Timing-Constrained Global Routing with Buffered Steiner Trees

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

vorgelegt von

Daniel Rotter

aus
Siegburg

Bonn, 15. März 2017

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

1. Gutachter:	Prof. Dr. Jens Vygen
2. Gutachter:	Prof. Dr. Stephan Held
Tag der Promotion:	2. Juni 2017
Erscheinungsjahr:	2017

Acknowledgements

This thesis would not have been possible without the support of many people.

First, I would like to thank my supervisors Prof. Dr. Jens Vygen and Prof. Dr. Stephan Held for their guidance and helpful discussions. Already during my Bachelor studies, Jens attracted my interest to Discrete Mathematics and accompanied me on the long journey through my Bachelor, Master, and PhD studies. Without Stephan's supervision I would have got stuck inside the IBM-jungle too many times. I would also like to thank Prof. Dr. Bernhard Korte for creating excellent working conditions at this institute.

Special thanks go to my former and present colleagues for helpful discussions and productive collaboration on several topics such as timing optimization, global routing, resource sharing, and pangea. Especially, I would like to thank Siad Daboul, Dr. Nicolai Hähnle, Dr. Dirk Müller, Pietro Saccardi, Rudolf Scheifele, and Dr. Ulrike Schorr.

I am thankful to many people at IBM for sharing their experience with me. In particular, I would like to thank Alexander J Suess for patiently answering questions about many timing related topics, Nancy Y Zhou and Steven Quay for helping me to run BuffOpt here in Bonn, and to Harald Folberth and Friedrich Schröder for the close and friendly collaboration with pangea.

I am grateful to Dr. Ulrich Brenner, Siad Daboul, Prof. Dr. Stephan Held, Dr. Dirk Müller, Dr. Ulrike Schorr, and Prof. Dr. Jens Vygen for proofreading this thesis or parts of it, and for valuable remarks and comments.

I would like to emphasize the importance of a good and friendly working atmosphere beyond everyday office life. Among many others I would like to thank Anna Borutzky, Dr. Ulrich Brenner, Siad Daboul, Dr. Nicolai Hähnle, Friederike Michaelis, Dr. Dirk Müller, Pietro Saccardi, Dr. Tomás Silveira Salles, Dr. Jan Schneider, Dr. Ulrike Schorr, and Kristina Stellwag for on-and off-season barbecues, both productive and funny business trips, celebrating carnival with me, for cheering me up during the most exhausting periods of my PhD studies, and for all occasions that required to order pizza to the conference room.

My biggest thank goes to my parents Gisela and Bernd Rotter for their great assistance during my whole life and to my girlfriend Eva Börgens for her lovely support and endless patience while I was writing this thesis.

Contents

Chapter 1

Introduction

Placement, routing, timing optimization – these are probably the most important steps of any physical chip design flow, i. e. in a simplified way, placing the circuits, connecting the pins, and optimizing the critical paths such that all signals arrive in time.

Several years ago, these steps have often been solved separately, but this approach seems to be insufficient for modern technologies. Since electronic devices have become smaller, faster, and more powerful, computer chips had to keep up with this development. Nowadays, the number of critical signals is overwhelmingly large while the available routing space is limited: roughly 1 km of wire needs to be packed on a chip that is not larger than $1\,\mathrm{cm}^2$.

"Optimizing the critical paths" very likely results in re-computing solutions for major parts of the chip if timing constraints are ignored during placement and routing.

The need for keeping an eye on timing during routing can be shown easily. Assume that the pins in Figure 1.1 have to be connected by horizontal and vertical wires. From a routing point of view, both solutions, 1.1(a) and 1.1(b), seem to be equally good as both trees are shortest possible. If we assume that the green pin s is a source pin from which we want to send a signal to t_1, t_2, and t_3, we might prefer Solution 1.1(b) since here, the paths from s to all sinks are shortest possible as well.

This is of course a simplified example. In practice, things are much more complicated. Instead of dealing with one instance only, we have to connect millions of sets of pins (called nets) which compete for the same routing space containing

(a) A shortest connection. A signal starting at s arrives late at t_3.

(b) Another shortest connection.

Figure 1.1: Two shortest connections of the pins consisting of horizontal and vertical wires only. Without taking timing constraints into account we might choose the left solution in which the signal starting at s arrives late at t_3.

- routing space on low layers allowing a dense packing of wires but on which signal delays are large, and
- routing space on high layers that yield space for only a small fraction of wires but that allow a fast signal transportation.

If the routing space needed by Solution 1.1(b) is used by other trees, we have to decide whether we choose a different solution (like Solution 1.1(a)) and take a detour or an unfavorable choice of layers into account, or if we re-route other nets in order to free the space needed by Solution 1.1(b).

Another difference between the simple example and practice is the estimate of the time signals need to traverse a connection. In practice the delay along a wire grows (roughly) quadratically with its length and replacing long wires by several shorter wires can result in overall better timing. This is done by a step called *buffering* in which additional gates with one input and one output are inserted such that the computed logical function remains unchanged.

Assume that in our example the horizontal wires need to be buffered. The yellow areas in Figure 1.2 depict the regions where we can insert a buffer. In all other regions, the available placement space is not sufficient.

(a) An optimally buffered connection if there is placement space everywhere.

(b) Restricted placement space completely changes wiring and buffering of an optimum solution.

Figure 1.2: Available buffer space influences both routing and buffering of a net. The horizontal wires need to be buffered but buffers can only be inserted in the yellow areas.

Here, we observe how the available placement space influences routing and buffering of the depicted net. Depending on placement constraints, Solution 1.2(b) can be optimum even if the path length from s to t_3 is large.

Although research is still far away from solving (most of) the tasks arising during physical design by one algorithm that takes all objectives into account, successful design flows nowadays have to combine tasks that have been solved separately in the past.

In this thesis we show how to solve the two central problems *buffering* and *global routing* by one single tool taking into account timing constraints, placement-, and routing-congestion. We investigate both theoretical and practical aspects of the problem.

In Chapter 2 we introduce the basics concepts of chip design needed in this thesis. On the routing side we will focus on global routing. During global routing, Steiner trees are computed in a coarse graph to quickly estimate global routability of a chip. On the timing side, we will concentrate on the buffering problem. In order to understand and solve that problem, we need knowledge about electrical properties of transistors and wires, and static timing analysis. Without re-defining it we use notations concerning the theory of graphs

and concepts in combinatorial optimization from the book by Korte and Vygen [KV12]. As the only exception to this we give a non-standard definition of a Steiner tree in Section 2.1.

In Chapter 3 we show how to model timing inside the *Min-Max Resource Sharing Problem*. The resource sharing approach has already proved to be effective for the *Standard Global Routing Problem* in theory and practice by Müller, Radke, Vygen [MRV11]. We extend their model and incorporate timing constraints. As most important sub-task we have to compute a solution for a single net. The resource sharing framework can be applied for various phases of optimization and for various delay models.

For the concrete goal of solving the combination of buffering and global routing, we have to compute a *buffered* Steiner tree in the global routing graph minimizing a sum of costs for timing, wiring congestion, placement congestion, and other metrics such as net length and power consumption. We show in Chapter 4 that this problem is already *NP*-hard for simple delay models that ignore slew effects such as the Elmore delay model and give a *fully polynomial approximation scheme* for instances with a constant number of pins. The key idea to obtain such an algorithm is to enumerate all possible routing topologies for a net and to compute buffered embeddings of it.

Already for medium-sized instances, a complete enumeration of all topologies is far beyond practicability. As an alternative we develop in Chapter 5 fast algorithms to compute *one* topology with provably good properties concerning timing and net length.

In contrast to the theoretical algorithms of Chapter 4 we concentrate on practical algorithms in Chapters 6 and 7. We divide the complex task of computing a buffered Steiner tree into the problem of computing a timing- and routing-congestion-aware Steiner tree (Chapter 6) and the task of buffering a given Steiner tree taking into account costs for routing congestion, placement congestion, timing, and electrical violations (Chapter 7).

Combining the results of Chapters 3 to 7 yields the tool BonnRouteBuffer that is part of the BonnTools suite of physical design optimization tools developed at the Research Institute for Discrete Mathematics, University of Bonn. We show results on real-world VLSI instances provided by our industrial cooperation partner IBM in Chapter 8.

As a byproduct of our main tool for global routing with buffered Steiner trees we obtain an algorithm for timing- and congestion-aware port assignment and assertion generation that can be used in early parts of physical design (Chapter 6).

Chapter 2

Basic Concepts of Chip Design

Life in the 21st century is certainly marked by the technology we are surrounded with. While 20 years ago the property of containing a *computer chip* was (at least almost) reserved for *computers*, surprisingly many devices are equipped with great computing power nowadays. TVs, mobile phones, watches, and even ordinary domestic appliances like refrigerators and heatings may contain computer chips superior to the largest super computers built in the 1980s.

One of the major requirements to make such a development possible was the production of *smaller* and *faster* computer chips. The process of creating these modern chips consisting of several billions of transistors is called *chip design*, or VLSI (=**v**ery **l**arge **s**cale **i**ntegration) design, and is probably one of the most fascinating applications of mathematical optimization. Due to its enormous complexity, a detailed introduction to VLSI design would certainly go beyond the scope of this thesis. Instead, we focus on the following aspects in this chapter:

1. In Section 2.3 we shortly explain how a chip is structured and introduce some notation to denote the parts of a chip.

2. To produce *smaller* chips we have to pack many wires on a small area. In Section 2.4 we will see how this is accomplished in today's computer chips. The core problem that arises thereby is the *Standard Global Routing Problem* that we also formulate.

3. To produce *faster* chips we have to speed-up signal delays through wires and the building blocks of a chip. Many different problems have to be solved to fulfill that task. After a short introduction to the basic concepts of signal propagation in Section 2.5.1 we consider one of these problems, the *Buffering Problem*, in detail in Section 2.5.5.

In addition to the chip design related topics mentioned above we will setup some mathematical notation. In large parts of this thesis we will build tree-like structures with certain properties like *small length*, *small delays*, and *small costs*. In Section 2.1 we give a slightly unusual but very general and flexible definition for *Steiner trees*. An important property of a Steiner tree is its *topology* which we define in Section 2.2.

For all other notation related to graph theory we refer to the book of Korte and Vygen [KV12]. In the whole thesis we denote the logarithm with basis 2 by log and the natural logarithm by ln. Logarithms with a basis $x \neq 2, e$ are denoted $\log_x$.

2.1 Steiner Trees

The most important structure of this thesis is the *Steiner tree*:

Definition 2.1 (Steiner tree) *Let G be a directed or undirected graph and $N \subseteq V(G)$. Let $s \in N$ be a special vertex (the source of N). A **Steiner tree** for N is a pair (A, κ) such that A is an arborescence rooted at s with $N \subseteq V(A)$ in which the set $\{\nu \in V(A) : \delta_A^+(\nu) = \emptyset\}$ of leaves of A is equal to $N \backslash \{s\}$ and κ is a function*

$$\kappa : V(A) \cup E(A) \to V(G) \cup (E(G) \cup \{\circ\}) \text{ such that}$$

- $\kappa(\nu) \in V(G)$ *for all* $\nu \in V(A)$,
- $\kappa(\zeta) \in E(G) \cup \{\circ\}$ *for all* $\zeta \in E(A)$,
- *for each* $(\nu, \omega) \in E(A)$ *either* $\kappa((\nu, \omega)) = \circ$ *and* $\kappa(\nu) = \kappa(\omega)$ *or*
  $$\kappa((\nu, \omega)) = \begin{cases} (\kappa(\nu), \kappa(\omega)) & \text{if } G \text{ is directed} \\ \{\kappa(\nu), \kappa(\omega)\} & \text{if } G \text{ is undirected,} \end{cases}$$
- $\kappa(v) = v$ *for* $v \in N$.

*If $N = \{s, t\}$, we call a Steiner tree for N an s-t **Steiner path**. The vertices in $V(A) \backslash N$ are called Steiner vertices or Steiner nodes.*

This definition is somehow unusual as in the literature a Steiner tree is usually defined as a connected and acyclic subgraph of G containing N.

Let $c : E(G) \cup \{\circ\} \to \mathbb{R}_{\geq 0}$ with $c(\circ) = 0$. If we are interested in finding Steiner trees (A, κ) minimizing

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)),$$

an optimum solution is always attained by (A, κ) such that $|\kappa^{-1}(e)| \leq 1$ for $e \in E(G)$. In this case we can identify A with the image of the restriction of κ to the set $\{\zeta \in E(A) : \kappa(\zeta) \neq \circ\}$ which is indeed a directed and acyclic subgraph of G containing N. We obtain the usual definition of a Steiner tree.

In this thesis we will mainly be interested in computing Steiner trees minimizing more complex cost functions including costs for delays. Restricting to solutions for which κ is injective would be too severe. Although we do not identify the complete Steiner tree with its image we identify $N \subseteq V(A)$ with its image in G under κ.

It is easy to see that a Steiner tree for N with source s in G exists if and only if all $t \in N$ are reachable from s in G. This condition can be checked in linear time by *breadth first search* or *depth first search* and is usually satisfied in our practical application.

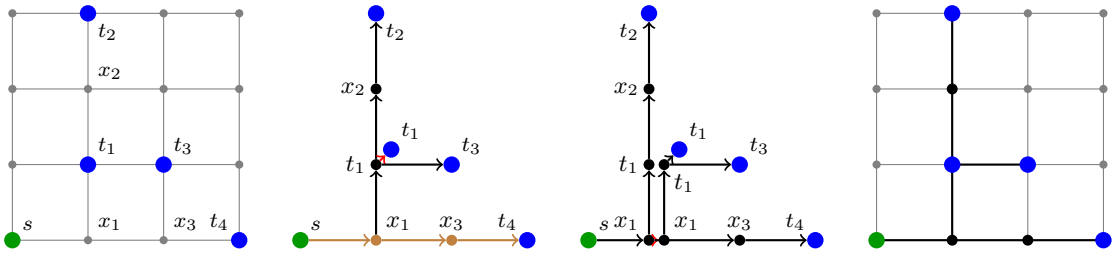
Henceforth we will assume that Steiner trees for our instances exist without explicitly mentioning it.

Sometimes we are interested in *rectilinear Steiner trees* rather than in Steiner trees in graphs. Using Definition 2.1 we can define a rectilinear Steiner trees as follows:

Definition 2.2 (Rectilinear Steiner tree) *Let $n \in \mathbb{N}$ and let $N \subset \mathbb{R}^n$ be a finite set containing an element $s \in N$. An n-**dimensional rectilinear Steiner tree** for N is a Steiner tree (A, κ) for N in the infinite and undirected graph*

$$\left(\mathbb{R}^n, \{\{p, q\} : p, q \in \mathbb{R}^n, p \neq q\}\right)$$

such that $\kappa(\nu)$ and $\kappa(\omega)$ differ in at most one coordinate for all $(\nu, \omega) \in E(A)$.



(a) An undirected graph G and a net N consisting of the blue and green nodes.

(b) A Steiner tree (A, κ) for N. The path $A_{[s,t_4]}$ is colored brown, the red edge is mapped to $\circ$ by κ.

(c) A Steiner tree that cannot be represented using the usual Steiner tree definition.

(d) Edges in G used by the two Steiner trees.

Figure 2.1: Two Steiner trees for which the sets $\kappa(E(A))$ are equal.

In nearly all applications we will be interested in the delay along paths inside Steiner trees.

Definition 2.3 *Let (A, κ) be a Steiner tree and let $\nu, \omega \in V(A)$ such that ω is reachable from ν in A. We define $A_{[\nu,\omega]}$ as the unique ν-ω path in A.*

By the assumption that A is rooted at s, each element of $N \backslash \{s\}$ is reachable from s. This property is important for our application as we will use Steiner trees to propagate "signals" from s to all remaining elements of N. Note that $\{\nu \in V(A) : \delta_A^+(\nu) = \emptyset\} = N \backslash \{s\}$ implies $V(A_{[s,t]}) \cap N = \{s, t\}$ for $t \in N$.

Examples of Steiner trees can be found in Figure 2.1.

2.2 Topologies

An important property of a Steiner tree is its *topology*.

Definition 2.4 (Topology) *A **topology** T for a net N with source s is an arborescence with $N \subseteq V(T)$ such that T is rooted at s and*

$$|\delta^+(v)| = \begin{cases} 0 & \text{if } v \in N \backslash \{s\}, \\ 1 & \text{if } v = s, \\ 2 & \text{otherwise.} \end{cases}$$

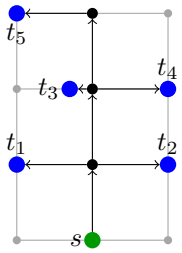
Vertices v with $|\delta^+(v)| = 2$ are called Steiner vertices or Steiner nodes.

Definition 2.5 *Let T be a topology and let $v, w \in V(T)$ such that w is reachable from v in A. We define $T_{[v,w]}$ as the unique v-w path in T.*

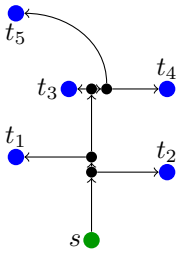
Definition 2.6 (Embedding of a topology) *An **embedding of a topology** T for N is a Steiner tree (A, κ) for N (in an arbitrary graph) such that there exists a function $\phi : V(T) \to V(A)$ with*

- *$\phi(v) = v$ for all $v \in N$,*
- *for $(v, w) \in E(T)$, $\phi(v) \in V\left(A_{[s,\phi(w)]}\right)$ and no vertex of $V(A_{[\phi(v),\phi(w)]}) \backslash \{\phi(v), \phi(w)\}$ is in the image of ϕ.*

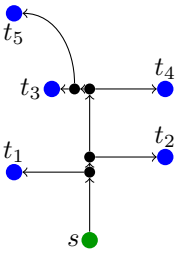
*We say that T is the **underlying topology** of (A, κ) if (A, κ) is an embedding of T.*



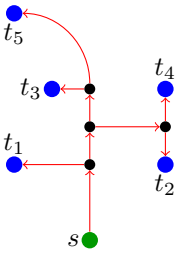
(a) Steiner tree (A, κ).



(b) A topology underlying to (A, κ).



(c) Another topology underlying to (A, κ).



(d) A topology not underlying to (A, κ).

Figure 2.2: Example of two different topologies underlying to the same Steiner tree and one topology that is not underlying to that tree. Here, $N = \{s, t_1, t_2, t_3, t_4, t_5\}$.

Note that each Steiner tree has an underlying topology but that this topology is not unique. See Figure 2.2 for an example. The property that (A, κ) is an embedding of T is independent of the function κ. Instead, we have an embedding relation $T \xrightarrow{\phi} A \xrightarrow{\kappa} G$.

From the strict degree constraints of a topology we can immediately derive some well-known properties:

Lemma 2.7 ("well known")

- *The number of Steiner nodes in a topology for N is $|N| - 2$.*
- *The number of different topologies for N is*

$$\prod_{i=1}^{|N|-2} (2i - 1) = \frac{(2|N| - 4)!}{2^{|N|-2}(|N| - 2)!}.$$

Proof For a topology T let S be the set of Steiner nodes. It holds that

$$\begin{aligned} |N| + |S| - 1 &= \sum_{\upsilon \in V(T)} |\delta_T^-(\upsilon)| \\ &= |E(T)| \\ &= \sum_{\upsilon \in V(T)} |\delta_T^+(\upsilon)| \\ &= 1 + 2|S| \end{aligned}$$

which implies the first fact. To prove the second fact we mention that for $k \geq 3$ all topologies for an instance with k pins can be obtained from the topologies for $k - 1$ pins by subdividing an edge and that the topologies created that way are pairwise different. By the first fact, the number of edges in a topology for k pins is equal to $2k - 3$ and hence,

$$(\#\text{ topologies for } k \text{ sinks}) = (2k - 3) \cdot (\#\text{ topologies for } k - 1 \text{ sinks}).$$

For a more detailed proof see Lemma 1.4 of [BZ15]. □

2.3 The Structure of a Computer Chip

Since this thesis will be largely technology independent, we can think of a computer chip as a collection of wires that connect smaller cells and the chip itself.

These cells can be *gates* implementing the basic logic functions such as AND, OR, and NOT, or they can be more complex chips again. Each of these cells contains pre-defined parts to which wires have to be connected. These parts are called *pins* and are usually very small such that we may consider them as points. The chip itself also contains pins, the so-called *primary pins*. In the context of signal propagation we distinguish between two types of pins, *input pins* and *output pins* (see Section 2.5.1). Especially if chips are built hierarchically, pins are also called *ports* (see Section 6.6).

The large set of pins is partitioned into subsets, called *nets*. Except for a few special cases that we do not consider in this thesis, each net contains exactly one *source* that distributes the signal to the sinks of the same net. A source is either a primary input pin or an output pin of a cell. All other pins of a net are primary output pins or input pins of cells. Wires have to be arranged such that they form a tree for each net as we will see in Section 2.4 in more detail.

For a more comprehensive description of the indeed much more complicated structure of computer chips see [KRV07][HHV15][Vyg16].

2.4 Packing of Wires

2.4.1 Routing Layers and Wire Codes

Wires are usually arranged in *routing layers*. Within each layer either only horizontal or only vertical wires are allowed. In the first case we say that a layer has *preferred direction* x while in the latter case it has *preferred direction* y. For the sake of simplicity we do not consider wires that go against the preferred direction (so-called *jogs*) in this introductory chapter although they are sometimes allowed in practice. Wires on adjacent routing layers can be connected by a wire in z-direction, a so-called *via*.

Wires have a certain width and require a certain distance (spacing) to each other. Widths and spacings are determined by a mapping layer $\mapsto$ (width, spacing) that is called *wire code*. Wire widths and spacings influence packing and timing optimization as follows:

- Wires with small width and spacing consume less routing space and are thus easier to pack.
- Wires with large width have a small resistance and wires with large spacing have small capacitance. These wires allow a faster signal propagation.

Each design has a *default wire code* that usually assigns smallest possible widths and spacings to all layers. The effect that even these smallest possible values are larger for higher layers than they are for the lowest layers is enforced by different usage of metal. The metal used on higher layers allows a faster signal propagation. This way, optimization algorithms that take into account packing (congestion) and timing constraints tend to put wires of trees for critical nets on higher layers while uncritical nets are connected by wires on the lower layers. If it is required to speed-up signal propagation on a wire on a low layer, it is possible to change the wire code such that a larger width and spacing is assigned there.

2.4.2 Global and Detailed Routing

The design step in which nets are connected by wires is called *detailed routing* (see e. g. [Ges+13][Ahr+15]). The output of this step is a set of Steiner trees (one for each net) in which each edge represents a wire in the available routing space (Section 2.4.1) with an assigned wire code.

Apart from the obvious requirements that wires must not overlap, the output of a detailed routing has to obey complex design rules and has to ensure sufficiently fast signal propagation. Among all solutions that satisfy these requirements we are looking for a solution with minimum total wire length.

The *Detailed Routing Problem* is hard in theory and practice. Even the simpler related problems of computing one shortest rectilinear Steiner tree ([GJ77], [GJ79] page 208f) and computing a set of edge disjoint paths in a grid graph [Sch09] are NP-hard.

To solve the problem in practice, there is a prior step called *global routing* which models the total routing space by a coarser grid graph. Instead of computing edge disjoint trees, we are looking for a set of Steiner trees in that graph obeying certain capacity constraints. Complex design rules are not taken into account. Finally, the solution of the global routing step serves as a guide for the detailed routing.

In its most basic form the *Global Routing Problem* can be defined as follows:

Standard Global Routing Problem

Instance: A directed or undirected graph G,
functions

$$\text{cap} : E(G) \to \mathbb{R}_{\geq 0}$$

and

$$\text{length} : E(G) \cup \{\circ\} \to \mathbb{R}_{\geq 0} \text{ with } \text{length}(\circ) = 0,$$

a set $\mathcal{N}$ of nets with $N \subseteq V(G)$ for $N \in \mathcal{N}$.

Output: A Steiner tree (A_N, κ_N) for N in G for each $N \in \mathcal{N}$ such that

$$\sum_{N \in \mathcal{N}} |\{\zeta \in E(A_N) : \kappa_N(\zeta) = e\}| \leq \text{cap}(e) \text{ for all edges } e \in E(G)$$

and among all these solutions

$$\sum_{N \in \mathcal{N}} \sum_{\zeta \in E(A_N)} \text{length}(\kappa_N(\zeta)) \text{ is minimum.}$$

In the above definition the space consumption of a wire is assumed to be 1. In the simplified setting of the *Standard Global Routing Problem*, this assumption makes sense since the selection of a non-default wire code is not necessary. In Chapter 3 we show how to incorporate timing constraints into the global routing problem. In this extended model, an edge ζ of a Steiner tree consumes a value $\text{usg}(\zeta)$ from its assigned edge in the global routing graph. This value is then equal to the sum of width and spacing of the wire and depends on the wire code.

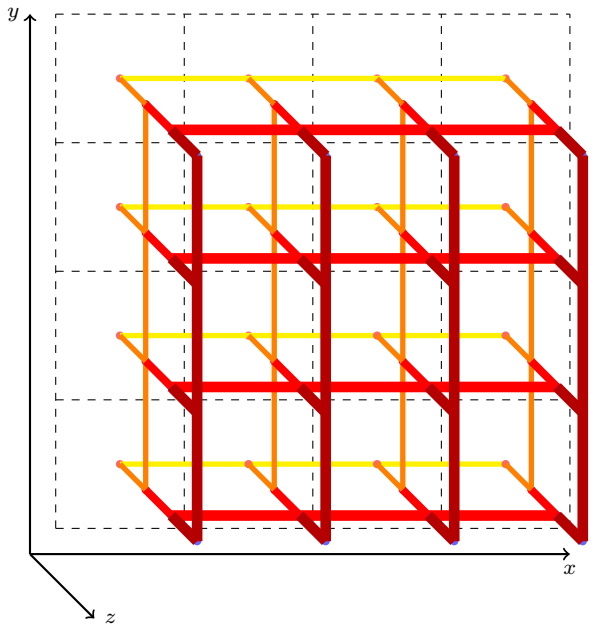


Figure 2.3: Partition of the chip area into tiles and the global routing graph formed by this partition in the case of four routing layers. Wires on higher layers are wider than wires on lower layers.

2.4.3 Global Routing Graph

In most practical applications the global routing graph is a 3-dimensional grid graph arising from a subdivision of the chip area as follows. Let $Z \in \mathbb{N}$ be the number of wiring planes, let $[0, W] \times [0, H]$ be the chip area, and let $0 = x_0 < x_1 < \ldots < x_w = W$, $0 = y_0 < \ldots < y_h = H$ be horizontal and vertical cuts ($w, h \in \mathbb{N}$). The vertices of the global routing graph represent the tiles. Two vertices are joined by an edge if one of the following conditions holds:

- They represent tiles with centers $\left(\frac{x_i+x_{i+1}}{2}, \frac{y_j+y_{j+1}}{2}, z\right)$ and $\left(\frac{x_{i+1}+x_{i+2}}{2}, \frac{y_j+y_{j+1}}{2}, z\right)$ of the same layer z with x as the preferred routing direction on z.
- They represent tiles with centers $\left(\frac{x_i+x_{i+1}}{2}, \frac{y_j+y_{j+1}}{2}, z\right)$ and $\left(\frac{x_i+x_{i+1}}{2}, \frac{y_{j+1}+y_{j+2}}{2}, z\right)$ of the same layer z with y as the preferred routing direction on z.
- They represent tiles with centers of the form $\left(\frac{x_i+x_{i+1}}{2}, \frac{y_j+y_{j+1}}{2}, z\right)$ and $\left(\frac{x_i+x_{i+1}}{2}, \frac{y_j+y_{j+1}}{2}, z+1\right)$.

Figure 2.3 depicts a global routing graph. An edge between adjacent tiles represents the routing space between them.

In the definition of the *Standard Global Routing Problem* a net is a subset of the vertex set. This can be achieved by projecting a pin to the node representing a tile that has non-empty intersection with that pin. To overcome the inaccuracy induced by these pin projections, Saccardi [Sac15] proposed a continuous global routing model that takes into account the actual pin positions.

Often, the layers of the global routing graph can be grouped into pairs such that for each of these layer pairs $\{z, z'\}$,

- $|z - z'| = 1$ and hence they have different preferred direction,
- wires on z and z' use the same metal,
- the default wire code assigns the same width and spacing to wires on z and z'.

After contracting the vias between two grouped layers, the global routing graph can be considered a special case of the more general global routing graph structure with vertex set $M \times \left\{1, \ldots, \frac{Z}{2}\right\}$ for a finite metric space (M, dist). The distance function dist allows us to define a geometric distance between two vertices as the distance between their projections onto M. We will use this property in later parts of this thesis (see e. g. Section 6.3.2).

2.5 Timing Optimization

In this section we briefly describe the basic concepts of timing optimization in VLSI design that are needed in this thesis. For a more comprehensive overview on timing optimization see [Hel08], [Sap04], and [Sch15].

2.5.1 Signals

The most important concept in timing optimization is a *signal*. A signal can be considered a voltage change over time at a certain pin. For the sake of simplicity we imagine that a chip has only two electrical potentials. The ground voltage V_0 represents the logical value 0. The positive voltage over ground, V_{dd}, represents the logic logical value 1. If the voltage changes from ground voltage V_0 to V_{dd} we speak of a *rising signal*. If the voltage changes from V_{dd} to V_0 we speak of a *falling signal*. The voltage change occurs gradually and requires a certain time (see Figure 2.4). The time at which the voltage at a pin has reached $50\%\, V_{\mathrm{dd}}$ is called *arrival time*. We call the time the signal needs to change from $10\%\, V_{\mathrm{dd}}$ to $90\%\, V_{\mathrm{dd}}$ (in case of a rising signal) or from $90\%\, V_{\mathrm{dd}}$ to $10\%\, V_{\mathrm{dd}}$ (in case of a falling signal) respectively, the *slew*.

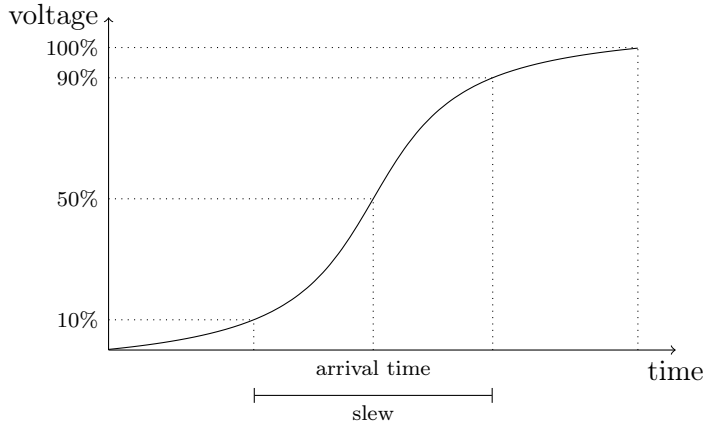
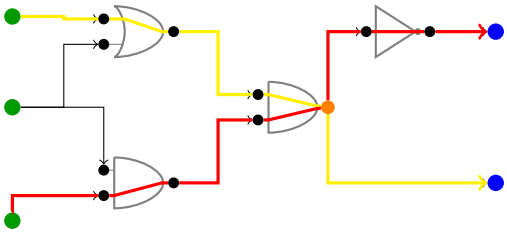
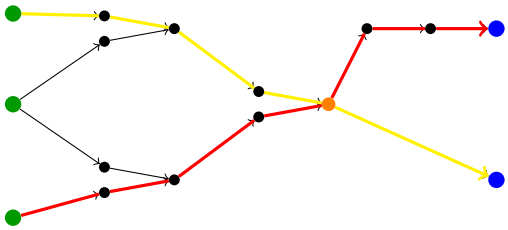


Figure 2.4: Example of a voltage change from V_0 to V_{dd}, i. e. a rising signal. The time at which the voltage reaches $50\%\, V_{\mathrm{dd}}$ is called *arrival time*. The time needed for the signal to change from $10\%\, V_{\mathrm{dd}}$ to $90\%\, V_{\mathrm{dd}}$ is called *slew*.



(a) A small chip with 16 pins and 4 gates. The green pins are the timing starting points, the blue pins are the timing end points.



(b) Corresponding standard timing graph consisting of 16 vertices, 9 net edges, and 7 gate edges.

Figure 2.5: Example of a small chip with corresponding standard timing graph. Two signal paths are highlighted (red and yellow). Each of these paths induces a signal at the orange pin. In this thesis we do not distinguish between different timing phases and assume that these signals are equal.

2.5.2 Standard Timing Graph

A signal at a source pin of a net induces signals at all sink pins of the same net. A signal at an input pin of a logic gate can induce signals at output pins of that gate. This dependency is modeled as an acyclic directed graph, the *standard timing graph*. The vertices of the standard timing graph are the pins of the chip. Two pins p, q are joined by an edge (p, q) if

- p is source and q is sink pin of the same net, or
- p is input and q is output pin of the same logic gate.

Figure 2.5 shows an example of a small timing graph.

The vertex set V of a standard timing graph can be written as $V = V_{\text{in}} \dot\cup V_{\text{out}} \dot\cup (V \backslash (V_{\text{in}} \cup V_{\text{out}}))$. Vertices in V_{in} represent pins at which signals start (primary input pins or latch output pins). The in-degree of these vertices is zero. Signals end in vertices in V_{out}. These vertices have no outgoing edges and represent timing endpoints (primary output pins and latch input pins). In Figure 2.5 the set V_{in} is colored green and V_{out} is colored blue. Maximal paths in the standard timing graph, i. e. V_{in} - V_{out} paths, are called *signal paths*. Their number can be exponential in the number of pins (see Figure 3.1 for an example). The yellow and the red path highlighted in Figure 2.5 are examples of signal paths.

A vertex of the standard timing graph can be reachable from several timing start points that originate different signals. In order to distinguish different signals (e. g. signals with different origins) we can associate them with a *phase*.
Taking different timing phases and their different arrival times, required arrival times, and delays into consideration is a straightforward task for all problems addressed in this thesis but requires a more complicated notation. For the sake of simplicity we do not distinguish between different timing phases and assume that there is exactly one signal.

2.5.3 Static Timing Analysis

The major task in (late mode) timing optimization is to make sure that the signal arrives in time at all timing end points. Static timing analysis (Hitchcock et al. [HSC82]) is a method to check this property.

Let D be a standard timing graph. We assume that we are given *arrival times* $\text{at}(v)$ at the timing start points $v \in V_{\text{in}} \subset V(D)$ and *required arrival times* $\text{rat}(w)$ at the timing

end points $w \in V_{\text{out}} \subset V(D)$. Let $d(e)$ be the time the signal needs to traverse an edge $e \in E(D)$. This value is called *delay* and there are several *delay models* that can be used to approximate it. In this thesis we will get to know three different delay models: the basic variant of the *Elmore delay model* (Section 4.2.1), the *linear delay model* (Section 6.1), and the *Elmore Delay Model with Slew Propagation* that takes slew effects into account (Section 7.1.3).

We say that all timing constraints are met if for each signal path P starting in $v \in V_{\text{in}}$ and ending in $w \in V_{\text{out}}$,

$$\text{at}(v) + \sum_{e \in E(P)} d(e) \leq \text{rat}(w).$$

Since the number of signal paths is usually very large, we cannot use this definition directly. Instead, we propagate the latest arrival time along the standard timing graph in topological order.

Starting with the given arrival times at timing start points we compute the latest arrival time at a vertex $u \in V(D) \backslash V_{\text{in}}$ recursively with the formula

$$\text{at}(u) = \max\{\text{at}(x) + d((x,u)) : (x,u) \in \delta_D^-(u)\}.$$

The signal arrives in time if $\text{at}(w) \leq \text{rat}(w)$ for all $w \in V_{\text{out}}$. This condition can be checked in linear time.

During early design phases, it will rarely be the case that all timing constraints are satisfied. Instead, we have to speed-up *critical* parts of the chip. To identify the most critical parts we first propagate required arrival times along the standard timing graph in reverse topological order. For $u \in V(D) \backslash V_{\text{out}}$ we recursively set

$$\text{rat}(u) = \min\{\text{rat}(x) - d((u,x)) : (u,x) \in \delta_D^+(u)\}.$$

We define the *worst slack* at a pin $u \in V(D)$ as

$$\text{wsl}(u) = \text{rat}(u) - \text{at}(u).$$

The smaller $\text{wsl}(u)$, the more critical pin u is.

2.5.4 Electrical Properties

Signal delays depend on electrical capacitances and slews that we must compute efficiently.

Capacitance. We need to compute the capacitance at the source pin or at the Steiner nodes of a Steiner tree for a net N. Given *capacitances* at all sink pins of N and the *capacitance per length* for all layer / wire code pairs, we can compute the missing capacitances in linear time by propagation in reverse topological order. The required capacitances and capacitance per length values are usually given with the library and the design.

Slew. For each timing start point $p \in V_{\text{in}}$ the slew $\text{slew}(p)$ at p comes with the design data. For each gate edge (v,w) of the standard timing graph we are given a function

$$\text{outslew}_{(v,w)} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$$

that depends on the slew at v and the capacitance at w. Slew changes induced by propagation along wires are given by functions

$$\text{wireslew}_{(z,\text{wc})} : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$$

for each pair (z, wc) of layer z and wire code wc. These functions depend on the length of the wire, the slew at v, and the capacitance at w.

In practice, these functions also depend on other parameters such as the type of the signal at v and w (e. g. *rising* or *falling* signal). For the sake of simplicity we ignore this fact here. As for the timing phases, distinguishing between rising and falling signals does not require new algorithmic ideas for all parts of this thesis.

After having computed Steiner trees for all nets and capacitances at all nodes of the Steiner trees, we can compute slews at all nodes by a linear number of evaluations of the above functions.

Except for monotonicity in each argument we do not make any assumptions on the functions outslew and wireslew but use them as black-box functions.

2.5.5 Buffering

Buffering is one of the main tasks in timing optimization and a main topic of this thesis. If consists of reducing electrical capacitances per net by inserting further gates, so-called *repeaters*. Repeaters can be either

- circuits implementing the identity function (so-called *buffers*), or
- circuits that turn rising signals into falling signals and vice versa (*inverters*).

We call a set containing all possible types of repeaters a *repeater library*. Repeaters of different type can differ by *size*, *power consumption*, *capacitance limit*, and their timing behavior. In this thesis we perform repeater insertion alongside with computation of Steiner trees. Our task will be to compute a *buffered Steiner tree* for each net:

Definition 2.8 (buffered Steiner tree) *Let L be a finite repeater library, let G be a directed or undirected graph, and let $N \subseteq V(G)$ be a net. We assume that for $l \in L$ we are given a set $V(l) \subseteq V(G)$ of nodes at which insertion of a repeater of type l is allowed.*

*A **buffered Steiner tree** is a Steiner tree (A, κ) for N in G together with a function $b : V(A) \to L \cup \{\square\}$ such that $b(t) = \square$ for $t \in N$ and $b(\nu) = l \Rightarrow \kappa(\nu) \in V(l)$.*

Setting $b(\nu) = l \in L$ represents to associate Steiner point v with a repeater of type l while $b(\nu) = \square$ represents to not associate ν with a repeater.

Of course one has to be careful that the logic function computed by the chip stays the same. In the context of buffering we deal with the netlist $\mathcal{N}$ obtained from the original netlist $\mathcal{N}_{\text{orig}}$ by removing all repeaters. For each sink pin t of a net $N \in \mathcal{N}$ we are given a *polarity*

$$\text{pol}(t) \in \{\text{invert}, \text{ident}\}.$$

This value is equal to the number of inverters on the path between the source s of N and t in $\mathcal{N}_{\text{orig}}$ and when computing a buffered Steiner tree $((A, \kappa), b)$ for N we have to make sure that the number

$$\left|\left\{\nu \in V(A_{[s,t]}) : b(\nu) \in L \text{ inverter}\right\}\right|$$

of inverters on the path between s and t is

- even if $\mathrm{pol}(t) = \mathrm{ident}$, and
- odd if $\mathrm{pol}(t) = \mathrm{invert}$.

There are several objectives to measure the quality of a buffered Steiner tree such as minimizing routing and placement congestion, and minimizing delays (see Chapters 4 and 7). The latter objective depends on the timing model we use. If we are using a delay model that considers electrical capacitances, the impact of buffering on delays is usually very large and without buffering (i. e. by setting $b(\nu) = \square$ for all Steiner nodes ν) we would always end up with a hopeless timing as we explain now.

In the presence of long interconnections, large wire capacitances and hence huge delays cannot be avoided. With buffering we can subdivide a long wiring path into several smaller pieces. Although we have to take the extra delay through the newly inserted repeaters into account, buffering can improve the timing of a design significantly. While the delay through a long wiring path can be almost quadratic in its length, the delay along an optimally buffered path is (almost) linear in its length.
In addition to timing improvements for long paths, buffering is important if nets have many sinks. For these nets, already the accumulated capacitances of the sink pins can result in a too bad timing. Buffering helps to replace a large net by several smaller ones.

Apart from the positive effect on the timing of a chip, capacitance reductions that result from a good buffering are important to obey electrical constraints. At source pins we are usually given a capacitance limit. Obeying these limits is necessary to reduce electromigration effects and to guarantee strengths of signals.
Besides capacitance limits there are slew limits at all sink pins as well as a slew limit for the timing phase that has to be obeyed by all signals. The latter is typically very tight and dominates slew limits at sink pins. Without capacitance reductions that result from buffering it would be impossible to obey slew limits.

2.5.6 Power Consumption

A chip consumes *power* and during timing optimization we have to keep this power consumption as small as possible. We distinguish between *static* and *dynamic* power consumption. Static power is consumed by each circuit when it is not switching. Dynamic power is consumed by a circuit when it is charging or discharging and is proportional to the product of its downstream capacitance and a *switching factor* that estimates the switching frequency of the circuit.

In the context of the buffering problem we assume that we are given a function

$$\mathrm{power} : L \cup \{\square\} \to \mathbb{R}_{\geq 0}$$

that determines the *static power consumption* of a repeater of type $l \in L$ and that assigns $\mathrm{power}(\square) = 0$. When we compute a buffered Steiner tree $((A, \kappa), b)$ we want to minimize the total static power consumption $\sum_{\nu \in V(A)} b(\nu)$.

For the sake of a simpler notation we do not address the dynamic power consumption directly in this thesis. In all parts in which we compute buffered Steiner trees, optimizing dynamic power consumption can be done analogously to optimizing the delay along a newly inserted repeater and the source gate of a net.

Chapter 3

Global Routing from a Timing Point of View

In this chapter we deal with the global routing problem introduced in Section 2.4.2.

One of the most important results on global routing is by Müller, Radke, and Vygen [MRV11] who approximated the *Standard Global Routing Problem* within a factor that is arbitrarily close to the approximation guarantee of an algorithm for the *Minimum Cost Steiner Tree Problem*. They also achieved good results on practical VLSI instances.

We show how to incorporate global static timing constraints into their approach by increasing the model by polynomial size only. Our approach works for many delay models, including the *Elmore delay model* (Section 4.2.1) and the *linear delay model* (Section 6.1).

The results of this chapter are joint work with Stephan Held, Dirk Müller, Rudolf Scheifele, Vera Traub, and Jens Vygen [Hel+17][Hel+15].

3.1 Timing: An Essential Objective for Global Routing

Global routing is an essential part of any modern physical design flow. It serves as preparation for detailed routing, is used for quick congestion estimation during placement, and is input to many steps in timing optimization. To achieve timing closure, it is necessary that nets on timing critical paths are routed on high layers (cf. Section 2.4.1) and connections to the most critical sinks are shortest possible. In some cases, wires need to have a wire code that allows even faster signal propagation at the cost of an increased routing space consumption. Making the right layer and wire code choices, and choosing efficient routing topologies trading-off delay and routing congestion, is a key task in global routing.

As many algorithms for global routing used in practice are not able to optimize timing directly there is a step called *layer and wire code assignment* in which wire codes and a range of wiring planes are assigned to timing-critical nets. These assignments then serve as constraints during non-timing-driven global routing. An example of a congestion driven layer assignment algorithm that uses a timing-unaware global router as black box for congestion analysis is CATALYST [Wei+13]. For more details on the layer assignment step see Section 8.2.

These two-step approaches have a big disadvantage: Layer ranges and wire codes can

only be assigned to entire nets. For nets containing both critical and uncritical sinks such an approach can involve a significant waste of routing resources on higher layers.

To overcome these limitations, several methods to address timing during global routing directly have been introduced.

For netlists consisting of two-terminal nets only, Albrecht et al. [Alb+02] gave a fully polynomial time approximation scheme (FPTAS) that finds a global routing minimizing buffer area, routing congestion, and sink delays using a multicommodity flow approach including net-based delay constraints.

Huang et al. [Hua+93] used net-based delay bounds and rejected Steiner trees violating these bounds. Hong et al. [Hon+97] generalized this idea and introduced path-based delay bounds. Now, Steiner trees leading to a delay violation of a path through the net are discarded. The major disadvantage of the algorithms by [Hua+93] and [Hon+97] is that routing capacities might be violated in order to meet net- or path based delay bounds that serve as hard constraints.

Differently from the previous authors, Vygen [Vyg04] introduced path-based delay bounds. By treating these similar to routing capacity constraints he was able to optimize timing together with routing congestion. As the number of critical paths is usually exponential in the size of the netlist, this approach does not yield a polynomial time algorithm. The algorithm by Vygen [Vyg04] is based on the *Min-Max Resource Sharing Problem* which we describe in more detail in Section 3.2.

Other notable approaches on global routing with timing constraints have been developed by Hu and Sapatnekar [HS02], Yan and Lin [YL04], and Yan, Lee, Chen, and Huang [Yan+06]. They all start with timing-driven but congestion-unaware Steiner trees for all nets that are embedded into the global routing graph minimizing congestion. The approaches differ in the way the trees are embedded. Recently, Samanta et al. [Sam+15] proposed to pre-compute a set of alternative timing-driven Steiner trees for each net.

3.2 Min-Max Resource Sharing

3.2.1 Resources and Customers: An Abstract View on Global Routing

In this section we consider the global routing problem from a more abstract point of view. Let N be a net and let (A, κ) be a Steiner tree for N in a global routing graph. The Steiner tree (A, κ) consumes from several *resources*:

- Since edges in A model wires on the actual chip, (A, κ) consumes a certain amount of *routing space*.
- A is also used to distribute signals from the electrical source s of N to its electrical sinks. All signal paths can be considered as resources with capacities equal to the difference between the required arrival time at the path's end point and the signal's arrival time at the starting point. The time a signal needs to traverse the unique path in A from s to a sink in $N \backslash \{s\}$ (and also the time needed to traverse the gate having s as its output pin) can be regarded as consumption of (A, κ) from these resources.
- Depending on the application we might also want to minimize *net length* or *power consumption* and notice that a Steiner tree consumes from these resources as well.

To obtain a mathematical model we consider a net as a *customer* consuming from a certain set of resources with given resource capacities. The goal of the *Min-Max Resource Sharing Problem* is to achieve that the total combined consumption of all customers from each resource is upper-bounded by its capacity, or, if this is not achievable, to bound the maximum violation of a resource capacity.

Min-Max Resource Sharing Problem

Instance: A set $\mathcal{C}$ of *customers*, a set $\mathcal{R}$ of *resources*,
a convex set B_c of solutions for each $c \in \mathcal{C}$,
convex functions $\text{usg}_{c,r} : B_c \to \mathbb{R}_{\geq 0}$ for all $c \in \mathcal{C}$, $r \in \mathcal{R}$.

Output: A solution $\text{sol}(c) \in B_c$ for all $c \in \mathcal{C}$ such that the maximum resource consumption

$$\max_{r \in \mathcal{R}} \sum_{c \in \mathcal{C}} \text{usg}_{c,r}(\text{sol}(c)) \quad \text{is minimum.}$$

A solution $\text{sol}(c) \in B_c$ for a customer $c \in \mathcal{C}$ is also called *block*. The functions $\text{usg}_{c,r}$ specify which percentage a solution for a customer c consumes from a resource r. Note that the minimum does not necessarily exist unless all B_c are compact (which will always be the case in our application). Since we focus on approximation algorithms, the existence of the minimum will not be important to us.

If the set B_c^I of solutions for a customer $c \in \mathcal{C}$ is finite we define B_c to be the set of formal convex combinations of B_c^I,

$$B_c := \left\{ \sum_{\text{sol} \in B_c^I} \mu_{\text{sol}} \cdot \text{sol} : \mu_{\text{sol}} \geq 0 \text{ for all sol} \in B_c^I, \sum_{\text{sol} \in B_c^I} \mu_{\text{sol}} = 1 \right\}.$$

A discrete function $\text{usg}_{c,r}^I : B_c^I \to \mathbb{R}_{\geq 0}$ can be extended to B_c by

$$\text{usg}_{c,r} \left(\sum_{\text{sol} \in B_c^I} \mu_{\text{sol}} \cdot \text{sol} \right) := \sum_{\text{sol} \in B_c^I} \mu_{\text{sol}} \cdot \text{usg}_{c,r}^I(\text{sol}).$$

For the *Standard Global Routing Problem*, $\mathcal{C}$ is the set of nets and $\mathcal{R}$ is the set of edges in the global routing graph G. Solutions B_N^I for a net $N \in \mathcal{C}$ are the Steiner trees for N. The function $\text{usg}_{N,e}^I$ tells which fraction of the available routing space at an edge $e \in E(G)$ is consumed by a Steiner tree for N. We obtain B_c and $\text{usg}_{N,e}$ as described above. We can think of $\text{usg}_{N,e}^I(\text{sol}(N))$ as the sum of wire widths and spacings of edges of Steiner tree $\text{sol}(N)$ for N mapped to e. By this interpretation it is easy to model usage of non-default wire codes and assignment of an extra spacing larger than the minimum spacing specified by the wire code. We will henceforth omit to mention spacings of wires explicitly.

Additional constraints and objectives such as minimizing total net length, total power consumption, or optimizing manufacturing yield can be modeled by adding further resources (see [MRV11], [Mül06], [Vyg04]).

Modeling timing constraints is more difficult. A naive approach would be to insert a resource for each signal path. Unfortunately, the number of these paths is usually exponential in the size of the netlist. An example of a netlist with an exponential number

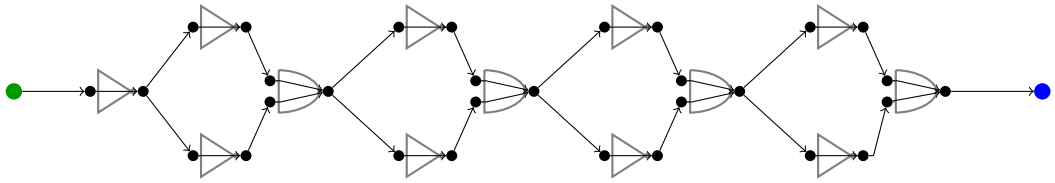


Figure 3.1: The number of timing paths can be exponential in the size of the netlist.

of signal paths is shown in Figure 3.1. Each path from the primary input (the green vertex) to the primary output colored in blue can contain the upper or the lower input pin of each of the four AND-gates, leading to a total number of $2^{\#\text{AND-gates}}$ maximal paths. Timing graphs that occur in practice are much more complex than depicted here and even enumerating all signal paths would lead to a hopeless running time. In this chapter we show how to model timing constraints by a *polynomial* number of additional resources and customers. For most practical instances, this number is even linear.

Block solvers. Usually, the sets B_c are not given explicitly but by oracle functions that optimize linear functions over them, called *block solvers*. For given resource prices $\text{price} : \mathcal{R} \to \mathbb{R}_{\geq 0}$ the block solver for a customer $c \in \mathcal{C}$ returns $\text{sol}(c) \in B_c$ approximately minimizing $\sum_{r \in \mathcal{R}} \text{price}(r) \cdot \text{usg}_{c,r}(\text{sol}(c))$.

Clearly, if these functions fail to return good solutions, we cannot hope for a good approximation for the overall problem. However, if we have block solvers with finite approximation ratios, we can find provably good solutions to the resource sharing problem as we present in the next section.

Note that the *Min-Max Resource Sharing Problem* is a generalization of the well-known *Multicommodity Flow Problem*: If all nets in our global routing application are two-terminal nets, finding (fractional) Steiner trees for all nets is identical to finding (fractional) paths. Hence, the *Standard Global Routing Problem* is identical to the *Multicommodity Flow Problem* in this special case. Garg and Könemann [GK07] solved several variants of it using a multiplicative price update strategy. The algorithms we present in the next section use similar strategies.

3.2.2 Algorithms for Min-Max Resource Sharing

In this section we give an overview over existing algorithms for the *Min-Max Resource Sharing Problem*. The common idea of these algorithms is the following: We define a price for each resource (initially all prices are 1) and use the block solvers to iteratively generate solutions for all customers that are approximately optimum with respect to the current prices. Whenever a solution consumes from a resource, we increase the resource's price and iterate this process with the new prices.
This way, "popular" resources become more and more expensive such that block solvers for some customers will eventually pick solutions not using these popular resources any more. In the end, customer c will receive the arithmetic mean of all its solutions. Note that the arithmetic mean is contained in B_c due to the convexity assumption. For a more formal description of this concept see Algorithm 1.

In most practical applications we prefer *integral* solutions over arbitrary elements of B_c. For instance, we cannot realize a convex combination of Steiner trees on the actual

Instance: Resources $\mathcal{R}$, customers $\mathcal{C}$, number of phases $p \in \mathbb{N}$, convex sets B_c for all $c \in \mathcal{C}$.
Output: Solutions $\text{sol}_c \in B_c$ for all $c \in \mathcal{C}$.

① set price$(r) := 1$ for all resources $r \in \mathcal{R}$
② **for** $i = 1$ **to** p **do**
③ **for** all customers $c \in \mathcal{C}$:
④ compute $\text{sol}_i(c) \in B_c$ approximately minimum w. r. t. the current prices
⑤ increase price(r) for resources r used by $\text{sol}_i(c)$
⑥ **return** $\left(\frac{1}{p} \cdot \sum_{i=1}^{p} \text{sol}_i(c)\right)_{c \in \mathcal{C}}$

Algorithm 1: General algorithm for solving *Min-Max Resource Sharing Problems*. Step ⑤ has to be specified.

chip. To round fractional solutions, we use a combination of *randomized rounding* [RT87] and traditional *rip-up and re-route* (see Section 3.8).

The first polynomial time approximation algorithm for the general version of the *Min-Max Resource Sharing Problem* is due to Grigoriadis and Khachiyan [GK94]. For each $\beta > 0$, their algorithm computes a $(1+\beta) \cdot \sigma$ approximation using $\widetilde{\mathcal{O}}\left(|\mathcal{C}| \cdot |\mathcal{R}| \cdot \beta^{-2}\right)$ calls to block solvers with approximation guarantee σ (step ④). Unfortunately, their result is restricted to the case that σ can be chosen arbitrarily close to 1. In 2008, Jansen and Zhang [JZ08] got rid of the restriction to σ while obtaining the same approximation ratio and running time as [GK94]. The fastest known algorithm is by Müller, Radke and Vygen [MRV11]. Instead of a quadratic dependence on the instance size, the running time of their algorithm only grows linearly in the number of resources and customers. More precisely, they proved the following result:

Theorem 3.1 ([MRV11]) *One can solve the Min-Max Resource Sharing Problem with approximation ratio* $(1+\beta) \cdot \sigma$ *for any* $\beta > 0$ *in* $\mathcal{O}(\theta(|\mathcal{C}| + |\mathcal{R}|) \log |\mathcal{R}|(\log\log|\mathcal{R}| + \beta^{-2}))$ *time. Here,* $\sigma \geq 1$ *is the worst approximation ratio of a block solver and* θ *is the time for an oracle call. If there exists a solution* $\{\text{sol}^*(c) : \text{sol}^*(c) \in B(c)$ *for* $c \in \mathcal{C}\}$ *such that* $\frac{1}{2} \leq \max_{r \in \mathcal{R}} \sum_{c \in \mathcal{C}} \text{usg}_{c,r}(\text{sol}^*(c)) \leq 2$*, the running time reduces to* $\mathcal{O}(\theta(|\mathcal{C}| + |\mathcal{R}|)\beta^{-2} \log |\mathcal{R}|)$.

One key idea to achieve an approximation ratio arbitrarily close to the approximation guarantee of the block solvers is to update prices in a *multiplicative* way. In the algorithm of Müller, Radke and Vygen [MRV11], a resource price p is updated to $p \cdot e^{\gamma \cdot \text{usg}}$ for a constant $\gamma > 0$ after a block solver has computed a new solution that consumes a fraction of usg from that particular resource. A detailed description of their core algorithm can be found in Algorithm 2.

3.3 Modeling Timing

As demonstrated by Figure 3.1, the number of paths in the timing graph can be exponentially large. Thus, adding a resource for each signal path would result in a set $\mathcal{R}$ of exponential size and the original algorithm of Müller, Radke and Vygen [MRV11] would no longer be a polynomial time algorithm. Modeling *timing-critical* paths only does not avoid this problem since firstly, this number can still be too large, and secondly, paths that have been uncritical before can become critical when their timing is ignored.

Instance: Resources $\mathcal{R}$, customers $\mathcal{C}$, number of phases $p \in \mathbb{N}$,
convex sets B_c for all $c \in \mathcal{C}$,
convex functions $\text{usg}_{c,r} : B_c \to \mathbb{R}_{\geq 0}$ for all $c \in \mathcal{C}$, $r \in \mathcal{R}$,
price adjust factor $\gamma > 0$.
Output: Solutions $\text{sol}_c \in B_c$ for all $c \in \mathcal{C}$.

① set $\text{price}(r) := 1$ for all resources $r \in \mathcal{R}$.
② set $X_c := 0$ and $x_{c,\text{sol}} := 0$ for all $c \in \mathcal{C}$ and $\text{sol} \in B_c$.
③ **for** $i = 1$ **to** p **do**
④ **while** there is $c \in \mathcal{C}$ with $X_c < i$:
⑤ compute $\text{sol}(c) \in B_c$ approximately minimum w. r. t. current prices
⑥ set $\xi := \min\{i - X_c, 1/\max\{\text{usg}_{c,r}(\text{sol}(c)) : r \in \mathcal{R}\}\}$
⑦ set $x_{c,\text{sol}} := x_{c,\text{sol}} + \xi$ and $X_c := X_c + \xi$
⑧ set $\text{price}(r) := \text{price}(r) \cdot e^{\gamma \cdot \xi \cdot \text{usg}_{c,r}(\text{sol}(c))}$ for all $r \in \mathcal{R}$
⑨ **return** $\left(\frac{1}{p} \sum_{\text{sol} \in B_c} x_{c,\text{sol}} \cdot \text{sol} \right)_{c \in \mathcal{C}}$

Algorithm 2: Algorithm of Müller, Radke, and Vygen [MRV11] for the *Min-Max Resource Sharing Problem*.

3.3.1 The Timing Graph

We now construct a directed graph D that helps us modeling timing in global routing more efficiently. The vertex set $V(D)$ consists of all vertices in the standard timing graph $\overline{D}$ introduced in Section 2.5.2 except for output pins of logic gates. More precisely, the vertex set $V(D)$ is defined as

$$V(D) = V_{\text{in}} \,\dot\cup\, V_{\text{gate}} \,\dot\cup\, V_{\text{out}},$$

where V_{in} is the set of timing starting points (primary input and latch output pins), V_{out} is the set of timing end points (primary output and latch input pins), and V_{gate} contains all input pins of the logic gates of the chip. Note that all signal paths start in V_{in} and end in V_{out}. The edges $E(D)$ of D correspond to signal propagation. Whenever we have a path $\overline{P}$ in $\overline{D}$, we will have a corresponding path P in D that arises from $\overline{P}$ by short cutting all subpaths of length 2 having a gate output pin as its middle vertex. More formally, $E(D)$ is the set

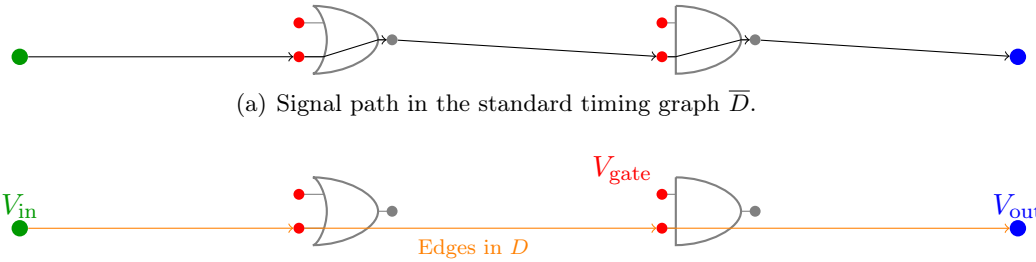
$$\{(v,w) : (v,w) \in E(\overline{D}) \text{ and } v, w \in V(D)\} \,\dot\cup$$
$$\{(v,w) : \text{there is a gate output pin } x \text{ such that } (v,x), (x,w) \in E(\overline{D})\}.$$

Figure 3.2 depicts D. Note that

$$|V(D)| \leq |V(\overline{D})| \text{ and } |E(D)| \leq |E(\overline{D})| + M,$$

where $M = \sum_{x \text{ output pin of a gate}} |\delta^-(x)| \cdot |\delta^+(x)|$. In practice, the number of input pins of a logic gate is a small constant and hence, $M = \mathcal{O}(|E(\overline{D})|)$, i. e. the size of D is linear in the size of $\overline{D}$. As $\overline{D}$ is acyclic, D is acyclic as well.

It is also possible to use $D = \overline{D}$ directly but not considering the gates' output pins results in faster convergence in practice and simplifies modeling the dependence of capacitances of solutions for net customers on gate delays.



(a) Signal path in the standard timing graph $\overline{D}$.

(b) Corresponding path in graph D. Vertices of D are primary input and latch output pins (V_{in}), input pins of logic gates (V_{gate}) and primary output, and latch input pins (V_{out}).

Figure 3.2: Directed graph D arising from the standard timing graph $\overline{D}$ described in Section 2.5.2. We use D to model timing within the Min-Max Resource Sharing model.

3.3.2 New Resources and Customers

The timing graph D can be used to model static timing within the resource sharing framework. Our model is based on the following simple and well-known observation:

Proposition 3.2 ("Folklore") *Let D be a directed graph and let $d : E(D) \to \mathbb{R}_+$, $a : V(D) \to \mathbb{R}_+$ be two functions.*

$$\begin{aligned} &\textit{If} && a(v) + d(e) \leq a(w) && \textit{for all } e = (v,w) \in E(D), \\ &\textit{then} && a(v) + \sum_{e \in E(P)} d(e) \leq a(w) && \textit{for all } v\text{-}w \textit{ paths } P \textit{ in } D. \end{aligned}$$

Proof For a path P with vertices $v_1, v_2, \ldots, v_k$ it holds that

$$\begin{aligned} a(v_1) + \sum_{e \in E(P)} d(e) &= a(v_1) + \sum_{i=2}^{k} d((v_{i-1}, v_i)) + \sum_{i=2}^{k-1} a(v_i) - \sum_{i=2}^{k-1} a(v_i) \\ &= \sum_{i=2}^{k} \Big(a(v_{i-1}) + d((v_{i-1}, v_i)) \Big) - \sum_{i=2}^{k-1} a(v_i) \\ &\leq \sum_{i=2}^{k} a(v_i) - \sum_{i=2}^{k-1} a(v_i). \end{aligned}$$

□

The result of Proposition 3.2 is not surprising since we have already used a similar result during the static timing analysis explained in Section 2.5.3: An exponential number of delay bounds on timing paths can be checked in linear time by defining *arrival times*. If we can find arrival times for all vertices of D such that

- the arrival time of the pin in V_{in} coincides with the signal's arrival time,
- arrival times of pins in V_{out} coincide with required arrival times, and
- the delay along an edge in D is not larger than the difference between the arrival times of its endpoints,

then, all signals arrive in time.

To make use of this knowledge in the resource sharing model we need a lower bound $a_{\min}(v)$ and an upper bound $a_{\max}(v)$ on the arrival time at a vertex $v \in V(D)$ in any timing-feasible solution. In Section 3.4 we describe how to choose these bounds.

In this section we show how to extend the resource sharing model based on the assumption that arrival time intervals $[a_{\min}(v), a_{\max}(v)]$ are given for all vertices. As in the resource sharing formulation of the *Standard Global Routing Problem* ([MRV11]) there will be a customer for each net and resources for routing congestion and other objectives like net length and power. However, these will no longer be the only customers and resources.

Timing resources and arrival time customers. Each edge $e = (v, w) \in E(D)$ is included as a *timing resource* with capacity $a_{\max}(w) - a_{\min}(v)$ and we assume that this capacity is strictly larger than zero. We include each vertex $v \in V_{\text{gate}} \cup V_{\text{out}}$ as *arrival time customer* whose feasible solutions is the set

$$[a_{\min}(v), a_{\max}(v)].$$

A solution $a(v) \in [a_{\min}(v), a_{\max}(v)]$ consumes an amount of

- $a_{\max}(v) - a(v)$ from resources $e \in \delta^-(v)$, and
- $a(v) - a_{\min}(v)$ from each $e \in \delta^+(v)$.

Net customers also consume from timing resources. Let N be a net with source pin s and let (A, κ) be a Steiner tree for N.

For $t \in N \backslash \{s\}$ we can compute the delay $\text{delay}_{(A,\kappa)}(s, t)$ along the unique s-t path in (A, κ). If s is the output pin of a gate g, we can also compute the delay $\text{delay}_g(u, s)$ through g between an input pin u of g and s.

In this thesis we assume that delays are positive and depend on (A, κ) only. This is the case for the linear delay model we use to estimate timing before buffering (Section 6.1) and for the Elmore delay model (Section 4.2.1) we use to measure timing in a buffered netlist. If delays are influenced by slews, this resource sharing model can still serve as a heuristic.

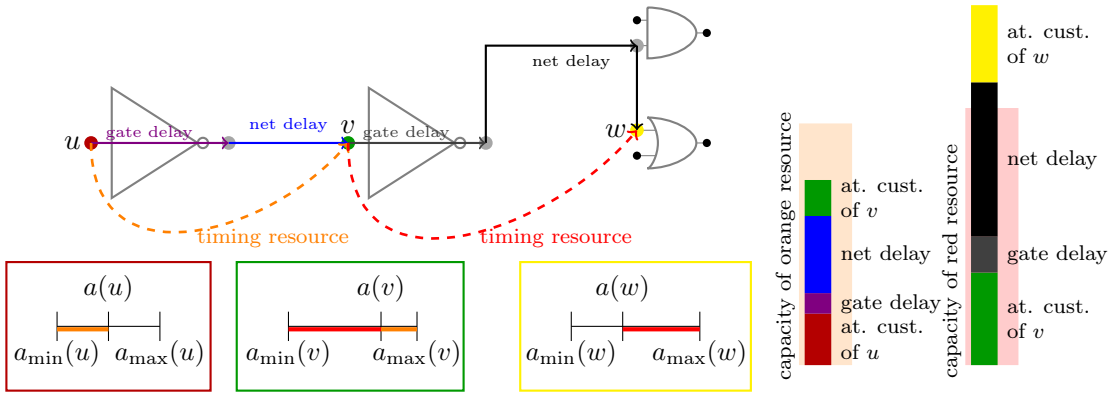
Net customer N consumes

- $\text{delay}_{(A,\kappa)}(s, t)$ from timing resource (s, t) for each $t \in N \backslash \{s\}$ if $s \in V_{\text{in}}$ and
- $\text{delay}_g(u, s) + \text{delay}_{(A,\kappa)}(s, t)$ from timing resource (u, t) for each input pin u of g and each $t \in N \backslash \{s\}$ if s is an output pin of gate g.

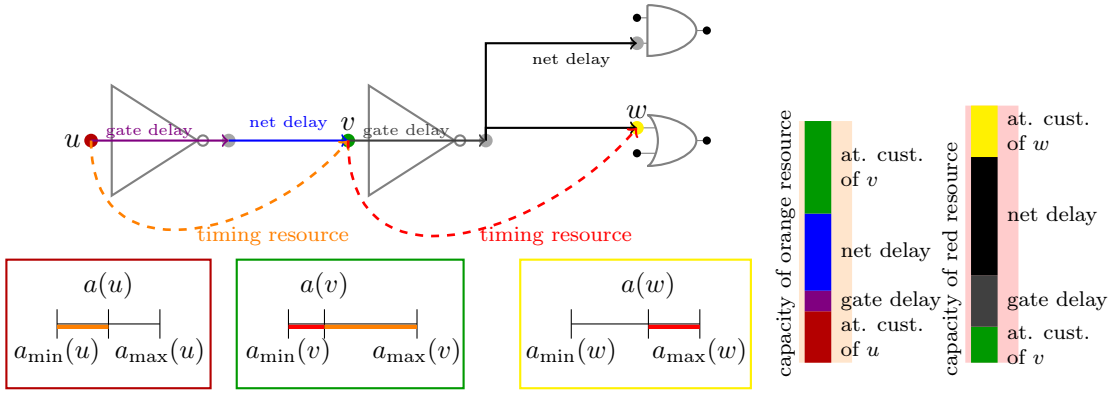
Timing relaxation resources. In addition to timing resources, we add a *relaxation resource* $\text{relax}(v)$ for timing endpoints $v \in V_{\text{out}}$ with $a_{\max}(v) > a_{\min}(v)$. This resource has capacity $\frac{a_{\max}(v) - a_{\min}(v)}{\beta}$, where $\beta > 0$ is the parameter from Theorem 3.1. Arrival time customer v is the only customer that consumes from it. Solution $a(v)$ consumes

$$\frac{a_{\max}(v) - a_{\min}(v)}{\beta} + a(v) - a_{\min}(v).$$

By this definition, each arrival time solution consumes at least 100% and selecting an arrival time later than $a_{\min}(v)$ leads to a violation and hence to a large resource price. This way, selecting an arrival time larger than $a_{\min}(v)$ is possible but very expensive.



(a) Delay violation due to bad choice of arrival times and Steiner tree for the black net.



(b) After re-choosing arrival times and rerouting the black net we achieve feasibility.

Figure 3.3: Interaction of timing resources and arrival time customers. The blue net customer consumes from the orange and the black net customer from the red resource. Their usage is equal to the delay through the Steiner tree and the pin's source gate. Arrival time customers also consume from timing resources. The earlier we choose the arrival time of v, the less we consume from the red timing resource but the more we consume from the orange one.

Example of the model Figure 3.3 shows how timing resources and arrival time customers interact. In Figure 3.3(a), the red timing resource is violated. The three customers consuming from it are the arrival time customers v and w, and the net connected by the black Steiner tree. The black Steiner tree itself is short – leading to small delay through the inverter in the middle. The detour on the path to w leads to a huge delay through the Steiner tree and hence to a large consumption from the red resource. As the arrival time at v is late and the arrival time at w is early, both arrival time customers consume a large amount from the red resource. On the other hand, the consumptions from the orange timing resource are small. Both u, v, and the net connected by the blue Steiner tree consume from that resource.

Figure 3.3(b) shows how to fix the violation of the red resource. Choosing an earlier arrival time at v leads to an increased consumption from the orange timing resource but to a decreased consumption from the red resource. We also choose a different black Steiner tree. Although the net length of the new tree (and hence the delay through the inverter in the middle) is significantly larger than before, the overall consumption from the red resource is smaller. The path to w through the tree is shortest possible now and thus, the consumption from the red resource is smaller.

3.4 Lower and Upper Bounds on Arrival Times

We now show how to obtain the arrival time intervals $[a_{\min}(v), a_{\max}(v)]$ for all vertices $v \in V(D)$ from which we can choose arrival times $a(v)$ that satisfy the first condition of Proposition 3.2. For the sake of faster convergence and better stability of the algorithm we want to achieve that these intervals are as small as possible.

3.4.1 Arrival Time Intervals Based on Lower Delay Bounds

For $e \in E(D)$ let $d_{\text{lb}}(e)$ be a lower bound on the delay along e. Independent of the delay model, $d_{\text{lb}}(e) = 0$ is always a valid choice. However, for some delay models better lower bounds can be computed in polynomial time.

By setting

$$a_{\text{lb}}^{\rightarrow}(v) = \begin{cases} \text{at}(v) & \text{if } v \in V_{\text{in}}, \\ \max\{a_{\text{lb}}^{\rightarrow}(u) + d_{\text{lb}}((u,v)) : (u,v) \in \delta^-(v)\} & \text{otherwise} \end{cases}$$

we can define lower bounds on the actual arrival time.

If $a_{\text{lb}}^{\rightarrow}(v)$ exceeds $\text{rat}(v)$ for $v \in V_{\text{out}}$ we will have no chance to meet all timing constraints and we have to relax required arrival times.

We define

$$\tau_r := \max\bigl\{0,\ \max\{a_{\text{lb}}^{\rightarrow}(v) - \text{rat}(v) : v \in V_{\text{out}}\}\bigr\}$$

and propagate (possibly relaxed) required arrival times using the formulas

$$a_{\text{lb}}^{\leftarrow}(v) = \begin{cases} \text{rat}(v) + \tau_r & \text{if } v \in V_{\text{out}}, \\ \min\{a_{\text{lb}}^{\leftarrow}(w) - d_{\text{lb}}((v,w)) : (v,w) \in \delta^+(v)\} & \text{otherwise.} \end{cases}$$

Setting $a_{\min}(v) = a_{\text{lb}}^{\rightarrow}(v)$ and $a_{\max}(v) = a_{\text{lb}}^{\leftarrow}(v)$ would already be a valid choice due to Proposition 3.3.

Proposition 3.3 *It holds that $a_{\text{lb}}^{\rightarrow}(v) \le a_{\text{lb}}^{\leftarrow}(v)$ for all $v \in V(D)$.*

Proof For $v \in V_{\text{out}}$, the statement is true by definition of τ_r. Otherwise, let $w \in V(D)$ such that $a_{\text{lb}}^{\leftarrow}(v) = a_{\text{lb}}^{\leftarrow}(w) - d_{\text{lb}}((v,w))$. By induction, $a_{\text{lb}}^{\rightarrow}(w) \le a_{\text{lb}}^{\leftarrow}(w)$ and hence, $a_{\text{lb}}^{\leftarrow}(v) = a_{\text{lb}}^{\leftarrow}(w) - d_{\text{lb}}((v,w)) \ge a_{\text{lb}}^{\rightarrow}(w) - d_{\text{lb}}((v,w)) \ge a_{\text{lb}}^{\rightarrow}(v)$. □

3.4.2 Shrinking Arrival Time Intervals with Upper Delay Bounds

For nodes on uncritical paths, the resulting intervals can be large as Figure 3.4 shows. In this simple example, v is completely uncritical and any time between 1 and 14 would be a feasible choice of arrival time. Unintuitively, arrival time selection for v seems to be a more difficult task than for the other nodes, where intervals contain one point only.

In the course of the resource sharing algorithm we might change $a(v)$ several times although any choice that leaves enough time for the incident edges would probably be equally good.

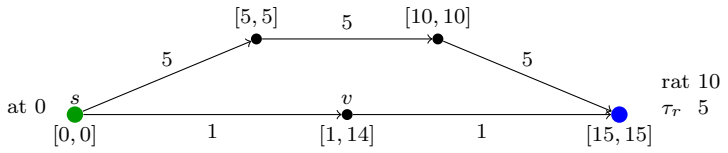


Figure 3.4: Example of arrival time ($a^{\rightarrow}_{\text{lb}}$) and required arrival time ($a^{\leftarrow}_{\text{lb}}$) propagation with respect to lower delay bounds d_{lb} shown at each edge. All arrival time intervals $[a^{\rightarrow}_{\text{lb}}, a^{\leftarrow}_{\text{lb}}]$ are feasible but at the uncritical vertex v the interval is large.

In practice, such a situation can lead to slow convergence and can be avoided by tightening the intervals. To do so, we define upper bounds $d_{\text{ub}}(e)$ on the delay along an edge $e \in E(D)$ as the delay in the worst reasonable solution.

As before, we can propagate arrival times and required arrival times with respect to delays d_{ub} to obtain

$$a^{\rightarrow}_{\text{ub}}(v) = \begin{cases} \text{at}(v) & \text{if } v \in V_{\text{in}} \\ \max\{a^{\rightarrow}_{\text{ub}}(u) + d_{\text{ub}}((u,v)) : (u,v) \in \delta^-(v)\} & \text{otherwise} \end{cases}$$

and

$$a^{\leftarrow}_{\text{ub}}(v) = \begin{cases} \text{rat}(v) & \text{if } v \in V_{\text{out}} \\ \min\{a^{\leftarrow}_{\text{ub}}(w) - d_{\text{ub}}((v,w)) : (v,w) \in \delta^+(v)\} & \text{otherwise.} \end{cases}$$

We can interpret $a^{\rightarrow}_{\text{ub}}(v)$ as the latest and $a^{\leftarrow}_{\text{ub}}(v)$ as the earliest reasonable arrival time at v. It is not necessary to choose $a(v) > a^{\rightarrow}_{\text{ub}}(v)$ as already $a(v) = a^{\rightarrow}_{\text{ub}}(v)$ leaves enough time for all edges on each path in D ending in v. Similarly, choosing $a(v) = a^{\leftarrow}_{\text{ub}}(v)$ leaves enough time for all edges on a path in D starting at v, and choosing $a(v) < a^{\leftarrow}_{\text{ub}}(v)$ is pointless.

Using this knowledge we define tightened arrival time intervals.

Definition 3.4 *Let $v \in V(D)$. If $a^{\rightarrow}_{\text{ub}}(v) > a^{\leftarrow}_{\text{ub}}(v)$, we define*

$$a_{\min}(v) := \max\{a^{\rightarrow}_{\text{lb}}(v), a^{\leftarrow}_{\text{ub}}(v)\} \quad \textit{and} \quad a_{\max}(v) := \min\{a^{\leftarrow}_{\text{lb}}(v), a^{\rightarrow}_{\text{ub}}(v)\}.$$

If $a^{\rightarrow}_{\text{ub}}(v) \leq a^{\leftarrow}_{\text{ub}}(v)$, we fix the arrival time at v to

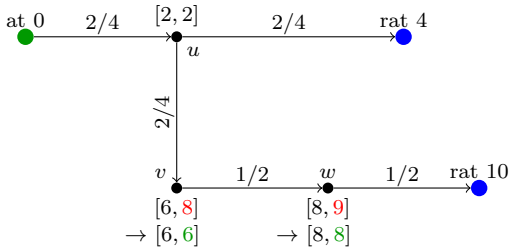
$$a_{\min}(v) := a_{\max}(v) := \frac{a^{\rightarrow}_{\text{ub}}(v) + a^{\leftarrow}_{\text{ub}}(v)}{2}.$$

We say that v is *critical* if $a^{\rightarrow}_{\text{ub}}(v) > a^{\leftarrow}_{\text{ub}}(v)$ and call v *uncritical* otherwise. If v is uncritical, any arrival time in $[a^{\rightarrow}_{\text{ub}}(v), a^{\leftarrow}_{\text{ub}}(v)]$ leaves enough time for all paths in D ending in v or starting with v and we can fix the arrival time to an arbitrary value within that interval. In this thesis we use the simple solution from Definition 3.4 and select the center of this interval. In the extended version of [Hel+17] we show how to distribute positive slack along uncritical paths by fixing the arrival time of an uncritical vertex v to

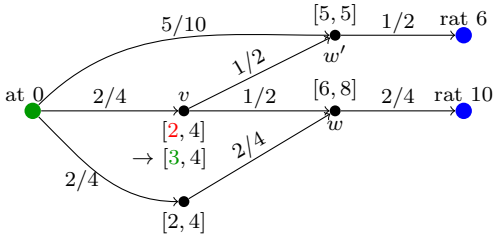
$$a_{\min}(v) := a_{\max}(v) := \ (1 - \mu(v)) \cdot a^{\rightarrow}_{\text{ub}}(v) + \mu(v) \cdot a^{\leftarrow}_{\text{ub}}(v)$$

for some choice of $\mu(v) \in [0,1]$ proportional to the fraction of the delay consumed at v in a longest path through v.

The next proposition tells us that we obtain arrival time intervals as desired.



(a) Taking into account that in each feasible solution the signal arrives at u at time 2 we can decrease $a_{\max}(v)$ from 8 to 6. Iterating this argument for (v, w), we can decrease $a_{\max}(w)$ to 8.



(b) In any feasible solution the signal arrives at w' at time 5 and selecting an arrival time earlier than 6 at w is not necessary. Using this observation we can increase $a_{\min}(v)$ to 3.

Figure 3.5: Examples in which we can tighten arrival time intervals further. Each edge e is labeled $d_{\mathrm{lb}}(e)/d_{\mathrm{ub}}(e)$. Arrival time intervals are shown next to internal vertices.

Proposition 3.5 *If* $0 \leq d_{\mathrm{lb}}(e) \leq d_{\mathrm{ub}}(e)$ *for all* $e \in E(D)$,

1. $[a_{\min}(v), a_{\max}(v)] \neq \emptyset$ *for all* $v \in V(D)$, *and*
2. $a_{\max}(v) + d_{\mathrm{ub}}((v, w)) \leq a_{\min}(w)$ *for all* $(v, w) \in E(D)$ *for which* v *or* w *is uncritical.*

Proof We first show 1. If v is uncritical, $a_{\min}(v) = a_{\max}(v)$ and 1 is fulfilled trivially. Let v be critical. If $a_{\min}(v) = a_{\mathrm{lb}}^{\rightarrow}(v)$ and $a_{\max}(v) = a_{\mathrm{lb}}^{\leftarrow}(v)$, $a_{\min}(v) \leq a_{\max}(v)$ follows from Proposition 3.3. Inequality $d_{\mathrm{lb}}(e) \leq d_{\mathrm{ub}}(e)$ guarantees $a_{\mathrm{lb}}^{\rightarrow}(v) \leq a_{\mathrm{ub}}^{\rightarrow}(v)$ and hence, yields 1 in the case $a_{\min}(v) = a_{\mathrm{lb}}^{\rightarrow}(v), a_{\max}(v) = a_{\mathrm{ub}}^{\rightarrow}(v)$. In the remaining cases, $a_{\min}(v) = a_{\mathrm{ub}}^{\leftarrow}(v) > a_{\mathrm{ub}}^{\rightarrow}(v) \geq a_{\max}(v)$ follows from the definition of criticality.

To prove 2 let $(v, w) \in E(D)$ with v or w uncritical. If v is uncritical and w is critical,

$$a_{\max}(v) = \tfrac{a_{\mathrm{ub}}^{\rightarrow}(v) + a_{\mathrm{ub}}^{\leftarrow}(v)}{2} \leq a_{\mathrm{ub}}^{\leftarrow}(v) \leq a_{\mathrm{ub}}^{\leftarrow}(w) - d_{\mathrm{ub}}((v, w)) \leq a_{\min}(w) - d_{\mathrm{ub}}((v, w)).$$

If v is critical and w is uncritical,

$$a_{\max}(v) + d_{\mathrm{ub}}((v, w)) \leq a_{\mathrm{ub}}^{\rightarrow}(v) + d_{\mathrm{ub}}((v, w)) \leq a_{\mathrm{ub}}^{\rightarrow}(w) \leq a_{\min}(w).$$

If both v and w are critical,

$$\begin{aligned} a_{\max}(v) + d_{\mathrm{ub}}((v, w)) &= \frac{1}{2}\Big(\big(a_{\mathrm{ub}}^{\rightarrow}(v) + d_{\mathrm{ub}}((v, w))\big) + \big(a_{\mathrm{ub}}^{\leftarrow}(v) + d_{\mathrm{ub}}((v, w))\big)\Big) \\ &\leq \frac{1}{2}\left(a_{\mathrm{ub}}^{\rightarrow}(w) + a_{\mathrm{ub}}^{\leftarrow}(w)\right) \\ &= a_{\min}(w). \end{aligned}$$

□

Assuming $d_{\mathrm{ub}}(e) = 3 \cdot d_{\mathrm{lb}}(e)$ in the example of Figure 3.4 we have $a_{\mathrm{ub}}^{\rightarrow}(v) = 3$ and $a_{\mathrm{ub}}^{\leftarrow}(v) = 7$. The uncritical vertex v gets the trivial arrival time interval $[5, 5]$.

3.4.3 Further Reduction of Arrival Time Intervals

In some cases the arrival time intervals can be reduced further. The right interval border $a_{\max}(v)$ of vertex v in Figure 3.5(a) is given by $a^{\rightarrow}_{\mathrm{ub}}(v) = 8$ and the computation of that arrival time assumed the maximum delay 4 along the edge entering u. Since u is critical, that edge has to be realized as fast as possible and the signal cannot arrive at v later than 6 in any timing-feasible solution (even if connection (u, v) is realized in the slowest possible way). As a consequence, we can reduce $a_{\max}(v)$ to 6. By the same argument we observe that we can also reduce $a_{\max}(w)$ and obtain an instance for which all arrival time intervals contain one point only.

To perform reductions of right interval borders we traverse the nodes in $V(D)$ in topological order. If for $v \in V(D)$,

$$a_{\min}(v) < a_{\max}(v) \quad \text{and} \quad a_{\max}(u) + d_{\mathrm{ub}}((u, v)) < a_{\max}(v) \text{ for all } (u, v) \in \delta^-(v),$$

we decrease $a_{\max}(v)$ to

$$a_{\max}(v) := \max\{a_{\min}(v), \max\{a_{\max}(u) + d_{\mathrm{ub}}((u, v)) : (u, v) \in \delta^-(v)\}\}. \tag{3.1}$$

Note that this choice guarantees that all timing intervals remain non-empty. Assuming that the signal does not arrive at a predecessor u of v later than $a_{\max}(u)$, we can conclude that the signal will also not arrive at v later than $a_{\max}(u) + d_{\mathrm{ub}}((u, v))$ and the update is feasible.

In the situation shown in Figure 3.5(b) we can increase the left interval border $a_{\min}(v)$ of vertex v. Here, the left interval borders of both successors are large – independent of the signal's arrival time at v. Both w and w' do not benefit from a small arrival time at v and choosing $a(v) = 3$ leaves enough time for the signal to arrive early enough at w and w' even if both connections leaving v are realized in the slowest possible way.

Similar to the previous algorithm we can increase lower bounds in linear time. This time we traverse V_{gate} in *reverse* topological order. If we encounter a vertex $v \in V_{\mathrm{gate}}$ with

$$a_{\min}(v) < a_{\max}(v) \quad \text{and} \quad a_{\min}(v) < a_{\min}(w) - d_{\mathrm{ub}}((v, w)) \text{ for all } (v, w) \in \delta^+(v),$$

we set

$$a_{\min}(v) := \min\{a_{\max}(v), \min\{a_{\min}(w) - d_{\mathrm{ub}}((v, w)) : (v, w) \in \delta^+(v)\}\}. \tag{3.2}$$

The next proposition states correctness of the algorithms.

Proposition 3.6 *After decreasing right interval borders and increasing left interval borders by the above algorithms the following statements are true*

1. *Proposition 3.5 still holds,*
2. *for all $v \in V(D)\backslash V_{\mathrm{in}}$ with $a_{\min}(v) < a_{\max}(v)$ there is $(u, v) \in \delta^-(v)$ such that $a_{\max}(u) + d_{\mathrm{ub}}((u, v)) \geq a_{\max}(v)$,*
3. *for all $v \in V_{\mathrm{gate}}$ with $a_{\min}(v) < a_{\max}(v)$ there is $(v, w) \in \delta^+(v)$ such that $a_{\min}(v) + d_{\mathrm{ub}}((v, w)) \geq a_{\min}(w)$.*

Proof This follows directly from the definition of the new interval borders. □

3.5 Properties of Low-Congestion Solutions

The next theorem tells us that all signals arrive in time if there is no violation of a timing resource. Consequently, we have expressed our timing constraints in terms of resource sharing correctly.

Theorem 3.7 *Assume that $\tau_r = 0$. Let $\mathcal{N}$ be the set of all nets and let $(\mathrm{sol}(c))_{c\in\mathcal{C}}$ be a solution, i. e.*

- *$\mathrm{sol}(c)$ is a (possibly fractional) routing of c if c is a net customer (i. e. $c \in \mathcal{N}$), and*
- *$\mathrm{sol}(c) \in [a_{\min}(c), a_{\max}(c)]$ is an arrival time if $c \in V(D)\backslash V_{\mathrm{in}}$ is an arrival time customer.*

The following holds:

1. *If $(\mathrm{sol}(c))_{c\in\mathcal{C}}$ does not violate any capacity constraints of a timing or relaxation resource, the (possibly fractional) routing $(\mathrm{sol}(N))_{N\in\mathcal{N}}$ meets all timing constraints.*
2. *If the routing $(\mathrm{sol}(N))_{N\in\mathcal{N}}$ meets all timing constraints and the delay along each edge $e \in V(D)$ defined by that solution lies between $d_{\mathrm{lb}}(e)$ and $d_{\mathrm{ub}}(e)$, there exist arrival times $\mathrm{sol}(v) \in [a_{\min}(v), a_{\max}(v)]$ for $v \in V_{\mathrm{gate}} \cup V_{\mathrm{out}}$ such that the solution $(\mathrm{sol}(c))_{c\in\mathcal{C}}$ with*
$$\mathrm{sol}(c) = \begin{cases} \mathrm{sol}(N) & \text{if } c = N \text{ is a net customer} \\ \mathrm{sol}(v) & \text{if } c = v \text{ is an arrival time customer} \end{cases}$$
does not violate any timing or relaxation resource.

Proof To prove Statement 1 we use Proposition 3.2 and show that for any signal path $P = v_1 v_2 \dots v_k$ in D consisting of nets $N_1, \dots, N_{k-1}$,

$$\mathrm{sol}(v_i) + d_i \le \mathrm{sol}(v_{i+1}) \text{ holds for } i = 1, \dots, k-1,$$

where $d_i := \mathrm{usg}_{N_i,(v_i,v_{i+1})}(\mathrm{sol}(N_i)) \cdot (a_{\max}(v_{i+1}) - a_{\min}(v_i))$ is the delay from v_i to v_{i+1} and $\mathrm{sol}(v_1) := \mathrm{at}(v_1)$ denotes the signal's arrival time at its origin $v_1 \in V_{\mathrm{in}}$.

Correctness of the inequality follows from the equivalence

$$\begin{aligned} &\mathrm{sol}(v_i) + d_i \le \mathrm{sol}(v_{i+1}) \Leftrightarrow \\ &(\mathrm{sol}(v_i) - a_{\min}(v_i)) + d_i + (a_{\max}(v_{i+1}) - \mathrm{sol}(v_{i+1})) \le a_{\max}(v_{i+1}) - a_{\min}(v_i). \end{aligned} \tag{3.3}$$

For $i > 1$, the summand $\mathrm{sol}(v_i) - a_{\min}(v_i)$ is exactly the usage of the arrival time customer v_i from timing resource (v_i, v_{i+1}) (and 0 for $i = 1$). The summand $a_{\max}(v_{i+1}) - \mathrm{sol}(v_{i+1})$ is equal to the usage of arrival time customer v_{i+1} from that resource. Hence, the left-hand side of the inequality is equal to the total amount of usage from resource (v_i, v_{i+1}) which is upper bounded by its resource capacity $a_{\max}(v_{i+1}) - a_{\min}(v_i)$ by assumption.
Since $\mathrm{relax}(v_k)$ is not violated and $\tau_r = 0$, $\mathrm{sol}(v_k) = a_{\min}(v_k) = \mathrm{rat}(v_k)$. Hence, $\mathrm{at}(v_1) + \sum_{i=1}^{k} d_i \le \mathrm{rat}(v_k)$.

Now we show Statement 2. For $e \in V(D)$ let $d(e)$ be the delay along e in routing $\mathrm{sol}(N)_{N\in\mathcal{N}}$. We use static timing analysis (Section 2.5.3) to compute arrival times $\mathrm{at}(v)$ for $v \in V(D)\backslash V_{\mathrm{in}}$ by forward propagation of delays $d(e)$ ($e \in E(D)$) starting with the initial arrival times $\mathrm{at}(v)$ for $v \in V_{\mathrm{in}}$. For the sake of simplicity we consider timing start points $v \in V_{\mathrm{in}}$ as arrival time customers with arrival time interval $[\mathrm{at}(v), \mathrm{at}(v)]$ in that proof.

For $v \in V(D)$ we define $\mathrm{sol}(v)$ as the projection of $\mathrm{at}(v)$ into $[a_{\min}(v), a_{\max}(v)]$, i. e.

$$\mathrm{sol}(v) = \min\{\max\{a_{\min}(v), \mathrm{at}(v)\}, a_{\max}(v)\}.$$

Let $e = (v, w) \in E(D)$. Since the routing solution is timing-feasible and by Proposition 3.2, $\mathrm{at}(v) + d(e) \leq \mathrm{at}(w)$ for all $e = (v, w) \in E(D)$. We show that the inequalities $\mathrm{sol}(v) + d(e) \leq \mathrm{sol}(w)$ hold as well by case distinction.

If v or w are uncritical, Proposition 3.5 implies $\mathrm{sol}(v) + d(e) \leq a_{\max}(v) + d_{\mathrm{ub}}(e) \leq a_{\min}(w) \leq \mathrm{sol}(w)$. It remains to consider the case that both v and w are critical.

- If $\mathrm{sol}(v) \leq \mathrm{at}(v)$ and $\mathrm{sol}(w) \geq \mathrm{at}(w)$, $\mathrm{sol}(w) \geq \mathrm{at}(w) \geq \mathrm{at}(v) + d(e) \geq \mathrm{sol}(v) + d(e)$.
- Now we consider the case $\mathrm{at}(v) < \mathrm{sol}(v)$, in particular $\mathrm{sol}(v) = a_{\min}(v)$.
  - If the left border of v has been increased during the arrival time interval reduction algorithm, $\mathrm{sol}(w) \geq a_{\min}(w) \overset{(3.2)}{\geq} a_{\min}(v) + d(e) = \mathrm{sol}(v) + d(e)$.
  - The case $\mathrm{sol}(v) = a^{\rightarrow}_{\mathrm{lb}}(v)$ is not possible as $d(e') \geq d_{\mathrm{lb}}(e')$ for all $e' \in E(D)$ and hence, $\mathrm{sol}(v) > \mathrm{at}(v) \geq a^{\rightarrow}_{\mathrm{lb}}(v)$.
  - In the remaining case, $\mathrm{sol}(v) = a^{\leftarrow}_{\mathrm{ub}}(v)$ and $\mathrm{sol}(w) \geq a_{\min}(w) \geq a^{\leftarrow}_{\mathrm{ub}}(w) \geq a^{\leftarrow}_{\mathrm{ub}}(v) + d(e) = \mathrm{sol}(v) + d(e)$.
- Finally, assume that $\mathrm{at}(w) > \mathrm{sol}(w)$ ($\Rightarrow \mathrm{sol}(w) = a_{\max}(w)$) and $\mathrm{sol}(v) > a_{\min}(v)$.
  - If the right border of w has been decreased during the arrival time interval reduction algorithm, $\mathrm{sol}(v) \leq a_{\max}(v) \overset{(3.1)}{\leq} a_{\max}(w) - d(e) = \mathrm{sol}(w) - d(e)$.
  - The case $\mathrm{sol}(w) = a^{\rightarrow}_{\mathrm{ub}}(w)$ is impossible due to the inequality $d(e') \geq d_{\mathrm{lb}}(e')$ for all $e' \in E(D)$.
  - If $a^{\leftarrow}_{\mathrm{lb}}(w) = \mathrm{sol}(w)$, $\mathrm{sol}(v) \leq a_{\max}(v) \leq a^{\leftarrow}_{\mathrm{lb}}(v) \leq a^{\leftarrow}_{\mathrm{lb}}(w) - d(e) = \mathrm{sol}(w) - d(e)$.

As the routing meets all timing constraints, $\tau_r = 0$ and $\mathrm{at}(v) \leq \mathrm{rat}(v)$ for $v \in V_{\mathrm{out}}$. We conclude that $\mathrm{sol}(v) \leq \mathrm{rat}(v)$ for all timing endpoints and hence, no timing endpoint relaxation resources are violated. With (3.3) we conclude that no timing resources are violated. □

It is also possible to give a theoretical lower bound on the worst slack if no resource is violated by more than a factor of $1 + \beta$.

Theorem 3.8 *Let $\mathrm{sol}(c)_{c \in \mathcal{C}}$ be a solution as in Theorem 3.7. For $e \in E(D)$ let $d(e)$ be the delay along e in that solution.*

If no resource is used by more than $1 + \beta$, then the worst slack

$$\min_{\substack{P \text{ path from} \\ v_0 \in V_{\mathrm{in}} \text{ to } v_k \in V_{\mathrm{out}}}} \left(\mathrm{rat}(v_k) - \mathrm{at}(v_0) - \sum_{e \in E(P)} d(e) \right)$$

is at least $-(\tau_r + \beta H)$, where

$$H := \max_{\substack{P \text{ path from} \\ V_{\mathrm{in}} \text{ to } V_{\mathrm{out}}}} \left(\sum_{e \in E(P)} d_{\mathrm{ub}}(e) \right) + \max_{\substack{P \text{ path from} \\ V_{\mathrm{in}} \text{ to } V_{\mathrm{out}}}} \left(\sum_{e \in E(P)} (d_{\mathrm{ub}}(e) - d_{\mathrm{lb}}(e)) \right) |E(P)|.$$

We omit a detailed proof here but refer to [Hel+17].

3.6 Block Solvers

The most important step in the algorithm of Müller, Radke, and Vygen [MRV11] is to find $\text{sol}(c) \in B(c)$ for customer $c \in \mathcal{C}$ such that

$$\sum_{r \in \mathcal{R}} \text{price}(r) \cdot \text{usg}(\text{sol}(c)) \text{ is (approximately) minimum.}$$

An algorithm performing this task is called *block solver*. The call to the block solver is done in step (4) of Algorithm 2. For the *Standard Global Routing Problem* the customers were exactly the nets in our netlist and the task of their block solvers is the *Minimum Cost Steiner Tree Problem*. The *Minimum Cost Steiner Tree Problem* can be approximated within a factor of 1.39 in polynomial time ([Byr+13], [Goe+12]) or within a factor of 2 in almost-linear time by the algorithm of Prim for the *Minimum Spanning Tree Algorithm* [Pri57]. This situation changes completely after adding timing resources and arrival time customers. Apart from the fact that we have a different type of customers now, the task of the block solver for the net customer gets more complicated when taking timing into account. In addition to resource usage costs for congestion, net length, and power resources (that can be translated to edge costs in the global routing graph), net customers have to pay for consumption from timing resources. The exact block solver depends on the particular delay model and for most common delay models, the resulting task is difficult. Often, constant factor approximations can only exist for special cases unless *P=NP*.

Since most of the later chapters of this thesis will be dedicated to the problem of finding block solvers of net customers for various delay models (Chapters 4 to 7) we restrict to block solvers for arrival time customers here.

Block solvers for arrival time customers. The task of the block solver for arrival time customer $v \in V_{\text{gate}}$ is to find an element $a(v) \in [a_{\min}(v), a_{\max}(v)]$ minimizing

$$\begin{aligned} f(a(v)) := &\sum_{(u,v) \in \delta^-(v)} \left(\text{price}((u,v)) \cdot \frac{a_{\max}(v) - a(v)}{a_{\max}(v) - a_{\min}(u)} \right) \\ &+ \sum_{(v,w) \in \delta^+(v)} \left(\text{price}((v,w)) \cdot \frac{a(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \right). \end{aligned}$$

Similarly, the block solver for an arrival time customer $v \in V_{\text{out}}$ of a timing endpoint has to find $a(v) \in [a_{\min}(v), a_{\max}(v)]$ minimizing

$$\begin{aligned} \overline{f}(a(v)) :=& \text{price}(\text{relax}(v)) \cdot \left(1 + \beta \cdot \frac{a(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(v)} \right) \\ &+ \sum_{(u,v) \in \delta^-(v)} \left(\text{price}((u,v)) \cdot \frac{a_{\max}(v) - a(v)}{a_{\max}(v) - a_{\min}(u)} \right) \end{aligned}$$

which is equivalent to minimizing

$$\begin{aligned} &\overline{f}(a(v)) - \text{price}(\text{relax}(v)) \\ &= \beta \cdot \text{price}(\text{relax}(v)) \cdot \frac{a(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(v)} + \sum_{(u,v) \in \delta^-(v)} \left(\text{price}((u,v)) \cdot \frac{a_{\max}(v) - a(v)}{a_{\max}(v) - a_{\min}(u)} \right). \end{aligned}$$

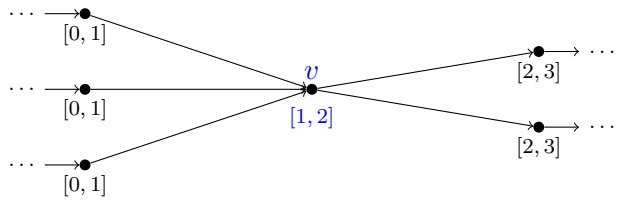


Figure 3.6: Problem of the simple block solver for timing resources. We assume that price adjust parameter γ in Algorithm 2 is 1 and the current usages of all timing resources are 0. Choosing arrival time 2 at v is optimum but after the price update, 1 is optimum.

This can be considered a special case of the task of a block solver for an arrival time customer in V_{gate}: Insert a new vertex v' and a new edge (v, v') for $v \in V_{\text{out}}$. The new edge represents relax(v) and gets price $\text{price}((v, v')) := \beta \cdot \text{price}(\text{relax}(v))$. After setting $a_{\max}(v') := a_{\max}(v)$ and $\delta^+(v) := \{(v, v')\}$ we can write $\overline{f}(a(v)) - \text{price}(\text{relax}(v))$ as $f(a(v))$.

In the remainder of Section 3.6 we consider a particular arrival time customer $v \in V_{\text{gate}} \cup V_{\text{out}}$ and show how to compute $a(v)$.

3.6.1 A Simple but Unstable Block Solver for Arrival Time Customers

Since f is linear with slope

$$\sum_{(v,w)\in\delta^+(v)} \frac{\text{price}((v,w))}{a_{\max}(w) - a_{\min}(v)} - \sum_{(u,v)\in\delta^-(v)} \frac{\text{price}((u,v))}{a_{\max}(v) - a_{\min}(u)}, \tag{3.4}$$

it attains its minimum at the borders and the following choice of $a(v)$ is optimum:

$$a(v) = \begin{cases} a_{\min}(v) & \text{if (3.4)} \geq 0 \\ a_{\max}(v) & \text{otherwise.} \end{cases}$$

Although this choice is optimum, it leads to slow convergence in practice. Even if preceding and succeeding resources are (nearly) balanced, one of the borders of the timing interval is returned. In case of poor lower and upper bounds (Section 3.4), these intervals are large and switching from $a_{\min}(v)$ to $a_{\max}(v)$ or vice versa between two iterations can change the situation completely.

Figure 3.6 shows how quickly a switch between interval borders can occur. Recall that the algorithm of Müller, Radke, and Vygen [MRV11] updates resource prices as $\text{price}(r) = e^{\gamma \cdot \xi \cdot \text{usg}_{c,r}(b(c))}$. In Figure 3.6, γ and ξ are chosen to be 1 and initially, the usages from timing resources are zero leading to a resource price of 1 for each resource. Thus, $a(v) = 2$ is the optimum arrival time at v. The solution $a(v) = 2$ induces a relative resource usage of $\frac{1}{2}$ from all timing resources corresponding to edges leaving v and hence, to a resource price of $\sqrt{e} > 1.6$ for these resources. From the other timing resources, nothing is consumed. Consequently, after choosing $a(v) = 2$ (and the resulting price update step), the other interval border $a(v) = 1$ has become optimum and the previous solution $a(v) = 2$ has become the overall worst solution.

① **for** $i = 1, \ldots, n$ **do**
② **compute** $a_i(v) \in \{a_{\min}(v), a_{\max}(v)\}$ minimizing $f(a_i(v))$
③ **set** $\text{price}(r) := \text{price}(r) \cdot e^{\gamma \cdot \frac{1}{n} \cdot \text{usg}_{v,r}(a_i(v))}$ for $r \in \mathcal{R}$
④ **return** $a(v) := \frac{1}{n} \sum_{i=1}^{n} a_i(v)$.

Algorithm 3: Iterated block solver for arrival time customers.

3.6.2 Stabilizing Arrival Time Computation by Iteration

This problem can be overcome by running the block solver for $n > 1$ iterations in each resource sharing phase, thereby selecting a fraction of $\frac{1}{n}$ from each computed solution (Algorithm 3).

Following the proof of Müller, Radke, and Vygen [MRV11], choosing a solution returned by the block solver by a small fraction only, does not invalidate the approximation ratio of their result. Decreasing ξ in line ⑥ of Algorithm 2 has an impact on running time only. For $n \to \infty$ we will end up with an arrival time that is still optimum after the price update:

Theorem 3.9 *Let $\gamma \geq 0$ and let*

$$g(a) := \sum_{(v,w) \in \delta^+(v)} \frac{\text{price}((v,w))}{a_{\max}(w) - a_{\min}(v)} \cdot e^{\gamma \cdot \frac{a - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)}} - \sum_{(u,v) \in \delta^-(v)} \frac{\text{price}((u,v))}{a_{\max}(v) - a_{\min}(u)} \cdot e^{\gamma \cdot \frac{a_{\max}(v) - a}{a_{\max}(v) - a_{\min}(u)}}.$$

Let $a(v)$ be the output of Algorithm 3, let a^* be the unique root of $g(a)$ and let $\overline{a}^* := \min\{\max\{a_{\min}(v), a^*\}, a_{\max}(v)\}$ be the projection of a^* into the arrival time interval $[a_{\min}(v), a_{\max}(v)]$.

It holds that $|a(v) - \overline{a}^*| \leq \frac{a_{\max}(v) + a_{\min}(v)}{n}$. In particular, the output of Algorithm 3 converges to $\overline{a}^*$ for $n \to \infty$.

Proof We may assume that $a_{\min}(v) < a_{\max}(v)$ as otherwise, the statement is trivial. First note that g is strictly monotonically increasing. We denote by $\text{price}^i(r)$ the price of resource $r \in \mathcal{R}$ and by f_i the function f at the beginning of iteration i. By definition of the price update step in Algorithm 2 it holds for $(v,w) \in \delta^+(v)$ and $(u,v) \in \delta^-(v)$ that

$$\text{price}^i((v,w)) = \text{price}^1((v,w)) \cdot e^{\frac{\gamma}{n} \cdot \sum_{i'=1}^{i-1} \frac{a_{i'}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)}} \quad \text{and}$$

$$\text{price}^i((u,v)) = \text{price}^1((u,v)) \cdot e^{\frac{\gamma}{n} \cdot \sum_{i'=1}^{i-1} \frac{a_{\max}(v) - a_{i'}(v)}{a_{\max}(v) - a_{\min}(u)}}.$$

If $a^* \leq a_{\min}(v)$, then $0 \leq g(a_{\min}(v))$ and for $i \leq n$:

$$\begin{aligned} f_i(a_{\min}(v)) &= \sum_{(u,v) \in \delta^-(v)} \text{price}^i((u,v)) \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(u)} \\ &< \sum_{(u,v) \in \delta^-(v)} \text{price}^1((u,v)) \cdot e^{\gamma \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(u)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(u)} \\ &\leq \sum_{(v,w) \in \delta^+(v)} \text{price}^1((v,w)) \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \\ &\leq f_i(a_{\max}(v)). \end{aligned}$$

Hence, we will always choose the left interval border and $a(v) = a_{\min}(v) = \overline{a}^*$. Analogously, if $a^* \geq a_{\max}(v)$, we will always choose the right interval border and end up with $a(v) = a_{\max}(v) = \overline{a}^*$.

In the case $a_{\min}(v) < a^* < a_{\max}(v)$ we show that for fixed n, Algorithm 3 selects solution $a_{\min}(v)$ at most q_n times, where

$$q_n = \left\lceil \frac{a_{\max}(v) - a^*}{a_{\max}(v) - a_{\min}(v)} \cdot n \right\rceil.$$

In the beginning of an iteration $i \leq n$ in which we have already selected the left interval border q_n times, it holds that

$$\begin{aligned}
&f_i(a_{\min}(v)) \\
&= \sum_{(u,v)\in\delta^-(v)} \text{price}^1((u,v)) \cdot e^{\frac{\gamma}{n}\cdot q_n \cdot \frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(v)-a_{\min}(u)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(u)} \\
&\geq \sum_{(u,v)\in\delta^-(v)} \text{price}^1((u,v)) \cdot e^{\gamma \cdot \frac{a_{\max}(v)-a^*}{a_{\max}(v)-a_{\min}(u)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(v) - a_{\min}(u)} \\
&= \sum_{(v,w)\in\delta^+(v)} \text{price}^1((v,w)) \cdot e^{\gamma \frac{a^*-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \\
&= \sum_{(v,w)\in\delta^+(v)} \text{price}^1((v,w)) \cdot e^{\frac{\gamma}{n}\left(n - n\cdot\frac{a_{\max}(v)-a^*}{a_{\max}(v)-a_{\min}(v)}\right)\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \\
&> \sum_{(v,w)\in\delta^+(v)} \text{price}^1((v,w)) \cdot e^{\frac{\gamma}{n}\left((i-1) - n\cdot\frac{a_{\max}(v)-a^*}{a_{\max}(v)-a_{\min}(v)}\right)\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \\
&\geq \sum_{(v,w)\in\delta^+(v)} \text{price}^1((v,w)) \cdot e^{\frac{\gamma}{n}((i-1)-q_n)\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}} \cdot \frac{a_{\max}(v) - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)} \\
&= f_i(a_{\max}(v)).
\end{aligned}$$

Thus, we select $a_i(v) = a_{\max}(v)$ in that iteration.

An analogous calculation shows that we select the right interval border $a_{\max}(v)$ at most $\overline{q}_n$ times with $\overline{q}_n := \left\lceil \frac{a^*-a_{\min}(v)}{a_{\max}(v)-a_{\min}(v)} \cdot n \right\rceil$.

From $\left(\frac{a_{\max}(v)-a^*}{a_{\max}(v)-a_{\min}(v)} \cdot n\right) + \left(\frac{a^*-a_{\min}(v)}{a_{\max}(v)-a_{\min}(v)} \cdot n\right) = n$ we conclude that $n \leq q_n + \overline{q}_n \leq n+1$ and hence,

- $a_i(v) = a_{\min}(v)$ must have been selected in at least $q_n - 1$ iterations, and
- $a_i(v) = a_{\max}(v)$ must have been selected in at least $\overline{q}_n - 1$.

Hence,

$$\begin{aligned}
a(v) &\leq \frac{1}{n}\left(\left(\frac{a_{\max}(v) - a^*}{a_{\max}(v) - a_{\min}(v)} \cdot n + 1\right) a_{\min}(v) + \left(\frac{a^* - a_{\min}(v)}{a_{\max}(v) - a_{\min}(v)} \cdot n + 1\right) a_{\max}(v)\right) \\
&= a^* + \frac{a_{\max}(v) + a_{\min}(v)}{n},
\end{aligned}$$

and

$$\begin{aligned}
a(v) &\geq \frac{1}{n}\left(\left(\frac{a_{\max}(v) - a^*}{a_{\max}(v) - a_{\min}(v)} \cdot n - 1\right) a_{\min}(v) + \left(\frac{a^* - a_{\min}(v)}{a_{\max}(v) - a_{\min}(v)} \cdot n - 1\right) a_{\max}(v)\right) \\
&= a^* - \frac{a_{\max}(v) + a_{\min}(v)}{n}.
\end{aligned}$$

□

Theorem 3.9 implies that Algorithm 3 needs

$$\mathcal{O}\Big((|\delta^+(v)| + |\delta^-(v)|) \cdot \tfrac{a_{\max}(v)+a_{\min}(v)}{\delta}\Big)$$

time to approximate $\overline{a}^*$ up to additive accuracy $\delta > 0$.

3.6.3 Stabilizing Arrival Time Computation with Newton's Method

Instead of spending this pseudo-polynomial running time we use Newton's method to approximate $\overline{a}^*$.

Theorem 3.10 *We can approximate $\overline{a}^*$ from Theorem 3.9 up to accuracy $\delta > 0$ in time*

$$\mathcal{O}\Big((|\delta^+(v)| + |\delta^-(v)|) \cdot \log\log\Big(\tfrac{a_{\max}(v)-a_{\min}(v)}{\delta}\Big)\Big).$$

Proof If $g(a_{\min}(v)) \geq 0$ or $g(a_{\max}(v)) \leq 0$ we return $\overline{a}^* = a_{\min}(v)$ or $\overline{a}^* = a_{\max}(v)$, respectively. So assume that $a_{\min}(v) < \overline{a}^* = a^* < a_{\max}(v)$.

Recall that function $g(a)$ is of the form

$$\begin{aligned} g(a) :=& \sum_{(v,w)\in\delta^+(v)} \frac{\text{price}((v,w))}{a_{\max}(w) - a_{\min}(v)} \cdot e^{\gamma\cdot\frac{a-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}} \\ & - \sum_{(u,v)\in\delta^-(v)} \frac{\text{price}((u,v))}{a_{\max}(v) - a_{\min}(u)} \cdot e^{\gamma\cdot\frac{a_{\max}(v)-a}{a_{\max}(v)-a_{\min}(u)}} \\ =& \sum_{(v,w)\in\delta^+(v)} \frac{\text{price}((v,w))}{a_{\max}(w) - a_{\min}(v)} \cdot e^{\gamma\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)}\cdot\left(\frac{a-a_{\min}(v)}{a_{\max}(v)-a_{\min}(v)}\right)} \\ & + \sum_{(u,v)\in\delta^-(v)} \left(-\frac{\text{price}((u,v))}{a_{\max}(v) - a_{\min}(u)} \cdot e^{\gamma\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(v)-a_{\min}(u)}}\right) \cdot e^{\gamma\cdot\left(-\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(v)-a_{\min}(u)}\right)\cdot\left(\frac{a-a_{\min}(v)}{a_{\max}(v)-a_{\min}(v)}\right)}. \end{aligned}$$

Hence, by substituting $x(a) := \frac{a-a_{\min}(v)}{a_{\max}(v)-a_{\min}(v)}$ and by defining

$$p(e) := \begin{cases} \frac{\text{price}((v,w))}{a_{\max}(w)-a_{\min}(v)} & \text{if } e = (v,w) \in \delta^+(v) \\ -\frac{\text{price}((u,v))}{a_{\max}(v)-a_{\min}(u)} \cdot e^{\gamma\cdot\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(v)-a_{\min}(u)}} & \text{if } e = (u,v) \in \delta^-(v), \end{cases}$$

and

$$q(e) := \begin{cases} \frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(w)-a_{\min}(v)} & \text{if } e = (v,w) \in \delta^+(v) \\ -\frac{a_{\max}(v)-a_{\min}(v)}{a_{\max}(v)-a_{\min}(u)} & \text{if } e = (u,v) \in \delta^-(v), \end{cases}$$

we can write $g = \overline{g} \circ x$, where $\overline{g}$ is of the form

$$\overline{g}(x) = \sum_{j=1}^{m} p_j e^{\gamma q_j x}$$

for $m = |\delta^+(v)| + |\delta^-(v)|$ and constants $p_i, q_i \in \mathbb{R}$ with $p_j \cdot q_j > 0$ and $|q_j| \leq 1$ for $j = 1, \ldots, m$.

Starting with an arbitrary $x_0 \in [0,1]$, Newton's method iteratively sets $x_{i+1} := x_i - \frac{\overline{g}(x_i)}{\overline{g}'(x_i)}$. By Taylor's theorem,

$$0 \;=\; \overline{g}(x(a^*)) \;=\; \overline{g}(x_i) + \overline{g}'(x_i)\cdot(x(a^*)-x_i) + \overline{g}''(\xi)\frac{(x(a^*)-x_i)^2}{2}$$

for some value $\xi \in [x(a^*), x_i] \cup [x_i, x(a^*)]$, which implies

$$|x(a^*) - x_{i+1}| \;=\; \frac{|\overline{g}''(\xi)|}{2|\overline{g}'(x_i)|}\cdot|x(a^*)-x_i|^2.$$

Now,

$$\frac{|\overline{g}''(\xi)|}{|\overline{g}'(x_i)|} = \frac{\left|\sum_{j=1}^{m} p_j q_j^2 \gamma^2 e^{\gamma q_j \xi}\right|}{\left|\sum_{j=1}^{m} p_j q_j \gamma e^{\gamma q_j x_i}\right|} \;\le \max_{j=1}^{m} \frac{|p_j q_j^2 \gamma^2 e^{\gamma q_j \xi}|}{|p_j q_j \gamma e^{\gamma q_j x_i}|} \le \max_{j=1}^{m} |q_j|\gamma \cdot e^{\gamma\cdot|\xi - x_i|} \;\le \gamma \cdot e^{\gamma\cdot|x^* - x_i|}$$

for all ξ, x_i and hence,

$$|x(a^*) - x_i| \;\le\; \frac{\gamma \cdot e^{\gamma\cdot|x(a^*)-x_{i-1}|}}{2}\cdot|x(a^*)-x_{i-1}|^2.$$

As starting point x_0 we choose a point with $|x_0 - x(a^*)| \le \frac{1}{2\gamma}$. Using the fact that $\overline{g}$ is monotonically increasing, this can be achieved by a constant number of iterations of binary search for fixed γ:
Initially, we set $y_{\min} := 0$ and $y_{\max} := 1$. While $y_{\max} - y_{\min} > \frac{1}{\gamma}$ we halve the length of interval $[y_{\max}, y_{\min}]$ without losing the property that root $x(a^*)$ is contained in it. If $\overline{g}\left(\frac{y_{\max}+y_{\min}}{2}\right) < 0$, we know $x(a^*) \ge \frac{y_{\max}+y_{\min}}{2}$ by monotonicity of $\overline{g}$ and we can update $y_{\min}$ to $\frac{y_{\max}+y_{\min}}{2}$. Otherwise, $x(a^*) \in \left[y_{\min}, \frac{y_{\max}+y_{\min}}{2}\right]$ an we set $y_{\max} := \frac{y_{\max}+y_{\min}}{2}$.
This binary search procedure needs $\mathcal{O}\left(\log\left(\frac{1}{\gamma}\right)\right) = \mathcal{O}(1)$ iterations and after termination, $x_0 := \frac{y_{\max}+y_{\min}}{2}$ fulfills $|x_0 - x(a^*)| \le \frac{1}{2\gamma}$.

Now we show by induction that for all $i \ge 0$,

1. $|x(a^*) - x_i| \le |x(a^*) - x_0|$ and
2. $|x(a^*) - x_i| \le \left(\frac{\gamma \cdot e^{\gamma|x(a^*)-x_0|}}{2}\right)^{2^i-1} \cdot |x(a^*) - x_0|^{2^i}$.

Since both statements are trivial for $i = 0$ we assume that $i > 0$. It holds that

$$\begin{aligned}
|x(a^*) - x_i| &\le \frac{\gamma \cdot e^{\gamma\cdot|x(a^*)-x_{i-1}|}}{2}\cdot|x(a^*)-x_{i-1}|^2 \\
&\le \left(\frac{\gamma \cdot e^{\gamma\cdot|x(a^*)-x_0|}}{2}\cdot|x(a^*)-x_0|\right)\cdot|x(a^*)-x_0| \\
&\le \frac{\sqrt{e}}{4}\cdot|x(a^*)-x_0| \\
&< 0.413\cdot|x(a^*)-x_0|
\end{aligned}$$

which implies 1. Statement 2 follows from

$$|x(a^*) - x_i| \leq \frac{\gamma \cdot e^{\gamma \cdot |x(a^*)-x_0|}}{2} \cdot \left(\left(\frac{\gamma \cdot e^{\gamma |x(a^*)-x_0|}}{2} \right)^{2^{i-1}-1} \cdot |x(a^*) - x_0|^{2^{i-1}} \right)^2$$
$$= \left(\frac{\gamma \cdot e^{\gamma |x(a^*)-x_0|}}{2} \right)^{2^i-1} \cdot |x(a^*) - x_0|^{2^i}.$$

We can use 2 to bound the error in iteration i as

$$|x(a^*) - x_i| \leq \left(\frac{\gamma \cdot e^{\gamma |x(a^*)-x_0|}}{2} \right)^{2^i-1} \cdot |x(a^*) - x_0|^{2^i} \leq \left(\frac{\gamma \cdot \sqrt{e}}{2} \right)^{2^i-1} \cdot \left(\frac{1}{2\gamma} \right)^{2^i} \leq \frac{1}{\gamma} \cdot \frac{1}{2^{2^i}}.$$

Thus, after $i = \log\log\left(\frac{a_{\max}(v)-a_{\min}(v)}{\delta \cdot \gamma}\right)$ Newton steps, each of which takes $\mathcal{O}(|\delta^+(v)| + |\delta^-(v)|)$ time, the absolute errors can be bounded as

$$|x(a^*) - x_i| \leq \frac{1}{\gamma} \cdot \frac{1}{2^{2^{\log\log\left(\frac{a_{\max}(v)-a_{\min}(v)}{\delta \cdot \gamma}\right)}}} = \frac{\delta}{a_{\max}(v) - a_{\min}(v)}$$

and $|a^* - x^{-1}(x_i)| = (a_{\max}(v) - a_{\min}(v)) \cdot |x(a^*) - x_i| \leq \delta$. □

3.7 Overall Algorithm

After extending our resource sharing model according to Section 3.3 we apply Algorithm 2. As block solver for arrival time customers we can either use a block solver from Section 3.6.1 or 3.6.2 (in which case we obtain a provably good overall algorithm) or the Newton based block solver from Section 3.6.3 (in which case we obtain fast convergence and good results). Note that the proof of Müller, Radke, and Vygen [MRV11] cannot be applied directly if we use the latter block solver as we cannot prove an approximation ratio with respect to the initial prices.

To accelerate convergence of arrival times even further, we iterate the computation of all arrival time customers for a constant number of times in each resource sharing phase. The overall algorithm is depicted in Algorithm 4.

Theorem 3.11 *Let $\beta > 0$. Given a routing oracle with approximation ratio σ and $p = \mathcal{O}(\beta^{-2} \log |\mathcal{R}|)$, lines (1)–(11) of Algorithm 4 compute a (fractional) solution that minimizes the maximum resource usage up to a factor $\sigma(1+\beta)$, assuming that this minimum is between $\frac{1}{2}$ and 2 if we use the block solver from Section 3.6.1 or 3.6.2 in (10).*

If $\sigma = 1$ and there is a global routing that satisfies all routing and timing constraints, the algorithm computes a fractional solution such that no edge is overloaded by more than a factor $1+\beta$ and the worst slack is at least $-\tau_r - \beta H$, where H is the constant from Theorem 3.8.

Proof This follows from optimality of the simple block solver from Section 3.6.1 and from Theorems 3.1, 3.7 and 3.8. □

① **for** $i = 1, \dots, p$ **do**
② **for** each net N **do**
③ **set** $X := 0$
④ **while** $X < 1$ **do**
⑤ Call routing oracle for N to obtain a solution (A, κ)
⑥ **set** $\xi := \min\left\{1 - X, \frac{1}{\max_{r \in \mathcal{R}} \mathrm{usg}_{N,r}((A,\kappa))}\right\}$
⑦ **set** $\mathrm{price}(r) := \mathrm{price}(r) \cdot e^{\gamma \cdot \xi \cdot \mathrm{usg}_{N,r}((A,\kappa))}$ for $r \in \mathcal{R}$
⑧ **for** $j = 1, \dots, n$ **do**
⑨ **for** each arrival time customer v **do**
⑩ $a(v) :=$ ComputeAT(v)
⑪ **set** $\mathrm{price}(r) := \mathrm{price}(r) \cdot e^{\gamma \cdot n^{-1} \cdot \mathrm{usg}_{v,r}(a(v))}$ for $r \in \mathcal{R}$
⑫ Iterated randomized rounding
⑬ Rip-up and re-route

Algorithm 4: Overall algorithm for timing-constrained global routing with constants, p, n, γ. Step ⑤ and ⑩ have to be specified.

3.8 Obtaining Integral Solutions

Having computed a fractional solution, the remaining task is to round the solutions of net customers to integral solutions such that the maximum resource violation does not increase too much. Müller, Radke, and Vygen [MRV11] proposed a randomized rounding approach to perform this task. Independently for all (net) customers N with solution $\sum_{i=1}^{k} x_i \mathrm{sol}_i$ where $x_i > 0$ for $i = 1, \dots, k$, $\sum_{i=1}^{k} x_i = 1$, and $\mathrm{sol}_i \in B(N)$ is an integral solution for $i = 1, \dots, k$, we select solution sol_i with probability x_i. Müller, Radke, and Vygen [MRV11] proved that the maximum relative resource violation does not increase by more than a factor of $1 + \delta$ with high probability. For a precise statement see Theorem 22 in [MRV11].

A common approach to decrease resource violations that have been introduced by randomized rounding is *rip-up and re-route*, i. e. replacing a solution of a customer by another solution. The new solution can either be chosen from the set of solutions that have been computed in the fractional phase but have not been selected, or it can be re-computed from scratch.

During *rip-up and re-route* it is necessary to update arrival times as these might have become sub-optimum at the end of Algorithm 2. Let $v \in V(D)$ be an arrival time customer. For $r \in \mathcal{R}$ let $\mathrm{price}^I(r, x) : \mathcal{R} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be the cost function used during rip-up and re-route, i. e. for a resource $r \in \mathcal{R}$, $\mathrm{price}^I(r, x)$ is the resource usage cost of r if a fraction of x is used from it. We may either still use exponential cost functions as in the resource sharing phase or we select a completely different function. The only assumptions we make are that all functions $\mathrm{price}^I(r, .)$ are strictly monotonically increasing, continuous and differentiable.

For $r \in \mathcal{R}$ let $\overline{\mathrm{usg}}(r)$ be the relative usage from r by all customers except for v. Let

$$h(x) := \sum_{(v,w) \in \delta^+(v)} \mathrm{price}^I\left((v,w), \overline{\mathrm{usg}}((v,w)) + \frac{x - a_{\min}(v)}{a_{\max}(w) - a_{\min}(v)}\right)$$
$$- \sum_{(u,v) \in \delta^-(v)} \mathrm{price}^I\left((u,v), \overline{\mathrm{usg}}((u,v)) + \frac{a_{\max}(v) - x}{a_{\max}(v) - a_{\min}(u)}\right).$$

Instance: A strictly monotonically increasing and differentiable function $h : [a_{\min}, a_{\max}] \to \mathbb{R}$ with root x^* such that $h(a_{\min}) < 0 < h(a_{\min})$, a precision $\delta > 0$.
Output: A value $x \in [a_{\min}, a_{\max}]$ such that $|x^* - x| < \delta$.

1. set $x_l := a_{\min}$, $x_u := a_{\max}$.
2. **while** $x_u - x_l > \delta$ **do**
3. set $x_1 := \frac{x_l + x_u}{2}$
4. set $x_2 := x_1 - \frac{h(x_1)}{h'(x_1)}$
5. **if** $\operatorname{sgn}(x_1) \neq \operatorname{sgn}(x_2)$ **do**
6. set $x_3 := \frac{h(x_1)x_2 - h(x_2)x_1}{x_1 - x_2}$
7. **else**
8. set $x_3 := x_2$
9. set $x_l := \max\{x \in \{x_l, x_1, x_2, x_3\} : h(x) \leq 0\}$,
10. set $x_u := \min\{x \in \{x_u, x_1, x_2, x_3\} : h(x) \geq 0\}$.
11. **return** x_l.

Algorithm 5: A combination of Newton's method and binary search.

We update the arrival time $a(v)$ to the projection of the root x^* of h into the interval $[a_{\min}(v), a_{\max}(v)]$ by Newton's method. As this method is not guaranteed to converge fast enough for general cost functions, we can combine this with a binary search as shown in Algorithm 5. The binary search guarantees that we need at most $\mathcal{O}(\log(|a_{\max}(v) - a_{\min}(v)|))$ iterations while Newton's method yields quadratic convergence as soon as we have approached x^* close enough. If root x^* lies between two solution candidates x_1 and x_2 in one iteration, we compute the intersection x_3 of the straight line through $(x_1, h(x_1))$ and $(x_2, h(x_2))$ with the y-axis.

Recently, Hähnle [Häh15] developed an alternative to the arrival time customer approach described in this chapter. Similar to Vygen [Vyg04] he modeled timing constraints within the *Min-Max Resource Sharing Problem* by adding a resource for each maximal path in the timing graph. By a modified algorithm and a new analysis, Hähnle [Häh15] showed how to obtain a polynomial running time although the number of resources is exponential. He reduced the dependency of the running time on the number n of signal paths to $\log(n)$ and achieved a running time of $\widetilde{\mathcal{O}}\left((|\mathcal{R}'| + m) \cdot \log(n)\right)$, where m is the number of edges in the timing graph and $\mathcal{R}'$ is the set of resources different from the signal path resources.

Chapter 4

Buffering-and-Routing Oracles

After having developed a general framework to incorporate timing constraints into a global routing algorithm, we concentrate on the buffering problem in this chapter. The goal of this thesis is to optimize the timing of a chip by buffering and by global routing without violating routing or placement constraints. A delay model that is largely used to estimate the timing of a buffered netlist is the Elmore Delay model [Elm48]. We prove that even the problem of finding a single Steiner path (cf. Definition 2.1) minimizing a weighted sum of Elmore delay and edge costs is *NP*-hard but can be approximated arbitrarily close to the optimum.
We generalize the FPTAS to obtain a polynomial time algorithm that finds an almost optimum buffered Steiner tree in the case that the number of sinks is constant.

Some results of this chapter are joint work with Nicolai Hähnle.

4.1 Minimum Cost Buffered Steiner Trees

With the timing-constrained global routing framework described in Chapter 3 we can directly address timing within a model that has been designed for the *Standard Global Routing Problem* originally. Instead of solving tasks from the fields *routing* and *timing* separately we can now obey constraints from both in one algorithm and output Steiner trees that are good with respect to congestion *and* delays.

In this thesis we go even one step further. Instead of just making global routing more timing-aware, we will solve the buffering problem completely within the resource sharing algorithm. In contrast to the situation of global routing where the solution of net customers were Steiner trees, we seek to find *buffered* Steiner trees (see Definition 2.8). In this sense, the block solver for net customers becomes a combination of a routing and a buffering oracle.

4.1.1 Buffer Space Resources

Besides routing congestion and timing, a buffering solution heavily affects the placement problem. Similar to routing space, placement space is a limited resource. In some areas of a chip standard logic is densely packed, leaving space for a few repeaters only. Other parts of the chip are completely reserved for larger macros and repeaters must not be inserted there at all.

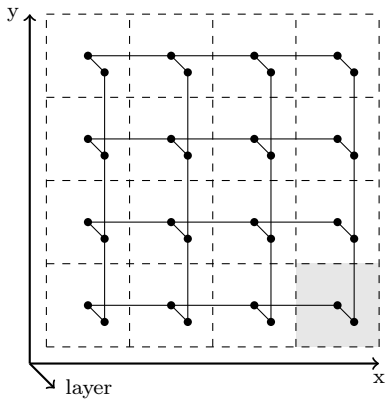


Figure 4.1: Global routing graph arising from a partition of a rectangular chip area into rectangular tiles in the case of two routing layers. The partition defines placement bins such as the gray area.

To model placement space inside the resource sharing algorithm we divide the chip area into *placement bins* $B_1 \dot\cup B_2 \dot\cup \ldots \dot\cup B_k$. For each bin B_i we add a *buffer space* resource with capacity equal to the available buffer space size(B_i) inside B_i. A function $\Psi : V(G) \to \{B_1, B_2, \ldots, B_k\}$ tells us to which placement bin a node of G belongs.

Let L be a repeater library and let $((A, \kappa), b)$ be a buffered Steiner tree for a net customer N. Let size(l) denote the size of repeater $l \in L$ and define size$(\square) := 0$. Net customer N consumes an amount of $\sum_{\substack{\nu \in V(A): \\ \Psi(\kappa(\nu)) = B_i}} \text{size}(b(\nu))$ from buffer space resource B_i.

In the global routing graph described in Section 2.4.3 there is a natural subdivision of the chip area into placement bins, namely

$$\Big\{[x_i, x_{i+1}] \times [y_j, y_{j+1}] : i \in \{0, \ldots, w-1\}, j \in \{0, \ldots, h-1\}\Big\},$$

where $0 = x_0 < x_1 < \ldots < x_w = W$, $0 = y_0 < \ldots < y_h = H$ is a subdivision of the chip area $[0, W] \times [0, H]$. The function Ψ maps a node representing a tile center $\left(\frac{x_i + x_{i+1}}{2}, \frac{y_j + y_{j+1}}{2}, z\right)$ to the bin $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$ containing it. Figure 4.1 illustrates this.

Note that for each repeater type $l \in L$ there is exactly one layer below the wiring layers on which we can place it. Usually, this the lowest possible layer but the pins of large repeaters can be on higher layers, too. It is trivial to extend this placement resource model and deal with repeaters that overlap with more than one tile. For the sake of simplicity and a simpler notation we omit this extension in this thesis.

4.2 Delay Models

In this section we describe two models that can be used to estimate the delay along a path inside a buffered Steiner tree $((A, \kappa), b)$ for a net N with source s.

For the rest of this chapter we use the notation of the case that G is a *directed* graph. All definitions and results can easily be adapted to the undirected case by replacing each undirected edge by two directed edges. Recall that for $\nu, \omega \in V(A)$ such that ω is reachable from ν we denote the unique ν-ω path in A by $A_{[\nu,\omega]}$ (Definition 2.3).

4.2.1 The Elmore Delay Model

The delay along a buffered Steiner tree $((A, \kappa), b)$ in a global routing graph G depends on various parameters. Beside its length, electrical properties of gates and wires such as capacitance and resistance have great impact on the delay along a path.

One popular model is the *Elmore* delay model [Elm48] that we define now. Let $\nu, \omega \in V(A)$ such that $\nu = s$ or $b(\nu) \in L$ and $P := A_{[\nu,\omega]}$ does not contain any internal repeaters, i. e. $b(\nu') = \square$ for $\nu' \in V(P)\backslash\{\nu, \omega\}$. Furthermore, we assume that we are given

- electrical capacitances $\text{cap}(\nu')$ for all $\nu' \in V(P)\backslash\{\nu\}$, and
- resistance $\text{res}(\nu)$ and total output capacitance $\text{outcap}(\nu)$ at ν.

By replacing each edge in the global routing graph by a copy for each wire code, we can achieve that each edge $e \in E(G)$ has pre-defined wire capacitance $\text{cap}(e)$ and resistance $\text{res}(e)$ that only depends on the edge itself. Recall that it is possible that adjacent vertices ν, ω are placed at the same position and $\kappa((\nu, \omega)) = \circ$. We set $\text{res}(\circ) = \text{cap}(\circ) = 0$.

The Elmore delay of P is defined as

$$\text{Elmore}(P) := \text{res}(\nu) \cdot \text{outcap}(\nu) + \sum_{\zeta=(\nu,\omega)\in E(P)} \text{res}(\kappa(\zeta)) \cdot \left(\frac{\text{cap}(\kappa(\zeta))}{2} + \text{cap}(\omega) \right). \quad (4.1)$$

In early design stages, the value $\ln(2) \cdot \text{Elmore}(P)$ is a reasonable estimate on the delay along P (see [Vyg16]). In this definition, the delays are independent of slews. To improve the accuracy of the Elmore delay model, modern timing engines incorporate slew effects. We call the resulting extension to the Elmore delay model the *Elmore Delay Model with Slew Propagation* (see Section 7.1.3). In later design stages, even more accurate delay models such as *rapid interconnection circuit evaluation* (RICE) [RP94] and *current-based delay models* [CW03] are preferred. Despite the higher accuracy, those delay models are harder to compute and lack explicit formulas which makes it more difficult to use them in optimization.

In the context of timing-constrained global routing we assume that resistances and capacitances of the edges in the global routing graph are given. The same holds for capacitances of input pins of logic gates, macros, latches, and of primary inputs as well as resistances of output pins.

This information allows us to define $\text{res}(s)$, $\text{cap}(t)$ of a sink pin $t \in N\backslash\{s\}$, and $\text{res}(\nu)$ and $\text{cap}(\nu)$ of a Steiner node ν with $b(\nu) \in L$.

For a Steiner point ν with $b(\nu) = \square$ we define recursively

$$\text{cap}(\nu) = \sum_{\zeta=(\nu,\omega)\in\delta_A^+(\nu)} (\text{cap}(\kappa(\zeta)) + \text{cap}(\omega)).$$

Similarly, we define $\text{outcap}(\nu)$ for all $\nu \in V(A)$ such that $\nu = s$ or $b(\nu) \in L$ by the same formula

$$\text{outcap}(\nu) = \sum_{\zeta=(\nu,\omega)\in\delta_A^+(\nu)} (\text{cap}(\kappa(\zeta)) + \text{cap}(\omega)).$$

The reason why we use a different notation to denote the total capacitance $\text{outcap}(\nu)$ that a vertex ν has to drive is that Steiner nodes associated with repeaters can appear as starting *and* as end points of paths.

Formula (4.1) enables us to compute the Elmore-value of any maximal subpath of $((A,\kappa),b)$ without internal repeaters. Let $t \in N\backslash\{s\}$ and let $\nu_1^{\text{rpt}},\dots,\nu_k^{\text{rpt}}$ be the nodes in $V(A_{[s,t]})$ with $b(.) \in L$ (in the order in which they appear in $A_{[s,t]}$). If $k \geq 1$, we define

$$\text{Elmore}(A_{[s,t]}) := \text{Elmore}\big(A_{[s,\nu_1^{\text{rpt}}]}\big) + \left(\sum_{i=1}^{k-1} \text{Elmore}\big(A_{[\nu_i^{\text{rpt}},\nu_{i+1}^{\text{rpt}}]}\big)\right) + \text{Elmore}\big(A_{[\nu_k^{\text{rpt}},t]}\big).$$

4.2.2 Generalized Non-Linear Delay Model

To demonstrate that the results of this chapter do not depend on the explicit formula for Elmore delay and to simplify notation we present a slightly more general delay model that we describe now.

Assume we are given the following functions:

- $\mu : E(G) \cup L \cup (N\backslash\{s\}) \to \mathbb{R}_{\geq 0}$,
- $F : (E(G) \cup L \cup \{s\}) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that $F(x,.) : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ is monotonically increasing for each $x \in E(G) \cup L \cup \{s\}$.

The function F determines the delay along an edge, a repeater, or the source and depends on a capacitance determined by the μ function. We set $\mu(\circ) := 0$ and define $F(\circ,.)$ to be the constant zero function. Similar to the Elmore delay case we compute $\text{cap}(\nu)$ for each vertex $\nu \in V(A)$ and $\text{outcap}(\nu)$ for each vertex $\nu \in V(A)$ such that $\nu = s$ or $b(\nu) \in L$ recursively by the formulas

$$\text{cap}(\nu) := \begin{cases} \mu(\nu) & \text{if } \nu \in N\backslash\{s\} \\ \mu(b(\nu)) & \text{if } b(\nu) \in L \\ \displaystyle\sum_{\zeta=(\nu,\omega)\in\delta_A^+(\nu)} \big(\text{cap}(\omega) + \mu(\kappa(\zeta))\big) & \text{otherwise,} \end{cases}$$

$$\text{outcap}(\nu) := \sum_{\zeta=(\nu,\omega)\in\delta_A^+(\nu)} \big(\text{cap}(\omega) + \mu(\kappa(\zeta))\big).$$

For $t \in N\backslash\{s\}$ we define the delay through the unique $s-t$ path in A as

$$\begin{aligned}\text{delay}_{((A,\kappa),b)}^{\mu,F}(t) := {} & F(s,\text{outcap}(s)) \\ & + \sum_{\zeta=(\nu,\omega)\in E(A_{[s,t]})} F(\kappa(\zeta),\text{cap}(\omega)) + \sum_{\substack{\nu\in V(A_{[s,t]}) \text{ s.t.} \\ b(\nu)\in L}} F(b(\nu),\text{outcap}(\nu)).\end{aligned}$$

The Elmore delay model is the special case of this model with

$$\begin{aligned} \mu(e) &= \text{cap}(e) && \text{for } e \in E(G), \\ \mu(t) &= \text{cap}(t) && \text{for } t \in N\backslash\{s\}, \\ \mu(l) &= \text{cap}(l) && \text{for } l \in L, \\ F(s,x) &= \text{res}(s)\cdot x, \\ F(l,x) &= \text{res}(l)\cdot x && \text{for } l \in L, \text{ and} \\ F(e,x) &= \text{res}(e)\cdot\left(\frac{\text{cap}(e)}{2} + x\right) && \text{for } e \in E(G). \end{aligned}$$

4.3 The Minimum Cost Buffered Steiner Tree Problem

Now we have collected all ingredients to state the problem of the net customer's oracle.

Minimum Cost Buffered Steiner Tree Problem

Instance: A repeater library L associated with functions

$$\text{size} : L \cup \{\square\} \to \mathbb{R}_{\geq 0} \text{ and } \text{power} : L \cup \{\square\} \to \mathbb{R}_{\geq 0}$$

such that $\text{size}(\square) = \text{power}(\square) = 0$.
A graph G with edge and placement costs

$$c : (E(G) \cup \{\circ\}) \cup V(G) \to \mathbb{R}_{\geq 0} \text{ with } c(\circ) = 0.$$

A net $N \subseteq V(G)$ with source s and sink delay costs

$$\lambda : N \backslash \{s\} \to \mathbb{R}_{\geq 0}.$$

A static power cost $c_{\text{power}} \in \mathbb{R}_{\geq 0}$.
Timing functions

$$\mu : E(G) \cup L \cup (N \backslash \{s\}) \to \mathbb{R}_{\geq 0}$$

and

$$F : (E(G) \cup L \cup \{s\}) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$$

such that $F(x, .)$ is monotonically increasing for each $x \in E(G) \cup L \cup \{s\}$.

Output: A buffered Steiner tree $((A, \kappa), b)$ for N in G minimizing

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \text{delay}^{\mu,F}_{((A,\kappa),b)}(t) +$$
$$\sum_{\nu \in V(A)} c(\kappa(\nu)) \cdot \text{size}(b(\nu)) + c_{\text{power}} \cdot \sum_{\nu \in V(A)} \text{power}(b(\nu)).$$

The cost $c(e)$ for an edge $e \in E(G)$ of the global routing graph contains all costs for using e such as resource prices of congestion resources and a net length resource. Costs $c(v)$ for $v \in V(G)$ are resource prices for buffer space resources while $\lambda(t)$ is the sum of prices over all timing resources entering a sink $t \in N \backslash \{s\}$. We also want to minimize the static power consumption of the newly inserted repeaters and add a static power resource into the resource sharing framework. Its price is c_{power}.

Optimizing dynamic power consumption can be done analogously to optimizing the delay through a repeater and through the source and requires insertion of an additional resource for dynamic power (cf. Section 2.5.6).

In this chapter we do not distinguish buffers and inverters and do not consider electrical constraints like slew- and capacitance constraints. In Section 4.7 we return to the question how to incorporate polarity and capacitance constraints. Obeying slew constraints and using delay models in which delays are influenced by slews is harder.

4.3.1 Hardness Results

It is easy to see that the *Minimum Cost Buffered Steiner Tree Problem* *NP*-hard. In fact, this is true even for special cases of it.
If $c_{\text{power}} = c(v) = \lambda(t) = 0$ for all $v \in V(G)$ and $t \in N\backslash\{s\}$, we obtain the classical *Minimum Cost Steiner Tree Problem* that has been proved to be *NP*-hard by Garey and Johnson [GJ77], [GJ79] page 209f.

Chuzhoy et al. [Chu+05] proved that for the following problem no $o(\log\log|N|)$ approximation algorithm exists unless every problem in *NP* can be solved in $\mathcal{O}(n^{\log\log\log n})$ time where n is the instance size:

Instance: An undirected graph G with functions $c, \rho : E(G) \cup \{\circ\} \to \mathbb{R}_{\geq 0}$ with $c(\circ) = \rho(\circ) = 0$.
A net N with source s and sink delay costs $\lambda : N\backslash\{s\} \to \mathbb{R}_{\geq 0}$.
Output: A Steiner tree (A, κ) for N in G minimizing

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N\backslash\{s\}} \lambda(t) \cdot \sum_{\zeta \in E(A_{[s,t]})} \rho(\kappa(\zeta)).$$

This is the special case of the *Minimum Cost Buffered Steiner Tree Problem* for which $L = \emptyset$, $F(s, .)$ is the constant zero function, for $e \in E(G)$, $F(e, .)$ is the constant function with value $\rho(e)$, and where G arises from an undirected graph by directing each edge in both directions. In Chapter 6 we will study that problem in greater detail. There, we refer to that problem as the *Minimum Cost Steiner Tree Problem with Linear Delays*.

Even if delays are measured by the Elmore delay model, the result of Chuzhoy et al. [Chu+05] shows that the task of a net customer's block solver is hard to approximate. Let $L = \{l\}$ such that l is allowed to be added at each node of G with price 0 (i. e. $c(v) = 0$ for all $v \in V(G)$). For $e \in E(G)$ let $\text{res}(e) := \rho(e)$ and $\text{cap}(e) := 0$. We set $c_{\text{power}} = \text{res}(s) = \text{res}(l) := 0$ and $\text{cap}(t) = \text{cap}(l) = 1$ for $t \in N\backslash\{s\}$.

If (A, κ) is a Steiner tree for N in G such that A is rooted at s, we define $b : V(G) \to L \cup \{\square\}$ by

$$b(v) = \begin{cases} \square & \text{if } v \in N \\ l & \text{otherwise.} \end{cases}$$

Then,

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N\backslash\{s\}} \lambda(t) \cdot \sum_{\zeta \in E(A_{[s,t]})} \rho(\kappa(\zeta)) = \sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N\backslash\{s\}} \lambda(t) \cdot \text{Elmore}(s, t).$$

On the other hand, if $((A, \kappa), b)$ is a buffered Steiner tree, we can assume that $b(v) = l$ for all $v \in V(A)\backslash N$ (inserting a repeater at each Steiner point does not increase capacitances and hence delays, and does not introduce any costs).

The problem of computing a shortest rectilinear Steiner tree in which all source-sink paths are shortest possible is known as the *Rectilinear Steiner Arborescence Problem*. Shi and Su [SS05] proved *NP*-hardness of that problem.

In Section 4.5.1 we show that the special case of the *Minimum Cost Buffered Steiner Tree Problem* with $|N| = 2$ is *NP*-hard as well. This is true even if delays are measured by the Elmore delay model.

4.3.2 Existing Algorithms for Special Cases

Due to its complexity, we will not be able to solve the general version of the *Minimum Cost Buffered Steiner Tree Problem*. Instead, we concentrate on special cases.

For the case that $c(e) = 0$ for $e \in E(G)$, $L = \emptyset$, and delays are measured by the Elmore delay model with edge resistances and capacitances of the form

$$\text{res}(e) = \text{const}_{\text{res}} \cdot \text{length}(e), \qquad \text{cap}(e) = \text{const}_{\text{cap}} \cdot \text{length}(e)$$

for all edges in $e \in E(G)$ (where length : $E(G) \to \mathbb{R}_{>0}$ is some function), Scheifele [Sch14] gave the first constant factor approximation algorithm. For general graph instances he obtains an approximation ratio of 4.11. If G is a 2-dimensional grid graph and the length of an edge is the ℓ_1-distance between its adjacent vertices, he achieves an approximation guarantee of 3.39.

For the *Rectilinear Steiner Arborescence Problem* a 2-approximation algorithm (Rao et al. [Rao+92], Córdova and Lee [CL94]) and a polynomial time approximation scheme (Lu and Ruan [LR00]) is known.

For the case that the global routing graph is a 2-dimensional grid graph there have been several heuristic approaches for simultaneous Steiner tree construction and buffering. Okamoto and Cong [OC96] combined the computation of a rectilinear Steiner arborescence with dynamic programming based buffering. For Steiner arborescence computation they used the A-Tree algorithm by Cong et al. [CLZ93]. The first algorithm for buffering a given Steiner tree by dynamic programming is due to Van Ginneken [Van90]. An overview of dynamic programming based algorithms for the buffering problem can be found in Section 7.2. Similar approaches as in [OC96] have been used by Hrkić and Lillis [HL02], and Hu et al. [Hu+03].

4.4 Cost-Delay Minimum Steiner Tree Problem with Loops

In this section we slightly change the problem formulation of the *Minimum Cost Buffered Steiner Tree Problem*. This allows a simplification of notation, a reduction of variables, and usage of a slightly more general delay model.

4.4.1 Shortening the Model: Eliminating Pin Properties

First, we may assume that the capacitance $\mu(t)$ is zero for each sink pin $t \in N \backslash \{s\}$. This can be achieved by adding a new vertex t' to G connected with t by an edge $e = (t, t')$. Edge e will receive a μ-value equal to the original μ-value of t and cost $c(e) = 0$. The function $F(e, .)$ can be set to be the constant zero function. Vertex t' replaces t in N.

By a similar technique we can assume that the source delay function $F(s, .)$ is constantly zero. As before, we add a new node s' as well as an edge $e = (s', s)$ to G and replace s by s' in N. We set $F(e, .)$ to the original function $F(s, .)$ and $c(e) = \mu(e) = 0$.

4.4.2 Shortening the Model: Representing Repeaters by Loops

Now we show how to avoid dealing with repeaters directly. Instead, we model the effect of inserting a repeater by an edge traversal as follows. For each repeater $l \in L$ and each

$v \in V(G)$ such that inserting repeater l at position v is allowed, we add a loop edge $e_l = (v, v)$ to G. Traversal of e_l models adding a repeater of type l at position v. We set $c(e_l) := c(v) \cdot \text{size}(l) + c_{\text{power}} \cdot \text{power}(l)$ and $F(e_l, .) := F(l, .)$. To model the capacitance change after repeater insertion we replace the former function μ by a function

$$\Delta : E(G) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$$

such that $\Delta(e_l, .)$ is the constant function with value $\mu(l)$. For edges $e \in E(G)$ that do not model repeater insertion we define $\Delta(e, .)$ to be the function $x \mapsto x + \mu(e)$ while $\Delta(\circ, .)$ is the identity function $\mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$.

The capacitance $\text{cap}(\nu)$ of a vertex $\nu \in V(A)$ is then recursively defined as $\text{cap}(\nu) = 0$ for $\nu \in N \backslash \{s\}$ and

$$\text{cap}(\nu) = \sum_{\zeta=(\nu,\omega)\in\delta^+(\nu)} \Delta(\kappa(\zeta), \text{cap}(\omega))$$

otherwise.

4.4.3 Problem Formulation

The following problem definition summarizes our new model and the resulting optimization problem. The *Minimum Cost Buffered Steiner Tree Problem* is a special case of this new problem.

Cost-Delay Minimum Steiner Tree Problem with Loops

Instance: A digraph G with edge costs $c : E(G) \to \mathbb{R}_{\geq 0}$, possibly containing loops.
A net $N \subseteq V(G)$ with source s and sink delay costs $\lambda : N \backslash \{s\} \to \mathbb{R}_{\geq 0}$.
A function $F : E(G) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that $F(e, .)$ is monotonically non-decreasing for each $e \in E(G)$.
A function $\Delta : E(G) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that $\Delta(e, .)$

- is a constant function if e is a loop edge and
- is monotonically non-decreasing with $\Delta(e, x) \geq x$ for all $x \in \mathbb{R}_{\geq 0}$ otherwise.

We define $c(\circ) = 0$, $F(\circ, .)$ to be the constant zero function, and $\Delta(\circ, .)$ to be the identity function

Output: A Steiner tree (A, κ) for N in G minimizing

$$\sum_{\zeta\in E(A)} c(\kappa(\zeta)) + \sum_{t\in N\backslash\{s\}} \lambda(t) \cdot \left(\sum_{\zeta=(\nu,\omega)\in E(A_{[s,t]})} F(\kappa(\zeta), \text{cap}(\omega)) \right),$$

where for $\omega \in V(A)$, $\text{cap}(\omega)$ is recursively defined as

$$\text{cap}(\omega) = \sum_{\zeta=(\omega,\omega')\in\delta_A^+(\omega)} \Delta(\kappa(\zeta), \text{cap}(\omega')).$$

Note that the recursive definition of cap already implies that $\text{cap}(t) = 0$ for $t \in N \backslash \{s\}$. We call the special case of the *Cost-Delay Minimum Steiner Tree Problem with Loops* where $N = \{s, t\}$ the *Cost-Delay Minimum Steiner Path Problem with Loops*.

4.4.4 Necessity of Conservative Edge Costs

Recall that for the *Standard Minimum Steiner Tree Problem* and for the *Shortest Path Problem* it is essential to restrict the edge costs such that there are no negative cycles. An edge cost function for which no cycle with negative total cost exists is called *conservative edge cost*. Finding a shortest path in a graph with non-conservative edge costs is *NP*-hard as a straightforward reduction from the *NP*-hard *Longest Path Problem* shows ([GJ79] page 213).

In our case, the restriction that $\Delta(e, x) \geq x$ for all non-loop edges $e \in E(G)$ and all $x \in \mathbb{R}_{\geq 0}$ is necessary to ensure that traversal of cycles does not decrease the overall cost of a Steiner tree. Note that this condition is fulfilled for the Elmore delay model.

It is easy to see that allowing e. g. a function $\Delta(e, .) = (x \mapsto \alpha \cdot x)$ for a constant $\alpha < 1$ may lead to undesired situations: If we do not enforce that the κ function of a Steiner tree (A, κ) is injective, no finite optimum solution might exist. If we require injectivity, we obtain an optimization problem for which no approximation algorithm exists unless $P = NP$. The last fact is even true for the *Cost-Delay Minimum Steiner Path Problem with Loops* and can be proved by a straightforward reduction from the *Hamiltonian s-t Path Problem*:

Lemma 4.1 *Let p be a polynomial. Unless $P = NP$ there is no $2^{p(|V(G)|)}$-approximation algorithm for the modification of the Cost-Delay Minimum Steiner Path Problem with Loops in which we omit the condition $\Delta(e, x) \geq x$ for non-loop edges $e \in E(G)$ and require that the returned buffered Steiner tree $((A, \kappa), b)$ has the property $|\kappa^{-1}(e)| \leq 1$ for all $e \in E(G)$.*

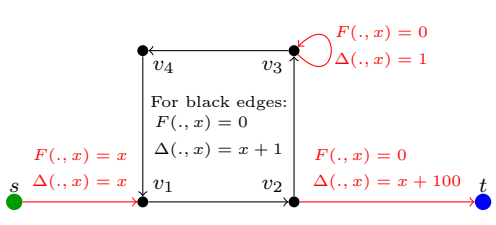
Proof Let $N = \{s, t\}$ and let G be a directed graph with $n := |V(G)|$ vertices. We construct a graph G' from G by adding vertices s', t' and edges (s', s), (t, t') with $F((s', s), x) = x$, $F((t, t'), x) = 0$, $\Delta((t, t'), x) = x + 1$ for all $x \in \mathbb{R}_{\geq 0}$. We set $\Delta(e, x) = \frac{x}{2^{p(n+2)+1}}$ and $F(e, x) = 0$ for all edges $e \in E(G)$. Edge costs $c(e)$ are zero for all edges $e \in E(G')$. Then, the total cost of an s'-t' path containing all vertices in G is more than a factor $2^{p(n+2)}$ smaller than the cost of any shorter s'-t' path. Hence, no $2^{p(|V(G')|)}$-approximation algorithm can exist unless $P = NP$. □

While for the *Shortest Steiner Tree Problems in Graphs* with non-negative (and hence conservative) edge costs, loops do not play any role, they are essential in our case as they model repeater insertion. With introduction of loop edges we have relaxed the condition $\Delta(e, x) \geq x$ that we require for non-loop edges $e \in E(G)$ and $x \in \mathbb{R}_{\geq 0}$. Consequently, optimum solutions can contain non-trivial cycles. Such a situation is depicted in Figure 4.2.

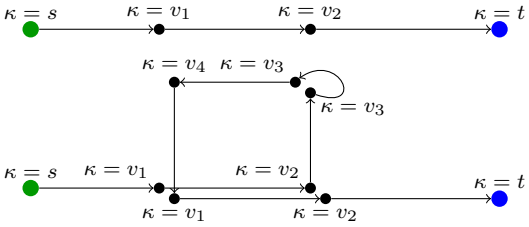
Despite these situations we can still bound the length of an optimum solution as the next theorem states. Having such a bound is necessary to obtain a polynomial time algorithm for (special cases of) the *Cost-Delay Minimum Steiner Tree Problem with Loops*.

Lemma 4.2 *Let $G, N, \lambda, F, c, \Delta$ be an instance of the Cost-Delay Minimum Steiner Tree Problem with Loops such that $N = \{s, t\}$. Let h be the number of loop edges in G. There exists an optimum solution $((A, \kappa), b)$ with $|V(A)| \leq (1 + h) \cdot |V(G)|$.*

Proof Without loss of generality we may assume that $\lambda(t) = 1$. Let $((A, \kappa), b)$ be an optimum solution with $|V(A)|$ minimum. Let $v \in \kappa(A)$ such that $\kappa^{-1}(v) = \{x_1, x_2, \ldots, x_k\}$ and x_i is reachable from x_j in A if and only if $i \geq j$ (i. e. the x_i appear in that order in path A).



(a) Graph G with functions Δ and F. Here, $N=\{s,t\}$, $c(e)=0$ for $e\in E(G)$ and $\lambda(t)=1$.



(b) Two Steiner trees. The above tree has cost 101, the tree below has cost 3 and uses a loop edge and a non-trivial cycle.

Figure 4.2: Instance of the *Cost-Delay Minimum Steiner Tree Problem with Loops* for which the optimum solution uses a non-trivial cycle in G.

If there is $i\in\{1,\dots,k-1\}$ such that $\mathrm{cap}(x_i)\geq\mathrm{cap}(x_{i+1})$, removing $A_{[x_i,x_{i+1}]}-x_i$ from A but connecting x_i with the successor of x_{i+1} in A (respectively identifying x_i with t if $x_{i+1}=t$) would not increase the cost of A but would decrease $|V(A)|$. We conclude that $A_{[x_i,x_{i+1}]}$ contains a loop edge for each $i\in\{1,\dots,k-1\}$.

Now, note that each loop edge e is used by A at most once as otherwise, removing the part between the first and the last traversal of e (including the first traversal but excluding the last one) decreases $|V(A)|$ without increasing the cost.

This implies that $k\leq h+1$ and the lemma follows. □

The bound of Lemma 4.2 is tight up to a constant factor:

Lemma 4.3 *For each $n,h\in\mathbb{N}$ there is an instance G,N,λ,F,c,Δ of the Cost-Delay Minimum Steiner Tree Problem with Loops with $|N|=2$, $|V(G)|=2+h+n$, and h loop edges in which each optimum solution (A,κ) has at least $|V(A)|=(h+1)(n+2)$ vertices.*

Proof Let G be the directed graph with vertices

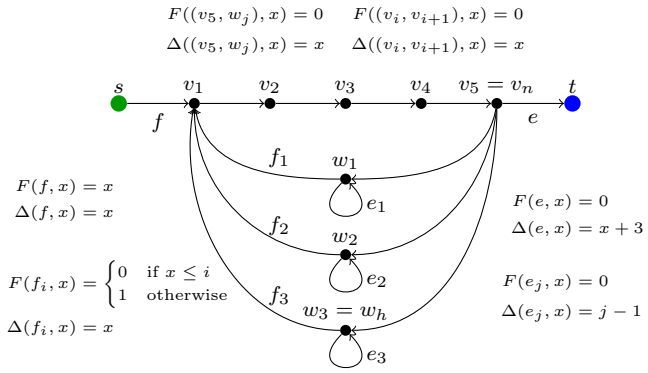
$$V(G)=\{s,t,v_1,\dots,v_n,w_1,\dots,w_h\}.$$

We add edges (s,v_1), (v_n,t), and (v_i,v_{i+1}) for $i=1,\dots,n-1$. For each $j\in\{1,\dots,h\}$ we insert edges (v_n,w_j), (w_j,v_1), and attach a loop edge e_j at w_j.

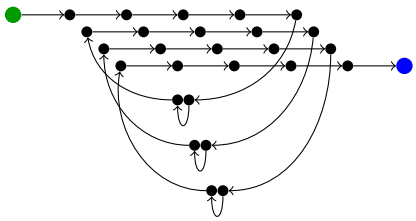
We define $N=\{s,t\}$ with $\lambda(t)=1$, $c(e)=0$ for all $e\in E(G)$, and for $x\in\mathbb{R}_{\geq 0}$

$$\begin{aligned}
F((s,v_1),x)&=x,\\
F((w_j,v_1),x)&=\begin{cases}0 & \text{if } x\leq j\\ 1 & \text{otherwise,}\end{cases} && \text{for all } j=1,\dots,h\\
F(e,x)&=0 && \text{for all edges } e\in E(G)\backslash\delta^-(v_1),\\
\Delta((v_n,t),x)&=x+h,\\
\Delta(e_j,x)&=j-1 && \text{for all } j=1,\dots,h,\\
\Delta(e,x)&=x && \text{for all non-loop edges } e\in E(G)\backslash\{(v_n,t)\}.
\end{aligned}$$

See Figure 4.3(a) for an illustration of G in the case $n=5$, $h=3$.



(a) Graph G and timing functions.



(b) A Steiner path with cost 0 and minimum number of vertices.

Figure 4.3: Instance for the *Cost-Delay Minimum Steiner Tree Problem with Loops* constructed in the proof of Lemma 4.3 with $n = 5$, $h = 3$. The number of edges in each optimum s-t Steiner path is large.

First observe that the Steiner path traversing edges

$$\begin{aligned}(s, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, w_1), e_1, (w_1, v_1),\\ (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, w_2), e_2, (w_2, v_1),\\ \ldots\\ (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, w_h), e_h, (w_h, v_1),\\ (v_1, v_2), \ldots, (v_{n-1}, v_n), (v_n, t)\end{aligned}$$

in that order has cost 0 (see Figure 4.3(b)).

To complete the proof of the lemma it remains to show that this path has the least number of vertices among all Steiner paths with cost 0. Let A be such an s-t Steiner path with cost 0. By definition of $F((s, v_1), .)$ and the Δ-functions, A must traverse e_1 which requires to traverse (w_1, v_1) afterwards.

Using the same argument we observe that each Steiner path with cost 0 that traverses edge (w_j, v_1) for $j \in \{1, \ldots, h-1\}$ must also traverse e_{j+1} and thus (w_{j+1}, v_1). Consequently, A traverses all edges entering v_1.

As each non-trivial cycle in G containing v_1 and a loop edge has $n+2$ vertices and since the part of A between the first and last occurrence (say including the first but excluding the last occurrence) must contain at least $n + 2$ vertices, we conclude $|V(A)| \geq (h + 1)(n + 2)$.

□

In instances arising in practice it is often the case that either no repeater or all repeaters can be inserted at a node. In this case, the bound of Lemma 4.2 improves to $(|L|+1)\cdot|V(G)|$ where L is the repeater library. This statement has essentially the same proof as Lemma 4.2 and tightness can easily be proved by attaching $|L|$ loop edges at each inner node of an s-t path of length $n + 2$. This graph defines G and each of the $|L|$ loop edges at each inner node represents a repeater of a certain type. By defining cost functions similar to the ones used in the proof of Lemma 4.3 we can make sure that an optimum Steiner path actually uses each of these loops.

4.5 Cost-Delay Minimum Steiner Path Problems

In this section we study the *Cost-Delay Minimum Steiner Path Problem with Loops*. Recall from Section 4.4.3 that this problem is the special case of the *Cost-Delay Minimum Steiner Tree Problem with Loops* where $N = \{s, t\}$.

If G does not even contain loops, we refer to the problem as the *Cost-Delay Minimum Steiner Path Problem*. Without loss of generality we assume $\lambda(t) = 1$ in this section.

Generally speaking, we are looking for a shortest Steiner path between two vertices s and t in this section (recall the definition of a Steiner path: Definition 2.1). The classical shortest path problem in which the cost of traversing an edge $e \in E(G)$ is $c(e) \geq 0$ is one of the best understood problems in combinatorial optimization. Well-known algorithms like Dijkstra's algorithm [Dij59] or the algorithms by Moore [Moo59], Bellman [Bel58], and Ford [For56] give optimum solutions in fast running time.

As the cost of traversing an edge is dependent on a capacitance in our case, the problem becomes much harder and the classical algorithms cannot be applied anymore.

In Section 4.5.1 we show that the *Cost-Delay Minimum Steiner Path Problem with Loops* is *NP*-hard even in very restricted special cases. Despite its hardness we can still give a fully polynomial time approximation scheme in Section 4.5.2. In Section 4.5.3 we show how to substantially improve the running time of that FPTAS in the situation of the *Cost-Delay Minimum Steiner Path Problem* (without loops).

4.5.1 *NP*-Hardness

Now we prove that the *Cost-Delay Minimum Steiner Path Problem* (and hence the *Cost-Delay Minimum Steiner Path Problem with Loops*) is *NP*-hard even if delays are measured by the Elmore delay model.

The results of this section are joint work with Nicolai Hähnle.

Our reductions use *NP*-completeness of the famous *Partition Problem* (see the book of Garey and Johnson, page 47 [GJ79]).

Lemma 4.4 ([GJ79]) *The following problem is NP-complete:*

Instance: *Rationals* $x_1, \ldots, x_n \in \mathbb{Q}_{>0}$ *with* $\sum_{i=1}^{n} x_i = 2$.
Question: *Is there a subset* $I \subseteq \{1, \ldots, n\}$ *such that* $\sum_{i \in I} x_i = 1$*?*

Theorem 4.5 *The Cost-Delay Minimum Steiner Path Problem is NP-hard if for each edge* $e \in E(G)$

- $\Delta(e, .)$ *is of the form* $x \mapsto x + \mu(e)$ *with* $\mu(e) \geq 0$ *and*
- $F(e, x)$ *is equal to* $F(e, x) := f(\Delta(e, x)) - f(x)$ *for a monotonically increasing, strictly convex, and twice differentiable function* $f : \mathbb{R}_{\geq 0} \to \mathbb{R}$.

This is true even if

- G *arises from an s-t path by replacing each edge by two parallel edges,*
- $c(e) \cdot \mu(e) = 0$ *for all* $e \in E(G)$*, and*
- $c(e) \cdot c(\overline{e}) = \mu(e) \cdot \mu(\overline{e}) = 0$ *for all pairs* e,$\overline{e}$ *of parallel edges.*

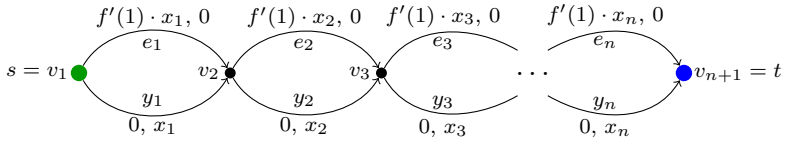


Figure 4.4: Graph G in the proof of Theorem 4.5. Each edge e is labeled $c(e)$, $\mu(e)$.

Proof Let f and F be as described in the theorem. First, observe that for any Steiner path (P, κ) in G,

$$\sum_{\zeta=(\nu,\omega)\in E(P)} F(\kappa(\zeta), \mathrm{cap}(\omega)) = f\left(\sum_{\zeta\in E(P)} \mu(\kappa(\zeta))\right) - f(0).$$

Let $x_1, \ldots, x_n$ be an instance of *Partition*. We construct an instance of the *Cost-Delay Minimum Steiner Path Problem* by

$$G = \Big(\{v_1, \ldots, v_{n+1}\}, \{e_i = (v_i, v_{i+1}), y_i = (v_i, v_{i+1}) : i = 1, \ldots, n\}\Big),$$

$c(e_i) = f'(1) \cdot x_i$, $c(y_i) = 0$, $\mu(e_i) = 0$, $\mu(y_i) = x_i$ for $i = 1, \ldots, n$, $v_1 = s$, $v_{n+1} = t$.
Figure 4.4 depicts G. For any Steiner path (P, κ) let

$$I(P, \kappa) := \Big\{i \in \{1, \ldots, n\} : P \text{ contains an edge } \zeta \text{ with } \kappa(\zeta) = y_i\Big\}.$$

Furthermore, define $g(x) := f'(1) \cdot (2 - x) + f(x) - f(0)$. Note that (P, κ) has cost

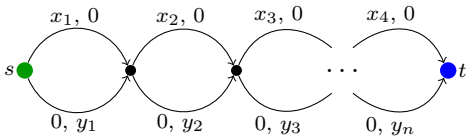
$$\begin{aligned}\sum_{\zeta\in E(P)} c(\kappa(\zeta)) + \sum_{\substack{\zeta=(\nu,\omega)\\ \in E(P)}} F(\kappa(\zeta), \mathrm{cap}(\omega)) &= f'(1) \cdot \sum_{i\notin I(P,\kappa)} x_i + f\left(\sum_{i\in I(P,\kappa)} x_i\right) - f(0)\\ &= f'(1) \cdot \left(2 - \sum_{i\in I(P,\kappa)} x_i\right) + f\left(\sum_{i\in I(P,\kappa)} x_i\right) - f(0)\\ &= g\left(\sum_{i\in I(P,\kappa)} x_i\right).\end{aligned}$$

We claim that $x_1, \ldots, x_n$ is a *yes*-instance for *Partition* if and only if there is a Steiner path with cost at most $g(1)$.

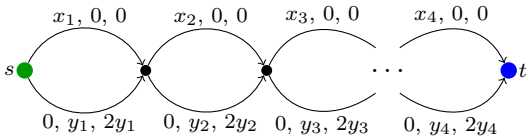
$\Rightarrow$: Let $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i\in I} x_i = \sum_{j\in\{1,\ldots,n\}\setminus I} x_j = 1$.
Let $P = \Big(\{\omega_1, \ldots, \omega_{n+1}\}, \{\zeta_1, \ldots, \zeta_n\}\Big)$ with

$$\kappa(\omega_i) = v_i \text{ for } i = 1, \ldots, n+1 \text{ and } \kappa(\zeta_i) = \begin{cases} e_i & \text{if } i \in I \\ y_i & \text{otherwise} \end{cases} \text{ for } i = 1, \ldots, n,$$

i. e. we take the "upper" edge (according to Figure 4.4) for indices in I and the lower edge otherwise. By construction, $I(P, \kappa) = \{1, \ldots, n\}\setminus I$ and thus, (P, κ) has cost $g(1)$.



(a) Instance $G, \overline{c}, \overline{\mu}$ of the problem of Theorem 4.5. Each edge e is labeled $\overline{c}(e)$, $\overline{\mu}(e)$



(b) Arising instance G, c, w, d of the *Cost-Delay Minimum Path Problem*. Each edge e is labeled $c(e)$, $w(e)$, $\mu(e)$

Figure 4.5: Graphs used in the proof of Theorem 4.6.

$\Leftarrow$: Let (P, κ) be a Steiner path with cost at most $g(1)$. Since f is strictly convex, g is strictly convex as well. As $g'(1) = -f'(1) + f'(1) = 0$ and $g'' \geq 0$ by convexity, 1 is a local, and hence global minimum of g.
By *strict* convexity, this minimum is unique which implies that $\sum_{i \in I(P,\kappa)} x_i = 1$.

□

The strict convexity assumption is indeed necessary. If f is a linear function and Δ and F are as described in Theorem 4.5, the resulting problem is equal to the *Standard Shortest Path Problem* which is of course not *NP*-hard. Analogously to Theorem 4.5 one can show that the corresponding *maximization* problem is *NP*-hard if f is strictly *concave*.

We can now show that finding a Steiner path minimizing the sum of linear edge costs and Elmore delay is *NP*-hard.

Theorem 4.6 *The Cost-Delay Minimum Steiner Path Problem is NP-hard if for each edge $e \in E(G)$*

- *$\Delta(e, .)$ is of the form $x \mapsto x + \mu(e)$ with $\mu(e) \geq 0$ and*
- *$F(e, x)$ is equal to $F(e, x) = r(e) \cdot \left(\frac{\mu(e)}{2} + x\right)$ with $r(e) \geq 0$.*

This is true even if

- *G arises from an s-t path by replacing each edge by two parallel edges,*
- *$c(e) \cdot \mu(e) = 0$ for all $e \in E(G)$, and*
- *$c(e) \cdot c(\overline{e}) = \mu(e) \cdot \mu(\overline{e}) = r(e) \cdot r(\overline{e}) = 0$ for all pairs e, $\overline{e}$ of parallel edges.*

Proof Let $G, \overline{c}, \overline{\mu}$ be an instance of the special case of the *Cost-Delay Minimum Steiner Path Problem* proved to be *NP*-hard in Theorem 4.5 where $f(x) = x^2$. We obtain an instance of the variant introduced in Theorem 4.6 by defining $c(e) = \overline{c}(e), r(e) = \overline{\mu}(e), \mu(e) = 2\overline{\mu}(e)$ (see Figure 4.5).

Let (P, κ) be an s-t- Steiner path in G, $E' = \{\zeta \in E(P) : c(\kappa(\zeta)) = 0\}$. Let $\zeta_1, \ldots, \zeta_k$ be the ordering of the edges in E' as they appear in P.

The cost of (P, κ) w. r. t. the cost function of the variant of Theorem 4.6 is then

$$\begin{aligned}
&\sum_{\zeta \in E(P) \setminus E'} c(\kappa(\zeta)) + \sum_{i=1}^{k} r(\kappa(\zeta_i)) \cdot \left(\frac{\mu(\kappa(\zeta_i))}{2} + \sum_{j=i+1}^{k} \mu(\kappa(\zeta_j)) \right) \\
&= \sum_{\zeta \in E(P) \setminus E'} c(\kappa(\zeta)) + \sum_{i=1}^{k} \left(\overline{\mu}(\kappa(\zeta_i)) \right)^2 + \sum_{i=1}^{k} \sum_{j=i+1}^{k} 2\overline{\mu}(\kappa(\zeta_i))\overline{\mu}(\kappa(\zeta_j)) \\
&= \sum_{\zeta \in E(P) \setminus E'} \overline{c}(\kappa(\zeta_i)) + \left(\sum_{\zeta \in E'} \overline{\mu}(\kappa(\zeta_i)) \right)^2 .
\end{aligned}$$

This is exactly the cost of (P, κ) w. r. t. the objective function of the variant of Theorem 4.5.

□

4.5.2 A Fully Polynomial Time Approximation Scheme

As we do not have a chance to solve the *Cost-Delay Minimum Steiner Path Problem with Loops* optimally (unless $P = NP$) we have to aim for approximation algorithms. In this section we give an approximation algorithm to compute a Steiner path with total cost at most $1 + \epsilon$ times the cost of an optimum solution. The running time will be polynomial in the size of the instance and $\frac{1}{\epsilon}$. Such an algorithm is called *fully polynomial time approximation scheme* (FPTAS) in literature.

The algorithm itself is similar to the algorithms of Moore [Moo59], Bellman [Bel58], and Ford [For56] for the minimum cost t-s path problem in the graph arising from G by reversing all edges. The major difference is that we round costs to powers of $1 + \epsilon$ and store labels with minimum capacitance for each cost-class at each node.

For the proof we need the following well-known lemma:

Lemma 4.7 *For all $x > 1$ it holds that $e^{1-\frac{1}{x}} \leq x \leq e^{x-1}$.*

Proof The second inequality is a simple consequence of the Mean-Value Theorem which yields $y \in [1, x]$ such that $\frac{e^{x-1}-e^{1-1}}{x-1} = e^{y-1} \geq 1$.
To prove the first inequality we note that the function $f(x) = 1 - \frac{1}{x} - \ln(x)$ has its only maximum at $x = 1$. The upper bound $f(x) \leq f(1) = 0$ implies $\ln(x) \geq 1 - \frac{1}{x} \Rightarrow x \geq e^{1-\frac{1}{x}}$.
□

Theorem 4.8 *Let $(G, N = \{s, t\}, c, F, \Delta)$ be an instance of the Cost-Delay Minimum Steiner Path Problem with Loops and $k \in \mathbb{N}$. We require that an s-t Steiner path in G with at most k edges exists. Denote by $n := |V(G)|$ the number of vertices and by $m := |E(G)|$ the number of edges in G. Let*

- $\text{cost}^{\downarrow} > 0$ *be a lower bound on the cost of all Steiner paths with positive cost that end in t,*
- $\text{cost}^{\uparrow} > 0$ *be an upper bound on the cost of all Steiner paths with at most k edges that end in t.*

For each $\epsilon > 0$ there is an algorithm that finds an s-t Steiner path (P, κ) with $|E(P)| \leq k$ and cost at most $(1+\epsilon)$ times the cost of an optimum s-t Steiner path with at most k edges.

Let θ be the maximum running time for an evaluation of one of the functions F, Δ, and the restriction of $\left\lceil \log_{1+\epsilon/2k}(.) \right\rceil$ to the interval $[\text{cost}^{\downarrow}, \text{cost}^{\uparrow}]$. The running time of the algorithm is

$$\mathcal{O}\left(k^2 \cdot m \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon} \cdot \theta\right).$$

Proof We define $\delta := \frac{\epsilon}{2k}$. Note that by this choice of δ and Lemma 4.7,

$$(1+\delta)^k \leq e^{\delta k} = e^{\epsilon/2} = \sum_{i=0}^{\infty} \frac{(\epsilon/2)^i}{i!} = 1 + \epsilon \cdot \sum_{i=1}^{\infty} \frac{\epsilon^{i-1}}{2^i i!} \leq 1 + \epsilon \cdot \sum_{i=1}^{\infty} \frac{1}{2^i} = 1 + \epsilon.$$

We proceed analogously to the algorithms of Moore [Moo59], Bellman [Bel58], and Ford [For56] and propagate labels $(v, i) \in (V(G), \mathbb{Z} \cup \{-\infty\})$ associated with a capacitance $\text{cap}(v, i) \in \mathbb{R}_{\geq 0}$. A label (v, i) with $\text{cap}(v, i) < \infty$ corresponds to a v-t Steiner path (P, κ) in G with

- total cost at most $(1+\delta)^i$ (where $(1+\delta)^{-\infty} := 0$) and
- capacitance of v in (P, κ) at most $\text{cap}(v, i)$.

Initially, we only have a label $(t, -\infty)$ with $\text{cap}(t, -\infty) = 0$ corresponding to the trivial t-t Steiner path. We iterate along all edges in G for k times. When we propagate along edge $e = (v, w)$ we do the following: For all labels (w, i) at w we create a label (v, i') associated with value $\text{cap}(v, i') = \Delta(e, \text{cap}(w, i))$.

If label (w, i) corresponds to the w-t Steiner path (P, κ), (v, i') corresponds to the following v-t Steiner path (P', κ'). P' arises from P by adding a new node ν and an edge ζ between ν and the starting point of P. We set $\kappa'(\nu) = v$, $\kappa'(\zeta) = e$, and $\kappa'(\omega) = \kappa(\omega)$ for $\omega \in V(P) \cup E(P)$.

The index $i' \in \mathbb{Z}$ is chosen minimum such that $(1+\delta)^i$ plus the cost increase induced by the new edge ζ is upper bounded by $(1+\delta)^{i'}$. We keep the new label unless we have an existing label of the form (v, i') with smaller capacitance.

A formal description of the algorithm can be found in Algorithm 6. We use the notation $(P(v, i), \kappa(v, i))$ to denote the Steiner path associated with a label (v, i).

We use the notation $\text{start}(P)$ to denote the starting point of a path P.

Approximation guarantee: Denote the total cost of a Steiner path (P, κ) by $\text{cost}(P, \kappa)$ and the capacitance of a node $\nu \in V(P)$ in (P, κ) by $\text{cap}_{(P,\kappa)}(\nu)$. The properties

$$\text{cost}(P(v, i), \kappa(v, i)) \leq (1+\delta)^i \quad \text{and} \quad \text{cap}_{P(v,i),\kappa(v,i)}(\text{start}(P(v, i))) \leq \text{cap}(v, i)$$

for all labels (v, i) are fulfilled by construction.

Let (P, κ) be any s-t Steiner path with vertices $V(P) = \{s = \nu_\eta, \nu_{\eta-1}, \ldots, \nu_1, \nu_0 = t\}$ and edges $E(P) = \{\zeta_\eta, \zeta_{\eta-1}, \ldots, \zeta_1\}$ (in that order) such that $\eta \leq k$. We may assume that $\kappa(\zeta) \neq \circ$ for all $\zeta \in E(P)$. For $j \in \{0, \ldots, \eta\}$ we define (P_j, κ_j) to be the ν_j-t sub-Steiner path of (P, κ), i. e. P_j consists of vertices $\nu_j, \ldots, \nu_0$ and edges $\zeta_j, \ldots, \zeta_1$. The function κ_j is the restriction of κ to $V(P_j) \cup E(P_j)$.

Instance:	An instance $(G, N = \{s,t\}, c, F, \Delta)$ of the *Cost-Delay Minimum Steiner Path Problem with Loops*, a natural number k.
Output:	An s-t Steiner path (P,κ) in G with $\lvert E(P)\rvert \leq k$ or *fail* if such a Steiner path does not exist.

① **set** $Q(v) := \{(v,i) : i \in \mathbb{Z} \cup \{-\infty\}\}$, $\mathrm{cap}(v,i) = \infty$ for all $v \in V(G)$, $i \in \mathbb{Z} \cup \{-\infty\}$.
② **set** $\mathrm{cap}(t,-\infty) = 0$, $P(t,-\infty) = (\{t\},\emptyset)$, $\kappa(t,-\infty) = (t \mapsto t)$
③ **for** $i = 1$ **to** k **do**
④ **for** $e = (v,w) \in E(G)$ **do**
⑤ **for** labels $(w,i) \in Q(w)$ with $\mathrm{cap}(w,i) < \infty$ **do**
⑥ PROPAGATE$((w,i),e)$
⑦ **if** $\mathrm{cap}(s,i) = \infty$ for all $(s,i) \in Q(s)$ **do**
⑧ **return** *fail*
⑨ **else**
⑩ **select** $(s,i) \in Q(s)$ with $\mathrm{cap}(s,i) \neq \infty$ and i minimum among these labels
⑪ **return** $(P(s,i), \kappa(s,i))$

function PROPAGATE(**label** (w,i), **edge** $e = (v,w)$):
① **set** $x := (1+\delta)^i + c(e) + F(e, \mathrm{cap}(w,i))$
② **set** $i' := \lceil \log_{1+\delta}(x) \rceil$ if $x > 0$ and $i' := -\infty$ if $x = 0$.
③ **set** $P := P(w,i)$, $\kappa := \kappa(w,i)$
④ **augment** P by a new vertex ν and an edge $\zeta = \{(\nu, \mathrm{start}(P))\}$
⑤ **set** $\kappa(\nu) = v$, $\kappa(\zeta) = e$
⑥ **if** $\mathrm{cap}(v,i') > \Delta(e, \mathrm{cap}(w,i))$ **do**
⑦ **set** $\mathrm{cap}(v,i') = \Delta(e, \mathrm{cap}(w,i))$
⑧ **set** $P(v,i') = P$, $\kappa(v,i') = \kappa$

Algorithm 6: FPTAS for the *Cost-Delay Minimum Steiner Path Problem with Loops*. We use the notation $\mathrm{start}(P)$ to denote the starting point of a path P.

Claim: Let $j \in \{0,\dots,\eta\}$. After the j-th iteration of the loop in line ③, $Q(\kappa(\nu_j))$ contains a label $(\kappa(\nu_j), i)$ such that

$$(1+\delta)^i \leq \mathrm{cost}(P_j, \kappa_j) \cdot (1+\delta)^j \quad \text{and} \quad \mathrm{cap}(\kappa(\nu_j), i) \leq \mathrm{cap}_{(P,\kappa)}(\nu_j).$$

Proof (of the claim) For $j = 0$ this is trivial. Let $j > 0$ and assume that after iteration $j-1$ of the loop in line ③, $Q(\kappa(\nu_{j-1}))$ contains a label $(\kappa(\nu_{j-1}), i)$ as desired. When we call PROPAGATE for label $(\kappa(\nu_{j-1}), i)$ and edge $\kappa(\zeta_j)$ in ⑥, we set

$$x := (1+\delta)^i + c(\kappa(\zeta_j)) + F(\kappa(\zeta_j), \mathrm{cap}(\kappa(\nu_{j-1}), i)) \quad \text{and} \quad i' := \lceil \log_{1+\delta}(x) \rceil .$$

Since

$$\begin{aligned}(1+\delta)^{i'} &= (1+\delta)^{\lceil \log_{1+\delta}\left((1+\delta)^i + c(\kappa(\zeta_j)) + F(\kappa(\zeta_j), \mathrm{cap}(\kappa(\nu_{j-1}),i))\right)\rceil} \\ &\leq (1+\delta)^{\lceil \log_{1+\delta}\left(\mathrm{cost}(P_{j-1},\kappa_{j-1})\cdot(1+\delta)^{j-1} + c(\kappa(\zeta_j)) + F\left(\kappa(\zeta_j), \mathrm{cap}_{(P,\kappa)}(\nu_{j-1})\right)\right)\rceil} \\ &\leq \mathrm{cost}\,(P_j) \cdot (1+\delta)^j\end{aligned}$$

and

$$\mathrm{cap}(\kappa(\nu_j), i') \leq \Delta(\kappa(\zeta_j), \mathrm{cap}(\kappa(\zeta_{j-1}), i)) \leq \Delta(\kappa(\zeta_j), \mathrm{cap}_{(P,\kappa)}(\zeta_{j-1})) = \mathrm{cap}_{(P,\kappa)}(\nu_j),$$

label (ν_j, i') fulfills the conditions of the claim after iteration j of the loop in line (3).

□(claim)

For the optimum s-t Steiner path (P^*, κ^*) with $|E(P^*)| \leq k$, the claim for $j = \eta$ yields a label (s, i) such that

$$(1+\delta)^i \leq (1+\delta)^k \cdot \text{cost}(P^*, \kappa^*) \leq (1+\epsilon) \cdot \text{cost}(P^*, \kappa^*).$$

Running time: We start by bounding the number of possible values for i such that a label (v, i) can have finite capacitance.

By definition of $\text{cost}^\downarrow$ and $\text{cost}^\uparrow$, the number I of different possible exponents of $(1+\delta)$ is at most

$$\begin{aligned} k + \left\lceil \log_{1+\delta}(\text{cost}^\uparrow) \right\rceil - \left\lceil \log_{1+\delta}(\text{cost}^\downarrow) \right\rceil + 2 &= \mathcal{O}\left(k + \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\log(1+\delta)}\right) \\ &= \mathcal{O}\left(k + \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\delta}\right). \end{aligned}$$

The last equality uses Lemma 4.7. With $x := 1+\delta$, the inequality $e^{1-\frac{1}{x}} \leq x$ of Lemma 4.7 implies $1 + \frac{1}{\delta} = \frac{1}{1-\frac{1}{\delta+1}} \geq \frac{1}{\ln(1+\delta)}$ and hence, $\frac{1}{\log(1+\delta)} = \mathcal{O}\left(\frac{1}{\delta}\right)$.

To perform PROPAGATE in time $\mathcal{O}(\theta)$ we store all labels (v, i) with $v \in V(G)$ and $i \in \left\{\left\lceil \log_{1+\delta}(\text{cost}^\downarrow)\right\rceil - 1, \ldots, \left\lceil \log_{1+\delta}(\text{cost}^\uparrow)\right\rceil + k\right\} \cup \{-\infty\}$ with corresponding capacitances in $|V(G)|$ arrays of size I (we do not store the other labels). Instead of storing Steiner paths directly we store the input to PROPAGATE of the call in which a label has been updated as predecessor information.

We obtain a total running time of

$$\mathcal{O}(kmI\theta) = \mathcal{O}\left(k^2 m\theta + km\theta \cdot \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\delta}\right) = \mathcal{O}\left(k^2 m\theta \cdot \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\epsilon}\right).$$

□

Corollary 4.9 *Denote* $\text{cost}^\downarrow, \text{cost}^\uparrow, \theta, m, n$ *as in Theorem 4.8 and assume that* $\log\left(\frac{\text{cost}^\uparrow}{\text{cost}^\downarrow}\right)$ *is polynomally bounded in the input size. There is an FPTAS for the Minimum Cost Buffered Steiner Path Problem with running time*

$$\mathcal{O}\left(m^3 \cdot n^2 \cdot \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\epsilon} \cdot \theta\right).$$

□

The condition that $\log\left(\text{cost}^\uparrow/\text{cost}^\downarrow\right)$ is polynomially bounded is naturally fulfilled if the functions $\Delta(e, .)$ are either constant or of the form $x \mapsto x + \mu(e)$, and if the result $F(e, x)$ is of polynomial size (i. e. can be expressed by a polynomial number of bits) for all input values x of polynomial size. This is the case when we model the problem of computing a cheapest path minimizing linear costs and costs for Elmore delay.
Computing $\left\lceil \log_{1+\delta}(x)\right\rceil$ for values x of polynomial size can be done by first computing an approximation y of $\log_{1+\delta}(x)$ with a bounded absolute error $\beta > 0$ and returning

$$\min\{i : (1+\delta)^i \geq x, i \in \mathbb{N} \cap \{\lfloor x - \beta \rfloor, \ldots, \lceil x + \beta \rceil\}\}.$$

4.5.3 Unbuffered Non-Linear Steiner Paths

Although the main motivation of this thesis is its application in buffering, the *Cost-Delay Minimum Steiner Path Problem* is interesting for other applications, too.

- A runner might get tired at some point and will get slower the more distance she or he has travelled.
- Millions of commuters get up early every day to arrive at their jobs before rush hour. The price for using a street depends on the traffic volume on it that is increasing in time.
- In later design steps of VLSI design we might want to find a path with minimum Elmore delay without making further design changes. In particular, we do not want to insert or remove repeaters.

In all of these applications the cost of traversing an edge is dependent on the path travelled before or afterwards. This can easily be modeled as a *Cost-Delay Minimum Steiner Path Problem*. We show how to use the property $\Delta(e,x) \geq x$ for all $e \in E(G), x \in \mathbb{R}_{\geq 0}$ to reduce the running time of the FPTAS presented in Theorem 4.8 significantly. Note that this property is indeed provided in all applications mentioned above.

The following theorem is joint work with Nicolai Hähnle.

Theorem 4.10 *Let $(G, N = \{s,t\}, c, F, \Delta)$ be an instance of the Cost-Delay Minimum Steiner Path Problem (i. e. G does not contain loops, in particular $\Delta(e,x) \geq x$ for all $e \in E(G), x \in \mathbb{R}_{\geq 0}$).*

As in Theorem 4.10 denote by $n := |V(G)|$ the number of vertices and by $m := |E(G)|$ the number of edges in G. Let $\text{cost}^{\downarrow} > 0$ be a lower bound on the cost of all Steiner paths with positive cost ending in t and let $\text{cost}^{\uparrow} > 0$ be an upper bound on the cost of all Steiner paths ending in t. Also, let θ be defined as in Theorem 4.10.

If $\log\left(\text{cost}^{\uparrow}/\text{cost}^{\downarrow}\right)$ is polynomially bounded in the input size, there is an FPTAS for the Cost-Delay Minimum Steiner Path Problem with running time

$$\mathcal{O}\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon} \cdot \left(m\theta + n \cdot \log\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon}\right)\right)\right).$$

Proof The proof is similar to the proof of Theorem 4.10. Instead of processing labels like in the algorithms of Moore [Moo59], Bellman [Bel58], and Ford [For56], we proceed in a Dijkstra [Dij59] manner, i. e. we store non-permanent labels in a heap and propagate from a "minimum" label in each iteration. To compare two labels (v,i) and (v',i') we say that $(v,i) < (v',i')$ if $i < i'$, or $i = i'$ and $\text{cap}(v,i) < \text{cap}(v',i')$.

A full description of the algorithm can be found in Algorithm 7.

Approximation guarantee: Note that by the non-negativity of F and c, the monotonicity of the $F(e,.)$ and $\Delta(e,.)$, and by the property $\Delta(e,x) \geq x$ for all $e \in E(G)$ and $x \in \mathbb{R}_{\geq 0}$, the labels chosen in (5) are always non-decreasing with respect to the relation $\leq$.

Let (P,κ) be any s-t Steiner path with $V(P) = \{s = \nu_\eta, \nu_{\eta-1}, \ldots, \nu_0 = t\}$, $E(P) = \{\zeta_\eta, \zeta_{\eta-1}, \ldots, \zeta_1\}$ (in that order). As in the proof of Theorem 4.10 let (P_j, κ_j) be the ν_j-t sub-Steiner path of P for $j = 0, \ldots, \eta$.

We also reuse the notation $\text{cost}(P,\kappa)$ to denote the cost of a path (P,κ) and $\text{cap}_{(P,\kappa)}(\omega)$ to denote the capacitance of a node $\omega \in V(P)$ in path (P,κ).

Instance: An instance $(G, N = \{s,t\}, c, F, \Delta)$ of the *Cost-Delay Minimum Steiner Path Problem.*
Output: An s-t Steiner path (P, κ) in G.

① **set** $Q(v) := \{(v,i) : i \in \mathbb{Z} \cup \{-\infty\}\}$, $\text{cap}(v,i) = \infty$ for all $v \in V(G)$, $i \in \mathbb{Z} \cup \{-\infty\}$.
② **set** $\text{cap}(t,-\infty) = 0$, $P(t,-\infty) = (\{t\}, \emptyset)$, $\kappa(t,-\infty) = (t \mapsto t)$
③ **set** $S := \emptyset$
④ **while** $\bigcup_{v \in V(G)} Q(v) \neq \emptyset$ **do**
⑤ **select** $(w,i) \in \left(\bigcup_{v \in V(G)} Q(v)\right) \setminus S$ such that $(i, \text{cap}(w,i))$ is lex. minimum
⑥ **if** $w = s$ **do**
⑦ **return** $(P(w,i), \kappa(w,i))$
⑧ **set** $S = S \cup \{(w,i)\}$
⑨ **for** $e = (v,w) \in \delta^-(w)$ **do**
⑩ PROPAGATE$((w,i), e)$ # function from Algorithm 6

Algorithm 7: FPTAS for the *Cost-Delay Minimum Steiner Path Problem.*

Claim: One of the following conditions holds for all $0 \leq j \leq \eta$:

1. At some point of the algorithm there exists a label $(\kappa(v_j), i) \in Q(\kappa(v_j))$ s. t.
$$(1+\delta)^i \leq \text{cost}(P_j, \kappa) \cdot (1+\delta)^j \quad \text{and} \quad \text{cap}(\kappa(v_j), i) \leq \text{cap}_{(P,\kappa)}(v_j).$$
2. $j > 0$ and the algorithm exits with a label (s,i) with
$$(1+\delta)^i \leq \text{cost}(P_{j-1}, \kappa) \cdot (1+\delta)^{j-1}.$$

Proof (of the claim) The proof of the claim is similar to the proof of the claim used in the proof of Theorem 4.10.

For $j = 0$ this is trivial. Let $j > 0$ and assume that there exists a label $(\kappa(\nu_{j-1}), i)$ fulfilling the first condition (if the second condition holds for $j-1$, it automatically holds for all larger indices and we are done as F is non-negative).

First, assume that $(\kappa(\nu_{j-1}), i)$ is selected in ⑤. After we call PROPAGATE with input label $(\kappa(\nu_{j-1}), i)$ and input edge $\kappa(\zeta_j)$, there must be a label $(\kappa(\nu_j), i')$ with

$$(1+\delta)^{i'} \leq \text{cost}(P_j) \cdot (1+\delta)^j \quad \text{and} \quad \text{cap}(\kappa(\nu_j), i') \leq \text{cap}_{(P,\kappa)}(\nu_j).$$

The proof of this property is equal to the proof of the claim used in Theorem 4.10.

If $(\kappa(v_{j+1}), i)$ is never selected, the algorithm must have stopped before. By monotonicity, the exit label $(s, \bar{i})$ must satisfy $\bar{i} \leq i$ and we are in Case 2. □(claim)

As in Theorem 4.10 the claim implies the approximation ratio.

Running time: As seen in the proof of the running time of Theorem 4.10 the number I of labels in $Q(v)$ with finite capacitance is

$$\mathcal{O}\left(n + \frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\delta}\right) = \mathcal{O}\left(\frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\delta}\right).$$

Since we never select from the set S of permanent labels in ⑦, we conclude that PROPAGATE is called at most I times for each edge.

To find minimum labels in (7), we use a Fibonacci heap [FT87]. During the whole algorithm, we have $\mathcal{O}(nI)$ INSERT and DELETEMIN operations as well as mI DECREASE KEY operations and thus, the total running time for all heap operations is $\mathcal{O}(I \cdot (m + n \cdot \log(n \cdot I)))$. Putting everything together we obtain a running time of

$$\begin{aligned}&\mathcal{O}\big(I \cdot (m + n \cdot \log(n \cdot I)) + Im\theta\big)\\ =&\mathcal{O}\left(n\frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon} \cdot \left(m\theta + n \cdot \log\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon}\right)\right)\right).\end{aligned}$$

□

4.6 Cost-Delay Minimum Steiner Trees with Fixed Topologies

In this section we deal with the question how to solve the *Cost-Delay Minimum Steiner Tree Problem with Loops* for nets N that consist of more than two pins.

By the result of Chuzhoy et al. [Chu+05] cited in Section 4.3.1 there is no $o(\log\log|N|)$ approximation algorithm unless every problem in NP can be solved in $\mathcal{O}(n^{\log\log\log n})$ time. This holds even if $F(e,.)$ is a constant function for each edge $e \in E(G)$ and all criticalities $\lambda(t)$ are equal. The best known approximation algorithm for this special case is due to Meyerson, Munagala and Plotkin [MMP00] and has approximation ratio $\mathcal{O}(\log(|N|))$.

We will now show how to approximate the *Cost-Delay Minimum Steiner Tree Problem with Loops* if we already know the topology of the output.

By using the results of Sections 4.5.2 and 4.5.3 we will combine embeddings of so-called *sub-topologies* to compute an overall embedding.

Definition 4.11 *Let T be a topology for N and $v \in V(T)$. The sub-topology $T(v)$ of T with source v is the graph with vertex set*

$$V(T(v)) := \{w \in V(T) : v \in V(T_{[s,w]})\}$$

and edge set

$$E(T(v)) := \{(u,w) \in E(T) : u, w \in V(T(v))\}.$$

An illustration of a sub-topology can be found in Figure 4.6. Note that for a node u, $u + (u,v) + T(v)$ is again a topology.

Theorem 4.12 *Let $G, N, F, c, \Delta, \lambda$ be an instance of the Cost-Delay Minimum Steiner Tree Problem with Loops and let T be a topology for N. Let*

- $\text{cost}^{\downarrow} > 0$ *be a lower bound on the cost of a Steiner path with positive costs in G*
- $\text{cost}^{\uparrow}$ *be an upper bound on the cost of any Steiner tree for N in G.*

Let θ be the maximum running time for one evaluation of F, Δ, and the restriction of $\left\lceil \log_{1+\epsilon/(2|N|(k+2))}(.) \right\rceil$ to the interval $[\text{cost}^{\downarrow}, \text{cost}^{\uparrow}]$.

For each $\epsilon > 0$ there is an algorithm that computes an embedding (A, κ) of T with cost at most $1+\epsilon$ times the cost of an optimum embedding of T. The running time is

$$\mathcal{O}\left(|N|^2 \cdot \left(\left(m + \frac{n|N|}{\epsilon}\right) m^2 \cdot n^2 \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon} \cdot \theta\right)\right).$$

By the second fact of Lemma 2.7, this theorem provides an FPTAS for instances of the *Cost-Delay Minimum Steiner Tree Problem with Loops* with $|N|$ constant and for which $\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})$ is polynomially bounded in the input size: enumerate and embed all possible topologies.

In Sections 4.6.1 and 4.6.2 we prove Theorem 4.12. The high-level idea is to obtain an embedding of T by composition of embeddings of the edges in T. As in the FPTAS of Theorem 4.8 we represent solution candidates by labels.

We identify nodes of T with nodes in the output that we produce. The reason for that is that we want to shorten notation by not mentioning the function ϕ of Definition 2.6 explicitly.

4.6.1 Preparation for the Proof of Theorem 4.12

Representing embedding of sub-topologies by labels

For an edge $(\mathfrak{v},\mathfrak{w}) \in E(T)$ we will create labels $(u,i)_{(\mathfrak{v},\mathfrak{w})}$ for all $u \in V(G)$ and $i \in \mathbb{Z} \cup \{-\infty\}$. Labels are associated with capacitances $\text{cap}_{(\mathfrak{v},\mathfrak{w})}(u,i)$ and if this value is finite, also with an embedding (A,κ) of the arborescence $\mathfrak{v} + (\mathfrak{v},\mathfrak{w}) + T(\mathfrak{w})$ with

- cost at most $(1+\delta)^i$,
- $\kappa(\mathfrak{v}) = u$, and
- $\text{cap}_{(A,\kappa)}(\mathfrak{v}) \leq \text{cap}_{(\mathfrak{v},\mathfrak{w})}(u,i)$. (4.2)

Merging labels at the same graph node

Let $\mathfrak{w}$ of T be a Steiner node with $\delta^+_T(\mathfrak{w}) = \{(\mathfrak{w},\overline{\mathfrak{w}}_1),(\mathfrak{w},\overline{\mathfrak{w}}_2)\}$. At $\mathfrak{w}$ we have to merge labels representing embeddings for $T(\overline{\mathfrak{w}}_1)$ and $T(\overline{\mathfrak{w}}_1)$. Let

$$M'(\mathfrak{w}) := \Big\{ \big\langle (u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}, (u,i_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \big\rangle : \quad u \in V(G),\ \text{labels } (u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)} \text{ and } (u,i_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \text{ s.t. } \text{cap}((u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}), \text{cap}((u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}) < \infty \Big\}$$

be the set of all label pairs with finite capacitance. We say that a pair $\big\langle (u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}, (u,i_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \big\rangle$ is *δ-dominated* by a pair $\big\langle (u,k_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}, (u,k_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \big\rangle$ if

$$\left\lceil \log_{1+\delta}((1+\delta)^{i_1} + (1+\delta)^{i_2}) \right\rceil \geq \left\lceil \log_{1+\delta}((1+\delta)^{k_1} + (1+\delta)^{k_2}) \right\rceil$$

and

$$\text{cap}_{((\mathfrak{w},\overline{\mathfrak{w}}_1))}(u,i_1) + \text{cap}_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}(u,i_2) \geq \text{cap}_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}(u,k_1) + \text{cap}_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}(u,k_2).$$

Let $M_\delta(\mathfrak{w})$ be a maximal subset of $M'(\mathfrak{w})$ with the property that no label pair $M_\delta(\mathfrak{w})$ is δ-dominated by another pair in $M_\delta(\mathfrak{w})$.

As seen in the running time proof of Theorem 4.8 we have up to $\mathcal{O}\left(\frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\delta}\right)$ labels at a vertex. This leads to a set $M'(\mathfrak{w})$ of size $\mathcal{O}\left(n \cdot \left(\frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\delta}\right)^2\right)$. The next lemma tells us that we can compute $M_\delta(\mathfrak{w})$ in a faster running time.

Lemma 4.13 *Using the notation of Theorem 4.12 the set $M_\delta(w)$ has size $\mathcal{O}\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\delta}\right)$ and can be computed in time $\mathcal{O}\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\delta^2} \cdot \theta\right)$.*

Proof Fix $u \in V(G)$ and write $l_i := (u,i)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}$ and $l'_i := (u,i)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}$ for labels $(u,i)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}$, $(u,i)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}$. Let $z < Z \in \mathbb{Z}$ such that all labels l_i and l'_i with $i \notin [z,\dots,Z] \cup \{-\infty\}$ have infinite capacitance.

Using bucket sort we can sort the l_i and l'_i by i in $\mathcal{O}(Z-z)$ time. We can assume that for $i<j$ the cap-value of l_i (respectively of l'_i) is not smaller than the cap-value of l_j (respectively of l'_j). If this is not the case, we erase l_j as it can never be contained in a non-δ-dominated label pair.

For simplicity we assume that we have labels

$$l_{-\infty}, l_z, l_{z+1}, \dots, l_Z \text{ and } l'_{-\infty}, l'_z, l'_{z+1}, \dots, l'_Z.$$

If label $l_{-\infty}$ does not exist, we create a dummy label $l_{-\infty}$ with cap-value ∞. If a label l_i is missing but l_{i-1} exists, we may set $l_i := l_{i-1}$ (respectively $l_i := l_{-\infty}$ for $i = z$). Similarly for the l'_i.

For $i,j \in \mathbb{Z} \cup \{-\infty\}$ define $g(i,j) := \left\lceil \log_{1+\delta}((1+\delta)^i + (1+\delta)^j) \right\rceil$. To create the set $M_\delta(\mathfrak{w})$ we find labels l_i, l'_j for each $\beta \in \{-\infty, z, z+1, \dots, Z\}$ such that $g(i,j) = \beta$ and i and j are maximal in the sense that $g(i+1,j), g(i,j+1) > \beta$. By exchanging roles of the l_i and l'_j it suffices to show how to find i and j with $i \geq j$.

If $\beta = -\infty$, $i = j = -\infty$ is the solution. Let $\beta \geq z$. If $i \leq \left\lfloor \beta - 1 - \frac{1}{\log(1+\delta)} \right\rfloor - 1$,

$$\begin{aligned} g(i,j) &\leq \left\lceil \frac{\log\left(2 \cdot (1+\delta)^{\beta - 2 - \frac{1}{\log(1+\delta)}}\right)}{\log(1+\delta)} \right\rceil = \left\lceil \frac{1 + \left(\beta - 2 - \frac{1}{\log(1+\delta)}\right)\log(1+\delta)}{\log(1+\delta)} \right\rceil \\ &\leq \frac{1}{\log(1+\delta)} + \left(\beta - 2 - \frac{1}{\log(1+\delta)}\right) + 1 = \beta - 1. \end{aligned}$$

Hence, i must lie between $\left\lfloor \beta - 1 - \frac{1}{\log(1+\delta)} \right\rfloor$ and β, and there are at most $\mathcal{O}\left(\frac{1}{\log(1+\delta)}\right) = \mathcal{O}\left(\frac{1}{\delta}\right)$ possible choices for i.

With $r := \log_{1+\delta}\left((1+\delta)^\beta - (1+\delta)^i\right)$, $\log_{1+\delta}\left((1+\delta)^i + (1+\delta)^r\right) = \beta$. This implies that l_i and $l'_{\lfloor r \rfloor}$ is a label pair with $g(i, \lfloor r \rfloor) = \beta$ and i, $\lfloor r \rfloor$ maximal.

As in the running time proof of Theorem 4.8 it follows that $Z - z = \mathcal{O}\left(\frac{\log(\text{cost}^\uparrow/\text{cost}^\downarrow)}{\delta}\right)$ which implies the lemma. □

The topology embedding algorithm

Let k be an upper bound on the number of edges in any optimum Steiner path between two distinct nodes in G. Let $\delta := \frac{\epsilon}{2\cdot(|N|\cdot(k+2))}$.

We embed edges of T into G in reverse topological ordering, i. e. we start with the edges entering the pins in $N \backslash \{s\}$ and end with the edge leaving the source s of N. During the embeddings we create labels of the form $(u,i)_{(\mathfrak{v},\mathfrak{w})}$ with $u \in V(G)$, $i \in \mathbb{Z} \cup \{-\infty\}$, $(\mathfrak{v},\mathfrak{w}) \in E(T)$ as described before .

To produce these labels we run Algorithm 6 on Page 57 until line (6) (including (6)). We refer to this algorithm as the *modified algorithm*.

If we embed an edge $(\mathfrak{v},t) \in E(T)$ entering a sink $t \in N \backslash \{s\}$, we run the modified algorithm on instance $\left(G, \{\mathfrak{v}, t\}, c, \lambda(t) \cdot F, \Delta\right)$ with precision $\frac{\epsilon}{|N|}$ ($\mathfrak{v}$ is considered the source

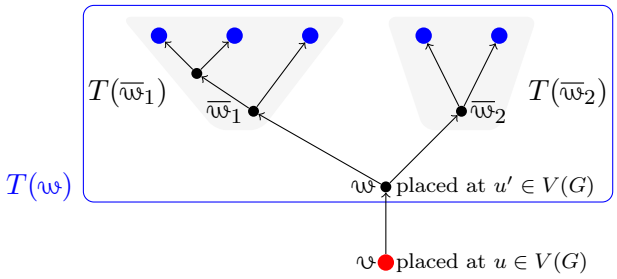


Figure 4.6: Sub-topologies of T in the proof of Theorem 4.12. $T(\mathfrak{w})$ consists of $\mathfrak{w}+(\mathfrak{w},\overline{\mathfrak{w}}_1)+T(\overline{\mathfrak{w}}_1)$ and $\mathfrak{w}+(\mathfrak{w},\overline{\mathfrak{w}}_2)+T(\overline{\mathfrak{w}}_2)$. To find an embedding of $T = T(\mathfrak{v})$ we combine embeddings of the topologies $\mathfrak{w}+(\mathfrak{w},\overline{\mathfrak{w}}_i)+T(\overline{\mathfrak{w}}_i)$ with an u-u' Steiner path.

of the dummy net $\{\mathfrak{v},t\}$ and is ignored by the modified algorithm). As parameter bounding the length of the longest path we choose $k+2$. With this choice, the δ-value in the proof of Theorem 4.8 coincides with the δ-value in this algorithm. For $u \in V(G)\backslash\{t\}$ the output labels $(u,i) = (u,i)_{(\mathfrak{v},t)}$ are already as desired. For label $(t,-\infty)$ the corresponding path (P,κ) might consist of zero edges. We extend it to a path with one edge ζ with $\kappa(\zeta) = \circ$. Labels (t,i) with $i > -\infty$ are not needed.

If we embed an edge $(\mathfrak{v},\mathfrak{w}) \in E(T)$ such that $\mathfrak{w}$ is a Steiner point (in T) with outgoing edges $(\mathfrak{w},\overline{\mathfrak{w}}_1)$, $(\mathfrak{w},\overline{\mathfrak{w}}_2)$, we do the following. First, we create the set $M_\delta(\mathfrak{w})$. Let $G'(\mathfrak{w})$ arise from G by adding a new vertex w and by inserting an edge (u,w) for each label pair $\big\langle (u,i_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}, (u,i_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \big\rangle \in M_\delta(\mathfrak{w})$. Edge (u,w) gets properties

$$\begin{aligned} c((u,w)) &= (1+\delta)^{\lceil \log_{1+\delta}((1+\delta)^{i_1}+(1+\delta)^{i_2})\rceil} \\ F((u,w),x) &= x \\ \Delta((u,w),x) &= x + \mathrm{cap}_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}(u,i_1) + \mathrm{cap}_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}(u,i_2). \end{aligned}$$

Let $\lambda := \sum_{t \in N \cap V(T(\mathfrak{v}))} \lambda(t)$. We run the modified algorithm on instance $(G'(\mathfrak{w}), \{\mathfrak{v},w\}, c, \lambda \cdot F, \Delta)$ with precision $\frac{\epsilon}{|N|}$ ($\mathfrak{v}$ is considered the source of net $\{\mathfrak{v},w\}$ and is again ignored by the modified algorithm). As parameter bounding the length of the output path we choose $k+2$ which is an upper bound on the number of edges in any optimum path in $G'(\mathfrak{w})$.

Let $u \in V(G)$. The algorithm provides labels of the form (u,i) associated with a u-w Steiner path (P,κ) with cost at most $(1+\delta)^i$. We create label $(u,i)_{(\mathfrak{v},\mathfrak{w})}$ with $\mathrm{cap}_{(\mathfrak{v},\mathfrak{w})}(u,i) = \mathrm{cap}(u,i)$. With ζ denoting the edge entering w in P, this label corresponds to the embedding of $\mathfrak{v}+(\mathfrak{v},\mathfrak{w})+T(\mathfrak{w})$ that consists of the combination of $(P-\zeta,\kappa)$ and the embeddings for $\mathfrak{w}+(\mathfrak{w},\overline{\mathfrak{w}}_1)+T(\overline{\mathfrak{w}}_1)$ and $\mathfrak{w}+(\mathfrak{w},\overline{\mathfrak{w}}_2)+T(\overline{\mathfrak{w}}_2)$. See Figure 4.6 for an illustration.

If we embed the edge $(s,\mathfrak{w})$ leaving the source, the topology $s+(s,\mathfrak{w})+T(\mathfrak{w})$ is equal to T. We return the Steiner tree associated with the label $(s,i)_{(s,\mathfrak{w})}$ for which i is minimum.

4.6.2 Proof of Theorem 4.12

We run the algorithm described in the previous section. As parameter k bounding the number of edges in every possibly optimum Steiner path we choose $k = m \cdot n$, which is a feasible choice by Lemma 4.2.

First, observe that the topology embeddings associated with labels created during the algorithm have Properties (4.2). For labels created during the embedding of an edge entering a sink this is true by the proof of Theorem 4.8. For the other labels this is true by construction of the modified graphs $G'(\mathfrak{w})$ with $\mathfrak{w} \in V(T)\backslash N$ (and again by Theorem 4.8).

Proof of approximation guarantee

To prove the approximation guarantee denote the number of edges in the longest path in T leaving a vertex $\mathfrak{v} \in V(T)$ by $\pi(\mathfrak{v})$.

Claim: For all edges $(\mathfrak{v}, \mathfrak{w}) \in V(T)$, all nodes $u \in V(G)$, and any embedding of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$

- for which the number of edges in each Steiner path is at most k,
- for which $\mathfrak{v}$ is mapped to u by the corresponding κ function,
- with cost c, and
- for which the capacitance of $\mathfrak{v}$ is χ,

we create a label $(u, i)_{(\mathfrak{v},\mathfrak{w})}$ such that $(1+\delta)^i \leq c \cdot (1+\delta)^{\pi(\mathfrak{v})\cdot(k+2)}$ and $\mathrm{cap}_{(\mathfrak{v},\mathfrak{w})}(u, i) \leq \chi$.

Proof (of the claim) For edges $(\mathfrak{v}, \mathfrak{w})$ with $\mathfrak{w} \in N\backslash\{s\}$ we have already shown this in the proof of Theorem 4.8. Assume that $\mathfrak{w}$ is a Steiner point with successors $\overline{\mathfrak{w}}_1$ and $\overline{\mathfrak{w}}_2$. Let $u \in V(G)$ and let (B^*, κ^*) be any embedding of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$ with $\kappa^*(\mathfrak{v}) = u$ and for which the total number of edges in each path is at most k. Let c be the cost of (B^*, κ^*) and let χ be the capacitance of $\mathfrak{v}$ in (B^*, κ^*).

Embedding (B^*, κ^*) consists of

- an embedding (B_r, κ_r) of topologies $\mathfrak{w} + (\mathfrak{w}, \overline{\mathfrak{w}}_r) + T(\overline{\mathfrak{w}}_r)$ with $\kappa^*(\mathfrak{w}) = \kappa_r(\mathfrak{w})$ $(r = 1, 2)$,
- an embedding of $(\{\mathfrak{v}, \mathfrak{w}\}, \{(\mathfrak{v}, \mathfrak{w})\})$ that corresponds to an u-w Steiner path (P^*, κ_{P^*}) in the modified graph $G'(\mathfrak{w})$ constructed during the algorithm before embedding $(\mathfrak{v}, \mathfrak{w})$. This Steiner path has at most $k+1$ edges. Node w is the artificial node in $G'(\mathfrak{w})\backslash V(G)$.

Denote the total cost of (B_r, κ_r) by c_r $(r = 1, 2)$. By induction we may assume that for $r = 1, 2$ we have created a label $(u', j_r)_{(\mathfrak{w},\overline{\mathfrak{w}}_r)}$ that satisfies the claim for $(\mathfrak{w}, \overline{\mathfrak{w}}_r) \in E(T)$, node $u' := \kappa^*(\mathfrak{w})$, and embedding (B_r, κ_r). In particular,

$$(1+\delta)^{j_1} \leq (1+\delta)^{\pi(\mathfrak{w})\cdot(k+2)} \cdot c_1, \text{ and} \tag{4.3}$$

$$(1+\delta)^{j_2} \leq (1+\delta)^{\pi(\mathfrak{w})\cdot(k+2)} \cdot c_2. \tag{4.4}$$

Without loss of generality we may assume that $\left\langle (u', j_1)_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}, (u', j_2)_{(\mathfrak{w},\overline{\mathfrak{w}}_2)} \right\rangle$ belongs to the set $M_\delta(\mathfrak{w})$ we computed before embedding $(\mathfrak{v}, \mathfrak{w})$ and the last edge ζ of P^* corresponds to that label pair.

By the proof of Theorem 4.8 we produce a label (u, i) with $\mathrm{cap}(u, i) \leq \chi$ and such that $(1+\delta)^i$ is upper bounded by $(1+\delta)^{|E(P^*)|} \leq (1+\delta)^{k+1}$ times the cost of P^*. Since P^* consists of ζ plus a u-u' Steiner path with cost $c - c_1 - c_2$, it holds that

$$(1+\delta)^i \leq (1+\delta)^{k+1} \cdot \left((1+\delta)^{\lceil \log_{1+\delta}((1+\delta)^{j_1} + (1+\delta)^{j_2}) \rceil} + (c - c_1 - c_2) \right).$$

By a simple calculation we verify that

$$(1+\delta)^{\lceil \log_{1+\delta}((1+\delta)^{j_1} + (1+\delta)^{j_2}) \rceil} \leq (1+\delta) \cdot \left((1+\delta)^{j_1} + (1+\delta)^{j_2} \right).$$

Consequently,

$$\begin{aligned}(1+\delta)^i &\leq (1+\delta)^{k+1} \cdot \left((1+\delta) \cdot \left((1+\delta)^{j_1} + (1+\delta)^{j_2}\right) + (c - c_1 - c_2)\right) \\ &\leq (1+\delta)^{k+2} \cdot \left((1+\delta)^{\pi(w)\cdot(k+2)} \cdot (c_1 + c_2) + (c - c_1 - c_2)\right) \\ &\leq (1+\delta)^{(\pi(w)+1)\cdot(k+2)} \cdot c.\end{aligned}$$

The second inequality uses Equations (4.3) and (4.4). Together with the observation $\pi(v) \geq \pi(w) + 1$ this proves the claim. □(claim)

Let $(s, \mathfrak{w})$ be the edge leaving s in T and let OPT be the cost of an optimum embedding of T. The claim yields a label $(s, i)_{(s,\mathfrak{w})}$ such that

$$(1+\delta)^i \leq (1+\delta)^{\pi(s)\cdot(k+2)} \cdot \mathrm{OPT} < (1+\delta)^{|N|\cdot(k+2)} \cdot \mathrm{OPT} \leq (1+\epsilon) \cdot \mathrm{OPT}.$$

Hence, the described algorithm is indeed an FPTAS.

Proof of running time

To achieve the claimed running time we have to speed-up the modification of Algorithm 6 when we use it on instances with the modified graphs $G'(\mathfrak{w})$. As the artificial node $w \in V(G'(\mathfrak{w}))\backslash V(G)$ has no outgoing edges, the edges entering w can only occur at the end of any path with endpoint w. Hence, we can omit to consider them in Step (4) of Algorithm 6 if the iteration counter i in Step (3) is larger than 1. When we process them in iteration $i = 1$, the set $Q(w)$ contains only one element, namely $(w, -\infty)$. Together with the observation that $\mathrm{cost}^{\uparrow}$ is an upper bound on the cost of any path in $G'(\mathfrak{w})$ and Theorem 4.8 we conclude that each invocation of the modified algorithm has running time

$$\mathcal{O}\left(|N| \cdot m^3 \cdot n^2 \cdot \frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon} \cdot \theta\right)$$

although it is sometimes called on a graph with $\mathcal{O}\left(|N|mn\left(\frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon}\right)\right)$ edges. The time to create a set $M_\delta(\mathfrak{w})$ is $\mathcal{O}\left(|N|^2m^2n^3 \cdot \frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon^2} \cdot \theta\right)$ by Lemma 4.13.

By Lemma 2.7, the number of edges in T is $\mathcal{O}(|N|)$. We obtain a total running time of

$$\begin{aligned}&\mathcal{O}\left(|N|^2 \cdot m^3 \cdot n^2 \cdot \frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon} \cdot \theta\right) + \mathcal{O}\left(|N|^3m^2n^3\frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon^2} \cdot \theta\right) \\ = \;& \mathcal{O}\left(|N|^2m^2n^2\frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon}\left(\frac{|N|n}{\epsilon} + m\right)\theta\right).\end{aligned}$$

□

4.6.3 Topology Embeddings in Graphs without Loops

In the case that G does not contain loops one can obtain an improved running time.

Theorem 4.14 *Let $\epsilon > 0$. In the situation of Theorem 4.12 we can embed topology T with approximation guarantee $1 + \epsilon$ and running time*

$$\mathcal{O}\left(|N|^2 \cdot n \cdot \frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon} \cdot \left(\left(\frac{|N|n^2}{\epsilon} \cdot \theta_{\log}\right) + m\theta + n \cdot \log\left(n \cdot \frac{\log(\mathrm{cost}^{\uparrow}/\mathrm{cost}^{\downarrow})}{\epsilon}\right)\right)\right)$$

if G does not contain any loops.

Sketch of the proof. The proof works exactly as the proof of Theorem 4.12. The only difference is that we use Algorithm 7 from Page 60 without lines (6) and (7) to produce labels. If G does not contain loops, $k = n - 1$ is an upper bound on the number of edges in any optimum Steiner path.

As in the running time proof of Theorem 4.12 we observe that if the end point w of a path search has no outgoing edge, all of its incoming edges are considered at most once. We conclude that the total running time for all $\mathcal{O}(|N|)$ invocations of the algorithm is

$$\mathcal{O}\left(|N|^2 \cdot n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon} \cdot \left(m(\theta_F + \theta_\Delta + \theta_{\log}) + n \cdot \log\left(n \cdot \frac{\log(\text{cost}^{\uparrow}/\text{cost}^{\downarrow})}{\epsilon}\right)\right)\right).$$

Together with Lemma 4.13 we obtain the claimed running time. □

4.7 Electrical and Polarity Constraints

With Theorems 4.12 and 4.14 we have obtained fully polynomial time approximation schemes for the *Minimum Cost Buffered Steiner Tree Problem* if the topology of a solution is known or if the number of sinks is constant. Recall that the *Minimum Cost Buffered Steiner Tree Problem* does not distinguish between buffers and inverters, and ignores capacitance and slew limits.

It is trivial to modify the algorithm such that it obeys capacitance limits. For all loop edges e modeling insertion of a repeater $l \in L$ with capacitance limit $\text{caplim}(l)$ we can simply replace function $F(e, .)$ by function $\overline{F}(e, .)$ with

$$\overline{F}(e, x) = \begin{cases} F(e, x) & \text{if } x \leq \text{caplim}(l), \\ \infty & \text{otherwise.} \end{cases}$$

With the same trick we can enforce that $\overline{F}(e, x) = \infty$ for all edges leaving source s and a value x larger than the capacitance limit of s.

Another class of constraints that can easily be incorporated into the algorithms are polarity constraints (Section 2.5.5). For each label $(u, i)_{(\mathfrak{v},\mathfrak{w})}$ we store a required polarity $\text{pol}(u, i)$. Initial labels (i. e. labels corresponding to the sinks) receive the sink's required polarity. Propagating along a loop edge modeling an inverter switches the polarity requirement. We allow to create two different labels of the form $(u, i)_{(\mathfrak{v},\mathfrak{w})}$ if they have distinct required polarities and only execute the updates in Steps (7) and (8) of the modification of Algorithm 6 if there is another label with smaller capacitance

- at the same node,
- in the same cost class, and
- with the same required polarity.

When computing the sets $M_\delta(\mathfrak{w})$ before embedding edges entering a Steiner node $\mathfrak{w}$ in the given topology, we only allow to merge labels which have the same required polarity. Finally, we output the Steiner tree corresponding to the best source label with required polarity ident. It is easy (but a bit technical) to verify that with these modifications, Theorem 4.12 remains to be an FPTAS and that the asymptotic running time does not increase.

It is an open problem to incorporate slew effects into the results from this chapter without loosing the provable quality.

Chapter 5

Topology Generation

The results of Chapter 4 are more of theoretical than of practical interest. Even for smaller nets the number of possible topologies grows extremely fast as the following table demonstrates (cf. Lemma 2.7). Enumerating all topologies to apply Theorem 4.12 or Theorem 4.14 is unacceptable unless the number of sinks is *very* small.

$\|N\|$	# topologies
2	1
3	1
4	3
5	15
6	105
7	945
8	10 395
9	135 135
10	2 027 025
11	34 459 425
12	654 729 075
13	13 749 310 575

In this chapter we show how to compute a topology with nice properties in terms of delay and net length. Such a topology can then be used as the initial topology for Theorem 4.12 or Theorem 4.14.

The focus in this chapter will be both theoretical and practical. Besides provable bounds for delays and lengths, the topology generation routines presented in this chapter are fast and provide good solutions on practical VLSI repeater tree instances. The latter property is reinforced by post-optimization.

Almost all results of this chapter are joint work with Stephan Held.

5.1 Placed Topologies

The main object of this chapter is a topology T for a net N together with a placement function $p : V(T) \to M$ that specifies a position for each node of T in a metric space (M, dist). We call a pair (T, p) *placed topology*.

In the case that the global routing graph is a 3-dimensional grid graph as described in Section 2.4.3, the 2-dimensional plane $\mathbb{R}^2$ together with the ℓ_1 norm

$$||(x_1, y_1) - (x_2, y_2)||_1 := |x_1 - x_2| + |y_1 - y_2|$$

is a canonical candidate for $(M, \mathrm{dist}) = (\mathbb{R}^2, \ell_1)$.

If we deal with a global routing graph G that does not allow a geometric interpretation but has a cost function $c : E(G) \to \mathbb{R}_{\geq 0}$, the metric closure of (G, c) yields the metric space.

5.1.1 Properties of Placed Topologies

Length. The distance function dist allows us to define the length of a placed topology (T, p) as

$$\text{length}(T, p) := \sum_{(\mathfrak{v},\mathfrak{w})\in E(T)} \text{dist}(p(\mathfrak{v}), p(\mathfrak{w})).$$

In most parts of this chapter the positions p are clear from the context and we will just write $\text{length}(T)$ instead of $\text{length}(T, p)$. We also use the above definition to define the length of other graph structures such as paths and branchings that have positions in M assigned to their nodes. By computing topologies with small length we hope to end up with a Steiner tree with small total costs for congestion and net length after applying the algorithm of Theorem 4.12.

Delay. To achieve small timing costs we need to define the delay of a placed topology. The following delay model was introduced by Bartoschek et al. [Bar+10] and estimates both the delay impact of path lengths and of the increased capacitance induced by sibling branches.

Let s be the source of net N, let $t \in N\backslash\{s\}$, and let $b \geq 0$ be a constant. Bartoschek et al. [Bar+10] estimate the delay from s to t in (T, p) as

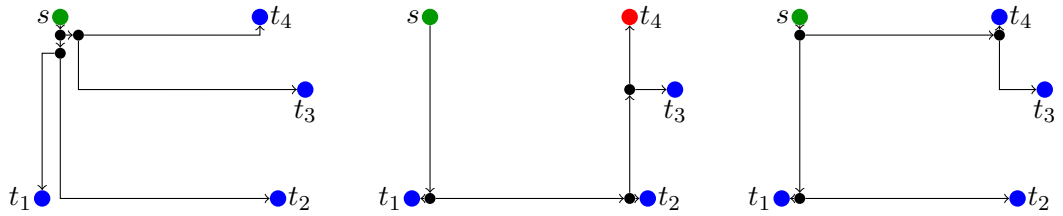
$$\text{delay}_{(T,p)}(s, t) = \left(\sum_{(\mathfrak{v},\mathfrak{w})\in E(T_{[s,t]})} \text{dist}(p(\mathfrak{v}), p(\mathfrak{w})) \right) + b \cdot (|E(T_{[s,t]})| - 1).$$

Due to the special graph structure of a topology defined in Definition 2.4, $(|E(T_{[s,t]})|-1)$ is exactly the number of bifurcations on the unique s-t path in T. Each of these bifurcations is assumed to increase the delay by the constant b. The delay along a path without bifurcations is assumed to be proportional to the length along it, which is a reasonable assumption in an optimally buffered path. Up to scaling we may assume that the delay along a path without bifurcations is *equal* to its length. If positions p are clear from the context, we write $\text{delay}_T(s, t)$ instead of $\text{delay}_{(T,p)}(s, t)$.

5.1.2 Contradicting Objectives

For many instances, short topologies and topologies with short delays look different. An example of such an instance is shown in Figure 5.1.

Computing short topologies is equivalent to the *Shortest Steiner Tree Problem* in (M, dist) and can thus be approximated within a constant factor from optimum (e. g. see [Byr+13; Goe+12]). An example of such a short topology can be found in Figure 5.1(b). A major drawback of such a solution is that delays on paths between the source and timing critical sinks can be large. In this example, the path between s and t_4 is roughly three times longer than a shortest path and has three bifurcations that slow down the signal delay even further. If t_4 is timing critical, the topology in Figure 5.1(b) is not a good solution. In contrast to the short topology, Figure 5.1(a) shows an example of a topology in which all path delays are small. Here, we could achieve that path lengths between s and the sinks $t \in N\backslash\{s\}$ are (almost) shortest possible by placing all Steiner points close to the source's position. The number of bifurcations on each source-sink path is exactly



(a) A topology with small delays but with huge length.

(b) A topology with small length but with $\text{delay}_{(T,p)}(s, t_4)$ large.

(c) A topology trading-off length and delays.

Figure 5.1: Trade-off between length and delays of a topology for $N = \{s, t_1, t_2, t_3, t_4\}$ embedded into $(\mathbb{R}^2, \ell_1)$.

two. In this example it is not possible to obtain a source-sink path with one bifurcation without creating paths with three bifurcations.
Whether the topology depicted in Figure 5.1(a) is an optimum solution from a timing point of view depends of course on the precise definition of delay optimality. In this section we consider the following two variants

Variant a) Given required arrival times $\text{rat}(t) \in \mathbb{R}_{\geq 0}$ for $t \in N \backslash \{s\}$ we want to compute a placed topology (T, p) such that

$$\text{delay}_{(T,p)}(s, t) \leq \text{rat}(t) \text{ for all } t \in N \backslash \{s\}$$

or decide that such a topology does not exist.

Variant b) Given criticalities $\lambda(t) \in \mathbb{R}_{\geq 0}$ for $t \in N \backslash \{s\}$ we want to compute a placed topology (T, p) minimizing

$$\sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \text{delay}_{(T,p)}(s, t).$$

In Section 5.1.3 we list existing algorithms that can compute optimum solutions to both variants. Since delay-optimum topologies can have large lengths, one is usually looking for a trade-off between delay and length. A placed topology trading-off these two contradicting objectives is called *shallow-light topology* in literature. An example can be found in Figure 5.1(c).

Similar to the two variants of delay-optimum topologies we consider the following two problems.

Shallow-Light Topology Problem with Required Arrival Times

Instance: A metric space (M, dist),
a net N with source $s \in N$ and positions $p : N \to M$,
a constant bifurcation delay penalty $b \geq 0$,
required arrival times $\text{rat} : N \backslash \{s\} \to \mathbb{R}_{\geq 0}$.

Output: A topology T for N and positions $p : V(T) \backslash N \to M$ such that

- $\text{delay}_{(T,p)}(s, t) \leq \text{rat}(t)$ for all $t \in T$ and
- $\text{length}(T, p)$ minimum

or find out that such a topology does not exist.

Shallow-Light Topology Problem with Criticalities

Instance: A metric space (M, dist),
a net N with source $s \in N$ and positions $p : N \to M$,
a constant bifurcation delay penalty $b \geq 0$,
criticalities $\lambda(t) \in \mathbb{R}_{\geq 0}$ for $t \in N \backslash \{s\}$.
Output: A topology T for N and positions $p : V(T) \backslash N \to M$ minimizing

$$\text{length}(T, p) + \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \text{delay}_{(T,p)}(s, t).$$

5.1.3 Delay-Minimum Placed Topologies

To compute shallow-light topologies, we have to compute delay-optimum topologies for Variant a) and b) as defined in the previous section.

Since it is always possible to achieve that all paths are shortest paths by placing all Steiner points to the source's position, it suffices to balance the number of the *depths* of the sinks in the computed topology.

For Variant a) we know upper bounds for depths of sinks in feasible solutions:

Definition 5.1 *For a net N with source s, positions $p : N \to M$, required arrival times* $\text{rat} : N \backslash \{s\} \to \mathbb{R}_{\geq 0}$*, and a bifurcation delay penalty $b \geq 0$ we define for each sink $t \in N \backslash \{s\}$:*

$$\text{bif}(t) := \begin{cases} |N| - 2 & \text{if } b = 0 \\ \left\lfloor \frac{\text{rat}(t) - \text{dist}(p(s), p(t))}{b} \right\rfloor & \text{if } b > 0. \end{cases}$$

Hence, to solve Variant a) it suffices to find a topology T for N such that

$$|E(T_{[s,t]})| - 1 \leq \text{bif}(t) \text{ for all } t \in N \backslash \{s\}.$$

Bartoschek et al. [Bar+10] gave a polynomial time algorithm that computes such a topology. Even stronger, they proved that their algorithm maximizes the *worst slack*

$$\min \left\{ \text{rat}(t) - \text{delay}_{(T,p)}(s, t) : t \in N \backslash \{s\} \right\}.$$

Their algorithm has running time of $\mathcal{O}(|N|^2)$ and is sketched in Section 5.3.1.

An even faster method to solve Variant a) or b) is the *Huffman Coding Algorithm* [Huf52] (Algorithm 8).

Although that algorithm is well known, formal proofs can rarely be found in literature. That's why we repeat the proofs of the most basic properties in the following.

Lemma 5.2 ("well known") *The Huffman Coding algorithm (Algorithm 8) has running time $\mathcal{O}(|N| \cdot \log(|N|))$.*
If sinks are sorted by their bif*-value or by their λ-values, respectively, the running time of the algorithm reduces to $\mathcal{O}(|N|)$.*

Instance: A net N with source s and
Variant a) values $\text{bif} : N\backslash\{s\} \to \mathbb{R}$.
Variant b) criticalities $\lambda(t) \in \mathbb{R}$ for $t \in N\backslash\{s\}$.
Output: A topology for N.

① **set** $X = N\backslash\{s\}$, $Y = \emptyset$, $T = (N, \emptyset)$
② **while** $|X \cup Y| > 1$ **do**
③ **let** $\mathfrak{v}, \mathfrak{w} \in X \cup Y$ be the two elements with
Variant a) largest bif-values
Variant b) smallest λ-values
④ **let** $\mathfrak{u}$ be a new node with $p(\mathfrak{u}) = s$ and
Variant a) $\text{bif}(\mathfrak{u}) = \min\{\text{bif}(\mathfrak{v}), \text{bif}(\mathfrak{w})\} - 1$
Variant b) $\lambda(\mathfrak{u}) = \lambda(\mathfrak{v}) + \lambda(\mathfrak{w})$
⑤ **insert** $\mathfrak{u}$ and edges $(\mathfrak{u}, \mathfrak{v})$, $(\mathfrak{u}, \mathfrak{w})$ to T
⑥ **set** $X = X\backslash\{\mathfrak{v}, \mathfrak{w}\}$, $Y = (Y\backslash\{\mathfrak{v}, \mathfrak{w}\}) \cup \{\mathfrak{u}\}$
⑦ **add** an edge between s and the unique remaining element of $X \cup Y$ to T

Algorithm 8: Huffman Coding algorithm for Variant a) and b) as defined in Section 5.1.2.

Proof We assume that the sinks are sorted. As after each iteration of the while-loop ②, the cardinality $|X \cup Y|$ is reduced by 1, it suffices to show that lines ③–⑥ can be implemented to run in constant time. To achieve that we store X and Y in two arrays sorted by their bif-value in non-increasing or by their λ-value in non-decreasing order, respectively.

Since the values $\text{bif}(\mathfrak{u}) = \min\{\text{bif}(\mathfrak{v}), \text{bif}(\mathfrak{w})\} - 1$ decrease or the values $\lambda(\mathfrak{u}) = \lambda(\mathfrak{v}) + \lambda(\mathfrak{w})$ increase in each iteration, appending $\mathfrak{u}$ to the back of Y in ⑥ maintains the sorting. Together with the observation that the elements $\mathfrak{v}$ and $\mathfrak{w}$ in line ③ can be found among the first two elements of each array and can implicitly be removed by storing the indices of the first elements that are supposed to be contained in each array, we can indeed perform ③–⑥ in constant time.

If sinks are not sorted in the input, the initial array storing X must be sorted. That takes $\mathcal{O}(|N| \cdot \log(|N|))$ time. We obtain the claimed running times. □

To prove optimality we need the well known inequality by Kraft [Kra49].

Lemma 5.3 ([Kra49]) *Let* (M, dist) *be a metric space, let* N *be a net with source* s*, and let* $\text{bif} : N\backslash\{s\} \to \mathbb{N}$ *be a function.*

There exists a topology T *for* N *in which the number of bifurcations on the* s*-*t *path is upper-bounded by* $\text{bif}(t)$ *for each* $t \in N\backslash\{s\}$ *if and only if*

$$\sum_{t \in N\backslash\{s\}} 2^{-\text{bif}(t)} \leq 1.$$

By setting the bif-function as in Definition 5.1 we can use Kraft's inequality to check if a feasible solution to Variant a) exists.

Lemma 5.4 ("well known") *If the initial set of sinks together with* bif*-values as defined in Definition 5.1 fulfill Kraft's inequality, the Huffman Coding algorithm returns a topology that is feasible for Variant a).*

Proof We show that the set $X \cup Y$ fulfills Kraft's inequality at each step of the algorithm.

Consider any iteration of the while-loop in line (2) and assume that Kraft's inequality holds before that iteration. Let $\mathfrak{v}$ and $\mathfrak{w}$ be as defined in line (3), without loss of generality $\text{bif}(\mathfrak{v}) \geq \text{bif}(\mathfrak{w})$. Let $\mathfrak{u}$ be the new element created in line (4) with bif-value $\text{bif}(\mathfrak{w}) - 1$ (i. e. $2^{-\text{bif}(\mathfrak{u})} = 2 \cdot 2^{-\text{bif}(\mathfrak{w})}$). Kraft sum $\sum_{\mathfrak{x} \in (X \cup Y \cup \{\mathfrak{u}\}) \setminus \{\mathfrak{v},\mathfrak{w}\}} 2^{-\text{bif}(\mathfrak{x})}$ equals the Kraft sum of instance $X \cup Y$ after decreasing $\text{bif}(\mathfrak{v})$ to $\text{bif}(\mathfrak{w})$.

In the remainder of this proof we show that even after decreasing the bif-value of $\mathfrak{v}$ to $\text{bif}(\mathfrak{w})$, Kraft's inequality is fulfilled. Assume for contradiction that this is not the case (in particular, $\text{bif}(\mathfrak{v}) > \text{bif}(\mathfrak{x})$ for $\mathfrak{x} \in (X \cup Y) \setminus \{\mathfrak{v}\}$). Let Z be a set of

$$|Z| = \left(1 - \sum_{\mathfrak{x} \in X \cup Y} 2^{-\text{bif}(\mathfrak{x})}\right) \cdot 2^{\min\{\text{bif}(\mathfrak{x}) : \mathfrak{x} \in X \cup Y\}}$$

new elements with an assigned bif-value $\text{bif}(\mathfrak{u}) = \min\{\text{bif}(\mathfrak{x}) : \mathfrak{x} \in X \cup Y\}$ for $\mathfrak{u} \in Z$. Since $\sum_{\mathfrak{x} \in X \cup Y \cup Z} 2^{-\text{bif}(\mathfrak{x})} = 1$ and $\text{bif}(\mathfrak{v}) > 0$,

$$2^{\text{bif}(\mathfrak{v})} = 2^{\text{bif}(\mathfrak{v})} \cdot \sum_{\mathfrak{u} \in X \cup Y \cup Z} 2^{-\text{bif}(\mathfrak{u})} = \sum_{\mathfrak{u} \in X \cup Y \cup Z} 2^{\text{bif}(\mathfrak{v}) - \text{bif}(\mathfrak{u})}$$

is an even number. On the other hand, the sum $\sum_{\mathfrak{u} \in X \cup Y \cup Z} 2^{\text{bif}(\mathfrak{v}) - \text{bif}(\mathfrak{u})}$ has exactly one odd summand (for $\mathfrak{u} = \mathfrak{v}$) which is a contradiction. □

Lemma 5.5 ("well known") *The Huffman Coding algorithm for Variant b) returns a topology T with $\sum_{t \in N \setminus \{s\}} \lambda(t) \cdot \text{delay}_T(s,t)$ minimum.*

Proof We prove optimality by induction on $|N|$. For $|N| \leq 2$ this is obvious. Let $|N| \geq 3$ and let $\mathfrak{v}, \mathfrak{w} \in X (= N \setminus \{s\})$ be the two sinks selected in line (2) in the first iteration. Let T be the topology returned by the Huffman Coding algorithm and let T^* be a topology with $\sum_{t \in N \setminus \{s\}} \lambda(t) \cdot \text{delay}_{T^*}(s,t)$ minimum. Let $\mathfrak{u} \in V(T^*) \setminus N$ be a Steiner point with $|E(T^*_{[s,\mathfrak{u}]})|$ maximum.

As $\lambda(\mathfrak{v})$ and $\lambda(\mathfrak{w})$ are minimum among all sinks we may assume that the successors of $\mathfrak{u}$ are exactly $\mathfrak{v}$ and $\mathfrak{w}$ (exchanging $\mathfrak{v}$ or $\mathfrak{w}$ with a successor of $\mathfrak{u}$ cannot make T^* worse). Hence, T^* consists of a topology $\overline{T}^*$ for net $N' := (N \setminus \{\mathfrak{v}, \mathfrak{w}\}) \cup \{\mathfrak{u}\}$ plus edges $(\mathfrak{u}, \mathfrak{v})$, $(\mathfrak{u}, \mathfrak{w})$. By induction hypothesis the Huffman Coding algorithm finds a solution $\overline{T}$ for instance N' with $\lambda(\mathfrak{u}) = \lambda(\mathfrak{v}) + \lambda(\mathfrak{w})$ such that

$$\sum_{t \in N' \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(\overline{T}_{[s,t]})| - 1) \leq \sum_{t \in N' \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(\overline{T}^*_{[s,t]})| - 1).$$

Note that the part of the algorithm for net N after the first iteration is equivalent to the algorithm for net N', i. e. T consists of $\overline{T}$ plus edges $(\mathfrak{u}, \mathfrak{v})$, $(\mathfrak{u}, \mathfrak{w})$. The inequality

$$\begin{aligned}
\sum_{t \in N \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(T_{[s,t]})| - 1) &= \left(\sum_{t \in N' \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(\overline{T}_{[s,t]})| - 1)\right) + b \cdot (\lambda(\mathfrak{v}) + \lambda(\mathfrak{w})) \\
&\leq \left(\sum_{t \in N' \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(\overline{T}^*_{[s,t]})| - 1)\right) + b \cdot (\lambda(\mathfrak{v}) + \lambda(\mathfrak{w})) \\
&= \sum_{t \in N \setminus \{s\}} \lambda(t) \cdot b \cdot (|E(T^*_{[s,t]})| - 1)
\end{aligned}$$

concludes the proof. □

5.2 Nonapproximability

In [HR13] we stated that the existence of a constant factor approximation for the *Shallow-Light Topology Problem with Required Arrival Times* would imply $P = NP$. A detailed proof of this statement can be found in [Rot12], we omitted it in the conference paper [HR13]. The stronger version that we prove now is joint work with Nicolai Hähnle who noticed that the analysis of [Rot12] can be strengthened.

From now on and for the rest of this chapter, *topologies* always mean *placed topologies* although we do not always mention the placement function explicitly.

Theorem 5.6 *There is no $|N|^\beta$-approximation algorithm for the Shallow-Light Topology Problem with Required Arrival Times for any constant $\beta < 1$ unless P=NP.*

Proof Assume, there is an approximation algorithm with approximation ratio $|N|^\beta$ for $\beta < 1$. We use this algorithm to decide an *NP*-complete variant of *Satisfiability*. Let $\mathcal{C}$ be a set of clauses over variables $X = \{x_1, \dots, x_n\}$ where $n = 2^k$ for some $k \in \mathbb{N}\backslash\{1\}$ and each literal appears in at most two clauses. It is *NP*-hard to decide if a set of clauses of this special form is satisfiable (the proof immediately follows from [Tov84]). Furthermore, we may assume that $|\mathcal{C}| \leq 2 \cdot n$.

Define $\overline{X} := \{\overline{x}_1, \dots, \overline{x}_n\}$ as the set of negated literals and interpret the variables in X as the non-negated literals. Let $\mathcal{C}'$ be a set of $2n - |\mathcal{C}|$ elements. We define (M, dist) as the metric closure of an undirected graph G which is defined as follows (see also Figure 5.2(a)). Let $\alpha = \frac{2+3\beta}{1-\beta}$, $m = \lceil n^\alpha \rceil$, $\epsilon = \frac{1}{m}$, and

$$V(G) = \mathcal{C} \cup \mathcal{C}' \cup \{s\} \cup X \cup \overline{X} \cup \{t_j^i : i \in \{1, \dots, n\}, j \in \{1, \dots, m\}\}.$$

Include edges

- $\{s, \chi\}$ for all $\chi \in X \cup \overline{X}$ with length 1,
- $\{\chi, C\}$ for all $\chi \in X \cup \overline{X}$, $C \in \mathcal{C}$ such that $\chi \in C$ with length 1,
- $\{\chi, C'\}$ for all $\chi \in X \cup \overline{X}$, $C' \in \mathcal{C}'$ with length 1,
- $\{\chi, t_j^i\}$ for all $\chi \in X \cup \overline{X}$ s.t. $\chi = x_i$ or $\chi = \overline{x}_i$, $j \in \{1, \dots, m\}$ with length ϵ.

Let $N := \{s\} \cup \mathcal{C} \cup \mathcal{C}' \cup \{t_j^i : i \in \{1, \dots, n\}, j \in \{1, \dots, m\}\}$, $b = 1$, $p(t) = t$ for all $t \in N$, $\text{rat}(C) = k+m+3$ for $C \in \mathcal{C} \cup \mathcal{C}'$, $\text{rat}(t_j^i) = 1+\epsilon+j+k$ for all $i \in \{1, \dots, n\}, j \in \{1, \dots, m\}$.

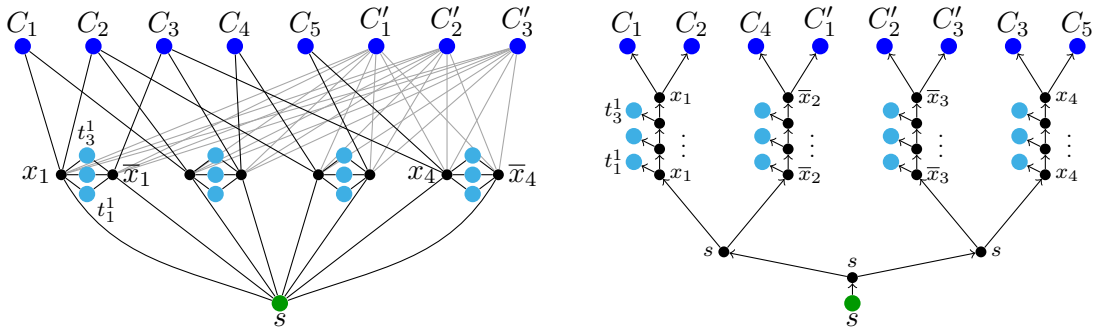
Using Definition 5.1 it holds that

$$\begin{aligned}\sum_{t \in N\backslash\{s\}} 2^{-\text{bif}(t)} &= 2n \cdot \left(2^{-(m+k+1)}\right) + n \cdot \sum_{j=1}^{m} 2^{-(j+k)} \\ &= n \cdot 2^{-(m+k)} + n \cdot \left(2^{-k} \cdot (1 - 2^{-m})\right) = n \cdot 2^{-k} = 1\end{aligned}$$

and by Lemma 5.3, a feasible topology for the constructed instance exists. As Kraft's inequality is satisfied with equality, the number of Steiner vertices on an s-t path is uniquely determined by $\text{bif}(t)$ for every $t \in N\backslash\{s\}$. The following claim proves the theorem.

Claim: If $\mathcal{C}$ is satisfiable, a feasible topology for N with length at most $3n+nm\cdot\epsilon$ exists (see Figure 5.2(b)). Otherwise, each feasible topology has cost at least $2n + k + (1 + \epsilon n) \cdot m > |N|^\beta \cdot (3n + nm \cdot \epsilon)$.

Proof (of the claim) We prove that $\mathcal{C}$ is satisfiable if and only if there is a feasible topology T for N in which no Steiner point ϑ with $|E(T_{[s,\vartheta]})| = k + m$ is placed at position s.



(a) The graph G which defines (M, dist) in the proof of Theorem 5.6 ($m = 3$).

(b) Feasible topology with small length. Labels indicate positions. $x_1 = x_4 = \text{true}$, $x_2 = x_3 = \text{false}$ satisfies all clauses.

Figure 5.2: Graph G and a light topology without delay violations in the proof of Figure 5.2(a). The corresponding instance of *Satisfiability* is $X = \{x_1, x_2, x_3, x_4\}$, $C_1 = \{x_1, x_2\}$, $C_2 = \{x_1, x_2, x_3\}$, $C_3 = \{\overline{x}_1, \overline{x}_2, x_4\}$, $C_4 = \{\overline{x}_2, x_3\}$, $C_5 = \{\overline{x}_3, x_4\}$. For simplicity, we choose $m = 3$.

First, we assume that $\mathcal{C}$ is satisfiable. Fix a satisfying truth assignment. We can define a function $\psi' : \mathcal{C} \to X \cup \overline{X}, \psi'(C) = \{\chi \in C : \chi \text{ is a true literal}\}$.

Note that $|\psi'^{-1}(\chi)| \leq 2$ for each true literal χ. By choice of $|\mathcal{C}'|$ we can extend ψ' to a function $\psi : \mathcal{C} \cup \mathcal{C}' \to X \cup \overline{X}$ such that $|\psi^{-1}(\chi)| = 2$ for each true literal χ. For each such χ place $m+1$ Steiner points $\mathfrak{v}_j^\chi$ at position χ ($j = 1, \ldots, m+1$) and include edges $(\mathfrak{v}_j^\chi, \mathfrak{v}_{j+1}^\chi)$ and $(\mathfrak{v}_j^\chi, t_j^i)$ for $j = 1, \ldots, m$ where $\chi \in \{x_i, \overline{x}_i\}$. Also include edges $(\mathfrak{v}_{m+1}^\chi, C)$ for each $C \in \psi^{-1}(\chi)$. The resulting branching can be extended to a topology T for N by a balanced binary tree for

$$\{\mathfrak{v}_1^\chi : \chi \text{ is a true literal}\} \cup \{s\}$$

in which all inner vertices are positioned at s. Figure 5.2(b) depicts this solution.
It is easy to see that this topology satisfies $\mathrm{delay}_T(s,t) = \mathrm{rat}(t)$ for each $t \in N \backslash \{s\}$ and hence, is feasible. The length of T is $3n + nm \cdot \epsilon$.

The set of Steiner points with distance $k+m$ from the source is exactly the set $\{\mathfrak{v}_m^\chi : \chi \text{ is a true literal}\}$. None of these vertices is placed at position s.

Conversely, let T be a feasible topology for N in which no Steiner point $\mathfrak{v}$ with $|E(T_{[s,\mathfrak{v}]})| = k+m$ is placed at position s. If there is a sink t' for which $T_{[s,t']}$ is not a shortest path, $|E(T_{[s,t']})| - 1$ must be at most $\left\lfloor \mathrm{rat}(t') - \sum_{(\mathfrak{v},\mathfrak{w}) \in E(T_{[s,t']})} \mathrm{dist}_G(p(\mathfrak{v}), p(\mathfrak{w})) \right\rfloor < \mathrm{bif}(t)$. By Lemma 5.3 we get a contradiction. Hence, all Steiner points must be placed in $\{s\} \cup X \cup \overline{X}$. Since

$$1 \geq \sum_{t \in T} 2^{-\min\{\mathrm{bif}(t), |E(T_{[s,t']})|\}} \geq \sum_{t \in T} 2^{-\mathrm{bif}(t)} = 1,$$

$\mathrm{bif}(t') = |E(T_{[s,t']})|$ for all $t \in N \backslash \{s\}$ which implies that the predecessor of terminal t_m^i ($i \in \{1, \ldots, n\}$) must be a Steiner point $\mathfrak{v}$ with $|E(T_{[s,\mathfrak{v}]})| = k+m$ and which reaches a sink in $\mathcal{C} \cup \mathcal{C}'$. Thus, it must be placed in $\{x_i, \overline{x}_i\}$. Its successor must be positioned at the same point. Let

$$V' := \{\mathfrak{v} \in V(T) \backslash N : |E(T_{[s,\mathfrak{v}]})| = k+m+1\}.$$

Since the successors of elements in V' are exactly the sinks in $\mathcal{C} \cup \mathcal{C}'$, $|V'| = |\mathcal{C} \cup \mathcal{C}'|/2 = n$. We conclude that for each $i = 1, \ldots, n$ there is exactly one $\chi \in \{x_i, \overline{x}_i\}$ such that there is a vertex of V' with position χ.
We use this property to define a truth assignment of $\{x_1, \ldots, x_n\}$ as follows:

$$\text{for each } i = 1, \ldots, n \text{ set } x_i \text{ to } \begin{cases} \text{true} & \text{if there is } \mathfrak{v} \in V' \text{ such that } p(\mathfrak{v}) = x_i \\ \text{false} & \text{otherwise.} \end{cases}$$

Let $C \in \mathcal{C}$ and let $\mathfrak{v} \in V'$ be the predecessor of C in T. Since $T_{[s,C]}$ is a shortest path, $p(\mathfrak{v})$ corresponds to a true literal containing C. Thus, $\mathcal{C}$ is satisfiable.

Assume that $\mathcal{C}$ is not satisfiable. Let T be any feasible topology for N. As seen before, all paths contained in T are shortest paths. Let γ be the number of Steiner points placed at s. All of the $|N| - 2 - \gamma = 2n + mn - 1 - \gamma$ other Steiner points are placed with distance 1 to the source. Since $\mathcal{C}$ is not satisfiable, at least one Steiner point $\mathfrak{v}$ such that $|E(T_{[s,\mathfrak{v}]})| = k + m$ and hence all Steiner points from which $\mathfrak{v}$ is reachable in T must be placed at position s. Thus, $\gamma \geq k + m$. For $\mathfrak{v} \in V(T) \backslash \{s\}$ let $\text{pred}(\mathfrak{v})$ denote the predecessor of $\mathfrak{v}$ in T. We have

$$\begin{aligned} \text{length}(T) &= \sum_{\mathfrak{v} \in V(T) \backslash \{s\}} \text{dist}_G(p(\mathfrak{v}), p(\text{pred}(\mathfrak{v}))) \\ &\geq \sum_{\mathfrak{v} \in V(T) \backslash \{s\}} (\text{dist}_G(s, p(\mathfrak{v})) - \text{dist}_G(s, p(\text{pred}(\mathfrak{v})))) \\ &= \sum_{t \in N \backslash \{s\}} \text{dist}_G(s, t) - \sum_{\mathfrak{v} \in V(T) \backslash N} \text{dist}_G(s, p(\mathfrak{v})) \\ &\geq 4n + (1 + \epsilon) \cdot nm - 2n - mn + 1 + \gamma \\ &> 2n + k + (1 + \epsilon n) \cdot m. \end{aligned}$$

By choice of α, $4 \leq n = n^{\alpha(1-\beta)-3\beta-1}$ and hence,

$$n^{(\alpha+3)\beta} \cdot 4n \leq n^{\alpha} \leq m < 2n + k + (1 + \epsilon n)m. \tag{5.1}$$

By choice of m and ϵ,

$$|N|^{\beta} \cdot (3n + nm \cdot \epsilon) = (n \cdot m + 2n)^{\beta} \cdot (3n + nm \cdot \epsilon) < n^{(\alpha+3)\cdot\beta} 4n. \tag{5.2}$$

(5.1) and (5.2) yield the result. □

Note that the Huffman Coding algorithm [Huf52] (Algorithm 8 for Variant a)) is an $|N|$ approximation algorithm as it produces a feasible solution with cost

$$\sum_{t \in N \backslash \{s\}} \text{dist}(p(s) - p(t)) < |N| \cdot \text{OPT},$$

where OPT denotes the length of an optimum feasible placed topology. In this sense, $\beta = 1$ is the smallest value such that the *Shallow-Light Topology Problem with Required Arrival Times* admits an $\mathcal{O}\left(|N|^{\beta}\right)$ approximation (unless *P*= *NP*).

5.3 Bicriteria Approximation

After we have found out that the *Shallow-Light Topology Problem with Required Arrival Times* is hopeless to be approximated within a non-trivial approximation factor, we will now allow to relax the delay constraints

$$\text{delay}_T(s,t) \leq \text{rat}(t).$$

Our goal is to obtain short topologies in which delay violations $\text{delay}_T(s,t) - \text{rat}(t)$ are not too large. In Section 5.4 we return to the *Shallow-Light Topology Problem with Criticalities*.

5.3.1 Previous Work

For the case that the bifurcation delay penalty b is zero there are several examples of algorithms that compute topologies bounding both path lengths and topology length. For the case $M = \{p(t) : t \in N\}$, Alpert et al. [Alp+95] gave an algorithm that combines the algorithms of Prim [Pri57] and Dijkstra [Dij59]. Both algorithms extend a current topology T by a new node $\mathfrak{w}$ with a position $p(\mathfrak{w}) \notin \{p(\mathfrak{v}) : \mathfrak{v} \in V(T)\}$ and an edge $(\mathfrak{v}, \mathfrak{w})$ with $\mathfrak{v} \in V(T)$. While Prim's algorithm selects $\mathfrak{v}$ and $p(\mathfrak{w})$ such that $\text{dist}(p(\mathfrak{v}), p(\mathfrak{w}))$ is minimum, Dijkstra's algorithm selects $\mathfrak{v}$ and $p(\mathfrak{w})$ such that $\text{delay}_{(T,p)}(s,\mathfrak{v}) + \text{dist}(p(\mathfrak{v}), p(\mathfrak{w}))$ is minimum. By choosing $\mathfrak{v}$ and $p(\mathfrak{w})$ minimizing

$$\xi \cdot \text{dist}(p(\mathfrak{v}), p(\mathfrak{w})) + (1-\xi) \cdot (\text{delay}_{(T,p)}(s,\mathfrak{v}) + \text{dist}(p(\mathfrak{v}), p(\mathfrak{w}))) \text{ for some } \xi \in [0,1]$$

we obtain the *Prim-Dijkstra Algorithm* by Alpert et al. [Alp+95]. Although that algorithm turns out to produce good results in practice, theoretical bounds are not known for $\xi \in (0,1)$.

Cong et al. [Con+92] developed an algorithm that trades-off topology length and the length of the longest path. For a given value $\epsilon > 0$ their algorithm computes a topology with length at most $1 + \frac{2}{\epsilon}$ times the cost of a minimum spanning tree for N. The length of each source-sink path is at most a factor $1 + \epsilon$ longer than $\max_{t \in N \setminus \{s\}} \{\text{dist}(p(s), p(t))\}$. Roughly speaking, the algorithm of Cong et al. [Con+92] operates in two phases. In the first phase, a short topology arising from a minimum spanning tree by applying local transformations to achieve the degree constraints is computed. Bounds on path lengths are ignored in that phase. This topology is then traversed by a *depth-first search*. When an edge $(\mathfrak{v}, \mathfrak{w})$ is traversed, we check if $\text{delay}_T(s, \mathfrak{w}) \leq (1+\epsilon) \cdot \text{dist}(p(s), p(\mathfrak{w}))$ holds (where T is the current topology). If this is not the case, edge $(\mathfrak{v}, \mathfrak{w})$ is replaced by a direct connection between s and $\mathfrak{v}$.

This algorithm could be improved by Khuller et al. [KRY95] who achieved path length bounds of $\text{delay}_T(s,t) \leq (1+\epsilon) \cdot \{\text{dist}(p(s), p(t)\}$ for each $t \in N \setminus \{s\}$ while obtaining the same bound on topology length. Their algorithm proceeds similar to the algorithm of Cong et al. [Con+92]. The only difference is that edge $(\mathfrak{v}, \mathfrak{w})$ is considered for a second time after all vertices reachable from $\mathfrak{w}$ in T have been visited. During that second traversal, it is checked if the delay to $\mathfrak{v}$ can be reduced by connecting it directly to $\mathfrak{w}$, i. e. $\text{delay}_T(s, \mathfrak{v}) > \text{delay}_T(s, \mathfrak{w}) + \text{dist}(p(\mathfrak{v}), p(\mathfrak{w}))$. This equation can be fulfilled if a direct connection between s and $\mathfrak{w}$ has been added before. If the equality is fulfilled, the edge entering $\mathfrak{v}$ in T is replaced by $(\mathfrak{w}, \mathfrak{v})$.

In this section we generalize the result of Khuller et al. [KRY95] to the case that $b > 0$. We also get rid of the requirement that the short topology computed in the first phase arose from a minimum spanning tree.

Figure 5.3: Median p of three points p_1, p_2, p_3.

Bartoschek et al. [Bar+10] proposed an algorithm for the *Shallow-Light Topology Problem with Required Arrival Times*. They first sort the sinks $N\backslash\{s\}$ by their bif-value (see Definition 5.1) in non-decreasing order. Let $t_1, \ldots, t_{|N|-1}$ be that ordering. Starting with the topology $(\{s, t_1\}, \{(s, t_1)\})$ they insert the remaining sinks $t_2, \ldots, t_{|N|-1}$ into the current topology (in that order). A sink t_i is inserted by subdividing an edge $(\mathfrak{v}, \mathfrak{w})$ by a new vertex $\mathfrak{u}$ and adding an edge between $\mathfrak{u}$ and t_i. Steiner node $\mathfrak{u}$ is positioned such that $p(\mathfrak{u})$ lies both on a shortest $p(\mathfrak{v})-p(\mathfrak{w})$ and a shortest $p(\mathfrak{v})-p(t_i)$ path and $\operatorname{dist}(p(\mathfrak{v}), p(\mathfrak{u}))$ is maximum. In the most relevant case that $(M, \operatorname{dist}) = (\mathbb{R}^2, \ell_1)$, this point is exactly the *median* of $p(t_i)$, $p(\mathfrak{v})$, and $p(\mathfrak{w})$.

Definition 5.7 (Median) *The median of three points $p_1, p_2, p_3 \in \mathbb{R}^2$ is the point $p \in \mathbb{R}^2$ for which p_x is the median of $(p_1)_x$, $(p_2)_x$, $(p_3)_x$ and p_y is the median of $(p_1)_y$, $(p_2)_y$, $(p_3)_y$.*

Figure 5.3 illustrates this definition. Bartoschek et al. [Bar+10] choose edge $(\mathfrak{v}, \mathfrak{w})$ such that it minimizes a convex combination

$$(1-\xi) \cdot \text{“length of the resulting topology”} - \xi \cdot \text{“worst slack of the resulting topology”},$$

where ξ is a parameter that can take values between 0 and 1.

Although the algorithm of Bartoschek et al. [Bar+10] turns out to be successful on practical repeater tree instances, provable delay bounds or topology length bounds are only known for the case $\xi = 0$ where the computed topology is at most as large as a minimum spanning tree for N, and for the case $\xi = 1$ where it maximizes the worst slack.

5.3.2 A Bicriteria-Approximation Algorithm

We now formulate a so-called bicriteria-approximation algorithm. This result is joint work with Stephan Held [HR13] and is based on the algorithm of Khuller et al. [KRY95].

Theorem 5.8 *Let (M, dist) be a metric space and let N be a net with source $s \in N$ and positions $p : N \to M$. Let $b \geq 0$ be a bifurcation delay penalty and let $\operatorname{rat} : N\backslash\{s\} \to \mathbb{R}_{\geq 0}$ be required arrival times such that a feasible solution for the Shallow-Light Topology Problem with Required Arrival Times exists (see Lemma 5.3). Let (T_{init}, p_{init}) be any placed topology. For each $\epsilon > 0$ we can compute a placed topology (T, p) such that*

$$\operatorname{delay}_{(T,p)}(s, t) \leq 2 \cdot b + \epsilon \cdot \operatorname{rat}(t) \text{ for all } t \in N\backslash\{s\} \text{ and} \tag{5.3}$$

$$\operatorname{length}(T, p) < \left(1 + \frac{2}{\epsilon}\right) \cdot \operatorname{length}(T_{init}, p_{init}) + \frac{4b \cdot (|N| - 1)}{\epsilon}. \tag{5.4}$$

The running time of the algorithm is $\mathcal{O}(|N| \log |N| + \psi)$, where ψ is the time needed to query $\operatorname{dist}(p(\mathfrak{v}), p(\mathfrak{w}))$ for all $(\mathfrak{v}, \mathfrak{w}) \in E(T_{init})$ and $\operatorname{dist}(s, t)$ for all $t \in N\backslash\{s\}$.

Note that $\psi = \mathcal{O}(|N|)$ in many applications, e. g. in the case that $(M, \mathrm{dist}) = (\mathbb{R}^2, \ell_1)$. In the remainder of this section we prove Theorem 5.8.

Description of the algorithm. Let $\mathfrak{s}'$ be the successor of s in T_{init} and let $\overleftrightarrow{T_{\text{init}}}$ be the directed graph with vertex set $V(T_{\text{init}})$ and edge set $\{(\mathfrak{v}, \mathfrak{w}), (\mathfrak{w}, \mathfrak{v}) : (\mathfrak{v}, \mathfrak{w}) \in E(T_{\text{init}})\}$. Note that $\overleftrightarrow{T_{\text{init}}}$ is Eulerian. We perform an Eulerian walk in $\overleftrightarrow{T_{\text{init}}} - s$ starting at $\mathfrak{s}'$. During the walk we keep track of a branching B and an estimate $d(\mathfrak{v})$ on the delay of the s-$\mathfrak{v}$ path in the final topology for each vertex $\mathfrak{v}$. Initially, set $B := T_{\text{init}} - s$ and $d(\mathfrak{s}') := \mathrm{dist}(p(s), p(\mathfrak{s}'))$ (see Figure 5.4(a) for an example). Throughout the whole algorithm, for vertices $\mathfrak{v} \in V(B)$ that are not roots (i. e. for vertices $\mathfrak{v} \in V(B)$ such that $|\delta_B^-(\mathfrak{v})| = 1$) we recursively set $d(\mathfrak{v}) := d(\mathfrak{u}) + b + \mathrm{dist}(p(\mathfrak{u}), p(\mathfrak{v}))$, where $(\mathfrak{u}, \mathfrak{v}) \in E(B)$ is the unique edge entering $\mathfrak{v}$ in B. By construction, each forward edge $(\mathfrak{v}, \mathfrak{w}) \in E(T_{\text{init}})$ is visited prior to its backward counterpart $(\mathfrak{w}, \mathfrak{v})$ and when $(\mathfrak{w}, \mathfrak{v})$ is visited, the tour finished visiting vertices in the sub-tree of T_{init} rooted at $\mathfrak{w}$.

When we visit a forward edge $(\mathfrak{v}, \mathfrak{w}) \in E(T_{\text{init}})$, we do nothing if $\mathfrak{w} \notin N \backslash \{s\}$. Otherwise, $\mathfrak{w} \in N \backslash \{s\}$ is a leaf and we check if

$$d(\mathfrak{w}) > (1 + \epsilon) \cdot \mathrm{rat}(\mathfrak{w}). \tag{5.5}$$

If this is the case, we delete the edge $(\mathfrak{v}, \mathfrak{w})$. The sink $\mathfrak{w}$ becomes a new root of B and we set $d(\mathfrak{w}) = \mathrm{dist}(p(s), p(\mathfrak{w})) + b \cdot \mathrm{bif}(\mathfrak{w})$ (see Figures 5.4(a) and 5.4(b)).

When we visit a backward edge $(\mathfrak{w}, \mathfrak{v}) \in E(\overleftrightarrow{T_{\text{init}}}) \backslash E(T_{\text{init}})$, we check whether it is better to merge the current sub-tree of B rooted at $\mathfrak{v}$ with the connected component of B containing $\mathfrak{w}$. More precisely, we check if

$$d(\mathfrak{v}) > d(\mathfrak{w}) + \mathrm{dist}(\mathfrak{w}, \mathfrak{v}) + b. \tag{5.6}$$

Note that by the definition of d, this can only be the case if the edge $(\mathfrak{v}, \mathfrak{w})$ is not in B anymore. If Condition (5.6) is true, we

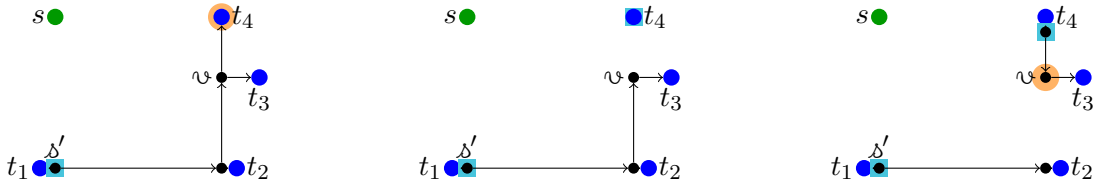
- delete the edge currently entering $\mathfrak{v}$ (unless $\mathfrak{v}$ is a root of B),
- subdivide the edge currently entering $\mathfrak{w}$ by a Steiner vertex placed at $p(\mathfrak{w})$ and connect it to $\mathfrak{w}$ if $\mathfrak{w}$ is not a root of B,
- create a new Steiner point $\mathfrak{u}$ placed at $p(\mathfrak{w})$, connect it to $\mathfrak{v}$ and $\mathfrak{w}$, and set $d(\mathfrak{u}) = d(\mathfrak{w})$ if $\mathfrak{w}$ is a root.

See Figure 5.4(c) for an illustration. The vertex $\mathfrak{u}$ is the new root of the connected component of B containing $\mathfrak{v}$ and $\mathfrak{w}$.

When we have finished the Eulerian walk, we make sure that $|\delta^+(\mathfrak{v})| = 2$ for all $\mathfrak{v} \in V(B) \backslash N$. If $|\delta^+(\mathfrak{v})| + |\delta^-(\mathfrak{v})| \leq 1$ for a Steiner point $\mathfrak{v}$, we delete it. If $\mathfrak{v} \in V(B) \backslash N$ has both out-degree and in-degree equal to one, delete it and connect its predecessor with its successor.

Let X be the set of roots of connected components of B (e. g. boxed vertices in Figure 5.4(c)). Note that $\mathfrak{s}' \in X$ unless there are no sinks left in the connected component of B containing $\mathfrak{s}'$ after the Eulerian walk. Set $\mathrm{rat}'(t') := d(t') + b$ for $t' \in X$. Let $\mathrm{bif}' : X \to \mathbb{N}$ be defined analogously to bif in Definition 5.1.

We have $\sum_{t' \in X} 2^{-\mathrm{bif}'(t')} \leq \frac{1}{2} + \frac{1}{2} \cdot \sum_{t \in N \backslash \{s\}} 2^{-\mathrm{bif}(t)} \leq 1$ and hence, a topology T_{toplvl} for net $X \cup \{s\}$ in which $\mathrm{delay}_{T_{\text{toplvl}}}(s, t') \leq \mathrm{rat}'(t')$ for all $t' \in X$ exists and can be computed



(a) Initial branching B when $(T_{\text{init}}, p_{\text{init}})$ is given by Figure 5.1(b). When visiting a forward arc $(\mathfrak{v}, t_4) \in E(T_{\text{init}})$, we check if $d(t_4) > (1+\epsilon) \cdot \text{rat}(t_4)$.

(b) If so, a new connected component of B with root t_4 is created.

(c) When visiting a backward arc $(t_4, \mathfrak{v})$, we reconnect $\mathfrak{v}$ if (5.6) holds.

Figure 5.4: The branching B at different stages of the Eulerian walk. The wide orange circles mark the head vertex of the currently visited edge in $\overleftrightarrow{T_{\text{init}}}$. The blue boxes mark roots in B.

in $\mathcal{O}(|X| \cdot \log(|X|)) \subseteq \mathcal{O}(|N| \cdot \log(|N|))$ time using Huffman Coding [Huf52]. All Steiner vertices in T_{toplvl} are placed at position $p(s)$ and hence,

$$\text{length}(T_{\text{toplvl}}) = \sum_{t' \in X} \text{dist}(p(s), p(t')).$$

Finally, the algorithm returns topology $T = T_{\text{toplvl}} + B$ (Figure 5.1(c) in our example). Positions of all nodes are chosen according to their position in T_{toplvl} or in B respectively.

Since the running time claim of Theorem 5.8 is clear, it suffices to prove worst slack and length bound.

Proof of the worst slack bound of Theorem 5.8. Let $t \in N \backslash \{s\}$ be a sink. After the first visit of t, $d(t) \leq (1+\epsilon) \cdot \text{rat}(t)$ by (5.5). Note that $d(t)$ increases only if an edge on the path from the root of its containing connected component and t is subdivided by a Steiner point during its second visit, i. e. after checking (5.6). Due to the subdivision, this can happen at most once. With $\text{rat}'(t') = d(t') + b$ for all $t' \in X$, we conclude that $\text{delay}_T(r, t) \leq (1+\epsilon) \cdot \text{rat}(t) + 2b$.

Proof of the length bound of Theorem 5.8. First note that

$$\text{length}(B) \leq \text{length}(T_{\text{init}}) - \text{dist}(s, \mathfrak{s}')$$

holds at the end of the Eulerian walk. Since $\text{length}(T) = \text{length}(B) + \text{length}(T_{\text{toplvl}})$, it suffices to estimate $\text{length}(T_{\text{toplvl}})$.

Let $X_1 := \{t_1, \ldots, t_k\}$ be the set of sinks for which Condition (5.5) was true when we traversed the edge entering it. We assume that the elements of X_1 are sorted by the time they are traversed by the Eulerian walk (i. e. we visited t_i before t_{i+1} for all $1 \leq i \leq k-1$). Let X be the set of roots of B at the end of the Eulerian walk as defined in the algorithm. By construction, for each $\mathfrak{x} \in X \backslash \{\mathfrak{s}'\}$ it holds that $p(\mathfrak{x}) = p(t_i)$ $(i \in \{1, \ldots, k\})$, where t_i is the unique sink from X_1 in the connected component of B rooted at $\mathfrak{x}$. Hence,

$$\text{length}(T_{\text{toplvl}}) = \sum_{\mathfrak{x} \in X} \text{dist}(s, \mathfrak{x}) \leq \text{dist}(s, \mathfrak{s}') + \sum_{i=1}^{k} \text{dist}(s, t_i).$$

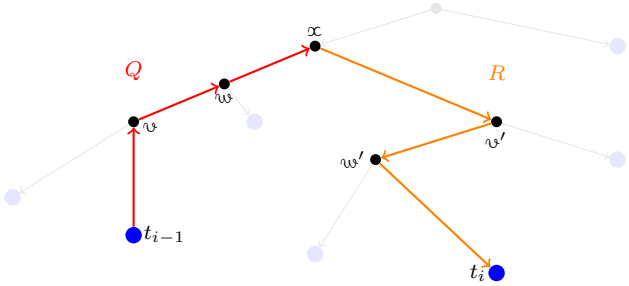


Figure 5.5: t_{i-1}-t_i sub-tour in the length bound proof of Theorem 5.8 for the case $t_{i-1} \neq s$.

In the remainder of the proof we show that

$$\sum_{i=1}^{k} \operatorname{dist}(s, t_i) < \frac{2}{\epsilon} \cdot \operatorname{length}(T_{\text{init}}) + \frac{4b \cdot (|N| - 1)}{\epsilon}.$$

Define $t_0 := s$ and $d(s) := 0$ at any time of the algorithm. Consider the time when we visit a sink $t_i \in X_1$ ($i \in \{1, \ldots, k\}$) in the Eulerian walk.
Let P_i be the t_{i-1}-t_i path in $\overleftrightarrow{T_{\text{init}}}$ and let $\mathfrak{x} \in V(P_i)$ such that P_i is the union of a t_{i-1}-$\mathfrak{x}$ path Q consisting of backward-edges and an $\mathfrak{x}$-t_i path R consisting of forward-edges only (see Figure 5.5). If $i = 1$, $\mathfrak{x} := s$ and Q is the trivial path with $E(Q) = \emptyset$. Let

- d_1 denote the function d at the time right before traversing the first edge of Q
- d_2 denote the function d at the time right before traversing the first edge of R
- d_3 denote the function d at the time when we check Condition (5.5) for t_i.

Due to Condition (5.6) it holds that

$$d_2(\mathfrak{w}) \leq d_1(\mathfrak{v}) + b + \operatorname{dist}(\mathfrak{v}, \mathfrak{w}) \text{ for all } (\mathfrak{v}, \mathfrak{w}) \in E(Q).$$

As we do not delete edges of R while traversing the t_{i-1}-t_i sub-part of the Eulerian walk,

$$d_3(\mathfrak{w}') = d_2(\mathfrak{v}') + b + \operatorname{dist}(\mathfrak{v}', \mathfrak{w}') = d_1(\mathfrak{v}') + b + \operatorname{dist}(\mathfrak{v}', \mathfrak{w}') \text{ for all } (\mathfrak{v}', \mathfrak{w}') \in E(R).$$

Consequently,

$$d_3(t_i) \leq d_1(t_{i-1}) + |E(P_i)| \cdot b + \operatorname{length}(P_i) = \operatorname{rat}(t_{i-1}) + |E(P_i)| \cdot b + \operatorname{length}(P_i).$$

The last equality follows from the equality $d_1(t_{i-1}) = \operatorname{rat}(t_{i-1})$ which is trivial for $i = 1$ and is true by construction and definition of d_1 for $i > 1$. By choice of t_i as a sink for which Condition (5.5) fails to hold, $(1 + \epsilon) \cdot \operatorname{rat}(t_i) < d_3(t_i)$ and hence,

$$(1 + \epsilon) \cdot \operatorname{rat}(t_i) < \operatorname{rat}(t_{i-1}) + |E(P_i)| \cdot b + \operatorname{length}(P_i).$$

Summing up over all $i = 1, \ldots, k$ yields

$$(1 + \epsilon) \cdot \sum_{i=1}^{k} \operatorname{rat}(t_i) < \sum_{i=0}^{k-1} \operatorname{rat}(t_i) + \sum_{i=1}^{k} (\operatorname{length}(P_i) + b \cdot |E(P_i)|).$$

Since the P_i are pairwise disjoint parts of the Eulerian walk through $\overleftrightarrow{T_{\text{init}}}$, $\sum_{i=1}^{k} \operatorname{length}(P_i) \leq 2 \cdot \operatorname{length}(T_{\text{init}})$ and $\sum_{i=1}^{k} |E(P_i)| \leq 2 \cdot |E(T_{\text{init}})| = 4(|N| - 1) - 2$. Combination of these inequalities (and the observation $\operatorname{dist}(s, t_i) \leq \operatorname{rat}(t_i)$ for all $\{i \in 1, \ldots, k\}$) concludes the proof:

$$\sum_{i=1}^{k} \operatorname{dist}(s, t_i) < \frac{2 \cdot \operatorname{length}(T_{\text{init}})}{\epsilon} + \frac{4b \cdot (|N| - 1)}{\epsilon}.$$

□

5.4 Shallow-Light Topologies with Criticalities

The bicriteria-approximation algorithm can be used to approximate the *Shallow-Light Topology Problem with Criticalities* as we show in this section.

An easy observation is that the Huffman Coding algorithm (Algorithm 8, Variant b)) is a 2-approximation in the case that $\lambda(t) \geq 1$ for all $t \in N \backslash \{s\}$. In the general case we can use the bicriteria algorithm to approximate the *Shallow-Light Topology Problem with Criticalities* as the next theorem shows.

Theorem 5.9 *Let N be a net with source s, positions $p : N \to M$ inside a metric space (M, dist), and sink criticalities $\lambda(t) \in \mathbb{R}_{\geq 0}$ for $t \in N \backslash \{s\}$. Let $b \geq 0$ be a bifurcation delay penalty and let $\beta \geq 1$ such that an approximation algorithm $\mathcal{A}$ for the Shortest Steiner Tree Problem with approximation guarantee β exists.*

There is an algorithm for the Shallow-Light Topology Problem with Criticalities that computes a placed topology (T, p) such that

$$\begin{aligned}
&\mathrm{length}(T,p) + \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \mathrm{delay}_{(T,p)}(s,t) \\
&\leq (1+\epsilon') \left(\mathrm{length}(T^*,p) + \sum_{t \in N \backslash \{s\}} \lambda(t) \mathrm{delay}_{(T^*,p)}(s,t) \right) + 2b \left(\sum_{t \in T} \lambda(t) + \tfrac{2}{\epsilon'}(|N|-1) \right),
\end{aligned}$$

where T^* is an optimum solution and ϵ' is the unique solution to $\left(1 + \frac{2}{\epsilon'}\right) \cdot \beta = (1+\epsilon')$.

If ψ is the time needed to run $\mathcal{A}$ on instance $N, (M, \mathrm{dist})$ and to query $\mathrm{dist}(s,t)$ for all $t \in N \backslash \{s\}$, the running time of the algorithm is $\mathcal{O}\left(|N| \log(|N|) + \psi\right)$.

Proof We use algorithm $\mathcal{A}$ (plus local transformations to ensure degree constraints) to compute a topology T_{init} such that $\mathrm{length}(T_{\mathrm{init}}) \leq \beta \cdot \mathrm{length}(T^*)$. We use the Huffman Coding Algorithm 8 to compute an optimum solution T_{toplvl} to Variant b) as defined in Section 5.1.2. In particular,

$$\sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \mathrm{delay}_{T_{\mathrm{toplvl}}}(s,t) \leq \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \mathrm{delay}_{T^*}(s,t).$$

Together with required arrival times $\mathrm{rat}(t) := \mathrm{delay}_{T_{\mathrm{toplvl}}}(s,t)$ for $t \in N \backslash \{s\}$ we obtain a feasible instance for the *Shallow-Light Topology Problem with Required Arrival Times*. Let T be the output of the bicriteria algorithm of Theorem 5.8 with initial topology T_{init} and $\epsilon := \epsilon'$. It holds that

$$\begin{aligned}
&\mathrm{length}(T,p) + \sum_{t \in N \backslash \{s\}} \lambda(t) \mathrm{delay}_{(T,p)}(s,t) \\
&\leq \left(1 + \frac{2}{\epsilon'}\right) \mathrm{length}(T_{\mathrm{init}}) + \frac{4b \cdot (|N|-1)}{\epsilon'} + \sum_{t \in N \backslash \{s\}} \lambda(t) \left((1+\epsilon') \cdot \mathrm{delay}_{T_{\mathrm{toplvl}}}(s,t) + 2b \right) \\
&= (1+\epsilon') \left(\frac{\mathrm{length}(T_{\mathrm{init}})}{\beta} + \sum_{t \in N \backslash \{s\}} \lambda(t) \mathrm{delay}_{T_{\mathrm{toplvl}}}(s,t) \right) + 2b \left(\frac{2}{\epsilon'}(|N|-1) + \sum_{t \in N \backslash \{s\}} \lambda(t) \right) \\
&\leq (1+\epsilon') \left(\mathrm{length}(T^*) + \sum_{t \in N \backslash \{s\}} \lambda(t) \mathrm{delay}_{T^*}(s,t) \right) + 2b \cdot \left(\sum_{t \in T} \lambda(t) + \frac{2}{\epsilon'} \cdot (|N|-1) \right).
\end{aligned}$$

The claim about the running time is clear. □

Corollary 5.10 *There are algorithms for the Shallow-Light Topology Problem with Criticalities with absolute error*

$$2b \cdot \left(\sum_{t \in T} \lambda(t) + \tfrac{2}{\epsilon'} \cdot (|N| - 1) \right)$$

and running time, relative error, and restriction according to the following table:

	running time	***relative error***	***restriction***
1.	$\mathcal{O}(\lvert E(G)\rvert + \lvert V(G)\rvert \cdot \log(\lvert V(G)\rvert))$	3.57	(M, dist) = *metric cl. of graph* G
2.	$\mathcal{O}(\lvert N\rvert \cdot \log(\lvert N\rvert))$	2.23	$(M, \mathrm{dist}) = (\mathbb{R}^2, \ell_1)$
3.	*"polynomial"*	1.88	(M, dist) = *metric cl. of graph* G
4.	*"polynomial"*	1.42	(M, dist) *is a Minkowski metric*

Proof For 1.–4. we select the following algorithms for $\mathcal{A}$ and apply Theorem 5.9.

- For 1. we use Mehlhorn's Algorithm [Meh88] which is a 2-approximation for the *shortest Steiner tree problem in graphs* and has running time $\mathcal{O}(|E(G)| + |V(G)| \log(|V(G)|))$.
- For 2. we compute a minimum spanning tree in the Delaunay triangulation (see [HS75]) which is a $\frac{3}{2}$-approximation by the result of Hwang [Hwa76]. Since the Delaunay triangulation can be computed in $\mathcal{O}(|N| \cdot \log(|N|))$ time and has $\mathcal{O}(|N|)$ edges, we obtain a total running time of $\mathcal{O}(|N| \cdot \log(|N|))$.
- For 3. we use the algorithm by Byrka et al. [Byr+13] which is a 1.39 approximation.
- For 4. we use the approximation scheme by Arora [Aro98] with $\epsilon = 0.004$.

□

5.5 Topology Optimization

The theoretical length bound is not very strong. A non-optimized implementation of the bicriteria algorithm of Theorem 5.8 would yield results far worse than the topologies computed by the algorithm of Bartoschek et al. [Bar+10].

In this section we show how to improve the results of the bicriteria algorithm without loosing the theoretical bounds. Although some optimizations can equally be applied to general metric spaces, we restrict to the case $(M, \mathrm{dist}) = (\mathbb{R}^2, \ell_1)$. The results of this section are joint work with Stephan Held.

5.5.1 Placement of Steiner Points in Delay-Optimum Solutions

Finding new positions for all Steiner nodes of a given topology such that delays along the source-sink paths are not increased and the total topology length is minimized is possible in polynomial time since the problem can be formulated as linear program (see e. g. Rockel [Roc16]). Maßberg [Maß15] gave a polynomial time combinatorial algorithm that combines dynamic programming with binary search to find such positions. Rockel [Roc16] observed that the linear program is the dual of a *Min-Cost-Flow problem* and obtained a second polynomial time combinatorial algorithm.

The most obvious reason why the output of the bicriteria algorithm is too long is the placement of Steiner points in the top-level topology. According to the following calculation

that holds for all topologies in which all source-sink paths are shortest paths, net length becomes smaller the larger the distances between the source and the Steiner points are:

$$\begin{aligned}\text{length}(T_{\text{toplvl}}) &= \sum_{(\mathfrak{v},\mathfrak{w})\in E(T_{\text{toplvl}})} (\text{dist}(p(s),p(\mathfrak{w})) - \text{dist}(p(s),p(\mathfrak{v})))\\ &= \sum_{\mathfrak{v}\in V(T_{\text{toplvl}})} (|\delta^-_{T_{\text{toplvl}}}(\mathfrak{v})| - |\delta^+_{T_{\text{toplvl}}}(\mathfrak{v})|) \cdot \text{dist}(p(s),p(\mathfrak{v}))\\ &= \sum_{t\in N\setminus\{s\}} \text{dist}(p(s),p(t)) - \sum_{\mathfrak{v}\in V(T_{\text{toplvl}})\setminus N} \text{dist}(p(s),p(\mathfrak{v})).\end{aligned}$$

In this sense, choosing position $p(s)$ for each Steiner point is the worst decision we can make with respect to length.

Instead, when nodes $\mathfrak{v}$ and $\mathfrak{w}$ are selected in (3) in Algorithm 8, we need to find the point that is both on a shortest $p(s)$-$p(\mathfrak{v})$ and on a shortest $p(s)$-$p(\mathfrak{w})$ path and has maximum distance to $p(s)$. For $(M,\text{dist}) = (\mathbb{R}^2,\ell_1)$, this point is exactly the median of $p(s)$, $p(\mathfrak{v})$, and $p(\mathfrak{w})$ (see Definition 5.7).

If the set of nodes from which we can choose in Step (3) is large, we can furthermore select $\mathfrak{v}$, $\mathfrak{w}$ close to each other, or such that the median of $p(s)$, $p(\mathfrak{v})$, and $p(\mathfrak{w})$ is far away from the source. The next lemma tells us how this can be accomplished if all sinks have a sufficiently large bif-value.

Lemma 5.11 *Let N be a net with source s and positions $p : N \to \mathbb{R}^2$, and let* bif $: N\setminus\{s\} \to \mathbb{N}$ *such that* $\text{bif}(t) = H := \lceil \log(|N|-1)\rceil$ *for all $t \in N\setminus\{s\}$. Let T^* be a (short) topology for N with respect to ℓ_1-distances. Then, we can compute a placed topology (T,p) with length at most $H \cdot \text{length}(T^*)$ and $\text{delay}_{(T,p)}(s,t) \le \text{rat}(t)$ for all $t \in N\setminus\{s\}$ in time $\mathcal{O}(|N|\log(|N|))$.*

The proof of this lemma is inspired by the results of Rao et al. [Rao+92].

Proof Let C be a Hamiltonian cycle through $N\setminus\{s\}$ with

$$\text{length}(C) := \sum_{\{v,w\}\in E(C)} ||p(v)-p(w)||_1 \le 2\cdot\text{length}(T^*).$$

Cycle C can be obtained by the double-tree algorithm starting with T^* in $\mathcal{O}(|C|)$ time. We use C to compute a matching M covering $2\cdot\left\lfloor\frac{|N|-1}{2}\right\rfloor$ sinks with

$$\text{length}(M) := \sum_{\{v,w\}\in E(M)} ||p(v)-p(w)||_1 \le \frac{1}{2}\cdot\text{length}(C) \le \text{length}(T^*).$$

Matching M is obtained in $\mathcal{O}(|N|)$ time by taking every second edge in C (for details see Rao et al. [Rao+92]). Choosing nodes $\mathfrak{v}$, $\mathfrak{w}$ for which $\{\mathfrak{v},\mathfrak{w}\} \in M$ in the first $|M|$ iterations of the Huffman Coding algorithm (Algorithm 8 Variant a)) is a valid choice. We place all Steiner vertices at the median of $p(s)$ and the positions of the selected vertices.

Let X' and Y' be the sets X and Y of the Huffman Coding algorithm after the first $|M|$ iterations respectively. $X' \cup Y'$ consists of the newly created Steiner vertices and at most one initial sink.

By choice of the Steiner points' positions, C can be considered a Hamiltonian cycle through $X' \cup Y'$ and by short-cutting we transform C into a cycle C' through $X' \cup Y'$ with

$$\text{length}(C') \leq \text{length}(C) \leq 2 \cdot \text{length}(T^*).$$

If $X' \cup Y'$ contains an initial sink (which is the case if and only if this set of initial sinks has odd cardinality), we decrease its bif-value from H to $H-1$. As seen in the proof of Lemma 5.4, this does not violate feasibility of the resulting instance. We decrease H by 1 and iterate the same procedure until there is only one sink left that we directly connect with s.

It is clear that the delay along the s-t path in the computed topology is at most $\text{rat}(t)$ for all $t \in N \backslash \{s\}$.

In each of the first $H-1$ iterations we produce edges with total cost at most $\text{length}(T^*)$. After the $(H-1)$-st iteration we have at most 2 elements left in $X \cup Y$ that are both placed within the bounding box of N. Thus, the length of the edges added in the last iteration plus the length of the edge connecting the final sink to s is upper-bounded by the length of the bounding box of N which is at most $\text{length}(T^*)$. We obtain the claim about the cost bound.

Let C_i be the cycle C in iteration i. After initially sorting all sinks by their bif-value, the running time of the i-th iteration is proportional to $|C_i|$. Equation $\sum_{i=1}^{H+1} |C_i| = \mathcal{O}(|N|)$ (by Lemma 2.7) yields the claimed running time. □

The algorithm of the proof of Lemma 5.11 can be generalized to general bif-values with $\sum_{t \in N \backslash \{s\}} 2^{-\text{bif}(t)} \leq 1$. For $H = \max\{\text{bif}(t) : t \in N \backslash \{s\}\}$ to 1 we compute a short maximum matching covering the sinks with bif-value $\geq H$ and compute a topology based on these matchings as in Lemma 5.11. The length of the computed topology will be smaller the smaller the number of iterations $\max\{\text{bif}(t) : t \in N \backslash \{s\}\}$ is. Counterintuively, we obtain larger lengths for uncritical instances. In fact, many instances contain uncritical sinks for which the original bif-value according to Formula 5.1 is larger than necessary. Lemma 5.12 shows how to efficiently decrease the bif-value of uncritical sinks such that we still have an instance for which a topology without delay violations exists.

Lemma 5.12 *Let N be a net with source s and let $\text{bif} : N \backslash \{s\} \to \mathbb{N}$ such that $\sum_{t \in N \backslash \{s\}} 2^{-\text{bif}(t)} \leq 1$. We can find $H \in \mathbb{N}$ minimum such that $\sum_{t \in N \backslash \{s\}} 2^{-\min\{\text{bif}(t), H\}} \leq 1$ in time $\mathcal{O}(|N| \log(|N|))$.*

Proof Initially, we sort the sinks by their bif-value. For all $H \in \mathbb{N}$ this yields a sorting with respect to values $\min\{\text{bif}(t), H\}$ as well.

We find the minimum value for H by binary search in the interval $[\lfloor \log(|N|) \rfloor, |N| - 2]$. For a candidate value of H we run Variant a) of the Huffman Coding Algorithm 8 with bif-values $\min\{\text{bif}(t), H\}$. By Lemma 5.2, one call of the algorithm needs time $\mathcal{O}(|N|)$ (since the input is already sorted). By Lemmas 5.3 and 5.4, the algorithm finds a topology T for which $|E(T_{[s,t]})| - 1 \leq \min\{\text{bif}(t), H\}$ for each $t \in N \backslash \{s\}$ if and only if $\sum_{t \in N \backslash \{s\}} 2^{-\min\{\text{bif}(t), H\}} \leq 1$. □

In the case that $H = \lceil \log(|X|) \rceil$ holds for the top-level instance $X \cup \{s\}$ in the bicriteria algorithm of Theorem 5.8 after application of Lemma 5.12, computing top-level topology T_{toplvl} by the algorithm of Lemma 5.11 can significantly improve the theoretical and practical length bound.

The following result is based on Lemma 3.2 of Elkin and Solomon [ES15] and appeared in [HR13] (Theorem 3).

Theorem 5.13 *Let N be a net with source s and let* $\mathrm{bif} : N\backslash\{s\} \to \mathbb{N}$ *such that* $\mathrm{bif}(t) = H := \lceil \log(|N|-1) \rceil$ *for $t \in N\backslash\{s\}$. Let T^* be any (short) topology for N with respect to ℓ_1-distances and let $\epsilon > 0$ such that*

$$\sum_{t \in N\backslash\{s\}} ||p(t) - p(s)||_1 \leq \tfrac{2}{\epsilon} \cdot \mathrm{length}(T^*).$$

Then, the placed topology (T, p) computed with Lemma 5.11 has length at most

$$\left(1 + \left\lceil \log\left(\frac{2}{\epsilon}\right) \right\rceil\right) \cdot \mathrm{length}(T*).$$

Proof We may assume that $|N|-1$ is a power of 2 as otherwise, we can insert $2^H - (|N|-1)$ additional sinks with bif-value H at source position. For $i = 1, \ldots, H+1$ we denote the set of edges produced in the i-th iteration by E_i (E_{H+1} consists of the edge outgoing of s in T only). Note that the number of sinks reachable from the endpoint of an edge in E_i is 2^{i-1}. As the statement follows directly from Lemma 5.11 if $\lceil \log\left(\frac{2}{\epsilon}\right) \rceil + 1 \geq H$, we may assume that $\lceil \log\left(\frac{2}{\epsilon}\right) \rceil \leq H - 2$. It holds that

$$\begin{aligned}
\sum_{t \in N\backslash\{s\}} ||p(s) - p(t)||_1 &= \sum_{i=1}^{H+1} \left(\sum_{(\mathfrak{v},\mathfrak{w}) \in E_i} |\{t \in N\backslash\{s\} : \mathfrak{w} \in V(T_{[s,t]})\}| \cdot ||p(\mathfrak{v}) - p(\mathfrak{w})||_1 \right) \\
&= \sum_{i=1}^{H+1} \left(2^{i-1} \cdot \sum_{(\mathfrak{v},\mathfrak{w}) \in E_i} ||p(\mathfrak{v}) - p(\mathfrak{w})||_1 \right) \\
&\geq \frac{2}{\epsilon} \cdot \sum_{i=\left\lceil \log\left(\frac{2}{\epsilon}\right) \right\rceil + 1}^{H+1} \left(\sum_{(\mathfrak{v},\mathfrak{w}) \in E_i} ||p(\mathfrak{v}) - p(\mathfrak{w})||_1 \right).
\end{aligned}$$

Thus,

$$\mathrm{length}(T^*) \geq \frac{\epsilon}{2} \sum_{t \in N\backslash\{s\}} ||p(t) - p(s)||_1 \geq \sum_{i=\left\lceil \log\left(\frac{2}{\epsilon}\right) \right\rceil + 1}^{H+1} \left(\sum_{(\mathfrak{v},\mathfrak{w}) \in E_i} ||p(\mathfrak{v}) - p(\mathfrak{w})||_1 \right)$$

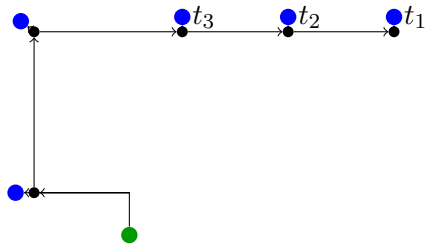
and

$$\mathrm{length}(T) = \sum_{i=1}^{H+1} \left(\sum_{(\mathfrak{v},\mathfrak{w}) \in E_i} ||p(\mathfrak{v}) - p(\mathfrak{w})||_1 \right) \leq \left(\left\lceil \log\left(\frac{2}{\epsilon}\right) \right\rceil + 1 \right) \cdot \mathrm{length}(T^*).$$

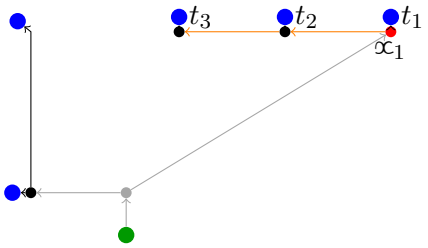
□

Note that the condition $H = \lceil \log(|N|-1) \rceil$ is always fulfilled in the case $b = 0$ and after applying Lemma 5.11. Using Theorem 5.13 for building top-level topology T_{toplvl} in Theorem 5.8 improves the length bound to

$$\left(2 + \left\lceil \log\left(\frac{2}{\epsilon}\right) \right\rceil\right) \cdot \mathrm{length}(T_{\mathrm{init}}) \quad \text{if } 0 < \epsilon \leq 2.$$



(a) Initial topology. Sink t_1 is supposed to be critical while all other sinks are uncritical.



(b) Branching (black and orange) and top-level topology (gray). Branching root $\mathfrak{x}_1$ is far away from s.

Figure 5.6: Instance for which placement of Steiner points results in large topology length. Here, we assume that a forward violation has been detected at t_1 and backward violations at the orange edges.

5.5.2 Changing Component Layout and Steiner Point Positions

In this section we concentrate on the placement of Steiner points inserted during the back-connect step performed after finding out that Equation (5.5) in the bicriteria algorithm of Theorem 5.8 is not fulfilled. Recall that during that step we place Steiner points at the position of the new edge's tail. This can result in poor results as Figure 5.6 shows.

These optimizations avoid the problems:

Optimization 5.14 (Rearrange Steiner points) *Traverse a branching in reverse topological order. When we visit a Steiner point $\mathfrak{v}$ that is not a root, we set $p(\mathfrak{v})$ to the median of its two children and its parent. If $\mathfrak{v}$ is a root, we set $p(\mathfrak{v})$ to the median of its children and s.*

Optimization 5.14 does neither increase the length nor decrease the worst slack of a branching B and can be performed in running time linear to the size of B.

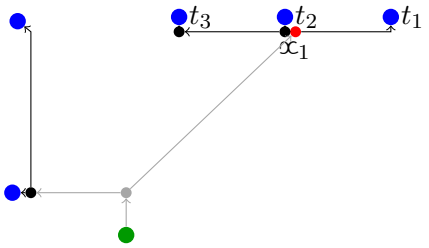
Optimization 5.15 (Change component layout) *Let $\mathfrak{x}$ be a root of a non-trivial connected component H of the branching B built by the algorithm. We replace all directed edges by undirected ones, delete $\mathfrak{x}$ but add an edge between its two children.*
We sort all edges $\{\mathfrak{v}, \mathfrak{w}\}$ by the distance between s and the median of $s, p(\mathfrak{v})$, and $p(\mathfrak{w})$ in non-decreasing order.
For an edge $\mathfrak{e} = \{\mathfrak{v}, \mathfrak{w}\}$, we subdivide $\mathfrak{e}$ by a Steiner point $\mathfrak{v}'$ placed on the median of $p(\mathfrak{v})$, $p(\mathfrak{w})$, $p(s)$ and direct all edges away from $\mathfrak{v}'$. We compute a required arrival time of $\mathfrak{v}'$ by propagating rat*-values of all sinks in B to $\mathfrak{v}'$ in reverse topological order.*
By Huffman Coding we can check if a top-level topology without delay violations for the resulting instance exists. If this is the case, we keep the changes to B; otherwise we revert the transformations and continue with the next edge.

Figure 5.7 illustrates the two optimizations.

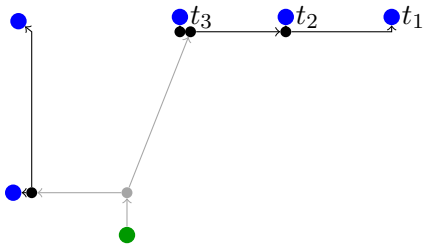
Theorem 5.16 *Applying Optimization 5.15 before building the top-level topology T_{toplvl} in the bicriteria algorithm of Theorem 5.8 does not violate Properties (5.3) and (5.4) of Theorem 5.8. The running time of applying Optimization 5.15 to one component H is*

$$\mathcal{O}(k \log k + |E(H)| \cdot (|E(H)| + k)),$$

where k is the number of connected components.



(a) Result after applying Optimization 5.14.



(b) Possible result after applying Optimization 5.15: We check if a feasible top-level topology exists after creating connection points nearer to the source by subdividing e. g. the left horizontal edge.

Figure 5.7: Effect of Optimization 5.14 and 5.15 on the instance of Figure 5.6.

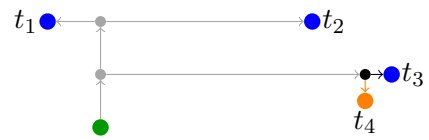
Proof First, note that in the iteration in which we select the edge connecting the former children of $\mathfrak{c}$, the achievable worst slack is at least as large as stated by Properties (5.3). We conclude that $\sum_{\mathfrak{v} \text{ root of } B} \operatorname{dist}(p(s), p(\mathfrak{v}))$ is not increased by Optimization 5.15 which implies Property (5.4). Property (5.3) is clear since we apply Huffman Coding in each iteration.
To achieve the claimed running time, we first sort the set of connected components by their bif-value in non-decreasing order. Transformation of the component and rat-propagation in each iteration takes $\mathcal{O}(|E(H)|)$ time while restoring the ordering of the set of branching roots as well as the application of Huffman-Coding takes $\mathcal{O}(k)$ time. □

5.5.3 Optimization with Greedy

Right before the computation of the top-level topology T_{toplvl} it is possible that two connected components with very different bif′-value are located close together. It is unlikely that these two components are joined during Huffman Coding although this would result in small length. In Figure 5.8, the black Steiner point is supposed to have a small bif′-value since it has a critical successor t_4. Sink t_2 is supposed to be uncritical and hence, has a larger bif′-value. Joining t_2 and the black node would result in a short topology although the Huffman Coding algorithm would probably not do that.

One idea is to check if we can include the non-critical component into the critical one by subdividing a non-critical edge of the critical component.
We will describe a procedure similar to the greedy algorithm by Bartoschek et al. [Bar+10] to merge components.

(a) Branching B. Sink t_4 is supposed to be critical while all other sinks are uncritical.



(b) Branching (black and orange) with and top-level topology (gray).

Figure 5.8: Branching B for which Huffman Coding finds a long top-level topology.

Optimization 5.17 (Optimization with greedy) *Let $C_1, \dots, C_k$ be the set of connected components (different from $\{s\}$) of a branching B and let $x_1, \dots, x_k$ be the corresponding roots. Initially, B is the branching in the bicriteria algorithm of Theorem 5.8 before building T_{toplvl}.*

We select a connected component C_i that we have not selected in any previous iteration and try to merge it into another component. We consider two different ways of merging components:

(i) We insert C_i into a different component C_j by subdividing an edge $e = (v, w) \in E(C_j)$ by a Steiner point u placed at the median of $p(v), p(w), p(x_i)$, and adding edge (u, x_i). See Figure 5.9(a) for a visualization.

(ii) We merge C_i and another component C_j by connecting x_i and x_j with a new Steiner point u placed at the median of $p(s), p(x_i), p(x_j)$. New edges (u, x_i) and (u, x_j) connect the new root u with the former roots x_i and x_j. An example of this operation is shown in Figure 5.9(b).

For all possible branchings $\widetilde{B}$ obtained by Operations (i) and (ii) we compute the maximal achievable worst slack $wsl(\widetilde{B})$ of a top-level topology connecting $\widetilde{B}$ by

- *computing required arrival times and* bif*-values of the roots of components of $\widetilde{B}$ by backward propagation of required arrival times at sinks, and by*
- *computing arrival times of the roots of components of $\widetilde{B}$ by applying Huffman-Coding to the set of these roots with the newly computed* bif*-values.*

Among all candidate branchings $\widetilde{B}$ with

$$wsl(\widetilde{B}) \geq 0 \quad \text{and} \tag{5.7}$$

$$\text{length}(\widetilde{B}) + \sum_{x \text{ root of } \widetilde{B}} ||p(s) - p(x)||_1 \leq \text{length}(B) + \sum_{x \text{ root of } B} ||p(s) - p(x)||_1 \tag{5.8}$$

we select the one minimizing $(1 - \xi) \cdot \text{length}(\widetilde{B}) - \xi \cdot wsl(\widetilde{B})$ for a trade-off parameter $\xi \in [0, 1]$.

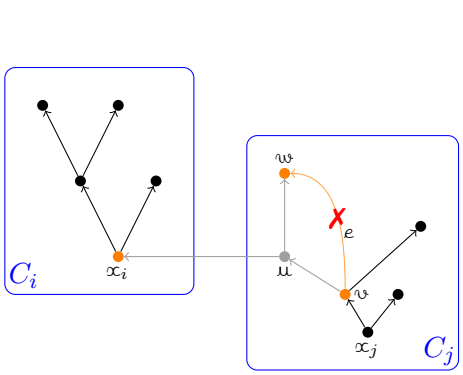
If such a branching $\widetilde{B}$ exists, we set $B := \widetilde{B}$ and iterate.

Figure 5.10 shows the effect of Optimization 5.17 on the instance shown in Figure 5.8. Note that the algorithm of Bartoschek et al. [Bar+10] is equivalent to Optimization 5.17 if we start with the initial branching $(N, \emptyset)$ consisting of singletons only, select components C_i according to the bif-values of their roots in non-decreasing order, and if we omit Conditions (5.7) and (5.8).

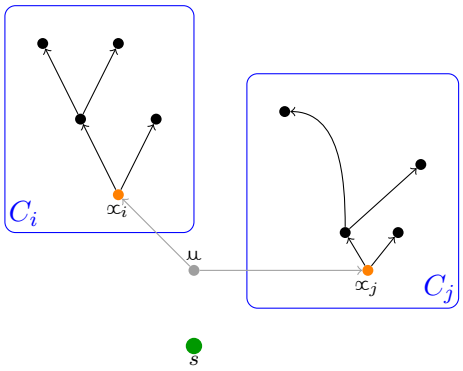
Theorem 5.18 *Applying Optimization 5.17 before building the top-level topology T_{toplvl} in the bicriteria algorithm of Theorem 5.8 does not violate Properties (5.3) and (5.4) of Theorem 5.8. The running time is $\mathcal{O}(k^2 \cdot (|E(B)| + k))$ where k is the number of connected components of the initial branching.*

Proof Since Conditions (5.7) and (5.8) guarantee that Properties (5.3) and (5.4) are preserved respectively, we only have to show how to obtain the claimed running time.

At the beginning of the algorithm we compute worst slack and bif-value of each root in the initial branching. Furthermore, we sort the roots by their bif-value in non-decreasing order. During the algorithm we keep wsl and bif values up-to-date and preserve the sorting.



(a) New branching resulting from subdivision of an edge e by a Steiner node $\mathfrak{u}$ that is connected to the root $\mathfrak{x}_i$ of another component.



(b) New branching resulting from insertion of a new Steiner node $\mathfrak{u}$ that is connected with two roots $\mathfrak{x}_i$ and $\mathfrak{x}_j$.

Figure 5.9: Examples of new branching candidates in Figure 5.8. For all candidates, components C_i and C_j are merged.

Let i be a fixed iteration index. Let B_i be the branching B at the beginning of that iteration and let k_i be the number of components of B_i. We try to merge component $C_{i'}$ of B_i with another component in iteration i. Let $e = (\mathfrak{v}, \mathfrak{w})$ be an edge of a component $C_j \neq C_{i'}$. If we subdivide e as described in (i), the new worst slack of $\mathfrak{x}_j$ will be the minimum of the old worst slack of $\mathfrak{x}_j$ and $\text{wsl}(\mathfrak{w}) - b$. If we merge $C_{i'}$ with another component C_j by inserting a new Steiner node $\mathfrak{u}$ connected with $\mathfrak{x}_i$ and $\mathfrak{x}_j$ as described in (ii), the worst slack of the new root $\mathfrak{u}$ is

$$\min\{\text{wsl}(\mathfrak{x}_i) - ||p(\mathfrak{u}) - p(\mathfrak{x}_i)||_1, \text{wsl}(\mathfrak{x}_j) - ||p(\mathfrak{u}) - p(\mathfrak{x}_j)||_1\} - b.$$

Clearly, determining $\text{length}(\widetilde{B})$ and $\sum_{\mathfrak{x} \text{ root of } \widetilde{B}} ||p(s) - p(\mathfrak{x})||_1$ for a candidate branching $\widetilde{B}$ can be done in constant time given these values for B_i.

Hence, checking if a candidate branching $\widetilde{B}$ satisfies (5.7) and (5.8), and determining the value $(1-\xi) \cdot \text{length}(\widetilde{B}) - \xi \cdot \text{wsl}(\widetilde{B})$ can be done in time $\mathcal{O}(k_i)$ by Huffman Coding. In addition, the worst slack information of the finally selected branching B_{i+1} allow us to update bif-values in constant time and the sorting of roots in $\mathcal{O}(k_i)$ time. We also update worst slack information at every node of the selected candidate branching as they are needed to evaluate candidate branchings in the next iteration. This takes time $\mathcal{O}(|E(B_i)| + |V(B_i)|) = \mathcal{O}(|E(B_i)| + k_i)$.

Using the fact that the expression $|E(B_i)| + k_i$ is increased by at most 2 in each of the at

(a) Branching B after applying Optimization 5.17. The delay on the critical path to t_4 did not increase.

(b) Since we achieved that t_2 is no longer a sink, we can build a shorter top-level topology.

Figure 5.10: Effect of Optimization 5.17 on the instance of Figure 5.8(a).

most k_1 iterations we obtain a total running time of

$$\mathcal{O}(k_i \cdot (|E(B_i)| + k_i)) = \mathcal{O}(k_1 \cdot (|E(B_1)| + k_1))$$

for iteration i and the claimed running time follows. □

5.6 Experimental Results

In this section we show results of the bicriteria algorithm of Theorem 5.8 and the optimizations described in Section 5.5 on practical repeater tree instances. As initial topologies we computed approximately shortest topologies. For up to 9 terminals we use the FLUTE algorithm by Chu and Wong [CW08] and for larger nets we ran an optimized variant of Prim's algorithm [Pri57] on the *Delaunay triangulation* [HS75] that has an approximation guarantee equal to Steiner ratio $\frac{3}{2}$ (see [Hwa76]).

Our testbed consists of 584 592 repeater tree instances from 32 real-world designs in 14 nm and 22 nm technology. The designs are provided by our cooperation partner IBM. Each instance has at least 4 pins, i. e. we excluded trivial instances with up to 3 pins. The largest instance has 12 063 pins. We deactivated pre-clustering of sinks for large instances.

The bifurcation delay penalties b vary from $3.4\,ps$ to $5.3\,ps$. Depending on the design this corresponds to the delay along wire of length between $5.5\,\mu m$ and $11.5\,\mu m$. The chosen delay parameters correspond to the lowest available layer although long, critical connections would rather be routed on higher layers. By this choice of delay parameters, wire delays are rather large and detours can result in bad worst slacks easily.

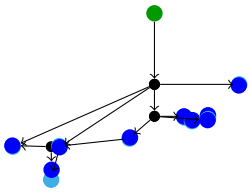
All tests were conducted on a machine with an Intel Xeon E5-2699 processor running at 2.20 GHZ.

5.6.1 Layout of Tables

Let $\mathcal{I}$ be the set of all 584 592 instances. For an algorithm $\mathcal{A}$ and $I \in \mathcal{I}$ let

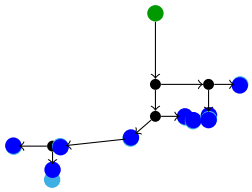
- $\text{length}(\mathcal{A}(I))$ be the length of the topology computed by algorithm $\mathcal{A}$ for instance I,
- $\text{delay}_{\mathcal{A}(I)}(t)$ be the delay from the source of I to sink t of I inside the topology computed by $\mathcal{A}$ for instance I,
- $\text{wsl}(\mathcal{A}(I)) := \min\{\text{rat}(t) - \text{delay}_{\mathcal{A}(I)}(t) : t \text{ sink in } I\}$ be the worst slack of the output of $\mathcal{A}$ on instance I, and
- $\text{sns}(\mathcal{A}(I)) := \sum_{t \text{ sink in } I} \min\left\{0, \text{rat}(t) - \text{delay}_{\mathcal{A}(I)}(t)\right\}$ be the sum of negative slacks of the output of $\mathcal{A}$ on instance I.

When comparing two algorithms $\mathcal{A}_{\text{main}}$ and $\mathcal{A}_{\text{ref}}$ we are interested in the **length ratios** $\frac{\text{length}(\mathcal{A}_{\text{main}}(I))}{\text{length}(\mathcal{A}_{\text{ref}}(I))}$ and the differences $\min\{0, \text{wsl}(\mathcal{A}_{\text{main}}(I))\} - \min\{0, \text{wsl}(\mathcal{A}_{\text{ref}}(I))\}$ and $\text{sns}(\mathcal{A}_{\text{main}}(I)) - \text{sns}(\mathcal{A}_{\text{ref}}(I))$ for $I \in \mathcal{I}$ (**worst slack difference** and **sum of negative slacks difference**). For these values we display maximum (**max**), minimum (**min**) and average (**av**) values over instance groups $\mathcal{I}' \subseteq \mathcal{I}$ containing instances of certain sizes. In addition, we report **total** length ratios $\frac{\sum_{I \in \mathcal{I}'} \text{length}(\mathcal{A}_{\text{main}}(I))}{\sum_{I \in \mathcal{I}'} \text{length}(\mathcal{A}_{\text{ref}}(I))}$ for these groups.



|$|N|$|: 19
Length: 168 μm
WSL: −63 ps

(a) Bicriteria for $\epsilon = 0.1$, without optimization.



|$|N|$|: 19
Length: 92 μm
WSL: −63 ps

(b) Bicriteria for $\epsilon = 0.1$, with optimization.

Figure 5.11: Instance on which optimization could reduce topology length without degrading the worst slack. For better visibility, very short edges are not drawn and some sink nodes are plotted in a different shade of blue.

As a rule of thumb we can say that $\mathcal{A}_{\text{main}}$ did a better job with respect to length if and only if numbers in the length ratio column of the tables in this section are smaller than 1. It did a better job with respect to worst slack and sum of negative slack if numbers in the respective slack difference column are larger than 0.

5.6.2 The Impact of Optimization

We ran the bicriteria algorithm with parameters $\epsilon = 0$, 0.1, 0.5, and 1.

In the main run ($\mathcal{A}_{\text{main}}$) we enabled the optimizations described in Section 5.5. We used the improved placement of Steiner points resulting from a better top-level topology (Section 5.5.1) and the optimization of component layouts (Section 5.5.2) for all instances. Since the optimization with greedy is more time consuming, we select different configurations based on the number k of connected components of the branching after the Eulerian walk. If $k \leq 100$, we proceed as described in Section 5.5.3. We use parameter $\xi := 1 - \epsilon$. If k is large, we cannot try all possible candidate branchings. If $101 \leq k \leq 1000$, we restrict to candidate branchings in which the selected component C_i is merged with the component containing the successor of the source in the initial topology. This component is usually the largest one as it contains the parts of T_{init} that have not been ripped-up during the Eulerian walk. To decrease the running time further, we avoid the application of Huffman Coding in each iteration if $101 \leq k \leq 1000$. Instead, when evaluating a candidate branching in which two components with roots α_i and α_j are replaced by a component with root μ, we replace Condition (5.7) by the stronger but computationally easier condition $2^{-\text{bif}(\mu)} \leq 2^{-\text{bif}(\alpha_i)} + 2^{-\text{bif}(\alpha_j)}$. If $k > 1000$, we do not optimize with greedy at all.

In the reference algorithm $\mathcal{A}_{\text{ref}}$ we disabled the optimization of component layouts and the optimization with greedy (Sections 5.5.2 and 5.5.3). The improved top-level topology generation from Section 5.5.1 remained active also for the reference run. With this setting, the reference algorithm coincides with the version of the bicriteria algorithm used in [HR13].

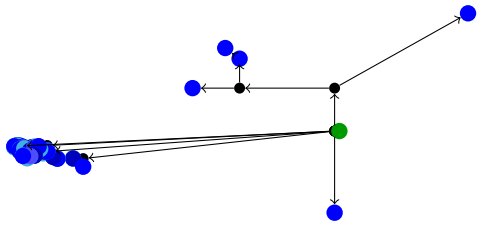
The results are shown in Table 5.1. Running times for the main algorithm range from 51 seconds for $\epsilon = 1$ to almost 2 minutes for $\epsilon = 0$. Roughly half of that time is spent in topology optimization. Computation of the initial topologies takes about 30 seconds. Taking into account that topology generation is a preparation step for the much slower buffer insertion, all these running times are negligible.

Average and total values in the *length ratio* columns of the table point out an overall positive effect of optimization on topology lengths. For the small values for ϵ (0 and 0.1), these improvements are substantial although delay bounds are very tight. With larger

$\lvert N\rvert$		wsl-diff	sns-diff	length	wsl-diff	sns-diff	length
# instances		[ps]	[ps]	ratio	[ps]	[ps]	ratio
		$\epsilon = 0$			$\epsilon = 0.1$		
≤ 10	max	10	546	1.71	31	110	1.62
	min	-10	-371	0.35	-87	-472	0.36
	av	0	0	0.99	1	1	0.99
# 468 838	total			0.97			0.98
$11 - 100$	max	10	1 600	1.89	39	3 290	1.73
	min	-10	-4 238	0.29	-103	-4 289	0.19
	av	0	-12	0.93	0	-10	0.88
# 112 216	total			0.90			0.86
> 100	max	10	1 974	1.95	28	24 930	1.62
	min	-5	-56 659	0.42	-65	-450 543	0.35
	av	0	-633	0.88	-2	-1 058	0.81
# 3 538	total			0.89			0.81
all	max	10	1 974	1.95	39	24 930	1.73
	min	10	-56 659	0.29	-103	-450 543	0.19
	av	0	-6	0.98	1	-7	0.96
# 584 592	total			0.94			0.91

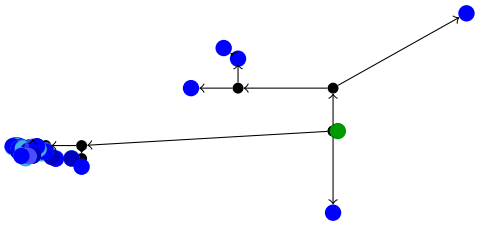
$\lvert N\rvert$		wsl-diff	sns-diff	length	wsl-diff	sns-diff	length
# instances		[ps]	[ps]	ratio	[ps]	[ps]	ratio
		$\epsilon = 0.5$			$\epsilon = 1$		
≤ 10	max	10	47	1.32	10	33	1.28
	min	-231	-524	0.36	-40	-154	0.53
	av	0	0	1.00	0	0	1.00
# 468 838	total			1.00			1.00
$11 - 100$	max	132	1 600	1.31	73	1 560	1.32
	min	-175	-6481	0.34	-220	-7 254	0.42
	av	-3	-56	0.93	-3	-65	0.96
# 112 216	total			0.92			0.96
> 100	max	86	21 696	1.18	78	8 966	1.46
	min	-285	-1 419 415	0.35	-286	-2 350 104	0.44
	av	-18	-2 688	0.82	-30	-3999	0.86
# 3 538	total			0.81			0.86
all	max	132	21 696	1.32	78	8 966	1.46
	min	-285	-1 419 415	0.34	-286	-2 350 104	0.42
	av	-1	-27	0.98	-1	-37	0.99
# 584 592	total			0.95			0.97

Table 5.1: Comparison between the bicriteria algorithm including all optimizations of Section 5.5 ($\mathcal{A}_{\text{main}}$) and the version of [HR13] that does not contain component layout optimization and optimization with greedy ($\mathcal{A}_{\text{ref}}$). We compare the results for two small values $\epsilon = 0, 0.1$ (on top) and for two larger values $\epsilon = 0.5, 1$ (at the bottom).
The columns entitled *length ratio* show the ratios $\frac{\text{length}(\mathcal{A}_{\text{main}})}{\text{length}(\mathcal{A}_{\text{ref}})}$. Numbers below 1 in these columns show that optimization decreases topology lengths. The wsl-diff columns show the differences $\min\{0, \text{wsl}(\mathcal{A}_{\text{main}})\} - \min\{0, \text{wsl}(\mathcal{A}_{\text{ref}})\}$ while the sns-diff columns display $\text{sns}(\mathcal{A}_{\text{main}}) - \text{sns}(\mathcal{A}_{\text{ref}})$. Negative values in the columns wsl-diff and sns-diff show that optimization degrades timing. A more detailed description of how the table is arranged can be found in Section 5.6.1.



$\|N\|$:	70
Length:	1 994 μm
WSL:	−74 ps

(a) Bicriteria algorithm for $\epsilon = 0.1$, without optimization.



$\|N\|$:	70
Length:	1 013 μm
WSL:	−74 ps

(b) Bicriteria algorithm for $\epsilon = 0.1$, with optimization.

Figure 5.12: Instance on which optimization could reduce topology lengths without degrading the worst slack.

values for ϵ (0.5 and 1), the initial topology is feasible for many instances and the effect of optimization is smaller.

On some instances optimization increased the length of the returned solution. The reason for this effect is that we run the faster non-optimized version of Huffman coding to evaluate worst slacks of branching candidates and the improved version of Huffman coding from Section 5.5.1 could find a longer solution for the new branching than for the old one. One possibility to overcome this would be to always use the optimized version and reject solutions that increase topology lengths. However, this situation occurs rarely and we prefer to take the runtime benefit from using the fast Huffman coding.

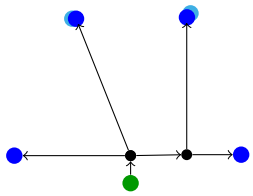
The overall improvement with respect to length goes on cost of a degraded timing. With average degradations between 0 and 1 ps the degradations of the average worst slack are tiny. For sinks t with large required arrival times the bound $(1 + \epsilon) \cdot \text{rat}(t)$ can become weak if $\epsilon > 0$. As optimization tries to shorten lengths as long as these delay bounds are met, timing can degrade a lot on such instances. Examples of instances for which topology optimization degrades the worst slack a lot are shown in Figures 5.13 and 5.14. In both instances, critical sinks are connected to the source by direct connections in the top-level topology of the non-optimized bicriteria algorithm. Optimization with greedy decreases the number of these direct connections but paths with detours arise. With $\epsilon = 0.5$, 50% detour is allowed and since delay parameters correspond to lower layers here, these detours result in huge worst slack degradations. To prevent that the optimization with greedy uses the freedom given by weak delay bounds, one can always choose $\xi = 1$ in the optimization with greedy algorithm.

Minimizing the sum of negative slacks is no direct optimization goal and the average degradations of the sum of negative slacks by the optimizations of Section 5.5 are small. The larger degradations can be avoided by excluding branching candidates in the optimization with greedy that result in a too large degradation of the sum of negative slacks. Examples of instances for which optimization could improve the results substantially can be found in Figures 5.11 and 5.12. On both instances we could decrease topology lengths by avoiding long edges of the top-level topology. On Figure 5.12(b) the length improvement is almost a factor 2. In both cases, optimization did not degrade the worst slack.

$\lvert N\rvert$		wsl-diff	length	wsl-diff	length
# instances		[ps]	ratio	[ps]	ratio
		$\epsilon = 0$		$\epsilon = 0.1$	
≤ 10	max	3	2.89	3	2.33
	min	-10	0.97	-99	0.97
	av	-2	1.03	-1	1.01
# 468 838	total		1.04		1.01
$11 - 100$	max	2	3.87	1	3.68
	min	-10	0.96	-106	0.95
	av	-4	1.29	-5	1.15
# 112 216	total		1.33		1.15
> 100	max	0	5.22	2	4.58
	min	-10	1.00	-104	1.00
	av	-6	1.87	-14	1.57
# 3 538	total		1.87		1.55
all	max	3	5.22	3	4.58
	min	-10	0.96	-106	0.95
	av	-2	1.08	-2	1.04
# 584 592	total		1.20		1.10

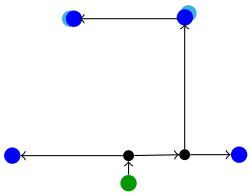
$\lvert N\rvert$		wsl-diff	length	wsl-diff	length
# instances		[ps]	ratio	[ps]	ratio
		$\epsilon = 0.5$		$\epsilon = 1$	
≤ 10	max	3	2.01	3	2.13
	min	-293	0.96	-547	0.98
	av	-3	1.01	-3	1.00
# 468 838	total		1.00		1.00
$11 - 100$	max	0	2.79	0	2.23
	min	-342	0.95	-562	0.95
	av	-16	1.09	-24	1.05
# 112 216	total		1.08		1.04
> 100	max	0	2.91	0	2.81
	min	-357	1.00	-704	1.00
	av	-57	1.33	-106	1.23
# 3 538	total		1.28		1.18
all	max	3	2.91	3	2.81
	min	-357	0.95	-704	0.95
	av	-6	1.03	-8	1.01
# 584 592	total		1.05		1.03

Table 5.2: Comparison between the bicriteria algorithm including all optimizations of Section 5.5 ($\mathcal{A}_{\text{main}}$) and bounds on length and worst slack. We compare the results for the two small values $\epsilon = 0, 0.1$ (on top) and for two larger values $\epsilon = 0.5, 1$ (at the bottom).
The columns entitled *length ratio* show the ratios $\frac{\text{length}(\mathcal{A}_{\text{main}})}{\text{length}(\mathcal{A}_{\text{ref}_1})}$, where $\mathcal{A}_{\text{ref}_1}$ is the algorithm computing the initial short topologies. The wsl-diff columns show the differences $\min\{0, \text{wsl}(\mathcal{A}_{\text{main}})\} - \min\{0, \text{wsl}(\mathcal{A}_{\text{ref}_2})\}$, where $\mathcal{A}_{\text{ref}_2}$ is the Huffman Coding Algorithm. A more detailed description of how the table is arranged can be found in Section 5.6.1.



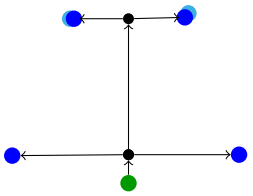
$\|N\|$:	7
Length:	2 329 μm
WSL:	−376 ps

(a) Bicriteria algorithm with $\epsilon = 0.5$, without optimization.



$\|N\|$:	7
Length:	2 007 μm
WSL:	−618 ps

(b) Bicriteria algorithm with $\epsilon = 0.5$, with optimization.



$\|N\|$:	7
Length:	2 011 μm
WSL:	−378 ps

(c) Greedy algorithm with $\xi = 0.5$.

Figure 5.13: Example of an instance for which optimization of the bicriteria algorithm with $\epsilon = 0.5$ degrades timing a lot. Here, the greedy algorithm with $\xi = 0.5$ gives a better solution.

5.6.3 Comparison between Bicriteria and Bounds for Length and Slack

We now compare the optimized bicriteria algorithm $\mathcal{A}_{\text{main}}$ from Section 5.6.2 with bounds on length and worst slack. We compare the lengths of the computed topologies with the lengths of the initial topologies. For instances with up to 9 sinks, the initial topologies are shortest possible as they have been computed by the FLUTE algorithm [CW08]. Initial topologies for larger instances are not necessarily optimum and can hence be larger than the output of the bicriteria algorithm. We compare worst slacks with the worst slacks of the output of the Huffman Coding Algorithm [Huf52]. Due to numerical issues it is possible that values of the form $\frac{\text{rat}(t)-\text{dist}(p(s)-p(t))}{b}$ are stored with a small error. These errors can result in an error of 1 when applying the $\lfloor . \rfloor$-operation during computation of $\text{bif}(t)$ and the worst slack of the Huffman topologies can be smaller that optimum by at most b.

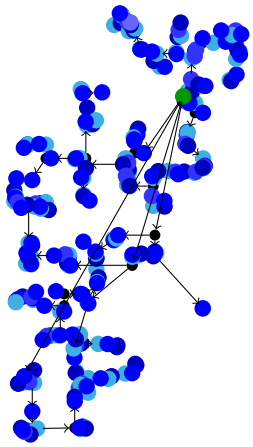
According to Table 5.2 for $\epsilon = 0$ we are close to the almost optimum timing of the Huffman coding topology. On average, the worst slacks lies only 2 ps below the almost-optimum. The maximum worst slack degradation is 10 ps which coincides with the worst slack guarantee of $2b$. In average, the topology lengths are 8% larger than the approximately shortest topologies. For the largest instances the average and total length ratios are much larger, resulting in a total length increase of 20% over all instances. Increasing ϵ from 0 to 0.1 does not degrade timing of most instances. The worst slack of timing outliers degrades to roughly 100 ps but we gain substantial length improvements.

Topology lengths can be decreased further by increasing ϵ. For $\epsilon = 1$, the total topology lengths are within only 3% from optimum. The improved lengths for $\epsilon > 0$ goes along with timing degradations. The average deviation from the almost optimum worst slack increases from 2 ps for $\epsilon = 0.0$, 0.1 to 8 ps for $\epsilon = 1$. For the largest values for ϵ, 0.5 and 1, there are some instances with a worst slack deviation of several hundred pico seconds. As these timing degradations are inacceptable in almost all scenarios, parameters $\epsilon = 0.5$ and 1 are usually not chosen in practice.

$\lvert N\rvert$		wsl-diff	sns-diff	length	wsl-diff	sns-diff	length
# instances		[ps]	[ps]	ratio	[ps]	[ps]	ratio
		$\epsilon = 0$			$\epsilon = 0.1$		
≤ 10	max	15	3 064	2.74	20	672	2.45
	min	-10	-323	0.22	-91	-565	0.29
	av	-2	3	0.94	-1	2	0.93
# 468 838	total			0.95			0.95
$11 - 100$	max	48	30 937	3.11	43	15 216	2.34
	min	-10	-7 079	0.11	-93	-7 323	0.26
	av	-2	106	0.95	-3	71	0.94
# 112 216	total			0.94			0.94
> 100	max	61	624 469	2.61	112	216 446	2.09
	min	-10	-50 412	0.15	-79	-177 589	0.30
	av	0	2 675	0.97	-4	1 812	1.01
# 3 538	total			0.90			1.04
all	max	61	624 469	3.11	112	216 446	2.34
	min	-10	-50 412	0.11	-93	-177 589	0.23
	av	-2	39	0.94	-2	28	0.92
# 584 592	total			0.94			0.94

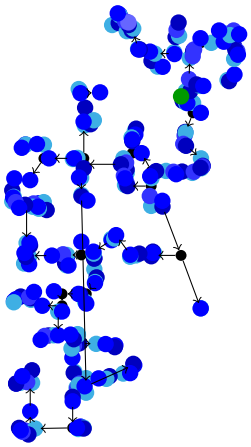
$\lvert N\rvert$		wsl-diff	sns-diff	length	wsl-diff	sns-diff	length
# instances		[ps]	[ps]	ratio	[ps]	[ps]	ratio
		$\epsilon = 0.5$			$\epsilon = 1$		
≤ 10	max	218	1 076	1.79	413	1 491	2.06
	min	-287	-776	0.43	-287	-1 148	0.75
	av	-2	-1	0.96	0	-1	1.00
# 468 838	total			0.98			1.00
$11 - 100$	max	252	13 200	2.04	475	8 835	2.17
	min	-325	-9 291	0.50	-533	-15 628	0.64
	av	-10	-55	0.97	-2	4	1.01
# 112 216	total			0.98			0.99
> 100	max	489	633 770	2.07	1724	2 551 448	2.63
	min	-249	-286 890	0.73	-401	-305 011	0.74
	av	-22	698	1.08	44	10 778	1.10
# 3 538	total			1.06			1.05
all	max	489	633 770	2.67	1724	2 551 448	2.63
	min	-325	-286 890	0.43	-1124	-305 011	0.64
	av	-4	-8	0.96	0	65	1.00
# 584 592	total			0.99			1.00

Table 5.3: Comparison between the bicriteria algorithm including all optimizations of Section 5.5($\mathcal{A}_{\text{main}}$) and the greedy topology algorithm by Bartoschek et al. [Bar+10] with trade-off parameter $\xi(\epsilon) = 1 - \epsilon$. We compare the results for two small values $\epsilon = 0, 0.1$ (on top) and for two larger values $\epsilon = 0.5, 1$ (at the bottom).
The columns entitled *length ratio* show the ratios $\frac{\text{length}(\mathcal{A}_{\text{main}})}{\text{length}(\mathcal{A}_{\text{ref}})}$. The wsl-diff columns show the differences $\min\{0, \text{wsl}(\mathcal{A}_{\text{main}})\} - \min\{0, \text{wsl}(\mathcal{A}_{\text{ref}})\}$ while the sns-diff columns display $\text{sns}(\mathcal{A}_{\text{main}}) - \text{sns}(\mathcal{A}_{\text{ref}})$. A more detailed description of how the table is arranged can be found in Section 5.6.1.



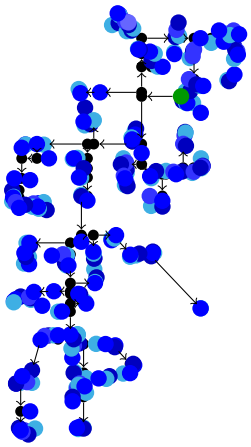
$\|N\|$:	249
Length:	6 616 μm
WSL:	−173 ps

(a) Bicriteria algorithm with $\epsilon = 0.5$, without optimization.



$\|N\|$:	249
Length:	4 944 μm
WSL:	−311 ps

(b) Bicriteria algorithm with $\epsilon = 0.5$, with optimization.



$\|N\|$:	249
Length:	5 058 μm
WSL:	−110 ps

(c) Greedy algorithm with $\xi = 0.5$.

Figure 5.14: Example of an instance for which optimization of the bicriteria algorithm with $\epsilon = 0.5$ degrades timing a lot. Here, the greedy algorithm with $\xi = 0.5$ gives better solutions.

5.6.4 Comparison between Bicriteria and Greedy

In this section we use an optimized version of the greedy topology algorithm by Bartoschek et al. [Bar+10] with trade-off parameter $\xi(\epsilon) = 1 - \epsilon$ as reference.

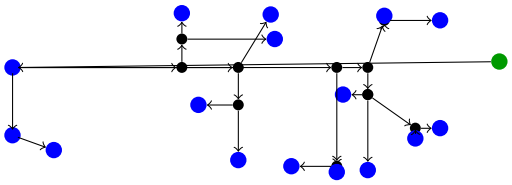
Table 5.3 shows the results. The greedy algorithm needed around 9 minutes to process all instances. Although this is slower than the bicriteria algorithm by a factor of 5, this is still a small running time. In practice, large instances are pre-clustered by a fast clustering algorithm of Maßberg and Vygen [MV08] and with this pre-clustering, the greedy algorithm is only slightly slower than the bicriteria algorithm.

Overall, the greedy algorithm produces topologies with slightly better worst slacks but with slightly worse sum of negative slacks. The only exception is the configuration $\epsilon = 0.5$ where the greedy algorithm seems to push more on timing. For parameters $\epsilon = 0.1$ ($\Rightarrow \xi = 0.9$) and $\epsilon = 0$ ($\Rightarrow \xi = 1$) that are most relevant in practice, the average worst slack degradation of the bicriteria algorithm lies around $2\,ps$ which corresponds to roughly $b/2$. In contrast to the greedy algorithm, the bicriteria algorithm has a configuration (i. e. $\epsilon = 0$) in which it is guaranteed not to produce timing outliers. When using larger values for ϵ such as 0.5 and 1, both algorithms produce solutions that have a worst slack far away from optimum.

For the smaller choices for ϵ, the bicriteria algorithm produces shorter topologies. The total net length reduction lies around 6%. The average net length improvement can be even higher. While the greedy algorithm is the most commonly used topology generation algorithm during design phases that target at maximizing the worst slack, the bicriteria algorithm appears to be well-suited in our application of timing-constrained global routing as the significantly shorter topology lengths improve routability a lot.

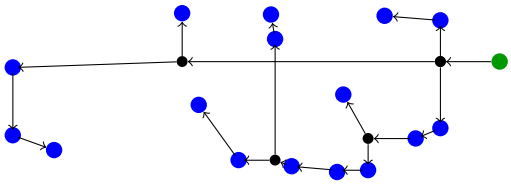
Instances for which the bicriteria algorithm yields worse results than the greedy algorithm can be found in Figures 5.13 and 5.14. Here, the initial short topologies contain detours and are bad w. r. t. timing. The bicriteria algorithm can repair most timing violations by long connections of the top-level topology (as this is done in the non-optimized version) or can recover most connections of the initial topology on cost of a degraded timing (as this is done in the optimized version). The greedy algorithm does not make use of a bad initial topology and builds relatively short topologies with good timing.

Instances for which the bicriteria achieves better results than the greedy algorithm are shown in Figure 5.15.



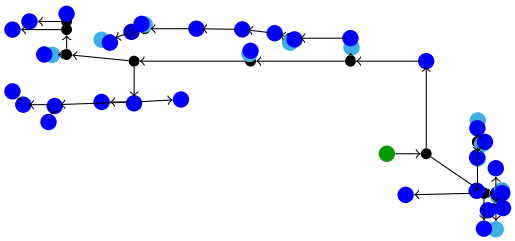
$|N|$: 17
Length: 258 μm
WSL: −47 ps

(a) Greedy algorithm with $\xi = 0.9$.



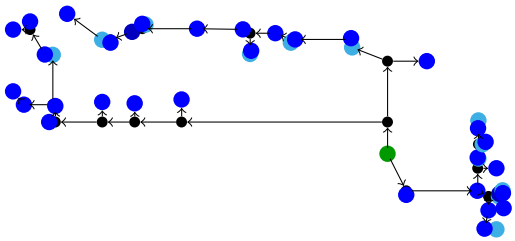
$|N|$: 17
Length: 174 μm
WSL: −32 ps

(b) Bicriteria algorithm with $\epsilon = 0.1$.



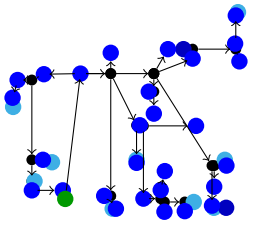
$|N|$: 45
Length: 597 μm
WSL: −71 ps

(c) Greedy algorithm with $\xi = 0.9$.



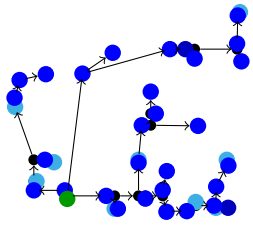
$|N|$: 45
Length: 529 μm
WSL: −61 ps

(d) Bicriteria algorithm with $\epsilon = 0.1$.



$|N|$: 40
Length: 644 μm
WSL: −40 ps

(e) Greedy algorithm with $\xi = 0.9$.



$|N|$: 40
Length: 473 μm
WSL: −28 ps

(f) Bicriteria algorithm with $\epsilon = 0.1$.

Figure 5.15: Instances for which the bicriteria algorithm with $\epsilon = 0.1$ behaves better than the greedy algorithm with $\xi = 0.9$.

Chapter 6

On the Way to a Practical Algorithm: Virtual Buffering

Although we can find an almost optimum solution to the *Minimum Cost Buffered Steiner Tree Problem* with a given topology in polynomial time, the practical running time is still too slow. To speed-up the repeater tree construction, we consider the Steiner tree construction problem and the problem of buffering a given Steiner tree separately from each other.

In this chapter we restrict ourselves to the first of these sub-problems and show how to compute a Steiner tree which trades-off congestion for *linear timing*. The linear delay model is a simple delay model that estimates the delay of a buffered Steiner tree without actually inserting repeaters. In this sense, such a Steiner tree can be considered a *virtual buffering*.

As in Chapter 4, we assume that the global routing graph is a directed graph although all results can equally be applied to the undirected case.

6.1 A Linear Delay Model for Steiner Trees

While computing a buffered Steiner tree as in Chapter 4 was computationally expensive, a topology with good properties w. r. t. both linear delays and lengths can be computed fast (Chapter 5). A natural question arising at this point is how fast we can compute a Steiner tree trading-off linear delays and congestion.

Definition 6.1 *Let G be a directed graph with delays $\rho : E(G) \cup \{\circ\} \to \mathbb{R}_{\geq 0}$ with $\rho(\circ) = 0$. Let (A, κ) be a Steiner tree in G for a net N with source s.*

For $t \in N \backslash \{s\}$ we define the linear delay between s and t in (A, κ) as

$$\text{linear\_delay}_{(A,\kappa)}(s,t) := \sum_{\zeta \in E(A_{[s,t]})} \rho(\kappa(\zeta)).$$

In the linear delay model, gate delays are not influenced by Steiner trees, i. e. the delay along a gate is constant for each gate. As for the buffered case, our goal is to compute a Steiner tree minimizing a weighted sum of edge costs $c : E(G) \to \mathbb{R}_{\geq 0}$ and delays. More precisely, we want to solve the following problem:

Minimum Cost Steiner Tree Problem with Linear Delays

Instance: A graph G with edge costs $c : E(G) \to \mathbb{R}_{\geq 0}$ and linear edge delays $\rho : E(G) \cup \{\circ\} \to \mathbb{R}_{\geq 0}$ with $\rho(\circ) = 0$. A net $N \subseteq V(G)$ with source s and sink delay costs $\lambda : N \backslash \{s\} \to \mathbb{R}_{\geq 0}$.

Output: A Steiner tree (A, κ) for N in G minimizing

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \text{linear\_delay}_{(A,\kappa)}(s, t).$$

This is of course the special case of the *Minimum Cost Buffered Steiner Tree Problem* for which $L = \emptyset$ and the functions $F(e, .)$ are constant for each edge $e \in E(G)$. Recall that Chuzhoy et al. [Chu+05] proved for this problem that no $o(\log \log |N|)$ approximation algorithm exists unless every problem in NP can be solved in $\mathcal{O}(n^{\log \log \log n})$ time (where n is the instance size).

6.2 Shortest Paths and Optimum Topology Embeddings

While we have seen that the general *Minimum Cost Buffered Steiner Tree Problem* is NP-hard for two-terminal nets, this is obviously not the case here (if $P \neq NP$).

By defining the traversal cost of an edge e as $c(e) + \lambda(t) \cdot \rho(e)$ (where $N = \{s, t\}$), we see that the two-terminal version of the *Minimum Cost Steiner Tree Problem with Linear Delays* is just a *Standard Shortest Path Problem* and hence can be solved optimally in time

$$\mathcal{O}(|E(G)| + |V(G)| \log(|V(G)|)).$$

The next theorem shows how to embed a given topology into G in an optimum way. This theorem is just an improved version of the theorems of Section 4.6 and is equal to Theorem 11 in [Hel+17].

Theorem 6.2 *Let G, N, c, ρ, λ be an instance of the Minimum Cost Steiner Tree Problem with Linear Delays. Let T be a topology for N.*

We can compute in $\mathcal{O}(|N| \cdot (|E(G)| + |V(G)| \log(|V(G)|)))$ time a Steiner tree (A, κ) for N with underlying topology T such that its cost

$$\sum_{\zeta \in E(A)} c(\kappa(\zeta)) + \sum_{t \in N \backslash \{s\}} \lambda(t) \cdot \text{linear\_delay}_{(A,\kappa)}(s, t)$$

is minimum among all embeddings of T.

Proof The algorithm is similar to the algorithms of the proof of Theorems 4.12 and 4.14. As in their proofs we process the edges of T in reverse topological ordering. While processing an edge $(\mathfrak{v}, \mathfrak{w}) \in E(T)$ we compute labels $(u, \alpha_{(\mathfrak{v},\mathfrak{w})}(u))_{(\mathfrak{v},\mathfrak{w})}$ for all $u \in V(G)$. These labels correspond to embeddings of the topology $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$ with total cost at most $\alpha_{(\mathfrak{v},\mathfrak{w})}(u)$ and in which $\mathfrak{v}$ has position u. We also specify λ-values for all Steiner points.

If we embed an edge $(\mathfrak{v}, t) \in E(T)$ with $t \in N \backslash \{s\}$, Dijkstra's algorithm [Dij59] on the graph $\big(V(G), \{(u', u) : (u, u') \in E(G)\}\big)$ with cost function $\text{cost}(e) = c(e) + \lambda(t) \cdot \rho(e)$

starting at $t \in V(G)$ produces these labels and we obtain corresponding embeddings by transforming paths (in G) into Steiner paths. As in Theorems 4.12 and 4.14 we have to extend the otherwise trivial embedding corresponding to $(t, -\infty)$ by an edge ζ mapped to $\kappa(\zeta) = \circ$ without increasing costs.

If we embed edge $(\mathfrak{v}, \mathfrak{w}) \in E(T)$ entering a Steiner node $\mathfrak{w} \notin N \backslash \{s\}$, let $\delta_T^+(\mathfrak{w}) = \{\overline{\mathfrak{w}}_1, \overline{\mathfrak{w}}_2\}$ and assume that for all $u \in V(G)$ we have already computed labels $(u, \alpha_{(\mathfrak{w},\overline{\mathfrak{w}}_i)}(u))_{(\mathfrak{w},\overline{\mathfrak{w}}_i)}$ as desired ($i = 1, 2$). We set $\lambda(\mathfrak{w}) := \lambda(\overline{\mathfrak{w}}_1) + \lambda(\overline{\mathfrak{w}}_2)$ and run Dijkstra's algorithm on the graph arising from G by reversing all edges and adding a new node w with incident edges (w, u) with $\rho((w, u)) = 0$ and $c((w, u)) = \alpha_{(\mathfrak{w},\overline{\mathfrak{w}}_1)}(u) + \alpha_{(\mathfrak{w},\overline{\mathfrak{w}}_2)}(u)$ for all $u \in V(G)$. Vertex w serves as start node and we use the cost function $\text{cost}(e) = c(e) + \lambda(\mathfrak{w}) \cdot \rho(e)$.

Due to the choice of costs of the edges leaving w, $w - u$ paths found by this algorithm naturally correspond to embeddings of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$ and the algorithm produces labels as desired.

Let $(s, \mathfrak{w}) \in E(T)$ be the unique edge leaving s in T. We output the embedding corresponding to $(s, \alpha_{(s,\mathfrak{w})}(s))$.

Since the running time of the overall algorithm is dominated by the $\mathcal{O}(|N|)$ applications of Dijkstra's algorithm, each taking time $\mathcal{O}(|E(G)| + |V(G)| \log(|V(G)|))$, the running time is clear.

To prove correctness we show that for each $(\mathfrak{v}, \mathfrak{w}) \in E(T)$ and $u \in V(G)$ there is no embedding of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$ where $\mathfrak{v}$ is positioned at u that has cost smaller than $\alpha_{(\mathfrak{v},w)}(u)$.

For $\mathfrak{w} \in N \backslash \{s\}$ this is clear by the correctness of Dijkstra's algorithm. For $\mathfrak{w} \in V(T) \backslash N$ let (A^*, κ^*) be an optimum embedding $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}) + T(\mathfrak{w})$ with $\kappa^*(\mathfrak{v}) = u$. If $\delta^+(\mathfrak{w}) = \{\overline{\mathfrak{w}}_1, \overline{\mathfrak{w}}_2\}$, (A^*, κ^*) consists of embeddings (A_i, κ_i) of $\mathfrak{w} + (\mathfrak{w}, \overline{\mathfrak{w}}_i) + T(\overline{\mathfrak{w}}_i)$ for $i = 1, 2$ plus an $\kappa^*(\mathfrak{w}) - u$ path. By induction and construction, the cost of the edges between w and u in the modified graph created for the application of Dijkstra's algorithm during processing of $(\mathfrak{v}, \mathfrak{w}) \in E(T)$ does not exceed the sum of costs of (A_1, κ_1) and (A_2, κ_2). By correctness of Dijkstra's algorithm, $\alpha_{(\mathfrak{v},\mathfrak{w})}(u)$ does not exceed the cost of (A^*, κ^*). □

Similar to Chapter 4 we have now developed an optimum algorithm for the special case of the *Minimum Cost Steiner Tree Problem with Linear Delays* where $|N|$ is constant. For larger sets of sinks a solution for the *Shallow-Light Topology Problem with Criticalities* (see Theorem 5.9) serves as a good starting topology.

6.3 Speed-up Techniques for Practical Instances

Although the running time of the algorithm contained in the proof of Theorem 6.2 is already smaller than its buffered extension (Theorem 4.12), it is still not fast enough for practical application.

Recall that in many applications, G is a 3-dimensional grid graph (see Section 2.4.3) and that the placement step, that is usually preceding the global routing step in any physical design flow, tries to place the circuits such that connected pins are not far away from each other. As a result, the pins of most nets are local in the sense that optimum Steiner trees for them are contained inside small sub-grids of G. Traversing the whole graph during each of the several million path searches would be way too time consuming.

6.3.1 Reducing Running Time by Limiting Search Areas

To achieve a fast running time of the overall algorithm it is necessary to reduce the number of potential Steiner points, i. e. the number of vertices $u \in V(G)$ for which we create a label $(u, \alpha_{(\mathfrak{v},\mathfrak{w})}(u))_{(\mathfrak{v},\mathfrak{w})}$ during embedding of a topology edge $(\mathfrak{v}, \mathfrak{w})$.

For positions that are far away from the bounding box of its terminals it is usually easy to find out that placing a Steiner point there cannot lead to an optimum solution. The cost of a topology embedding that is easy to compute (e. g. an embedding where all Steiner points are placed at the source's position) or an embedding that we have computed in an earlier phase of the resource sharing algorithm (Chapter 3) might already be smaller than the length of a shortest path to that position. When running Dijkstra's algorithm we do not have to wait until labels at these distant positions are created.

Simultaneous embedding of sibling paths. Instead of excluding Steiner point candidates explicitly, we use a dynamic approach to determine when we can stop a Dijkstra path search. Assume that we are given a topology T for a net N. Let $\mathfrak{v} \in V(T)$ and assume that we have already computed labels corresponding to embeddings of the subtopologies rooted at the successors of $\mathfrak{v}$ (see the proof of Theorem 6.2). If $\mathfrak{v}$ is the source of N, we can certainly stop the label generation as soon as we have labeled the source's position permanently. Assume that $\mathfrak{v}$ is a Steiner point with outgoing edges $(\mathfrak{v}, \mathfrak{w}_1)$ and $(\mathfrak{v}, \mathfrak{w}_2) \in E(T)$. Instead of embedding these edges one after the other, we run both path searches used for the embeddings simultaneously. This approach is similar to the largely used *bi-directional Dijkstra* (see [Nic66]).

Each point u that is visited by both path searches is a possible location for $\mathfrak{v}$. The combined cost of the embeddings of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}_1) + T(\mathfrak{w}_1)$ and $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}_2) + T(\mathfrak{w}_2)$ with $\kappa(\mathfrak{v}) = u$ is equal to the sum of keys of the two labels at u. Since the keys of the labels selected in each iteration of Dijkstra's algorithm are monotonically increasing, the cost of these embeddings will be larger for Steiner point candidates found in later iterations. After the first node in the global routing graph has been labeled permanently by both path searches, we compute a bound on the cost of the embeddings of $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}_1) + T(\mathfrak{w}_1)$ and $\mathfrak{v} + (\mathfrak{v}, \mathfrak{w}_2) + T(_2)$ and omit to process labels that have a key exceeding this bound. In the case that we embed a *placed* topology, the costs of labels at the first node u that is permanently labeled by both path searches plus an estimate on the cost (with respect to the cost function $c(.) + \lambda(\mathfrak{v}) \cdot \rho(.)$) of the path between u and the position of the predecessor of $\mathfrak{v}$ in T can be used as such a bound. In the next section we show how to compute an estimate on path costs by a landmark based A^* approach.

Bounded embedding tolerances. In practice, Steiner point positions computed by the optimized version of the bicriteria algorithm of Chapter 5 are very good unless the design is highly congested. Geometric information obtained by the positions of the endpoints of edges in the placed topology can serve as a *guideline* during the path searches.

In the case that the global routing graph is a 3-dimensional grid graph as described in Section 2.4.3, we can use placement information of the initial topology to impose a limited *embedding tolerance* on topology edges. Let $i \in \{1, 2\}$ and let BB be the bounding box of

- all nodes in the global routing graph where we have an initial label for the path search used to embed $(\mathfrak{v}, \mathfrak{w}_i)$, and
- the position of $\mathfrak{v}$ in the initial placed topology.

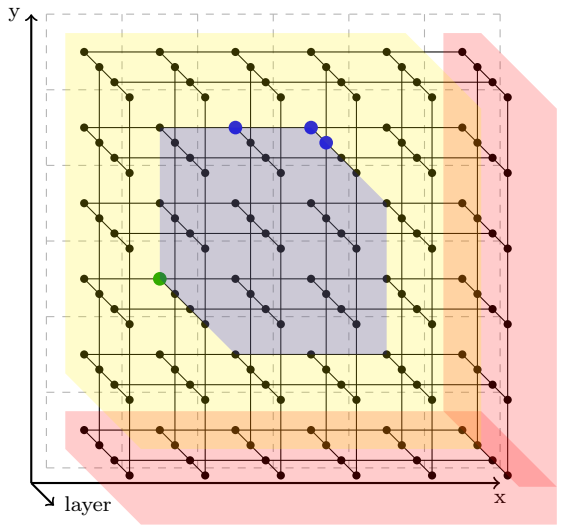


Figure 6.1: Visualization of embedding tolerance. In this example we wish to find a path from one of the blue vertices to the green vertex. The bounding box of these pins (blue box) is extended in each direction (yellow box). During the path search we forbid all vertices within the red boxes.

Let $\text{tol} > 0$ be a parameter. The value of tol will serve as a trade-off between quality and running time and can be chosen dependent on the the timing criticality of $(\mathfrak{v}, \mathfrak{w}_i)$ in T and on the aspect ratio of BB. We extend BB by tol in each direction and restrict the path search to nodes within this extended bounding box. An example of such a restricted routing area can be found in Figure 6.1. In the picture we choose $\text{tol} = 1$ and restrict the path search to the nodes within the blue and yellow box.

6.3.2 Future Costs

In the case that the initial topology is a *placed* topology, which is always the case if we are embedding two-terminal nets, path searches have a *target*. By exchanging the original cost function with its *reduced costs* we can decrease the number of labels produced during target directed path searches (see [GH05]).

Recall that the cost functions we use during Dijkstra's algorithm are of the form

$$\text{cost}((v,w)) = c((v,w)) + \lambda \cdot \rho((v,w)).$$

Definition 6.3 *For functions $\pi_c, \pi_\rho : V(G) \to \mathbb{R}_{\geq 0}$ such that for all $(v,w) \in E(G)$,*

- $c((v,w)) + \pi_c(w) - \pi_c(v) \geq 0$ *and*
- $\rho((v,w)) + \pi_\rho(w) - \pi_\rho(v) \geq 0$,

we call the function $V(G) \to \mathbb{R}_{\geq 0}$, $v \mapsto \pi_c(v) + \lambda \cdot \pi_\rho(v)$ a feasible potential and the edge cost function

$$\text{cost}_{\pi_c + \lambda\pi_\rho}((v,w)) = \text{cost}((v,w)) + \pi_c(w) - \pi_c(v) + \lambda \cdot (\pi_\rho(w) - \pi_\rho(v))$$

*reduced costs. We also call π_c respectively $\pi_\rho(w)$ a **feasible potential** or **potential function** if it satisfies the above inequalities.*

Note that for $s, t \in V(G)$ and any s-t- path P in G,

$$\text{cost}(P) = \text{cost}_{\pi_c + \lambda\pi_\rho}(P) + (\pi_c(s) + \lambda\pi_\rho(s)) - (\pi_c(t) + \lambda\pi_\rho(t))$$

and hence, a shortest path with respect to the original cost function is also a shortest path with respect to reduced costs.

The definition of π_c and π_ρ, as well as the running time that is necessary to compute these values, heavily depend on the structure of the global routing graph. We now define π_c and π_ρ in the case that the vertices of the global routing graph G can be written as

$$M \times \{1, \dots, Z\} \text{ for a finite metric space } (M, \mathrm{dist})$$

and a finite number $Z \in \mathbb{N}$ of routing layers (see Section 2.4.3). In this case we can define the geometric distance $||v, w||$ between two vertices $v, w \in V(G)$ as the distance with respect to dist of the projections of v and w onto M.

Different wire codes that influence delays and space consumption of wires can be modeled easily by inserting further routing layers. Henceforth, we will omit mentioning wire codes explicitly.

For the functions π_ρ and π_c that we define now, $\pi_c(v) + \lambda\pi_\rho(v)$ is a lower bound on the cost on the shortest path between a node $v \in V(G)$ and the target t of the path search.

Definition of π_ρ by geometric lower bounds. For an edge $e \in E(G)$ connecting two nodes on the same routing layer, the estimated delay $\rho(e)$ usually depends on the geometric length of e and the layer only (resp. the combination of layer and wire code). If $e \in E(G)$ connects different layers, $\rho(e)$ depends on the layers only.

For $z \in \{1, \dots, Z\}$ we are usually given a *delay per length* value $\rho_{\text{wire}}(z) \in \mathbb{R}_{\geq 0}$ for z while for a pair $z_1, z_2 \in \{1, \dots, Z\}$ of adjacent wiring planes (i. e. $|z_1 - z_2| = 1$) we are given a *via delay* $\rho_{\text{via}}(z_1, z_2) \in \mathbb{R}_{\geq 0}$ such that $\rho_{\text{via}}(z_1, z_2) = \rho_{\text{via}}(z_2, z_1)$. For $v \in V(G)$ we denote the layer on which v is located by $\text{layer}(v)$. Using this notation we write

$$\rho((v, w)) = \begin{cases} ||v, w|| \cdot \rho_{\text{wire}}(\text{layer}(v)) & \text{if layer}(v) = \text{layer}(w), \\ \rho_{\text{via}}(\text{layer}(v), \text{layer}(w)) & \text{otherwise.} \end{cases}$$

Let $t \in V(G)$ be the target of the path search. For $v \in V(G)$ we define

$$\pi_\rho(v) := \min_{z \in \{1,\dots,Z\}} \left\{ ||v, t|| \cdot \rho_{\text{wire}}(z) + \sum_{z'=\min\{\text{layer}(t),z\}}^{\max\{\text{layer}(t),z\}-1} \rho_{\text{via}}(z', z'+1) + \sum_{z'=\min\{\text{layer}(v),z\}}^{\max\{\text{layer}(v),z\}-1} \rho_{\text{via}}(z', z'+1) \right\}.$$

This can indeed be used to obtain a feasible potential as the next lemma shows:

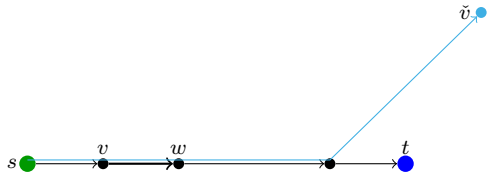
Lemma 6.4 *For any edge $(v, w) \in E(G)$ it holds that $\rho((v, w)) + \pi_\rho(w) - \pi_\rho(v) \geq 0$.*

Proof Let H be the graph with vertex set $V(H) = M \times \{1, \dots, Z\}$ and edge set

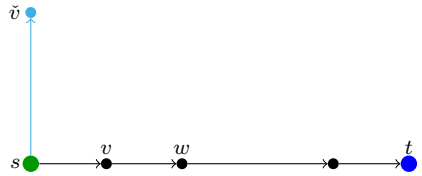
$$E(H) = \{(v, w) : \text{layer}(v) = \text{layer}(w) \text{ or } (|\text{layer}(v) - \text{layer}(w)| = 1 \text{ and } ||v, w|| = 0)\}.$$

For $v \in V(H)$ the length of a shortest v-t path in H with respect to ρ-costs is equal to $\pi_\rho(v)$ and hence, $\pi_\rho(w) + \rho(e) \geq \pi_\rho(v)$ for all $e = (v, w) \in E(H)$.

Now, the lemma follows from the fact that G is a subgraph of H. □



(a) Large parts of the shortest s-t and s-$\check{v}$ paths coincide. The reduced costs of edge (v, w) are small.



(b) The shortest s-t and s-$\check{v}$ path are completely different. The reduced costs are equal to the original costs.

Figure 6.2: Idea of the landmark-based approach by Goldberg and Harrelson [GH05] who choose the potential function $\max\{0, \text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(t, \check{v})\}$. Figure 6.2(a) is taken from [GH05] (Figure 2).

Henke [Hen16] defined several more advanced future cost functions on weighted grid graphs. On instances that contain many blockages that extend to all available routing layers it would be more accurate to use one of those.

Definition of π_c by landmark-based future costs. As the c-cost of an edge is usually not dependent on the length of the edge only, defining good future costs for cost function c is harder. We use the landmark-based A^* approach by Goldberg and Harrelson [GH05] to define π_c. This strategy has already be used by Müller [Mül09].

We start by explaining the high-level idea of their approach. Assume that we want to find a shortest s-t path and we already know shortest paths from all vertices in G to a particular node $\check{v} \in V(G)$. The vertex $\check{v}$ is called *landmark*. If we are lucky, large parts of the shortest s-$\check{v}$ and the shortest s-t path coincide and we can use our knowledge on the s-$\check{v}$ path to achieve that reduced edge costs on the s-t path are small.

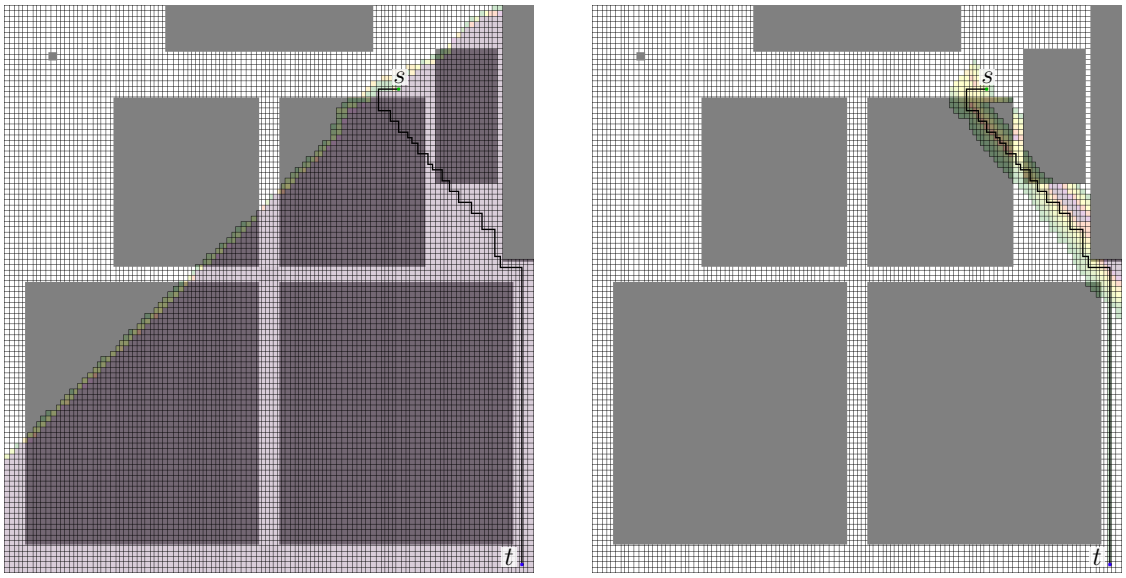
Figure 6.2, that is inspired by Figure 2 from [GH05], depicts this idea. For $v \in V(G)$ let $\text{dist}_{(G,c)}(v, \check{v})$ be the c-cost of a shortest v-$\check{v}$ path in G. Since the edge (v, w) in Figure 6.2(a) lies on a shortest v-$\check{v}$ path, $\text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(w, \check{v}) = c((v, w))$. In other words, the reduced cost of (v, w) with respect to the feasible potential $v \mapsto \text{dist}_{(G,c)}(v, \check{v})$ is zero and we are encouraged to use that edge. Note that using the potential $v \mapsto \text{dist}_{(G,c)}(v, \check{v})$ is identical to using the potential $v \mapsto \text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(t, \check{v})$ because $\text{dist}_{(G,c)}(t, \check{v})$ is constant.

In the case that shortest paths between s and $\check{v}$ and between t and $\check{v}$ are different (Figure 6.2(b)), it is not a good idea to use the potential defined before. The reduced edge cost of (v, w) would be even larger than the original costs and we would be discouraged to use (v, w). Instead, we would prefer to use the trivial potential $v \mapsto 0$ in that case.

Of course we are not able to find out if an edge (v, w) lies in the intersection of a shortest v-$\check{v}$ path with a shortest t-$\check{v}$ path sufficiently fast. By checking whether $\text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(t, \check{v})$ is positive or not, and by using the trivial potential function in case of negativity, we can at least distinguish between the two extreme cases depicted in Figure 6.2. By Lemma 2.2 of [GH05], $v \mapsto \max\{0, \text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(t, \check{v})\}$ is indeed a potential function and if $\check{V} \subseteq V(G)$ is a set of landmarks,

$$\pi_c(v) := \max_{\check{v} \in \check{V}} \left\{ \max\{0, \text{dist}_{(G,c)}(v, \check{v}) - \text{dist}_{(G,c)}(t, \check{v})\} \right\}$$

is again a potential function.



(a) Path search without future cost.

(b) Path search with future cost.

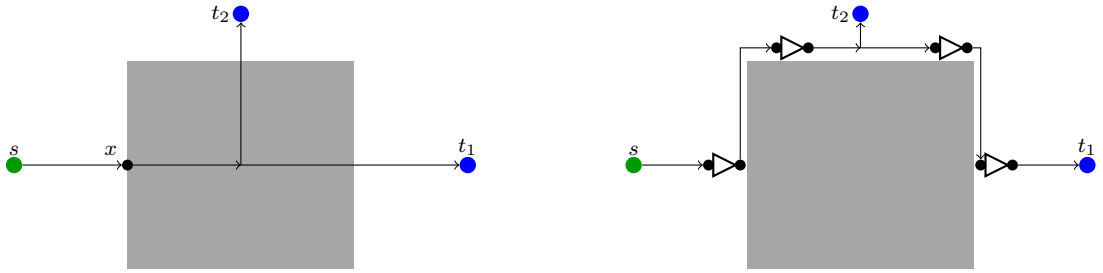
Figure 6.3: Visualization of the amount of vertices labeled during the path search from t to s. Pictures are shown in 2 dimensions although the path searches were performed in a 3 dimensional grid graph. Each rectangle represents 12 nodes. The darker the color of a rectangle the more of the represented vertices were visited. If none of these vertices were visited, the rectangle is white. The large gray boxes are blockages.

To speed-up Dijkstra's algorithm it is important to choose sets $\check{V}$ such that for many path searches there exists a landmark for which the target lies "between" the source and the landmark as in Figure 6.2(a). Goldberg and Harrelson [GH05] show how to find suitable landmark sets in general graphs and for geometric graphs such as road graphs. If G is a 3-dimensional grid graph as in Section 2.4.3, it is often a good idea to select the corners of the chip area on both the lowest and the highest routing layer.

The major drawback of the landmark approach of Goldberg and Harrelson [GH05] is of course the fact that we need to pre-compute the distances to all landmarks. Since we make millions – or even billions – of path searches in the same graph during a whole timing-constrained global routing, this effort pays-off. According to the price update strategy of the resource sharing algorithm by Müller, Radke, and Vygen [MRV11] (Algorithm 1), prices never decrease and hence, a feasible potential remains feasible during the whole algorithm.

At some points of the resource sharing algorithm the previously computed distances to the landmarks will only yield poor lower bounds for the new prices and we need to re-compute the distances. To detect these situations during the algorithm we compare the actual cost of all computed shortest paths with the lower bound provided by the feasible potential $\pi_c(v) + \lambda\pi_\rho(v)$. Whenever the ratio between the actual cost and the estimated cost has become too large for a sufficiently large number of path searches, we update landmark distances. To avoid that the number of landmark re-computations becomes too large, we strictly forbid re-computations if the number of path searches since the last re-computation is too small.

Good future costs have an enormous impact on the number of permanently labeled vertices as Figure 6.3 shows.



(a) Steiner tree with small linear delay that does not admit a good buffering. The capacitance at x is too large in each buffering solution.

(b) Avoiding to route over the blockage can improve the results after buffering although linear delays increase.

Figure 6.4: While computing Steiner trees with minimum linear delays as input to buffering we have to take blockages on the placement layer into account. In the picture we assume that the gray area is an area without placement space. We are not allowed to insert repeaters there but we are allowed to put wires on top of it.

6.4 Reach-Aware

Recall that the main goal of this thesis is the computation of *buffered* Steiner trees minimizing the sum of costs for placement congestion, routing congestion, and timing.

With respect to routing congestion, the output of the algorithm of Theorem 6.2 is already well-suited as a starting point for the actual buffer insertion (see Chapter 7). To achieve a large correlation between linear delays (Definition 6.1) and the delays after buffering we must take placement congestion into account already during Steiner tree computation. While Steiner trees with small linear delays usually have small delays after they are buffered in a timing-optimum way, we will probably get poor results if limited placement space prevents us from inserting repeaters at optimum positions. Especially on chips that have large areas without placement space (*blockages*) this is a severe problem.

Figure 6.4(a) shows an example of a Steiner tree with small linear delays. Due to the large blockage in the middle of the picture we cannot prevent a large electrical capacitance at point x and hence, we cannot achieve a good timing after buffering. To avoid such a situation we have to make sure that connected components over placement blockages are not too large. Instead of completely forbidding routing space over these blockages (which is too restrictive in nearly all cases), our goal is to compute a so-called *reach-aware Steiner tree* that we define now for rectilinear Steiner trees.

In the following we denote by $\pi : \mathbb{R}^3 \to \mathbb{R}^2$ the canonical projection $(x, y, z) \mapsto (x, y)$ and by $\text{layer} : \mathbb{R}^3 \to \mathbb{R}$ the function $(x, y, z) \to z$.

Definition 6.5 (Reach-aware rectilinear Steiner tree) *Let $\mathcal{B}$ be a finite set of axis-parallel rectangles (blockages) and let (A, κ) be a 3-dimensional rectilinear Steiner tree (Definition 2.2). Let $\Gamma_z \in \mathbb{R}_{\geq 0}$ for all $z \in \text{layer}(\kappa(V(A)))$ (reach lengths).*

We may assume that for all $(\nu, \omega) \in E(A)$, all inner points of the (possibly degenerated) line between $\pi(\kappa(\nu))$ and $\pi(\kappa(\omega))$ either lie completely inside or completely outside the set $\bigcup_{B \in \mathcal{B}} B$. In the first case we call an edge blocked.

*We say that (A, κ) is **reach-aware** if for all connected subgraphs A' of A consisting of*

blocked edges only,

$$\sum_{(\nu,\omega)\in E(A')} \frac{||\pi(\kappa(\nu)) - \pi(\kappa(\omega))||_1}{\Gamma_{\text{layer}(\kappa(\nu))}} \leq 1. \tag{6.1}$$

Note that we can check in linear time if a given Steiner tree is reach-aware.

6.4.1 Reach-Aware 2-Dimensional Steiner Trees

In the case that the layers on all edges are identical, the previous definition says that a rectilinear Steiner tree is reach-aware if and only if the length of each connected component over a blockage does not exceed a certain threshold.

The problem of computing such a Steiner tree with minimum total geometric length is called the *Length-Restricted Rectilinear Steiner Tree Problem.*

Müller-Hannemann and Peyer [MP03] introduced that problem and gave a 2-approximation algorithm with running time $\mathcal{O}((|N| + |\mathcal{B}|)^2 \log(|N| + |\mathcal{B}))$. Under the mild assumption that each component of the blocked area has a constant number of corners, Held and Spirkl [HS14] achieved the same approximation guarantee in almost linear running time $\mathcal{O}((|N| + |\mathcal{B}|) \log(|N| + |\mathcal{B})^2)$.

If the reach length is 0, we obtain the well-studied *Obstacle Avoiding Rectilinear Steiner Tree Problem* for which numerous fast 2-approximation algorithms exist, see e. g. [Fen+06], [Lin+08], [LZM08], and [Liu+09].

Bihler [Bih15] extended the algorithm by Held and Spirkl [HS14]. In the case that each component of the blocked area has a constant number of corners, he provides a 2-approximation algorithm with running time $\mathcal{O}((|N| + |\mathcal{B}|) \log(|N| + |\mathcal{B})^2)$ that can simultaneously handle

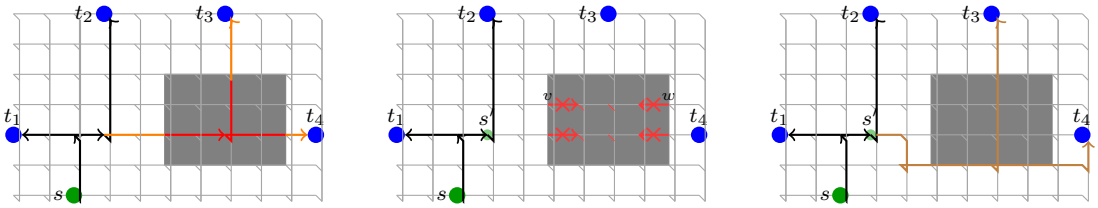
- blockages over which routing is allowed if reach length constraints are obeyed,
- blockages that completely block the routing space (i. e. reach length 0),
- blockages that block the routing space in horizontal direction but allow length-restricted routing in vertical direction, and
- blockages that block the routing space in vertical direction but allow length-restricted routing in horizontal direction.

The *Length-Restricted Rectilinear Steiner Tree Problem* is certainly far away from the problem of computing a 3-dimensional rectilinear reach-aware Steiner tree minimizing the objective function of *Minimum Cost Steiner Tree Problem with Linear Delays.* First, it does not take into account different reach lengths on different layers and second, it completely ignores delay constraints.

However, we can still use the algorithm of the previously mentioned authors to compute an initial short topology T_{init} for the bicriteria algorithm (see Section 5.3). The resulting topology will be a better starting point for the algorithm of Theorem 6.2.

6.4.2 Reach-Awareness by Restricting the Routing Area

In addition to a better choice of starting topology we enforce reach-awareness by restricting the global routing graph. This simple method has already been used for a long time in BonnRouteGlobal and BonnRoute [Ges+13], [Ahr+15], [Hel+15]. We could improve



(a) Initial Steiner tree that is not reach-aware. The red subgraph is too large.

(b) Remainder of the initial tree after removing the component over the blockage. The red edges are forbidden in G'. Horizontal edges are forbidden in the indicated direction only.

(c) Final reach-aware Steiner tree consisting of the remainder of the initial tree and a tree for $\{s', t_3, t_4\}$.

Figure 6.5: Visualization of the algorithm that computes reach-aware Steiner trees by forbidding edges described in Section 6.4.2. We assume that traversing 4 or more edges over blockages results in a reach violation.

running time and flexibility by a new implementation. We explain the approach in the case that the global routing graph is the 3-dimensional grid graph introduced in Section 2.4.3.

In a first round, we compute a Steiner tree (A, κ) without considering reach-awareness at all (see Figure 6.5(a)). For practical VLSI instances the fraction of reach-aware Steiner trees that we build without explicitly trying to is typically large. For the trees which indeed violate reach constraints we rip-out connected components over blockages that violate Inequality (6.1) in Definition 6.5. In Figure 6.5(a) this is exactly the component consisting of the red edges. To gain more flexibility in the subsequent embedding step we successively remove antennas, i. e. Steiner nodes with degree 1 and edges incident to them. These paths are colored orange in Figure 6.5(a). The set of erased edges can be written as a collection $(A_1, \kappa_1), \dots, (A_k, \kappa_k)$ of Steiner trees. For $i = 1, \dots, k$ the set

$$N_i := \{\nu : \nu \text{ is the root of } A_i\} \cup \{\kappa(\nu) : \nu \text{ is a leaf of } A_i\} \subset V(G)$$

together with criticalities

$$\lambda(\kappa(\nu)) = \sum_{\substack{t \in N \setminus \{s\} \text{ reachable} \\ \text{from } \nu \text{ in } A}} \lambda(t)$$

for leaves ν of A_i form an instance of the *Minimum Cost Steiner Tree Problem with Linear Delays*. In Figure 6.5(b), there is one new instance that needs to be reconnected. This instance has source s' and sinks t_3 and t_4.

We solve the resulting instances in a subgraph G' of G with the property that all paths in G' satisfy Definition 6.5. Together with the restriction that we do not allow to create Steiner points over blockages we end up with a reach-aware Steiner tree after combining the remainder of (A, κ) with the computed reach-aware Steiner trees for the sub-instances N_i. Note that Steiner points over blockages can easily be forbidden by removing labels $(u, \alpha_{(\mathfrak{v},\mathfrak{w})}(u))_{(\mathfrak{v},\mathfrak{w})}$ for which the projection of u to $\mathbb{R}^2$ lies within such a blocked area after the path search we perform to embed an edge $(\mathfrak{v}, \mathfrak{w})$ of the initial topology (see Theorem 6.2).

To build G' we gradually remove certain edges from G. First, we remove all via edges over blockages. Whenever we use an edge e over a blockage we have to follow the whole

straight path over the blockage until we reach a node that is not located over a blockage. The reason is that by construction of G we can only switch between horizontal and vertical direction by traversing a via edge. If this straight path violates Inequality (6.1), we delete blocked edge e. The resulting graph G' can be pre-computed globally. In Figure 6.5(b) the edges in $E(G)\backslash E(G')$ are printed in red. The four red horizontal wires are forbidden in one direction only. In the picture we assume that traversing four consecutive edges over blockages results in a violation of reach length. When we enter the horizontal edge at point v, we have to follow the direct horizontal path to point w. As this path consists of 4 edges we have to forbid the edge going from v to the right.

We finally output the tree consisting of the remainder of the initial tree and the computed Steiner trees for the sub-instances N_i. An example of the final tree is depicted in Figure 6.5(c). The Steiner tree for the sub-instance $\{s', t_3, t_4\}$ is printed in brown.

In the rare case that a pin (i. e. an element of a net) is inaccessible in G', no reach-aware Steiner tree without vias over blockages exists. In this case, we do not perform the path search for embedding the unique edge incident to that pin in G', but in our original graph G. To avoid unnecessarily long paths over blockages also in this case, we increase the cost of edges in $E(G)\backslash E(G')$ by a large value (e. g. by $|V(G)| \cdot \max\{c(e) : e \in E(G)\}$).

In this case we indeed cannot guarantee to find a reach-aware path, even if such a path exists. To avoid such an undesired behavior it is possible to run the path search introduced in Theorem 4.10. As cost function c we choose the cost function of the Dijkstra path search with linear delays (which is of the form $c(e) + \lambda \cdot \rho(e)$). As the function Δ we use $\Delta((v, w), x) = x + \frac{||\pi(v)-\pi(w)||_1}{\Gamma_{\text{layer}(v)}}$ and F can be defined as

$$F(e, x) = \begin{cases} 0 & \text{if } x \leq 1 \\ \infty & \text{otherwise.} \end{cases}$$

By Definition 6.5 and by Theorem 4.10, we will obtain a reach-aware path if such a path exists. The path itself will be nearly optimum. This improved approach can certainly be used as a replacement for all path searches that we would perform in G' otherwise. However, from a running time point of view the standard path search in G' is preferable.

6.5 Experimental Results

We implemented the timing-constrained global routing approach described in Chapter 3 as extension to the resource sharing based 3D global router BonnRouteGlobal [Ges+13] that is part of the BonnTools suite developed by the Research Institute for Discrete Mathematics and is used inside the IBM design environment. As a delay model we used the linear delay model introduced in Section 6.1. The topology embedding algorithm of Theorem 6.2 together with all speed-up techniques and improvements described in this chapter serves as block solver for the net customers. In this section we refer to this algorithm as TCGRLin (**T**iming **C**onstrained **G**lobal **R**outing with **Lin**ear delays)

We ran this algorithm on 11 microprocessor units in 14 nm and 22 nm technology that were provided by IBM. All netlists are unbuffered and do not contain layer and wire code assignments.

For all runs we used 16 threads on a machine with a 2.20 GHZ Intel Xeon E5-2699 processor. The experiments we present here coincide with the experiments for the linear delay model presented in [Hel+17] although the testbed is different.

Linear delay parameters. As global routing graph for all these designs we used the standard global routing graph of Section 2.4.3. In this setting, linear delays $\rho((v,w))$ along edges (v,w) with wire code wc can be written as

$$\rho((v,w)) = \begin{cases} ||v,w||_1 \cdot \rho_{\text{wire}}(\text{layer}(v), \text{wc}) & \text{if layer}(v) = \text{layer}(w) \\ \rho_{\text{via}}(\text{layer}(v), \text{layer}(w), \text{wc}) & \text{otherwise} \end{cases}$$

for parameters ρ_{wire} and ρ_{via} for each layer / wire code pair (cf. Section 6.3.2).

To compute these constants we use a library preprocessing by Bartoschek et al. [Bar+09]. The wire delay ρ_{wire} of a wire on a given plane and with a given wire code is equal to the delay per length in a long repeater chain. For more details we refer to the PhD thesis of Bartoschek [Bar14] or to Section 7.3.5. As via delay ρ_{via} we compute the delay through a via, assuming a default output capacitance and input slew.

The underlying input topologies for Theorem 6.2 are computed with the optimized variant of the bicriteria algorithm of Chapter 5 with bifurcation penalty $b = 0$. The reason for choosing $b = 0$ is not that the embedding algorithm cannot handle the case $b > 0$ but rather that there is no reference algorithm that is able to deal with bifurcation delay penalties.

Clustering. Note that the bicriteria algorithm requires uniform delays per length although the parameters ρ_{wire} differ much between the routing layers and wire codes. For example, on the largest unit U11 the wire delay on the lowest layer is 0.5 ps per micrometer and 0.09 ps per micrometer on the highest layer. For that reason we start by clustering nearby sinks using the clustering algorithm of Maßberg and Vygen [MV08]. Each set of clustered sinks is connected by the optimized bicriteria algorithm (Chapter 5) using the wire delay on the lowest layer for the default wire code. As source for each cluster we use a dummy node placed on the projection of the net's source into the bounding box of the sinks that belong to that cluster. These sources are then connected to the net's source by a bicriteria topology that uses the delay per length parameters on the highest layer and for the wire code that makes the fastest signal propagation possible. The combination of the topologies connecting the sinks of each cluster and the topology connecting the dummy sources yields the initial topology that is embedded by the embedding algorithm of Theorem 6.2.

Lower and upper delay bounds. The timing-constrained global routing algorithm requires lower and upper delay bounds (Section 3.4). Since gate delays are not influenced by Steiner trees in the linear delay model, it suffices to define these bounds for wire delays.

Let t be a sink pin and let s be the source of the net containing t. Let $\text{layer}(s)$ and $\text{layer}(t)$ be the layer on which s and t are located respectively. The delay

$$\min_{z \text{ layer}} \Bigg(\Bigg(\sum_{z'=\min\{\text{layer}(s),z\}}^{\max\{\text{layer}(s),z\}-1} \min_{\substack{\text{wc} \\ \text{wire code}}} \rho_{\text{via}}(z', z'+1, \text{wc}) \Bigg) + \text{dist}(s,t) \cdot \min_{\substack{\text{wc} \\ \text{wire code}}} \rho_{\text{wire}}(z, \text{wc}) + \Bigg(\sum_{z'=\min\{\text{layer}(t),z\}}^{\max\{\text{layer}(t),z\}-1} \min_{\substack{\text{wc} \\ \text{wire code}}} \rho_{\text{via}}(z', z'+1, \text{wc}) \Bigg) \Bigg)$$

is a lower bound on the wire delay between s and t (cf. Section 6.3.2 and Henke [Hen16]). This bound assumes that the distance $\text{dist}(s,t)$ is realized by wires with the same wire

delay. Since in our instances, each layer has an adjacent layer for which the wire delay ρ_{wire} with respect to all wire codes is similar, this is a reasonable assumption (cf. Section 2.4.3). To improve this lower bound in situations in which all layers have different characteristics, it is possible to compute an *s*-*t* path with minimum delay directly.

Defining an *upper* bound on delay is more difficult. As before, we start by computing the delay along a straight path. Instead of selecting optimum layers and wire codes, we select the lowest available layer and use the default wire code. We multiply this straight path delay with a *detour factor* that accounts for both

1. detours caused by the choice of the embedded topology, and
2. detours caused by the embedding of edges of that topology.

We also add a small constant to the resulting bound.

We start by computing a short topology for each instance. The ratio between the ℓ_1-length of the source-sink path in that short topology, and the geometric ℓ_1-distance between source and sink serves as an upper bound on the detour inside the topology (although it is no strict upper bound). The detours caused by embedding are quite small on most instances. We multiply the detour factor with an additional factor of 1.5 and relax this bound further by adding the delay needed to send the signal from one tile to its neighboring tile and back. This delay bound is no strict upper bound but turns out to be suitable in practice. For the largest unit U11 this bound is only violated for 0.003% of all timing resources. None of them lie on timing paths that do not meet their timing specification at the end. On the other units the situation is similar.

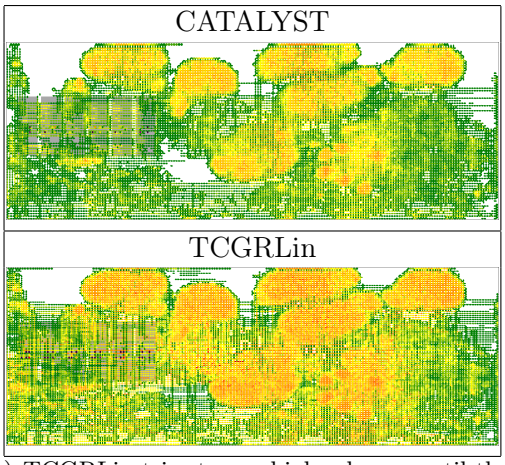
Reference algorithm. We compare TCGRLin with the layer assignment algorithm CATALYST [Wei+13] that is part of the IBM design flow. CATALYST computes a timing and congestion-aware layer assignment by making calls to a 2-dimensional global router. This assignment is then passed as constraint to the timing-unaware version of BonnRouteGlobal. In addition to these layer assignments, the reference run pre-assigns net length bounds to critical nets and bounds the source-to-sink distances inside topologies for high fan-out nets. These bounds are respected by BonnRouteGlobal. In addition, we post-optimize the output of CATALYST with BonnLayerOpt, the layer assignment tool of the BonnTools suite. This step is necessary for a fair comparison since CATALYST estimates delays based on 2-dimensional Steiner trees with a star-like topology instead of propagating delays along global wires as we do in the final BonnLayerOpt step.

Results. The results are shown in Table 6.1. The first lines ("Bounds") show worst slacks (**wsl**) and sum of negative slacks (**sns**) in the case that all delays are equal to their lower bound. We also ran the timing-unaware version of BonnRouteGlobal without any assignments and embedded approximately shortest Steiner trees without considering congestion or timing. The best routing overload among all runs is reported as "bound" on overload (**ol**) while shortest obtained wiring lengths (**wl**) and the smallest number of required **vias** yield "bounds" for these two metrics.

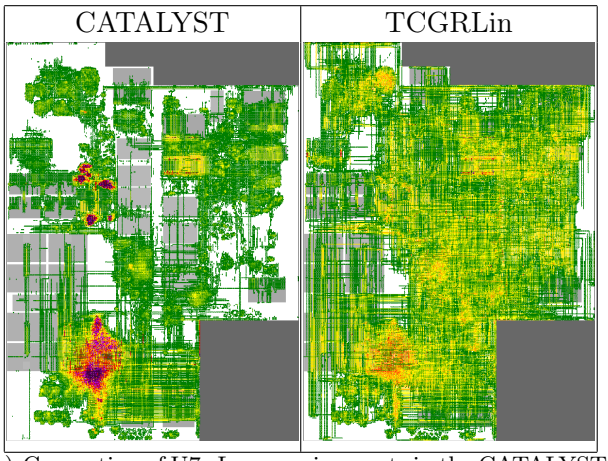
The remaining lines show worst slack (**wsl**), sum of negative slacks (**sns**), routing overload (**ol**), wiring length (**wl**), number of required vias (**vias**), and total running time (**wall time**) for the reference run and for TCGRLin. For our algorithm we report both results of the fractional solution after the resource sharing phase (Lines ①–⑪ in Algorithm 4) and of the final integral solution.

Unit (#nets, cycle time)	Experiment	wsl [ps]	sns [ns]	ol	wl [m]	vias [K]	wall time [h:mm:ss]
U1	"Bounds"	11	0	0	0.80	25	-
(10 188,	CATALYST [Wei+13]	-114	-4	0	0.80	28	0:03:40
250 ps)	TCGRLin (fractional)	-32	-1	1	0.80	31	0:00:15
	TCGRLin (final)	-26	-1	0	0.80	30	0:02:31
U2	"Bounds"	-62	-44	0	3.37	195	-
(21 385,	CATALYST [Wei+13]	-228	-220	8 032	4.14	225	0:04:58
240 ps)	TCGRLin (fractional)	-118	-107	8	3.47	214	0:01:43
	TCGRLin (final)	-90	-96	36	3.43	210	0:04:01
U3	"Bounds"	-57	-5	0	1.77	834	-
(140 380,	CATALYST [Wei+13]	-83	-59	0	2.07	1 259	0:07:53
240 ps)	TCGRLin (fractional)	-58	-13	0	1.88	1 228	0:01:36
	TCGRLin (final)	-58	-12	0	1.86	1 229	0:04:43
U4	"Bounds"	-44	-6	4	5.04	938	-
(156 800,	CATALYST [Wei+13]	-72	-36	4	5.10	1 700	0:09:52
264 ps)	TCGRLin (fractional)	-52	-10	53	5.18	1 738	0:05:22
	TCGRLin (final)	-52	-9	53	5.20	1 816	0:10:06
U5	"Bounds"	-306	-457	0	9.78	952	-
(165 379,	CATALYST [Wei+13]	-782	-3 163	12	10.78	2 191	0:12:00
274 ps)	TCGRLin (fractional)	-307	-755	1 002	9.90	1 499	0:02:16
	TCGRLin (final)	-306	-682	0	9.91	1 614	0:06:43
U6	"Bounds"	-6	-0	6	8.73	1 172	-
(304 718,	CATALYST [Wei+13]	-30	-0.3	6	9.29	1 877	0:07:35
790 ps)	TCGRLin (fractional)	-40	-0.2	9	8.87	1 792	0:02:17
	TCGRLin (final)	-30	-0.1	7	8.88	1 949	0:08:35
U7	"Bounds"	-99	-12 709	16	13.51	1 461	-
(355 234,	CATALYST [Wei+13]	-100	-12 766	42 000	15.18	2 918	0:23:07
208 ps)	TCGRLin (fractional)	-102	-12 881	177	13.85	2 579	0:08:21
	TCGRLin (final)	-100	-12 810	19	13.86	2 833	0:22:22
U8	"Bounds"	-237	-32	0	8.54	2 147	-
(361 684,	CATALYST [Wei+13]	-329	-88	124	8.59	2 982	0:13:32
184 ps)	TCGRLin (fractional)	-244	-46	0	8.83	2 985	0:04:05
	TCGRLin (final)	-248	-45	0	8.85	3 132	0:13:15
U9	"Bounds"	-36	-3	0	12.37	1 712	-
(413 199,	CATALYST [Wei+13]	-58	-26	0	13.26	3 140	0:11:12
208 ps)	TCGRLin (fractional)	-40	-13	0	12.60	3 016	0:03:20
	TCGRLin (final)	-38	-10	0	12.62	3 406	0:11:54
U10	"Bounds"	-116	-3	933	10.32	2 021	-
(477 869,	CATALYST [Wei+13]	-193	-227	1 860	11.11	3 265	0:12:40
208 ps)	TCGRLin (fractional)	-134	-42	1 734	10.93	3 302	0:03:55
	TCGRLin (final)	-125	-48	1 539	10.92	3 485	0:15:04
U11	"Bounds"	-80	-1 380	0	36.37	7 539	-
(1 257 242,	CATALYST [Wei+13]	-181	-1 868	9	36.79	10 993	0:55:45
184 ps)	TCGRLin (fractional)	-91	-1 479	0	37.02	10 983	0:16:23
	TCGRLin (final)	-85	-1 463	0	37.14	12 073	0:49:22

Table 6.1: Comparison between CATALYST [Wei+13] and our timing-constrained global routing algorithm with linear delay model and the topology embedding algorithm of Theorem 6.2 as block solver (TCGRLin). All netlists were unbuffered and timing is evaluated with the linear delay model (Definition 6.1). For TCGRLin we show results for fractional solutions after the resource sharing phase and for and results of the final solutions.



(a) TCGRLin tries to use higher layers until the congestion target is met.



(b) Congestion of U7. Layer assignments in the CATALYST run create congestion.

Figure 6.6: Congestion plots for instances U4 (left) and U7 (right). White, green, and yellow indicate that an edge is used by a small fraction only. Orange and red edges are used to almost their full amount and purple edges indicate a violation of their resource capacity. The images show the maxima over all routing layers.

On all units TCGRLin is as least as good as the CATALYST algorithm with respect to timing. On U6 and U7 both algorithms compute solutions with a similar worst slack, on all other units, timing improvements were significant with 20 ps and more. On U3, U5, U7, U9, and 11, the achieved worst slacks are close to optimum.

Except for the routing-critical unit U10 (where a routing without overload does not seem to exist), TCGRLin could find a feasible or almost feasible solution with respect to routability. Small overloads on U2, U4, U6, and U7 exhibit inaccuracies of the global routing model at macro borders and when connecting pins that belong to macros. Typically, they do not impact detailed routability.

A congestion plot of U4 can be found in Figure 6.6(a). Here, every edge is colored according to the fraction to which its corresponding congestion resource is used. Light colors such as white, green, and yellow indicate that an edge is used by a small fraction only. Orange and red edges are used to almost their full amount and purple edges indicate a violation of their resource capacity. The images show the maxima over all routing layers. In the congestion plot of TCGRLin, most edges are printed in orange. In these areas, the highest layers are used by a fraction of up to 95% which is the congestion target for the designs. Many of these edges are yellow or green in the congestion plot for the CATALYST run which indicates that they are used by 80% or less.

The two units with the fewest nets (U1 and U2) are integration instances. Although the number of nets and the number of pins in each net is small, these instances are challenging global routing tasks as they contain many blockages and the distances between the pins are large. Similar to these integration designs, units U5, U7, and U10 have many blockages and contain nets with sinks that are far apart from each other. Here, good timing solutions can only be achieved if most parts of the long connections are realized on the highest available layers. Simultaneously, the many blockages make it hard to find a routable solution. Except for U7, the layer assignment algorithm failed to compute good solutions with respect to timing for these instances. On U2 and U7 the solution was not routable. In contrast to the layer assignment algorithm, the timing-constrained global routing algorithm could

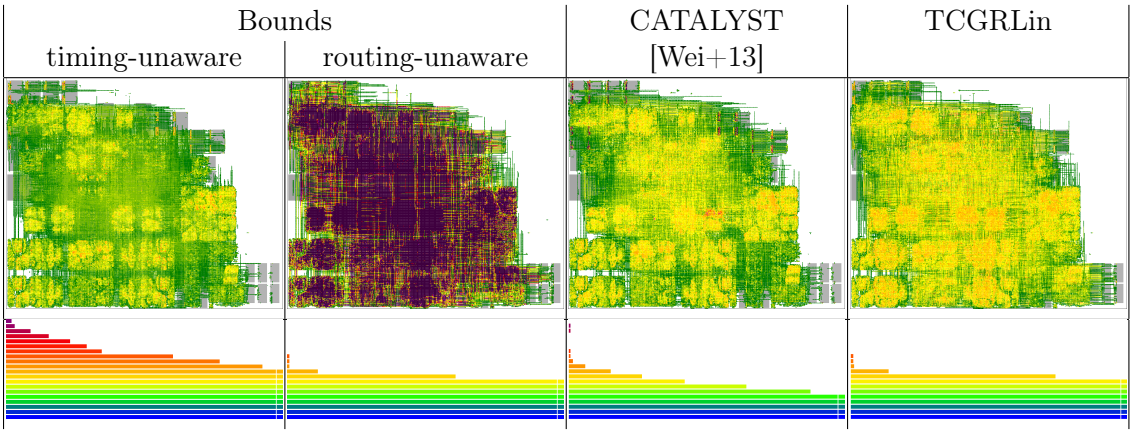


Figure 6.7: Congestion plots and slack histograms on instance U11.

avoid hot-spots by locally using lower layers in routing-critical areas. As an example of such a behavior congestion plots of U7 can be found in Figure 6.6(b). The CATALYST algorithm created large routing hot-spots while other parts show a small utilization of routing resources. The results suggest that especially on these integration instances the new approach is superior to the classical layer assignment algorithm.

With respect to wire length and vias none of the algorithms is dominating. On some units, selecting timing-driven topologies in TCGRLin increases the total wire length and the more flexible usage of different routing layers consumes more vias than the layer assignment algorithm. However, in many cases, layer assignments worsen routability a lot and many nets need to take detours to avoid congestion. On these units, the CATALYST runs produce more wiring length and need more vias.

The comparison between fractional and final solutions in TCGRLin shows that we can recover the quality of the fractional solution after rounding and rip-up and re-route. Since the fractional solution also contains global routes from early phases, integral solutions are sometimes even better in all metrics.

The results suggest that on unbuffered instances layer assignment approaches have a limited power to fulfill timing constraints. First, assignments of large nets to high layers are often inhibited as they create congestion by also forcing connections to uncritical sinks to the limited routing space on the assigned layers. Second, the timing-unaware global routing may choose unfavorable topologies. In contrast, the delay prices and the implicit delay bounds computed throughout TCGRLin ensure that the eventually critical connections are short and use layers that make fast signal propagation possible. The large timing benefit does not go along with larger routing overloads or higher running times.

A visual comparison between TCGRLin and CATALYST can be found in Figure 6.7. Below congestion plots in the upper image there are slack histograms that show the slack distribution of all gates, where each gate is represented by its worst slack. Here, the yellow bar represents all gates with slack zero, lower bars show uncritical gates that have positive slacks, and the upper bars represent gates with negative slacks.

The pictures show that the slack distribution of the result of TDGRLin is similar to the slack distribution of a solution that does not take routing into account at all. Layer utilizations are large with TDGRLin (larger than with e. g. a timing-unaware algorithm) but we do not create any violations of congestion resources. In contrast to that, the reference

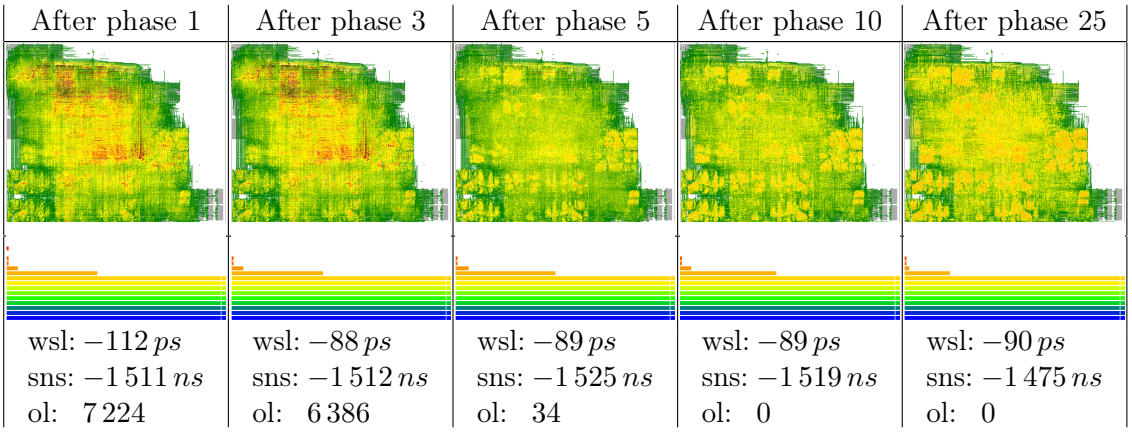


Figure 6.8: Development of timing and congestion on instance U11 during the resource sharing phases.

algorithm creates some local hot-spots at macro borders and ends up with sub-optimum timing. The pictures in Figure 6.8 show the progress of our algorithm during the resource sharing phases. Already after the first five phases we have eliminated most timing problems with an almost feasible fractional global routing. In the remaining phases, we fix the remaining timing violations, improve the sum of negative slacks, and remove the overflow on the few violated edges.

6.6 Port and Assertion Generation

Apart from the fact that Steiner trees computed by timing-constrained global routing with the linear delay model are a good basis for buffering, the linear delay model has another nice application in the *floor planning* step – one of the earliest parts of physical design.

6.6.1 Hierarchichal Design Flows

Due to the increasing complexity of computer chips, hierarchical approaches are very popular. Instead of designing a whole chip at once, it is subdivided into smaller units that are optimized individually. In a later step, solutions for the single units are combined to obtain a solution for the whole chip. In this step the units are considered as large macros with pre-defined ports that need to be connected to ports of other units and to primary ports of the top-level. Of course, these units can then be designed hierarchically again.

This way, the tremendous amount of design and computation work can be distributed to different designers and to different machines more easily. By reduction of instance sizes it becomes possible for the single designer and machine to put more effort into optimization. Hierarchical approaches also have advantages with respect to stability and turn-around time since local changes do no longer require to load data or to even re-compute solutions for the whole chip. Although the flexibility of optimization tools is limited by imposing hierarchy on a chip, results of hierarchical design flows are often even better than the results of a flat design. Explanations for this paradox can be found in the complexity of the chip design task. Since even most tiny sub-tasks that arise in the theory of VLSI design are *NP*-hard, optimization tools are non optimum. Reasonable subdivision into sub-units serve as a guide for heuristic approaches.

The success of hierarchy depends on the amount of interconnection between the sub-units and between the units and the top-level. If signal paths traverse several units, it is difficult to optimize the timing of a single unit without knowing the delay of the path through the remaining units. To enable timing optimization on a unit-level, *I/O-assertions* for the unit's ports need to be generated. These assertions include

- arrival times, capacitance limits, and output slews for primary input ports
- required arrival times, capacitances, and slew limits for primary output ports.

In Section 6.6.3 we show how to use timing-constrained global routing with the linear delay model to create these assertions.

As wiring inside the unit and wiring of the top-level interconnections share the same routing space, one has to be careful with routing space consumption. When we compute a routing of a sub-unit, we must not use all the routing space on high performance layers although that seems to be a good solution from the unit's perspective.

To enable the application of hierarchy, classical design flows have strictly divided the routing layers between unit-level and top-level. For a unit ceiling z that is strictly smaller than the number Z of available wiring planes, the unit internal routing is allowed to use layers $1, \ldots, z$ while layers $z+1, \ldots, Z$ above the unit are reserved for the top-level instances. A major drawback of this approach is that uncritical connections of the top-level nets are forced to use routing resources on high performance layers that must not be used for the critical nets on unit-level. Since the amount of wires that fit on the highest layers is usually small, congestion problems may arise. To overcome this issue, the macros that represent units on the top-level instance are often not packed as densely as possible. This way, an unnecessarily large amount of unused space on the placement layer is created.

Figure 6.9 shows a (simplified) example of a top-level instance. The spacing between the sub-units is quite large resulting in a large amount of unused placement space. No routing of the top-level nets uses routing space within the units while nets of the units are completely connected within the units. These connections are not shown in the picture. For the hierarchical design flow to be successful it is essential that all ports of the units (here depicted as black and gray boxes) have good assertions. The ports' positions also need to be determined beforehand and have a large impact on timing and routability.

6.6.2 Abutted Hierarchy and Port Assignment

To reduce the amount of unused placement space and to avoid the necessity of a priory separation of routing space we follow an abutted hierarchy approach. Development of this approach is joint work with Harald Folberth, Stephan Held, Pietro Saccardi, and Friedrich Schröder. Our main goal is to achieve that units can be placed without enforcing space in between.

We assume that we are given both positions and I/O-assertions for initial ports of the units. A netlist tells us how these ports are connected. As shown by Bartoschek et al. [Bar+10] we can determine an estimate $\rho(e)$ on the delay for traversing an edge e in the global routing graph in an almost optimally buffered netlist. Recall that we have already used these estimates in Sections 5.1.1 and 6.5.

We run a timing-constrained global routing with the linear delay model that is allowed to use the whole available routing space to connect the initial nets (Figure 6.10(a)). Unit internal connections can either be completely ignored in that step, or the unit-level nets

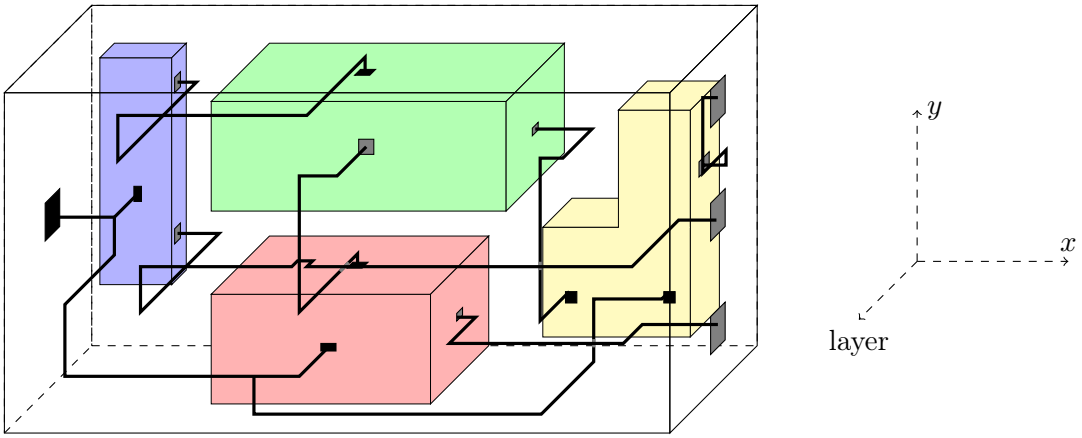


Figure 6.9: A top-level instance connecting ports of sub-units and ports of the top-level. Input ports are plotted black and output ports are gray. In a classical design flow wires connecting these ports may use layers above the units' ceilings and routing space in the gaps between units only. Routing space within the units is reserved for unit internal connections.

can be included into the netlist. If we decide for the latter solution, we enforce that unit internal connections do not cross their unit's boundary by restricting their routing area.

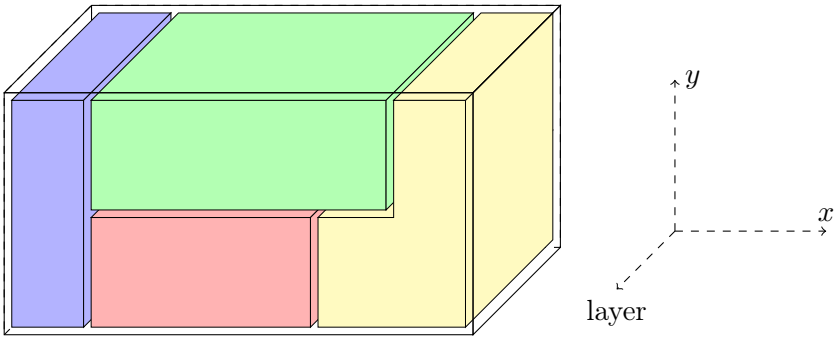
The solution of the initial timing-constrained global routing is used to replace the initial ports with new ports at all positions at which the Steiner trees (for top-level nets) cross unit borders. Simultaneously, the Steiner trees determine how these new ports are connected and hence define new nets that can be classified into the following three categories (Figure 6.10(b)):

1. Nets that connect an initial output port of a unit to a new output port at the border of the same unit. The new port replaces the initial port in a net of the unit's netlist. In Figure 6.10(b) and Figure 6.10(c), an example of such a net is marked with 1. The left gray port of the green unit is replaced by the new white port on the bottom left of the green unit.

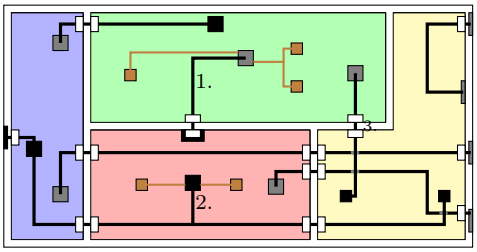
2. Nets that connect a new input port of a unit to new output ports of the same unit and to initial input ports of that unit. By replacing each initial input port to all pins to which this port is connected in the unit's netlist, we receive a new net that we add to the unit's netlist. In Figure 6.10(b) and Figure 6.10(c) such a situation is marked with 2.

3. Nets that connect new output ports of units (respectively primary input ports of the top-level instance) to new input ports of different units (respectively to primary output ports of the top-level). These nets form the new top-level instance. If the spacing between the units is indeed small, these nets are trivial. These trivial connections are drawn in blue in Figure 6.10(c). The edge marked by 3. is an example.

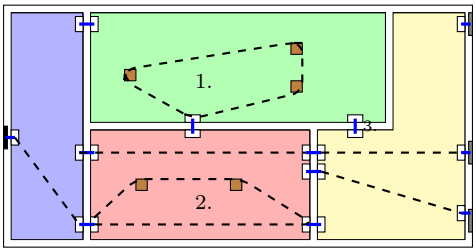
We obtain netlists for unit-level and top-level instances. Due to strict design rules it is usually forbidden to access ports by Steiner trees containing jogs, vias, or wire segments parallel to unit boundaries close to unit boundaries. To forbid these structures we restrict the routing space close to unit boundaries.



(a) Abutted units. Each of the units gets the full layer range.



(b) Initial ports (black and gray) with Steiner trees (black) that define the units' new ports (white). For better visibility we draw in 2 dimensions and draw connections to the interior of the unit for two ports only (brown).



(c) Nets after creation of new ports. Only nets within units (dashed) and trivial top-level nets (blue) remain.

Figure 6.10: Timing-constrained global routing can be used to define a new set of ports. By abutting units we reduce the amount of unused space on the placement layer and we do no longer require to divide routing space between unit-level and top-level.

6.6.3 Assertion Generation

The arrival time customer framework can be used to generate assertions for the newly created ports. Let (A, κ) be a Steiner tree for a top-level net N that crosses at least one unit border. Let s be its source. Arrival time customers yield an arrival time at s: If s is a primary port, the arrival time $\mathrm{at}(s)$ is given with the instance. Otherwise, the source port is a unit's output port and we can use the solution of the arrival time customers at the preceding input port plus the time it takes to propagate through the unit. Similarly, we obtain a required arrival time $\mathrm{rat}(t)$ at each sink $t \in N \backslash \{s\}$.

Let $s'_1, \ldots, s'_k$ be the points where (A, κ) crosses the source's unit border if s is an output port of a unit, and let $s'_1 = s$, $k = 1$ if s is a top-level port. For each sink $t \in N \backslash \{s\}$ that is neither a top-level port nor an input port of the source's unit let t' be the point nearest to t where $A_{[s,t]}$ crosses the border of the unit to which t belongs. For sinks that will not be moved during port assignment we set $t' := t$. We assign arrival times $\mathrm{at}(s'_i) := \mathrm{at}(s)$ for $i = 1, \ldots, k$ and $\mathrm{at}(t') := \mathrm{rat}(t)$ to the sinks. By propagating the linear delay values $\rho(e)$ from the s'_i along (A, κ) in topological ordering, we can define arrival times $\mathrm{at}(v)$ at the other points v where (A, κ) crosses unit borders. These values serve as assertions for arrival times and required arrival times for the corresponding ports.

Together with the delay-per-length values that define the values $\rho(e)$, Bartoschek et al. [Bar+10] compute default capacitances and slews for given layers and wire codes. At input ports these values can be used as assertions for capacitance limits and output slew.

At output ports they can be used as assertions for slew limit and capacitance.

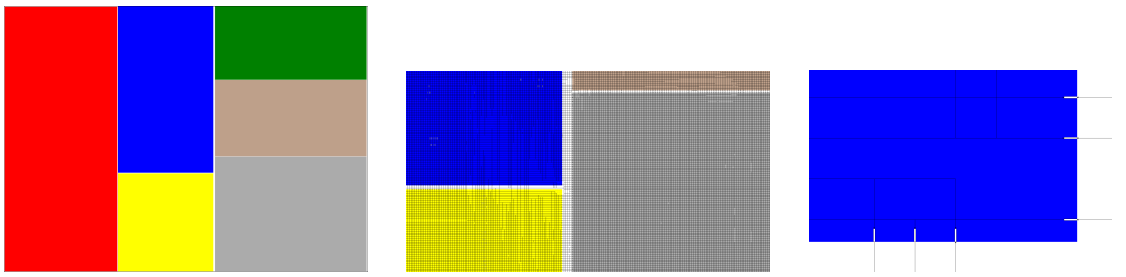
Let $t \in N\backslash\{s\}$ and let $s' \in \{s'_1, \ldots, s'_k\}$ such that t' is reachable from s' in A. One can think of several improvements to refine assertions depending on the slack $\mathrm{wsl}(t') := \mathrm{at}(t') - \mathrm{at}(s') - \mathrm{linear\_delay}_{(A,\kappa)}(s', t')$.

Distributing positive slacks. In case of positive slack on a path we distribute the slack to all units. We compute a delay scaling factor $\alpha > 1$ such that $\max_{t \in N\backslash\{s\}}\{\mathrm{wsl}(t')\}$ becomes zero if we increase all delays by a factor of α. It is also possible to scale delays non-uniformly.

Distributing negative slacks. In case of negative slacks, we distribute the slack between source and sink units. In all other units, and for top-level connections, negative slack is usually unwanted. To accomplish this, we increase $\mathrm{at}(t')$ by $-\frac{\mathrm{wsl}(t')}{2}$ for all $t \in N\backslash\{s\}$ with $\mathrm{wsl}(t') < 0$. For $i = 1, \ldots, k$ we decrease $\mathrm{at}(s'_i)$ by $\frac{1}{2} \cdot \max\{-\mathrm{wsl}(t') : t \text{ reachable from } s'_i \text{ in } A\}$. This method can be refined further by distributing the negative slack non-uniformly between the source and the sink continent.

Improved assertions by buffered Steiner trees. Instead of computing assertions based on the linear delay model, we can directly compute buffered Steiner trees. Instead of using default values for capacitance and slew, we can propagate the actual slews, slew limits, capacitances and capacitance limits along the fully buffered Steiner tree and obtain more accurate assertions. When computing these buffered Steiner trees it is important that these are electrically correct and have a good timing with respect to a delay model that measures slew effects. Buffered Steiner trees computed with the algorithms presented in Chapter 4 do not fulfill these requirements. In Chapter 7 we show how to compute a low-cost buffering of a given Steiner tree. In addition to routing and placement congestion, these costs model penalties for capacitance and slew violations, and for slew-dependent delays.

The methods described in this section are not only of theoretical interest but are actually used in practice at IBM. An example output of a real-world top-level design for which ports and assertions have been created as described are shown in Figure 6.11.



(a) Top-level view on the design. Each unit is plotted in a different color. The top-level routing is not shown.

(b) Zoom into the top-level routing shows distribution of the wires that define ports.

(c) Zoom into Figure 6.11(b) shows created ports (white boxes).

Figure 6.11: Real-world output of port assignment. Pictures are shown in 2-dimensions although the used global routing graph is a 3-dimensional graph.

Chapter 7

Buffering a Given Steiner Tree

Buffering algorithms in today's physical design flows typically work in two steps.

1. In a first step a Steiner tree or a topology is computed.
2. This Steiner tree or topology is then buffered.

Since we already discussed algorithms for the computation of topologies (Chapter 5) and Steiner trees (Chapter 6) in previous chapters, we concentrate on the second step in this chapter. As before we try to minimize costs for delays, routing congestion, placement congestion, and other costs including costs for net length and power. We also try to obey capacitance limits and slew limits.

After having recalled existing buffering algorithms (Section 7.2) we will introduce a new approach to realize Step 2. This algorithm unites classical dynamic programming algorithms (e. g. Van Ginneken [Van90]) with the *Fast Buffering Algorithm* by Bartoschek [Bar14] and Bartoschek et al. [Bar+09]. We discuss two versions of it. In a *basic version* we propagate many candidates and obtain good solutions. We show how to make use of information based on repeater library and input characteristics to prune most candidates without degrading the results too much. We obtain a *fast mode* of our algorithm that can be used even for large designs.

After the bicriteria algorithm of Section 5.3 to compute a routing topology and the topology-embedding algorithm contained in the proof of Theorem 6.2, the algorithm presented in this chapter is the last building block of a (heuristic) buffering-and-routing oracle that can be used in the resource-sharing-based timing-constrained global routing framework introduced in Chapter 3. This oracle will be fast enough for practical application.

7.1 The Minimum Cost Steiner Tree Buffering Problem

In this section we give a detailed formulation for the *Minimum Cost Steiner Tree Buffering Problem*. Since this chapter is targeted at improving results on *practical* VLSI instances we have to make several deviations from the setting in the previous chapters. In particular, we no longer make use of the simple version of Elmore Delay model ([Elm48]) described in Section 4.2.1 that does not consider slew effects.

7.1.1 Connecting the Detailed Pin Shapes

The first deviation concerns the global routing graph. Recall that the standard global routing graph G_{coarse} introduced in Section 2.4.3 has a vertex for each tile of a partition of the routing area and joins two vertices if they represent adjacent tiles. All pins of a net are mapped to a graph node representing a tile that has non-empty intersection with that pin. We call these nodes *global* pins. From a timing point of view this strategy has many drawbacks.

- If the distance between the nodes representing source and sink of the same net is larger than the actual distance between the pins, we might insert unnecessary repeaters.
- If the distance between global pins is shorter than between the corresponding detailed pins, we underestimate capacitances and slews. This can lead to timing degradations and electrical violations.

This situation is even worse considering the fact that nodes to which we can assign repeaters need a position in the coarse graph G_{coarse}. Refining the partition would certainly decrease these over- or underestimates but can also increase running times.

To avoid this problem we do not restrict ourselves to G_{coarse} during buffering. Instead, we consider the infinite graph G_{fine} with vertex set $V(G_{\text{fine}}) = [0, W] \times [0, H] \times \{1, \dots, Z\}$, where $W \in \mathbb{R}_{>0}$ is the width, $H \in \mathbb{R}_{>0}$ is the height of the chip image, and $Z \in \mathbb{N}$ is the number of wiring planes. We have an edge between two nodes (x, y, z) and (x', y', z') in G_{fine} if they have exactly two coordinates in common. For simplicity, we also add edges between any pair of points in the same tile on the same routing layer – even if we create jogs that way. As before, different wire code can be modeled as parallel edges in G_{fine}.

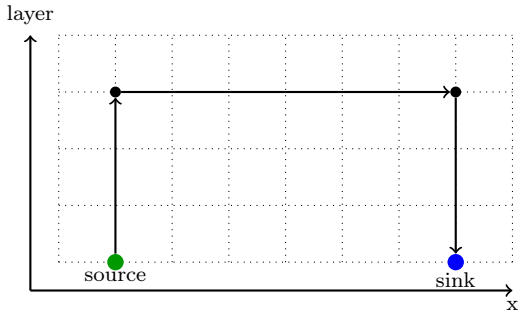
It is easy to transform a Steiner tree $(A_{\text{global}}, \kappa_{\text{global}})$ in G_{coarse} connecting the global pins into a Steiner tree (A, κ) in G_{fine} that connects the original shapes such that the number of crossings of tile borders are identical for both trees. Kiefner [Kie16] has shown how to minimize tree length as first and delays as secondary objective.

As before, we measure congestion at tile borders (that are represented by edges in G_{coarse}). When we query congestion cost or consumption of an edge $e = \{(x, y, z), (x', y', z')\}$ $\in E(G_{\text{fine}})$ there are two cases. Either e connects points in the same tile, and hence does not consume from congestion resources, or e is a straight path segment and consumes from all congestion resources of tile borders that are crossed by e. Using this definition of congestion for edges in G_{fine}, $(A_{\text{global}}, \kappa_{\text{global}})$ and (A, κ) consume exactly the same amount from all congestion resources.

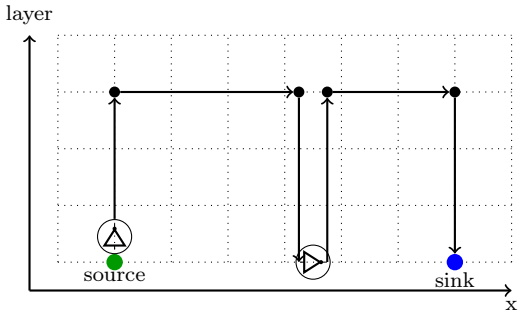
When we discuss algorithms to buffer a given Steiner tree we rather think of Steiner trees in G_{fine} instead of G_{coarse}.

7.1.2 Steiner Tree Transformations

Input to the *Minimum Cost Steiner Tree Buffering Problem* is a Steiner tree (A, κ), say in the graph G_{fine} defined in the previous section. In principle we would like to obtain a *buffered* Steiner tree $((A, \kappa), b)$ as defined in Section 4.1. However, just computing a function $b : V(A) \to L$ for a discrete repeater library L would be insufficient due to formal reasons.



(a) Initial Steiner tree consisting of a long path on high layers. We cannot insert repeaters into the Steiner tree directly. We assume that the sink has ident polarity.



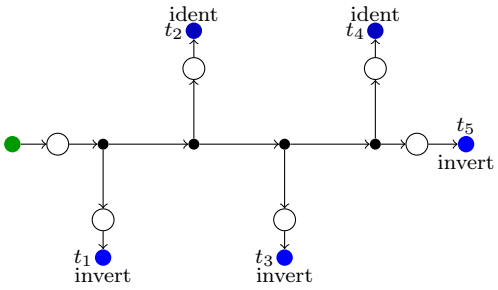
(b) We subdivide the upper path three times and place the middle Steiner node on the placement layer. We assign an inverter to that node. To meet the polarity constraint we have to insert a second inverter. We subdivide the edge leaving the source by a Steiner node at root's position and assign the second inverter to it.

Figure 7.1: During buffering we allow subdivisions of the initial Steiner tree.

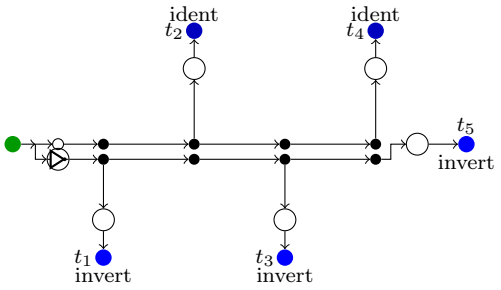
- It is possible that function κ does not map any Steiner node to a graph node in which buffer insertion is allowed. For example, (A, κ) could be a 3-dimensional rectilinear Steiner tree for which all wires in x- or in y- direction use a high layer. Without allowing insertion of paths containing Steiner points mapped to graph nodes on which buffer insertion is allowed (e. g. nodes on the lowest layer), we cannot insert any repeater and cannot even output a logically correct solution. Such a situation can be found in Figure 7.1.

- Edges in (A, κ) could be long. Again, we must allow to subdivide edges to achieve good results with respect to timing and to meet all polarity constraints. As before, Figure 7.1 depicts such a situation.

- Sometimes, polarity constraints force us to insert many inverters. To overcome that problem we allow local topology changes as shown in Figure 7.2.



(a) Initial Steiner tree for a net with 5 sinks. Sinks t_2 and t_4 have ident polarity; t_1, t_3, and t_5 have invert polarity. Without topology changes we have to insert at least three inverters.



(b) After local topology changes there is a legal buffering with one inverter.

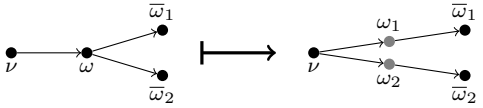
Figure 7.2: In order to save inverters we allow local topology changes during buffering. In this picture we assume that we may assign repeaters to the white Steiner nodes.

The following definition summarizes the allowed transformations to the initial Steiner tree.

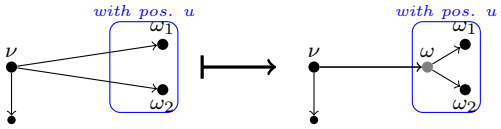
Definition 7.1 (Feasible modification) *We say that a Steiner tree (A', κ') is a feasible modification of a Steiner tree (A, κ) if we can reach (A', κ') from (A, κ) by successively applying the following operations:*

1. *Multiple subdivision of an edge in A (without changing the placement of the adjacent vertices)*

2. *Let $\omega \in V(A)$ such that $\delta_A^+(\omega) = \{(\omega, \overline{\omega}_1), (\omega, \overline{\omega}_2)\}$, and $\delta_A^-(\omega) = \{(\nu, \omega)\}$. Let ω_1 and ω_2 be new vertices placed at the position of ω. Replace ω and its incident edges by ω_1, ω_2, (ν, ω_1), (ν, ω_2), $(\omega_1, \overline{\omega}_1)$, $(\omega_2, \overline{\omega}_2)$.*



3. *Let $\omega_1, \omega_2 \in V(A)$ such that ω_1 and ω_2 have the same position u and the same predecessor ν. Insert a new vertex ω with position u and replace (ν, ω_1), (ν, ω_2) by (ν, ω), (ω, ω_1), (ω, ω_2).*



We say that Operation 2 *sets ω into parallel mode* while Operation 3 *resolves the parallel mode* at ω_1 and ω_2. Observe that the Steiner trees in Figures 7.1(b) and 7.2(b) are feasible modifications of the Steiner trees in Figures 7.1(a) and 7.2(a) respectively.

7.1.3 Elmore Delay Model with Slew Propagation

Due to its simplicity, the Elmore delay model (Section 4.2.1) is a frequently used delay model in timing optimization. For modern technologies the timing impact of slew effects on delays is significant and ignoring them can lead to poor results. For that reason we extend the simple version of the Elmore delay model of Section 4.2.1 by slew propagation. We call the resulting delay model the *Elmore Delay Model with Slew Propagation*.

We now describe how to compute timing information of a buffered Steiner tree $((A, \kappa), b)$ for a net N with source s in a graph G. We can think of G being the graph G_{fine} introduced in Section 7.1.1. Recall the function $\kappa : V(A) \cup E(A) \to V(G) \cup (E(G) \cup \{\circ\})$ for a graph G as defined in Section 2.1. As in Section 4.1, a function $b : V(A) \to L \cup \{\square\}$ assigns (Steiner) nodes of A to repeaters of a finite repeater library L or to $\square$ if we do not insert a repeater at a node.

Instead of using a particular delay model directly, we assume that we are given delay functions

$$\begin{aligned} d_{\text{wire}} &: E(G) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, \\ d_{\text{repeater}} &: L \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, \text{ and} \\ d_{\text{source}} &: \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}. \end{aligned}$$

The above functions are treated as black-box functions but we assume that d_{source} is monotonically increasing, and d_{wire} and d_{repeater} are monotonically increasing in the second and the third argument (when the other arguments are fixed). We will explain later how these functions yield a delay model.

Capacitances. The last input of all delay functions is a *capacitance* that can be computed by propagation in reverse topological order (see Sections 2.5.4 and 4.2.1). We assume given capacitances $\text{cap}(t)$ for all sinks $t \in N\backslash\{s\}$ as well as the input capacitance $\text{cap}(l)$ for all repeaters $l \in L$. A function $\text{cap} : E(G) \to \mathbb{R}_{\geq 0}$ determines the capacitance increase along an edge in G. For wiring edges the latter function is usually just the product of the length of the edge and a value that determines the *capacitance increase per length*. This value depends on the edge's wire code and layer. For via edges the capacitance increase is a constant that also depends on the wire code and the connected layers. For the sake of a simpler notation we define $\text{cap}(\circ) := 0$.

We recursively define the capacitance of a node $\nu \in V(A)\backslash(N\backslash\{s\})$ by

$$\text{cap}(\nu) := \begin{cases} \text{cap}(b(\nu)) & \text{if } b(\nu) \in L \\ \sum_{(\nu,\omega)\in\delta_A^+(\nu)} \text{cap}(\omega) + \text{cap}(\kappa((\nu,\omega))) & \text{otherwise} \end{cases}$$

and the downstream capacitance of a node ν that is associated with a repeater as

$$\text{outcap}(\nu) := \sum_{(\nu,\omega)\in\delta_A^+(\nu)} \text{cap}(\omega) + \text{cap}(\kappa((\nu,\omega))).$$

Capacitance violations. Source pins of all repeaters and of the net N itself have capacitance limits. These are given by a function $\text{caplim} : \{s\} \cup L \to \mathbb{R}_{\geq 0}$. We denote the total amount of capacitance violations by

$$\begin{aligned} \text{capvio}(((A,\kappa),b)) := {} & \max\{0, \text{cap}(s) - \text{caplim}(s)\} \\ & + \sum_{\nu\in V(A):b(\nu)\in L} \max\{0, \text{outcap}(\nu) - \text{caplim}(b(\nu))\}. \end{aligned}$$

Slews. Wire and repeater delays (d_{wire}, d_{repeater}) can depend on slews. To propagate slews we assume functions

$$\text{outslew} : (E(G) \cup L) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \quad \text{and} \quad \text{outslew}_{\text{source}} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}.$$

As before, we assume that $\text{outslew}_{\text{source}}$ is monotonically increasing and outslew is monotonically increasing in the second and the third argument. After having computed $\text{cap}(\nu)$ for all $\nu \in V(A)$ and $\text{outcap}(\nu)$ for $\nu \in V(A)$ with $b(\nu) \in L$ we recursively set

$$\text{slew}(\nu) := \begin{cases} \text{outslew}_{\text{source}}(\text{cap}(s)) & \text{if } \nu = s \\ \text{outslew}(\kappa((\omega,\nu)), \text{slew}(\omega), \text{cap}(\nu)) & \text{if } \delta_A^-(\nu) = \{(\omega,\nu)\} \text{ and } b(\nu) = \square \\ \text{outslew}(b(\nu), \text{inslew}(\nu), \text{outcap}(\nu)) & \text{otherwise,} \end{cases}$$

where $\text{inslew}(\nu)$ is defined as $\text{outslew}(\kappa((\omega,\nu)), \text{slew}(\omega), \text{cap}(\nu))$ for Steiner nodes ν associated with repeater $b(\nu) \in L$ and having entering arc (ω,ν) (cf. Section 2.5.4).

Slew limits and slew violations. Similar to capacitance limits we have to obey slew limits at all sink pins. Note that the set of sink pins include input pins of the newly inserted repeaters and obeying these limits means bounding the value of inslew(ν) for a Steiner node $\nu \in V(A)$ assigned to a repeater $b(\nu) \in L$. Let

$$\text{slewlim} : (N \backslash \{s\}) \cup L \to \mathbb{R}_{\geq 0}$$

be the function that yields the slew limits of sinks of N and of the repeater's sinks. We denote the total amount of slew violations of $((A, \kappa), b)$ by

$$\begin{aligned}\text{slewvio}(((A,\kappa),b)) := &\sum_{t \in N \backslash \{s\}} \max\{0, \text{slew}(t) - \text{slewlim}(t)\} \\ &+ \sum_{\nu \in V(A), b(\nu) \in L} \max\{0, \text{inslew}(\nu) - \text{slewlim}(b(\nu))\}.\end{aligned}$$

During the algorithms considered in this chapter we often propagate slew limits through wires and repeater, i. e.

- For $(\nu, \omega) \in E(A)$ such that ω has capacitance cap(ω) and slew limit slewlim(ω), what is the largest slew at ν that does not cause a violation?
- For $\nu \in V(A)$ with $b(\nu) \in L$ such that the output pin of the repeater assigned to ν drives capacitance outcap(ν) and has slew limit slewlim(ν), what is the largest slew at the input slew of that repeater that does not create a slew violation?

As all slew functions are monotonically increasing with increasing input slew, we can answer both questions by binary search. Sometimes, it is even possible to let the timing engine propagate slew limits directly. In this chapter we just assume that we can efficiently propagate slews both backwards and forwards along (sub-trees of) buffered Steiner trees.

Wire delay. Let $\zeta = (\nu, \omega) \in E(A)$. If $\kappa(\zeta) = \square$, its delay is 0. Otherwise, $\kappa(\zeta) \in E(G)$ and we define the delay of traversing e as $d_{\text{wire}}(\kappa(\zeta), \text{slew}(\nu), \text{cap}(\omega))$.

Repeater delay. Let $\nu \in V(A)$ such that $b(\nu) \in L$. The delay needed to traverse the repeater assigned to ν is $d_{\text{repeater}}(b(\nu), \text{inslew}(\nu), \text{outcap}(\nu))$.

Source delay. The delay $d_{\text{source}}(\text{cap}(s))$ measures the delay through the source gate (if s is an output pin of a logic gate). That delay depends on the slews of the input pins of the gate. Since these slews do not depend on N, we treat them as being encoded in function d_{source}. If s is a primary input or latch output, the functions d_{source} are usually the constant zero function.

Delay along a source-sink path. With the above definitions we can define the delay along the unique path $A_{[s,t]}$ from s to a sink $t \in N \backslash \{s\}$ as

$$\begin{aligned}\text{delay}_{((A,\kappa),b)}(t) := d_{\text{source}}(\text{cap}(s)) &+ \sum_{\substack{\zeta=(\nu,\omega)\in E(A),\\ \kappa((\nu,\omega))\neq\square}} d_{\text{wire}}(\kappa(\zeta), \text{slew}(\nu), \text{cap}(\omega)) \\ &+ \sum_{\substack{\nu\in V(A),\\ b(\nu)\in L}} d_{\text{repeater}}(b(\nu), \text{inslew}(\nu), \text{outcap}(\nu)).\end{aligned}$$

We omit to explicitly mention the dependence on the functions d_{wire}, d_{repeater}, d_{source}, cap, outslew, and $\text{outslew}_{\text{source}}$ in that notation.

7.1.4 Problem Formulation

With the previous sections we are now able to give a formal definition to the overall problem that we wish to solve in this chapter.

Minimum Cost Steiner Tree Buffering Problem

Instance: A repeater library L consisting of buffers and inverters.
A graph G with edge and placement costs $c : E(G) \cup V(G) \to \mathbb{R}_{\geq 0}$.
A net $N \subseteq V(G)$ with source s.
Sink polarities $\text{pol} : N\backslash\{s\} \to \{\text{ident}, \text{invert}\}$.
Delay costs $\lambda : N\backslash\{s\} \to \mathbb{R}_{\geq 0}$.
Power consumptions $\text{power} : L \to \mathbb{R}_{\geq 0}$ and a power price $c_{\text{power}} \in \mathbb{R}_{\geq 0}$.
Capacitance and slew violation penalties pen_{cap} and pen_{slew}.
A Steiner tree (A, κ) for N.
Timing functions

$$\begin{array}{rcll} \text{cap} & : & (N\backslash\{s\}) \cup L & \to \mathbb{R}_{\geq 0} \\ \text{caplim} & : & \{s\} \cup L & \to \mathbb{R}_{\geq 0} \\ \text{slewlim} & : & (N\backslash\{s\}) \cup L & \to \mathbb{R}_{\geq 0} \\ d_{\text{wire}} & : & E(G) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} & \to \mathbb{R}_{\geq 0} \\ d_{\text{repeater}} & : & L \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} & \to \mathbb{R}_{\geq 0} \\ d_{\text{source}} & : & \mathbb{R}_{\geq 0} & \to \mathbb{R}_{\geq 0} \\ \text{outslew} & : & (E(G) \cup L) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} & \to \mathbb{R}_{\geq 0} \\ \text{outslew}_{\text{source}} & : & \mathbb{R}_{\geq 0} & \to \mathbb{R}_{\geq 0}. \end{array}$$

Output: A buffered Steiner tree $((A', \kappa'), b)$ for N in G such that (A', κ') is a feasible modification of (A, κ) and for all $t \in N\backslash\{s\}$,

$$\left|\left\{\nu \in V(A'_{[s,t]}) : b(\nu) \in L \text{ and } b(\nu) \text{ inverter}\right\}\right| \text{ even} \Leftrightarrow \text{pol}(t) = \text{ident}.$$

Our goal is to minimize

$$\begin{aligned} &\sum_{\substack{\zeta \in E(A'),\\ \kappa(\zeta) \neq \circ}} c(\kappa(\zeta)) && + \sum_{\nu \in V(A')} c(\kappa(\nu)) \cdot \text{size}(b(\nu)) \\ &+ \sum_{t \in N\backslash\{s\}} \lambda(t) \cdot \text{delay}_{((A',\kappa'),b)}(t) && + c_{\text{power}} \cdot \sum_{\substack{\nu \in V(A'),\\ b(\nu) \in L}} \text{power}(b(\nu)) \\ &+ \text{pen}_{\text{slew}} \cdot \text{slewvio}(((A', \kappa'), b)) && + \text{pen}_{\text{cap}} \cdot \text{capvio}(((A', \kappa'), b)). \end{aligned}$$

In this definition we optimize *static* power only. As described in Section 2.5.6, optimizing dynamic power is also easily possible (see Section 2.5.6).

Dealing with capacitance and slew violations does not really fit into the resource sharing model (Section 3.2). In principle one would like to define the feasible solutions for a net customer as the set of (convex combinations of) buffered Steiner trees for the nets without any violations.

However, this is not always possible:

- An input slew of the source gate could be large and introduce slew violations at the subsequent sink pins in *any* buffered Steiner tree.
- An input capacitance of a sink pin could exceed the capacitance limit of the source of N, and of any repeater.

In the definition of the *Minimum Cost Steiner Tree Buffering Problem* we allow to produce capacitance and slew violations but pay a price if we do so. We can think of the penalties pen_{cap} and pen_{slew} as large numbers with $\text{pen}_{\text{cap}} > \text{pen}_{\text{slew}}$. In particular, they should be larger than the resource prices of any resource such that we would rather violate resources than introducing violations.

7.2 Previous Work

The particular problem formulation of Section 7.1.4 arises from the need of a *cost*-based buffering algorithm that we can use as a block solver for the resource sharing algorithm (Chapter 3) together with the Steiner tree algorithms from Chapter 6.

Most buffering algorithms known in the literature rather target at maximizing the worst slack

$$\min_{t \in N \setminus \{s\}} \left\{ \text{rat}(t) - \text{at}(s) - \text{delay}_{((A,\kappa),b)}(t) \right\},$$

where $\text{rat}(t)$ is the required arrival time at t and $\text{at}(s)$ is the arrival time at s.

7.2.1 Buffering by Dynamic Programming

The most natural and commonly used algorithmic paradigm to achieve that goal is *dynamic programming*. The first version of a dynamic programming algorithm for a buffering problem has been developed by Van Ginneken [Van90]. His algorithm works in the case that delays are independent of slews (which is the case for the Elmore delay model of Section 4.2.1) and that L consists of one buffer only (and no inverter).

While traversing an initial tree (A, κ) in reverse topological order he computes sets of non-dominated candidates that define solutions for the sub-tree of A rooted at any node in $V(A)$. Tree (A, κ) is not transformed during the algorithm but assumed to be subdivided beforehand. Van Ginneken [Van90] shows that the total running time for computing all these sets and hence of his algorithm is $\mathcal{O}(|V(A)|^2)$.

In the following years there have been several extensions of Van Ginneken's dynamic programming approach. In 1996, Lillis et al. [LCL96] showed how to deal with a larger repeater library L containing both buffers and inverters. They obtained a total running time of $\mathcal{O}(|L|^2 \cdot |V(A)|^2)$. In the same paper they also proposed a way to minimize power consumption during buffer insertion. Without further effort, these ideas can be used to minimize costs for delay, wiring, and placement congestion.

The $\mathcal{O}(|L|^2 \cdot |V(A)|^2)$ running time of the algorithm by Lillis et al. has been reduced to $\mathcal{O}(|L|^2 \cdot |V(A)| \cdot \log(|V(A)|))$ by Shi and Li [SL05] by introducing a new pruning technique. Li and Shi [LS06] could get rid of the quadratic dependency on the library size. They achieved a running time of $\mathcal{O}(|L| \cdot |V(A)|^2)$. Li et al. [LZS12] presented a variant that has running time $\mathcal{O}(|L|^2|V(A)| + |L||V(A)||N|)$. If repeater library and net have constant size, this is a linear time algorithm.

Other extensions to Van Ginneken's core algorithm are related to the delay model. Alpert et al. [ADQ99] published a version that can be used for the Elmore delay model with slew propagation and even for *higher-order delay models*. Chen and Menezes [CM99] showed how to model noise effects.

One drawback of Van Ginneken style algorithms is that they do not make any transformations to the initial tree. A natural approach is to cut each wire of the input tree into small and uniformly sized pieces. Choosing small pieces improves the quality of dynamic programming algorithms on cost of a higher running time. Alpert and Devgan [AD97] presented a strategy for wire segmentation that yields a compromise between running time and quality. Alpert et al. [AHQ04] showed how to choose potential repeater positions based on the library and on properties of the pins of the net.

Dynamic programming can also be applied to compute a buffering of a given Steiner tree that (approximately) minimizes static power consumption or other costs associated with each repeater type. The algorithm of Hu et. [Hu+07] minimizes a cost function

$$\sum_{\nu \in V(A),\, b(\nu) \in L} \text{power}(b(\nu)) \tag{7.1}$$

under the constraint that the input slew at each sink and repeater does not exceed a certain constant α. To compute slews, they use the simplified slew model by Kashyap et al. [Kas+04].

The more natural task of finding a cheapest solution (w. r. t. cost function (7.1)) that meets required arrival time constraints at every sink has been proved to be *NP*-hard by Shi et al. [SLA04]. This is true even if the cost function only attains integral values and if delays are measured with the version of the Elmore delay model in which slew effects are ignored.

Hu et al. [HLA09] gave a fully polynomial time approximation scheme for the task identified as NP-hard by Shi et al. [SLA04]. For $\epsilon > 0$ they compute a solution with the following properties.

- The total cost (7.1) is at most $1 + \epsilon$ times larger than a cheapest timing-feasible solution.
- The Elmore delay along each source-sink path does not exceed $1 + \epsilon$ times the largest delay that meets the required arrival time constraint for that sink.

The running time of their algorithm is

$$\mathcal{O}\left(\frac{|N|^2 \cdot |V(A)|^2 \cdot |L|^2}{\epsilon^3} + \frac{|N|^3 |L|^2}{\epsilon}\right).$$

Romen [Rom15] extended this algorithm to general costs in his master's thesis at the Research Institute for Discrete Mathematics, University of Bonn. This master's thesis was co-supervised by me. For $\epsilon > 0$ the algorithm by Romen [Rom15] computes a buffering of a given Steiner tree (A, κ) minimizing the weighted sum

$$\sum_{\nu \in V(A),\, b(\nu) \in L} \Big(c(\kappa(\nu)) \cdot \text{size}(b(\nu)) + c_{\text{power}} \cdot \text{power}(b(\nu))\Big) + \sum_{t \in N \setminus \{s\}} \lambda(t) \cdot \text{Elmore}_{(A,\kappa)}(t)$$

up to a factor $1 + \epsilon$ in time

$$\mathcal{O}\left(\log(|V(A)|) \cdot \left(\frac{|V(A)|^3 |N|}{\epsilon^2} + \frac{|V(A)|^3 |L|}{\epsilon}\right)\right).$$

7.2.2 The Fast Buffering Algorithm

A fast and successful approach for buffering is due to Bartoschek et al. [Bar+09], [Bar14]. Unlike the algorithms cited in Section 7.2.1, their *Fast Buffering Algorithm* allows modifications of the initial topology. Since the fast variant of the buffering algorithm presented in Section 7.3 is an extension to that algorithm, we have a closer look on their algorithm now. For a more detailed description we refer to [Bar14].

Roughly speaking, the algorithm of Bartoschek et al. [Bar+09], [Bar14] is a dynamic program that processes one single candidate only. The key ingredient to determine the most promising candidate is a careful *library pre-processing*.

For a given routing layer z, a wire code wc, and a value $\xi_{\text{power}} \in [0, 1]$, we pre-compute a buffering of a long path on layer z using wire code wc. All repeaters in that solution have the same type and are placed equidistantly. The buffering minimizes

$$\xi_{\text{power}} \cdot \text{“delay of the buffered path”} + (1 - \xi_{\text{power}}) \cdot \text{“power consumption of that path”}.$$

Bartoschek et al. [Bar+09][Bar14] show how to compute such a solution.

We call a triple $(z, \text{wc}, \xi_{\text{power}})$ *buffering mode* and the repeater type inserted in the corresponding repeater chain *default repeater* of a buffering mode. The stationary slews of the chains serve as *slew targets* during the main algorithm.

Let (A, κ) be an initial Steiner tree for a net N with source s in a 2-dimensional grid graph. During the algorithm a set of *clusters* is propagated bottom-up along A. Formally, a cluster C is a triple $(S(C), P(C), M(C))$. $S(C)$ is the set of sink pins that correspond to C. That set can contain sink of N and input pins of previously inserted repeaters, and is partitioned into (possibly empty) subsets $S^+(C)$ and $S^-(C)$ that differ by polarity. In the case that both $S^+(C)$ and $S^-(C)$ are non-empty sets, we might want to *merge* both parts. Due to polarity restrictions this can only be done by inserting an inverter driving either $S^-(C)$ or $S^+(C)$. Position $P(C)$ is a possible position for such an inverter. If $S^+(C) = \emptyset$ or $S^-(C) = \emptyset$, position $P(C)$ is not needed and can be chosen arbitrarily. We say that a cluster C with $S^+(C) \neq \emptyset, S^-(C) \neq \emptyset$ is in *parallel mode*. The object $M(C)$ is a buffering mode assigned to C. Additionally, we keep track of required arrival times, slew limits, and slew targets at the sinks of all clusters.

Initially, we create clusters $(S, \kappa(\nu), M)$ for all $\nu \in V(A)$ with $S = \begin{cases} \{\nu\} & \text{if } \nu \in N \backslash \{s\} \\ \emptyset & \text{otherwise.} \end{cases}$

Buffering modes M are selected according to the criticality of ν. Bartoschek et al. [Bar+09] [Bar14] proposed a *Min-Cost-Flow*-based approach to assign a buffering mode to the initial clusters. The default slew of the chosen buffering mode serves as a slew target for the initial cluster.

We process (A, κ) in reverse topological order until we reach source s. While we do so, we move and merge clusters until there is only one remaining cluster at s.

We also modify (A, κ). Apart from the feasible transformations described in Section 7.1.2, we replace parts of (A, κ) connecting a set S of sink pins and a Steiner node $\nu \in V(A)$ by an approximately shortest rectilinear Steiner tree for $S \cup \{\nu\}$ whenever we assign a repeater to a node ν. For details on these transformations we refer to [Bar14]. Note that *Fast Buffering* operates in 2-dimensions and, formally, repeater insertion includes insertion of via stacks to reach the placement layer. The different layer and wire code characteristics are encoded in the buffering modes.

Move. Let $\zeta = (\nu, \omega) \in E(A)$ and let C be a cluster at ω. While processing ζ we move C to position $\kappa(\nu)$. First, we extract a short Steiner tree for $S^+(C) \cup \{\omega\}$ (unless $S^+(C) = \emptyset$), and $S^-(C) \cup \{\omega\}$ (unless $S^-(C) = \emptyset$) with root ω.

For a point p on the straight line segment between $\kappa(\nu)$ and $\kappa(\omega)$ we extend either of these trees by two new vertices ν_1, ν_2, and edges (ν_1, ν_2), (ν_2, ω). For both trees, the position of both new vertices will be p and we assign the default repeater for buffering mode $M(C)$ to ν_2. By backward propagation along these trees assuming the slew target at the sources ν_1 and ν_2, we check if we meet slew limit and slew target (for $M(C)$) at these sources. Using binary search we can approximate the point p^* closest to $\kappa(\nu)$ for which that is the case (respectively $p^* = \kappa(\omega)$ if such a point does not exist).

If $p^* = \kappa(\nu)$, we move C to position p^*, update timing values at C, and *merge* C with the cluster at ν. The merge step will be described in the next paragraph.

If $p^* \neq \kappa(\nu)$, we insert a repeater. If both $S^+(C)$ and $S^-(C)$ are non-empty, we insert an inverter at position $P(C)$, either driving all sinks in $S^+(C)$ or in $S^-(C)$. We obtain a new cluster with sinks $S^-(C) \cup \{\chi\}$ or $S^+(C) \cup \{\chi\}$, where χ is the sink pin of the newly inserted inverter. All sinks of the new cluster have the same polarity and we continue to move it to position $\kappa(\nu)$. Among all $2|L|$ possibilities to insert a repeater in one side of the parallel cluster C we select the one lexicographically minimizing

- the total capacitance violations,
- the total slew violations,
- the following convex combination for a parameter $\xi_{\text{buf}} \in [0, 1]$:
  $(1 - \xi_{\text{buf}}) \cdot \text{power}(\text{“repeater”}) - \xi_{\text{buf}} \cdot \text{“resulting worst slack”}$.

(7.2)

In the case that C is not in parallel mode, we insert a new repeater at an unblocked position closest to p^* and obtain a new cluster containing the repeater's sink pin only. As before, we select the repeater type minimizing (7.2).

Merge. After we have moved a cluster C to its *parent cluster* C', we perform a merge. We say that C' is the *parent cluster* of C if C is the cluster at a node $\omega \in V(A)$ and C' has node $\nu \in V(A)$ with $(\nu, \omega) \in E(A)$.

If $|S^+(C)| \cdot |S^-(C')| = |S^+(C')| \cdot |S^-(C)| = 0$, we can just add the sinks in $S(C)$ to $S(C')$. Otherwise, we select the solution minimizing (7.2) from a list of possible *merge configurations*. Examples of merge configurations are

- merge clusters without inserting a further repeater,
- resolve parallel mode of C by inserting a repeater at position $P(C)$,
- resolve parallel mode of C and C', and add a further repeater driving all sinks of the resolved cluster C.

Bartoschek ([Bar14] page 76, Figure 6.1 and Figure 6.9) listed and described 15 reasonable merge configurations and the situations in which they can be applied. Since we do not need the explicit configurations in this thesis, we omit details here.

If C' becomes a parallel cluster during the merge step, we update $P(C')$ to the current position.

Connect root. After we have reached the source, we make sure that all polarity constraints are met and connect the sinks of the root pin. To do this, we enumerate all possibilities to successively insert 0,1, or 2 repeaters in the last remaining cluster. Inserting

a repeater into a cluster C is done by extracting short rectilinear Steiner trees connecting the sinks of the sets $S^+(C)$ or $S^-(C)$ with a Steiner node at source position associated with the repeater. The repeater's input pin replaces the sinks driven by its output pin in $S(C)$.

Among the solutions for which all sinks that remain in the final cluster have ident polarity, we select the one minimizing (7.2). When evaluating a solution we also take timing and electrical violations into account that we obtain after propagating through the source gate.

7.3 An Algorithm for Cost-Based Buffering

In this section we describe an algorithm for cost-based buffering that uses ideas from the *Fast Buffering Algorithm* [Bar+09][Bar14] described in Section 7.2.2 and the classical dynamic programming buffering algorithms [Van90][LCL96] listed in Section 7.2.1.

The algorithm maintains and propagates a set of *candidates*, buffering and modifying an initial Steiner tree (A, κ) as it does so. There will be two flavors of the algorithm. In a basic version we create a possibly exponential number of candidates, leading to good solutions but high running times. By using elements of the *Fast Buffering Algorithm* we are able to prune or avoid computation of almost all candidates and achieve that their number is constant at each position. We show how to select *the right candidates* to gain large speed-ups without too large loss in quality. We refer to the second variant as *fast version*.

All results of this section concerning the basic version of the algorithm are joint work with Rodion Permin who in particular implemented a variant of it [Per16]. Permin worked on the cost-based buffering problem during his master's thesis at the Research Institute for Discrete Mathematics, University of Bonn, under my co-supervision.

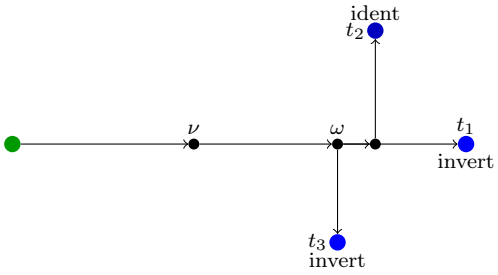
Before we describe the algorithm in detail, we define the basic data structures and operations such as *propagation*, *repeater insertion*, and *pruning*.

For this section we fix an instance for the *Minimum Cost Steiner Tree Buffering Problem* and use the notation of the problem formulation given in Section 7.1.4. Furthermore, we assume that the global routing graph G is equal to the graph G_{fine} described in Section 7.1.1 as this is the situation we have to solve in practice.

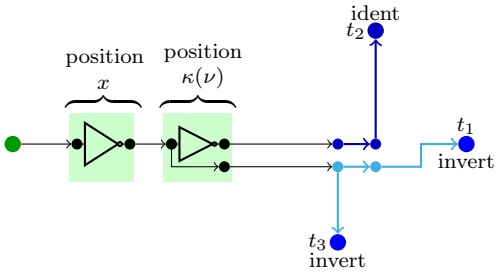
7.3.1 Candidates and Candidate Pairs

The algorithm maintains a set of *candidates* representing sub-trees in the final solution. A *candidate* C is associated with the set $\text{sinks}(C) \subseteq N \backslash \{s\}$ of sinks in that sub-tree, and a polarity $\text{pol}(C) \in \{\text{ident}, \text{invert}, \text{undefined}\}$ that specifies if the path between s and the root of the represented sub-tree must contain an even or an odd number of inverters to achieve that all sinks in $\text{sinks}(C)$ meet their polarity constraint. The first case is indicated as $\text{pol}(C) = \text{ident}$ and we have $\text{pol}(C) = \text{invert}$ in the second case. We define $\not\oslash$ to be the invalid candidate that represents the empty sub-tree with no sinks. We set $\text{sinks}(\not\oslash) = \emptyset$ and $\text{pol}(\not\oslash) = \text{undefined}$.

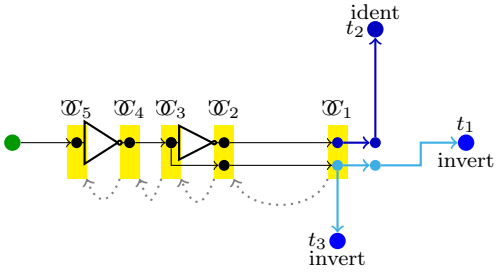
Candidates are created by bottom-up propagation along the initial Steiner tree (A, κ) and are linked with A by *candidate pairs*.



(a) Initial Steiner tree for a net with 3 sinks.



(b) Final buffered Steiner tree. For simplicity, we consider repeaters as points.



$\text{node}(\mathfrak{C}_1) = \omega, \quad p(\mathfrak{C}_1) = \kappa(\omega), \quad \text{type: parallel}$
$\text{node}(\mathfrak{C}_2) = \nu, \quad p(\mathfrak{C}_2) = \kappa(\nu), \quad \text{type: parallel}$
$\text{node}(\mathfrak{C}_3) = \nu, \quad p(\mathfrak{C}_3) = \kappa(\nu), \quad \text{type: invert}$
$\text{node}(\mathfrak{C}_4) = \nu, \quad p(\mathfrak{C}_4) = x, \quad \text{type: invert}$
$\text{node}(\mathfrak{C}_5) = \nu, \quad p(\mathfrak{C}_5) = x, \quad \text{type: ident}$

$\text{ident}(\mathfrak{C}_1) = \big(\{t_2\}, \text{ident}\big), \quad \text{invert}(\mathfrak{C}_1) = \big(\{t_1, t_3\}, \text{invert}\big)$
$\text{ident}(\mathfrak{C}_4) = \not{C}, \quad \text{invert}(\mathfrak{C}_4) = \big(\{t_1, t_2, t_3\}, \text{invert}\big)$
$\text{ident}(\mathfrak{C}_5) = \big(\{t_1, t_2, t_3\}, \text{ident}\big), \quad \text{invert}(\mathfrak{C}_4) = \not{C}$

(c) Some candidate pairs and candidates. Candidates $C \neq \not{C}$ are labeled $(\text{sinks}(C), \text{pol}(C))$.

Figure 7.3: Candidate pairs representing a final solution. Together with backtrace information indicated as gray arrows (e. g. "$\mathfrak{C}_2$ can be obtained from $\mathfrak{C}_1$ by inserting an inverter of type $l \in L$ at the ident part") we can obtain the solution from the candidates.

A *candidate pair* $\mathfrak{C}$ is a 4-tuple

$$\mathfrak{C} = (\text{node}(\mathfrak{C}), p(\mathfrak{C}), \text{ident}(\mathfrak{C}), \text{invert}(\mathfrak{C})),$$

where $\text{node}(\mathfrak{C}) \in V(A)$, $p(\mathfrak{C}) \in V(G)$ is $\kappa(\text{node}(\mathfrak{C}))$ or a point on the edge entering $\text{node}(\mathfrak{C})$, and $\text{ident}(\mathfrak{C})$, $\text{invert}(\mathfrak{C})$ are candidates.

As in the *Fast Buffering Algorithm* we allow modifications of the initial Steiner tree, in particular to set a node into parallel mode (see Definition 7.1, Item 2). As a consequence, the sub-tree of A rooted at ν can fall into two parts that differ by polarity in a possible final solution. These two sub-trees are represented by candidates $\text{ident}(\mathfrak{C})$ and $\text{invert}(\mathfrak{C})$. It is possible that one of these candidate is the invalid candidate $\not{C}$ but we require that $\text{pol}(\text{ident}(\mathfrak{C})) = \text{ident}$ if $\text{ident}(\mathfrak{C}) \neq \not{C}$ and $\text{pol}(\text{invert}(\mathfrak{C})) = \text{invert}$ if $\text{invert}(\mathfrak{C}) \neq \not{C}$. By this property we can define the *type* of a candidate pair:

Definition 7.2 (Type of a candidate pair) *A candidate pair $\mathfrak{C}$ has type*

- ***ident*** *if* $\text{ident}(\mathfrak{C}) \neq \not{C}$ *and* $\text{invert}(\mathfrak{C}) = \not{C}$,
- ***invert*** *if* $\text{ident}(\mathfrak{C}) = \not{C}$ *and* $\text{invert}(\mathfrak{C}) \neq \not{C}$,
- ***parallel*** *if* $\text{ident}(\mathfrak{C}) \neq \not{C}$ *and* $\text{invert}(\mathfrak{C}) \neq \not{C}$,

Given candidate pairs and backtrace information we can re-construct the represented solutions. An example how candidates and candidate pairs represent possible final solutions can be found in Figure 7.3. The solution shown in Figure 7.3(b) arises from the instance shown in Figure 7.3(a) by buffering. Five candidate pairs are shown in yellow boxes in Figure 7.3(c).

Quality of candidates. To be able to evaluate the quality of solutions we associate each candidate $C \neq \not{C}$ with

$\underline{\text{cap}(C)}$**:** the capacitance at the root of the sub-tree represented by C,

$\underline{\text{slewlim}(C)}$**:** the maximum slew such that no slew violation is induced in the represented sub-tree as long as the slew at its root is not larger than this value, and

$\underline{\text{cost}(C)}$**:** the cost of the sub-tree according to Definition 7.1.4.

While $\text{cap}(C)$ can be computed easily, exact computation of slew limits is impossible unless we know the slew of the sub-tree's root r. Similarly, computation of delays between r and the sinks in $\text{sinks}(C)$, and of slew violations requires knowledge about the slews (see Section 7.1.3). These delays and slew violations are needed to compute costs.

Instead of working with the exact values, we make use of the library pre-processing and Min-Cost-Flow-based buffering mode assignment by Bartoschek [Bar14] (see Section 7.2.2). This yields a target slew $\text{slew}_{\text{target}}$ at any node of the initial Steiner tree that we can use as an estimate on the root's slew.

If a candidate has a slew violation (in particular if $\text{slewlim}(C) < 0$), it will no longer be possible to obtain candidates with a non-negative slew limit by propagating from it. Therefore, we relax violated slew limits by setting them to the slew target after we have paid the price for the violation by increasing $\text{cost}(C)$.

We define $\text{cap}(\not{C}) = \text{cost}(\not{C}) = 0$ and $\text{slewlim}(\not{C}) = \infty$ for the invalid candidate $\not{C}$.

Creation and propagation. Initially, we create a candidate $C(t)$ with $\text{sinks}(C(t)) = \{t\}$ and $\text{pol}(C(t)) = \text{pol}(t)$ contained in an initial candidate pair $\mathfrak{C}(t)$ with $\text{node}(\mathfrak{C}(t)) = p(\mathfrak{C}(t)) = t$ for each sink $t \in N \backslash \{s\}$.

All other candidates and candidate pairs can be obtained by

- propagation along a wire or via,
- repeater insertion, or
- merge with another candidate or candidate pair.

Note that for all these operations the resulting candidates and candidate pairs as well as all quality estimates can be obtained from their predecessors by a constant number of invocations of the timing functions presented in Section 7.1.3.

7.3.2 Dominance

To obtain an acceptable running time we have to keep the number of candidate pairs small. By the *NP*-hardness result of Section 4.5.1 that already holds in the case of 2-terminal nets and for the basic version of the Elmore delay model, we certainly won't be able to bound the number of created candidate pairs by a polynomial unless we accept to prune candidate pairs that could possibly lead to optimum solutions (or unless P=*NP*). However, in some cases, candidate pairs cannot lead to an optimum solution as they are *dominated*.

Definition 7.3 (Dominance of candidates) *Let C and C' be candidates with* $\text{sinks}(C) = \text{sinks}(C')$ *and* $\text{pol}(C) = \text{pol}(C')$. *$C$ is dominated by C' if*

$$\text{slewlim}(C) \leq \text{slewlim}(C'), \qquad \text{cap}(C) \geq \text{cap}(C'), \qquad \textit{and} \qquad \text{cost}(C) \geq \text{cost}(C').$$

Using Definition 7.3 we can define dominance of candidate pairs.

Definition 7.4 (Dominance of candidate pairs) *A candidate pair $\mathfrak{C}$ is dominated by a pair $\mathfrak{C}'$ if all of the following conditions hold:*

- $p(\mathfrak{C}) = p(\mathfrak{C}')$,
- $\text{node}(\mathfrak{C}) = \text{node}(\mathfrak{C}')$,
- $\text{ident}(\mathfrak{C})$ *is dominated by* $\text{ident}(\mathfrak{C}')$,
- $\text{invert}(\mathfrak{C})$ *is dominated by* $\text{invert}(\mathfrak{C}')$.

Formally speaking, if $\mathfrak{C}$ is dominated by $\mathfrak{C}'$, implementing the solution encoded by $\mathfrak{C}'$ instead of $\mathfrak{C}$ would never lead to a worse solution. This is not completely true since slew limits and delay costs are computed using an estimate $\text{slew}_{\text{target}}$ on the input slew, but we would still like to gain the running time benefit.

Even more, we will prune a candidate pair if it is *almost-dominated* by another pair.

Definition 7.5 (Almost-domination) *Let $\gamma_{\text{cost}} > 1$, $\gamma_{\text{cap}} > 0$, and $\gamma_{\text{slewlim}} > 0$ be fixed constants. We say that a candidate C is almost-dominated by a candidate C' if* $\text{sinks}(C) = \text{sinks}(C')$, $\text{pol}(C) = \text{pol}(C')$ *and*

$$\left\lfloor \frac{\text{slewlim}(C)}{\gamma_{\text{slewlim}}} \right\rfloor \leq \left\lfloor \frac{\text{slewlim}(C')}{\gamma_{\text{slewlim}}} \right\rfloor,$$
$$\left\lceil \frac{\text{cap}(C)}{\gamma_{\text{cap}}} \right\rceil \geq \left\lceil \frac{\text{cap}(C')}{\gamma_{\text{cap}}} \right\rceil,$$
$$\left\lceil \log_{\gamma_{\text{cost}}}(\text{cost}(C)) \right\rceil \geq \left\lceil \log_{\gamma_{\text{cost}}}(\text{cost}(C')) \right\rceil.$$

A candidate pair $\mathfrak{C}$ is almost-dominated by a pair $\mathfrak{C}'$ if

- $p(\mathfrak{C}) = p(\mathfrak{C}')$,
- $\text{node}(\mathfrak{C}) = \text{node}(\mathfrak{C}')$,
- $\text{ident}(\mathfrak{C})$ *is almost-dominated by* $\text{ident}(\mathfrak{C}')$,
- $\text{invert}(\mathfrak{C})$ *is almost-dominated by* $\text{invert}(\mathfrak{C}')$.

The rounding used in Definition 7.5 is similar to the rounding of costs into buckets used in the FPTAS of Section 4.5.2. To obtain better running times we also round capacitances and slew limits here.

During the algorithm we have to check if a new candidate pair is almost-dominated by any previously computed candidate pair and to determine the previous candidate pairs that are almost-dominated by the new pair. To do this fast, we sort the set of candidate pairs for the same node and at the same position by $\text{cap}(\text{ident}(.))) + \text{cap}(\text{invert}(.)))$. To check if a new candidate pair $\mathfrak{C}$ is almost-dominated, we scan through all previous pairs with total capacitance at most $\text{cap}(\text{ident}(\mathfrak{C}))) + \text{cap}(\text{invert}(\mathfrak{C})))$. The candidate pairs almost-dominated by $\mathfrak{C}$ must have a total capacitance of at least this value. During insertion of $\mathfrak{C}$ we make sure to maintain that sorting.

7.3.3 Infeasible Repeater Positions

Not all repeater types can be placed at all positions. In our setting $G = G_{\text{fine}}$ (Section 7.1.1) we assume that we are allowed to insert repeaters of any type in all positions on the lowest layer and that we are not allowed to insert any repeater on higher layers. Blockages on the placement layer can be modeled by infinite placement costs.

A natural operation for a buffering algorithm in this standardized setting is to create stacked vias leading from a position u on a higher layer to the point $u_{\downarrow}$ on the lowest

Instance: An instance of the *Minimum Cost Steiner Tree Buffering Problem.*
Output: A buffered Steiner tree.

① Subdivide edges of (A, κ) as described in Section 7.3.5.
② Pre-process the library and assign buffering modes as in Section 7.2.2.
③ Create initial candidate pairs representing sub-trees containing the sinks only.
④ Sort $V' = \{\nu \in V(A) : |\delta_A^+(\nu)| > 1\} \cup \{s\}$ in reverse topological ordering.
⑤ **for** $\nu \in V'$ **do**
⑥ **for each** $\zeta \in \delta_A^+(\nu)$ **do**
⑦ $P_\zeta :=$ maximal path in A starting with ζ without internal vertices in V'.
⑧ Apply the MOVE step of Section 7.3.6 to ζ.
⑨ $\overline{\mathfrak{O}}_\zeta :=$ set of final candidate pairs at ν.
⑩ Apply the MERGE step of Section 7.3.8 to merge the sets $\overline{\mathfrak{O}}_\zeta$ $(\zeta \in \delta_A^+(\nu))$.
⑪ Select and apply the best candidate pair as in Section 7.3.9.

Algorithm 9: Overview of the basic version of the algorithm for cost-based buffering described in Section 7.3.

layer that shares x- and y- coordinate with u, and creating stacked vias from $u_\downarrow$ to u after having placed a repeater at $u_\downarrow$. This way we can realize repeater insertion at all positions $u \in V(G)$ as a sequence of via propagations and a repeater insertion although insertion is forbidden on u. From a Steiner tree point of view, these operations result in a feasible modification of Type 1 as described in Definition 7.1. For an illustration of a repeater insertion on a higher layer see Figure 7.4(c). We are not allowed to insert the depicted inverter at position $\kappa(\nu)$ directly but can reach $\kappa(\nu)_\downarrow$ with stacked vias.

In a more general setting in which $G \neq G_{\text{fine}}$, inserting repeaters at forbidden positions can be modeled in a similar way by defining close points $u_\downarrow$ in which repeater insertion is allowed.

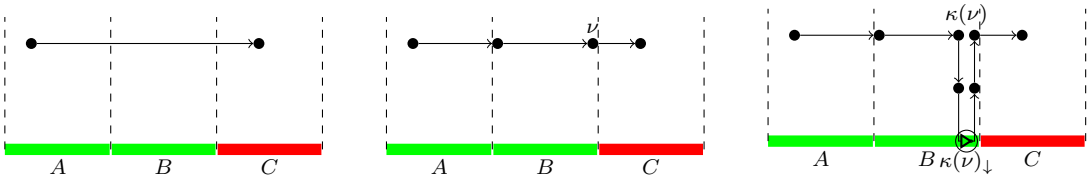
7.3.4 Overview of the Algorithm

We can now start with the description of the algorithm. A high-level description is given in this section and the details can be found in the subsequent sections.

After a pre-processing step (Section 7.3.5) and after having created initial candidate pairs representing sub-trees consisting of the sinks only (Section 7.3.1) we process the initial tree (A, κ) in reverse topological order. We use a Dijkstra-based [Dij59] move step to propagate candidate pairs along maximal paths with the property that all internal nodes have out-degree one (Section 7.3.6). At the Steiner nodes with out-degree at least two we merge candidate pairs (Section 7.3.8). After we have propagated all candidate pairs to the source of N we select a best candidate and obtain our final solution by backtracing (Section 7.3.9). Algorithm 9 shows a schematic overview.

7.3.5 Pre-Processing

The algorithm starts by dividing all wiring edges of (A, κ) crossing a tile border by a Steiner node placed in the tile with cheaper placement cost (see Figure 7.4). If both tiles have the same placement costs, we arbitrarily choose one of the adjacent tiles. Moreover, we subdivide the edges entering the sinks by a Steiner node at sink position. This enables



(a) Edge in the initial Steiner tree crossing two tile borders. The edge is a wiring edge on a higher layer. Placing a repeater in tile A or B is cheap while the placement resource price for tile C is large.

(b) We create Steiner nodes near each crossing point but make sure that the Steiner node ν near the crossing point of tile B and C belongs to the tile with cheaper placement cost (B).

(c) Placing a repeater at position $\kappa(\nu)$ can be simulated by a sequence of via insertions and a repeater insertion at point $\kappa(\nu)_\downarrow$ on the placement layer directly below $\kappa(\nu)$. We pay the placement resource price for tile B.

Figure 7.4: In the pre-processing step of the algorithm of Section 7.3 we subdivide all wiring edges to ensure that there are points near all tile border crossings. Placement resource prices are taken into account when crossing points are positioned.

us to place a repeater directly before a sink which is sometimes useful if sinks have a large capacitance. By subdividing the edge leaving the source s of net N two times and placing the new Steiner nodes at the sink's position, we will be able to obtain good results even if s is the output pin of a small gate. Recall that this situation is accounted for by the *root connect* step in the *Fast Buffering Algorithm*.

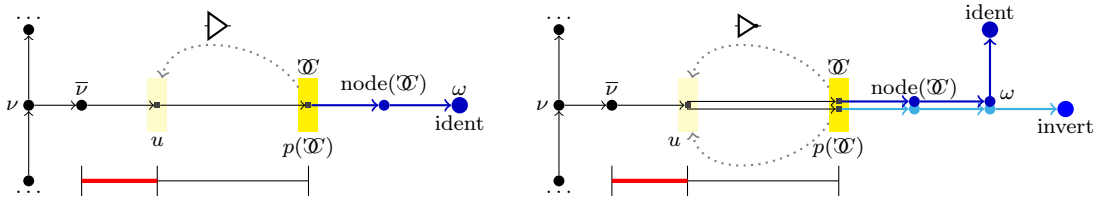
Additionally, we run the library pre-processing step and the buffering mode assignment of the *Fast Buffering Algorithm* (Bartoschek et al. [Bar+09], Bartoschek [Bar14], Section 7.2.2). As a result we obtain a slew target for each node of (A, κ) that we require for candidate propagation. We also store the (stationary) output capacitance of the repeaters in the long repeater chain computed during the library pre-processing for each buffering mode. This yields a capacitance target of which we make use during the move and merge step of the faster variant of the algorithm.

7.3.6 The Move Step

Let P be a maximal path in A with the property that none of its internal vertices has out-degree larger than 1. Let $\nu, \omega \in V(A)$ be its endpoints (i. e. $P = A_{[\nu,\omega]}$).

We use a Dijkstra-like [Dij59] algorithm to propagate initial candidate pairs with node ω and position $\kappa(\omega)$ along P to ν. If $\omega \in N \backslash \{s\}$, there is only one initial candidate pair: The pair representing the sub-tree consisting of ω only. If ω is a Steiner node with out-degree larger than one, the initial candidate pairs are a result of the merge step (see Section 7.3.8). All these initial candidate pairs are stored in a heap with key $\text{cost}(\text{ident}(\mathfrak{C})) + \text{cost}(\text{invert}(\mathfrak{C}))$.

While the heap is non-empty we do the following. First, we erase a candidate pair $\mathfrak{C}$ with minimum key from the heap. If $\text{node}(\mathfrak{C}) = \nu$, we mark it as a *final candidate pair*. Otherwise, we create new candidate pairs by propagation along the edge $e = \kappa((\overline{\nu}, \text{node}(\mathfrak{C})))$, where $(\overline{\nu}, \text{node}(\mathfrak{C}))$ is the edge entering $\text{node}(\mathfrak{C})$ in A. We create two types of new candidate pairs: candidate pairs that arise from $\mathfrak{C}$ by propagation along a wire or a via *without* inserting new repeaters and candidate pairs that arise by repeater insertion.



(a) Non-parallel mode. Creating a new candidate pair left of u would introduce an electrical violation.

(b) Parallel mode. Creating a new candidate pair left of u would introduce an electrical violation. The new candidate ends the parallel mode.

Figure 7.5: Propagation of candidate pairs with repeater insertion. We apply the move step of Section 7.3.6 on the path from ν to ω. By binary search we look for position u "between" $p(\mathfrak{O})$ and $\kappa(\overline{\nu})$ such that we do not create an electrical violation. We create a new candidate with position u that represents repeater insertion.

Propagation without repeater insertion. We propagate $\mathfrak{O}$ to position $\kappa(\overline{\nu})$. The resulting candidate pair is added to the heap unless it is almost-dominated by a candidate pair that is currently contained in the heap or that has been removed before. The new candidate pair $\mathfrak{O}'$ has $\text{node}(\mathfrak{O}') = \overline{\nu}$ and $p(\mathfrak{O}') = \kappa(\overline{\nu})$. Note that $\mathfrak{O}$ and $\mathfrak{O}'$ have the same type.

Propagation with repeater insertion. When we create new candidate pairs by insertion of further repeaters we have to distinguish if $\mathfrak{O}$ is of parallel type or not.

If $\mathfrak{O}$ is *not* of parallel type, we do the following. For each repeater $l \in L$ we look for a point u on the straight path segment between $p(\mathfrak{O})$ and $\kappa(\overline{\nu})$ (if e is a wiring edge) respectively in $\{p(\mathfrak{O}), \kappa(\overline{\nu})\}$ (if e is a via). We want to create a new candidate pair $\mathfrak{O}(u, l)$ arising by propagation to u and insertion of a repeater of type l in the valid part as described in Section 7.3.3. Position u is chosen such that the downstream capacitance of the inserted repeater does not exceed the capacitance limit of l and such that the slew limit of the new candidate is not smaller than the slew target. By binary search we can find u as far away from $p(\mathfrak{O})$ as possible. If no such point exists, we choose $u = p(\mathfrak{O})$. This is the case if the capacitance of the valid part of $\mathfrak{O}$ is already larger than the capacitance limit of l or if a repeater of type l cannot drive $\mathfrak{O}$ at all without violating the slew limit. We add $\mathfrak{O}(u, l)$ to the heap unless it is almost-dominated by a candidate pair contained in the heap or already removed from it, and set $\text{node}(\mathfrak{O}(u, l)) = \overline{\nu}$ if $u = \kappa(\overline{\nu})$. An example of non-parallel repeater insertion during the move-step can be found in Figure 7.5(a).

If $\mathfrak{O}$ *is* of parallel type, we create candidate pairs resolving the parallel mode. For each *inverter* $l \in L$ and each polarity in $\text{pol} \in \{\text{ident}, \text{invert}\}$ we use binary search to find the point u as far away from $p(\mathfrak{O})$ as possible such that after propagation to position u and insertion of an inverter of type l in the pol-part of $\mathfrak{O}$ we obtain a candidate pair without capacitance violations and for which no slew limit is smaller than the slew target. As before, we choose u on the straight path segment between $p(\mathfrak{O})$ and $\kappa(\overline{\nu})$ if e is a wiring edge and in $\{p(\mathfrak{O}), \kappa(\overline{\nu})\}$ if e is a via. If a point such as u does not exist, we choose $u := \kappa(\overline{\nu})$. We add the resulting candidate pair to the heap unless it is almost-dominated. An example of parallel repeater insertion during the move-step can be found in Figure 7.5(b).

When we compute the positions u we only take electrical violations into account. Positions for which *delays* are optimum can be between $p(\mathfrak{O})$ and u. In our practical application, tile sizes are small compared to optimum distances between repeaters and

the error we make by computing u based on electrical violations only is also small. In the presence of large tile sizes we can make use of the stationary slew $\text{slew}_{\text{target}}$ in the pre-computed optimum repeater chain (Bartoschek et al. [Bar+09]) and compute u such that $\mathfrak{O}(u, l)$ does not result in a slew larger than $\text{slew}_{\text{target}}$ at $\mathfrak{O}$. This can be achieved by reducing the slew limit of every valid part of $\mathfrak{O}$ to $\text{slew}_{\text{target}}$.

Every time we add a candidate pair $\mathfrak{O}$ to the heap, we erase all candidate pairs $\mathfrak{O}' (\neq \mathfrak{O})$ for which $\mathfrak{O}$ almost-dominates $\mathfrak{O}'$ from the heap. Note that due to non-negativity of the cost function, a candidate pair $\mathfrak{O}_1$ can never be almost-dominated by a pair $\mathfrak{O}_2$ that is created after $\mathfrak{O}_1$ is extracted from the heap without the property that $\mathfrak{O}_1$ almost-dominates $\mathfrak{O}_2$ as well. When pruning almost-dominated candidates we have to make sure that whenever $\mathfrak{O}_1$ and $\mathfrak{O}_2$ almost-dominate each other, we never erase the earlier candidate pair.

Although almost-domination is already sufficient to prune most candidate pairs, their number can still be exponential in general (although it is possible to give polynomial runtime bounds when making restrictions to the input, see Permin [Per16]). In order to obtain acceptable running times in practice, Permin [Per16] developed and implemented useful speed-up techniques such as future costs and caching of transitions that lead to almost-dominated solutions. By increasing the values γ_{cost}, γ_{cap}, γ_{slewlim} we can obtain even further speed-ups – even though at the cost of worse solutions.

In the next section we show how we can prune and avoid most candidate pairs and thus, obtain a polynomial running time. These techniques are used in the *fast version* of the algorithm.

7.3.7 Speed-up Techniques for the Move Step in the Fast Version

Candidate pruning. In the *fast version* we prune candidate pairs until the number of pairs of each type and at each position of the form $\kappa(\nu)$ for $\nu \in V(P)$ is bounded by a constant k_{internal}.

During pruning we make sure that we neither prune too many candidate pairs of which the total capacitance is below the target capacitance $\text{cap}_{\text{target}}$ (see Section 7.3.5) nor below $\text{cap}_{\text{limit}} := \max\{\text{caplim}(l) : l \in L\}$.

Let $\overline{\mathfrak{O}}$ be a set of candidate pairs with the same type and with the same position in $\{\kappa(\nu) : \nu \in V(P)\}$ and let

$$\overline{\mathfrak{O}}_1 := \{\mathfrak{O} \in \overline{\mathfrak{O}} : \text{cap}(\text{ident}(\mathfrak{O})) + \text{cap}(\text{invert}(\mathfrak{O})) \leq \text{cap}_{\text{target}}\}$$
$$\overline{\mathfrak{O}}_2 := \{\mathfrak{O} \in \overline{\mathfrak{O}} : \text{cap}(\text{ident}(\mathfrak{O})) + \text{cap}(\text{invert}(\mathfrak{O})) \leq \text{cap}_{\text{limit}}\}.$$

We define the cost of a candidate pair $\mathfrak{O}$ in the canonical way as $\text{cost}(\text{ident}(\mathfrak{O})) + \text{cost}(\text{invert}(\mathfrak{O}))$. First, we add the $\min\left\{|\overline{\mathfrak{O}}_1|, \left\lceil \frac{k_{\text{internal}}-1}{2} \right\rceil\right\}$ cheapest candidate pairs from $\overline{\mathfrak{O}}_1$ to a set $\overline{\mathfrak{O}}_{\text{kept}}$ of kept candidates. Then, we add the $\min\left\{|\overline{\mathfrak{O}}_2 \backslash \overline{\mathfrak{O}}_{\text{kept}}|, \left\lfloor \frac{k_{\text{internal}}-1}{2} \right\rfloor\right\}$ cheapest candidate pairs from $\overline{\mathfrak{O}}_2 \backslash \overline{\mathfrak{O}}_{\text{kept}}$ to $\overline{\mathfrak{O}}_{\text{kept}}$ and fill up with the $k_{\text{internal}} - |\overline{\mathfrak{O}}_{\text{kept}}|$ cheapest candidate pairs in $\overline{\mathfrak{O}} \backslash \overline{\mathfrak{O}}_{\text{kept}}$. Finally, we erase all candidate pairs in $\overline{\mathfrak{O}} \backslash \overline{\mathfrak{O}}_{\text{kept}}$.

The best speed-up due to pruning is obtained if we already prune during Dijkstra's algorithm. To avoid problems after pruning candidate pairs that are predecessors of pairs

that we do not prune, we can re-define the keys of the elements stored in a heap such that we select candidate pairs $\mathfrak{C}$ lexicographically minimizing

$$\Big(\text{node}(\mathfrak{C}),\ \text{cost}(\text{ident}(\mathfrak{C})) + \text{cost}(\text{invert}(\mathfrak{C}))\Big),$$

where the sorting of the nodes is the reverse topological ordering of A.

Since we will actually prune most candidate pairs created during the move operation we should also avoid creating them in the first place.

Avoiding candidate creation. Let $\mathfrak{C}$ be a candidate pair and let $(\nu, \text{node}(\mathfrak{C})) \in E(P)$. During the move step we count the number of pairs with node ν and at position $\kappa(\nu)$ that we have extracted from the heap. If this number has reached $2 \cdot k_{\text{internal}}$, we will no longer create further candidate pairs arising from $\mathfrak{C}$. As soon as the number has reached $2 \cdot k_{\text{internal}}$ we perform the pruning mentioned above.

In addition to this strict method, we can use characteristics of the input library to avoid creation of too many candidates. To use this method we have to assume that we can sort repeaters in $L = \{\ell_1, \ldots, l_{|L|}\}$ by their *strength* such that for a given candidate C we can determine a smallest index $i \in \{1, \ldots, |L|\} \cup \{\infty\}$ such that

$$\text{caplim}(l_j) \geq \text{cap}(C) \quad \text{and} \quad \text{outslew}(l_j, \text{slew}_{\text{target}}, \text{cap}(C)) \leq \text{slewlim}(C)$$

if and only if $j \geq i$. In practice, such a sorting is usually possible. Let $\mathfrak{C}$ be a candidate pair extracted from the heap containing a valid candidate $C \in \{\text{ident}(\mathfrak{C}), \text{invert}(\mathfrak{C})\}$. If the index i of the smallest feasible repeater for C is ∞, we avoid creating candidate pairs arising from $\mathfrak{C}$ by propagation along an edge or by inserting a repeater driving C that is not the largest possible repeater. If $i < \infty$, we avoid creating candidate pairs by inserting repeaters with index $< i$ that drives C.

Limiting the number of final candidates. Unless path P is outgoing of s, the set of final candidate pairs will be the input to a merge step (Section 7.3.8). Keeping the number of them small usually has an even greater impact on the running time than bounding the number of internal candidate pairs.

To decrease the set of final candidate pairs, we only keep the k_{final} pairs with smallest *score* for a constant k_{final}:

Definition 7.6 (Score of a candidate pair) *The* ***score*** *of a valid candidate pair* $\mathfrak{C}$ *is*

$$\text{cost}(\mathfrak{C}) + \lambda(\mathfrak{C}) \cdot \min\{d_{repeater}(l, slew_{target}, \text{cap}(\mathfrak{C})) : l \in L\},$$

where $\lambda(\mathfrak{C})$ *is the sum of prices over all timing resources entering the sinks in* $\text{sinks}(\text{ident}(\mathfrak{C})) \cup \text{sinks}(\text{invert}(\mathfrak{C}))$*, and*

$$\text{cap}(\mathfrak{C}) := \text{cap}(\text{ident}(\mathfrak{C})) + \text{cap}(\text{invert}(\mathfrak{C})), \text{cost}(\mathfrak{C}) := \text{cost}(\text{ident}(\mathfrak{C})) + \text{cost}(\text{invert}(\mathfrak{C})).$$

Intuitively, the score is created to penalize candidate pairs with high capacitances. It allows us to compare the timing benefit of a repeater insertion on the overall solution with the additional cost accompanied with the insertion.

At this point we note that it is not difficult to construct examples in which we prune or avoid candidate pairs representing the optimum solution. This is not surprising as the *Minimum Cost Steiner Tree Buffering Problem* is *NP*-hard (cf. Section 4.5.1).

7.3.8 The Merge Step

Let $\nu \in V(A)$ with $\delta_A^+(\nu) = \{(\nu, \omega_i) : i = 1, \ldots, |\delta_A^+(\nu)|\}$. As we proceed in reverse topological ordering, we have already computed sets $\overline{\mathfrak{O}}_i$ of final candidate pairs obtained from the move steps for the paths starting with the edges (ν, ω_i) for $i = 1, \ldots, |\delta_A^+(\nu)|$.

Without loss of generality we assume that $|\delta_A^+(\nu)| = 2$ as otherwise, we can iteratively merge the candidate pairs in $\overline{\mathfrak{O}}_i$ for $i > 3$ with the candidate pairs resulting from the previous merge steps.

In the basic version, the merge step consists just of merging all candidates in $\overline{\mathfrak{O}}_1$ with all candidates with $\overline{\mathfrak{O}}_2$ and pruning almost-dominated candidates. Furthermore, in the implementation of Permin [Per16], only the k candidate pairs with smallest cost (with respect to cost(ident(.)) + cost(invert(.))) of each type are kept for an input parameter k. All other pairs are erased. The remaining candidate pairs are used as initial candidates for the move step along the path in A ending in ν.

In the fast version we use a more sophisticated merge step. First, we extend the sets $\overline{\mathfrak{O}}_1$ and $\overline{\mathfrak{O}}_2$ by computing new candidate pairs obtained from the previous ones $\mathfrak{O}$ by inserting a further repeater (where we use Section 7.3.3 if repeater insertion at position $\kappa(\nu)$ is not allowed). More precisely, for each $i = 1, 2$ we extend $\overline{\mathfrak{O}}_i'$ as follows:

- For all candidate pairs $\mathfrak{O}$ of type pol $\in \{\text{ident}, \text{invert}\}$ we add all candidate pairs obtained from $\mathfrak{O}$ by inserting a repeater of type $l \in L$ to $\overline{\mathfrak{O}}_i'$.
- For all candidate pairs $\mathfrak{O}$ of parallel type we compute all $2 \cdot |\{l \in L : l \text{ inverter}\}|$ possible candidate pairs obtained from $\mathfrak{O}$ by resolving parallel mode.

After merging all candidate pairs $\overline{\mathfrak{O}}_1$ with candidate pairs $\overline{\mathfrak{O}}_2$ we obtain a set $\overline{\mathfrak{O}}$ of merged pairs. As above, we extend $\overline{\mathfrak{O}}$ by adding candidates modeling insertion of further repeaters. Note that all merge configurations described by Bartoschek ([Bar14] page 76, Figure 6.1 and Figure 6.9) can be obtained that way.

Among this set we select the k_{final} candidate pairs with smallest *score* (see Definition 7.6) for the constant k_{final} that we already used at the end of the move step in the fast mode. The set of initial candidates for the move step along the path in A ending in v consists of these pairs only.

As before, the score can penalize candidate pairs with high capacitances. Inserting a repeater next to a merge point can be a good idea as it shields capacitances on a side branch and thus, decreases delay costs on the incoming path. On the contrary, candidate pairs representing such a solution usually have higher costs as the price for repeater insertion is already contained in the cost stored in their candidates. With the definition of the score we can compare the timing benefit of a repeater insertion with the additional cost.

7.3.9 Choosing a Final Solution

After having arrived at the source we select the cheapest among all candidate pairs $\mathfrak{O}$ with $\text{node}(\mathfrak{O}) = p(\mathfrak{O}) = s$ and $\text{invert}(\mathfrak{O}) = \not\subset$. Unlike for the intermediate candidates for which we had to make use of a slew estimate, we can evaluate the objective function for these final candidate pairs exactly.

Chapter 8

BonnRouteBuffer: A Tool for Global Buffering

In this chapter we put together the results of the previous chapters and obtain a tool for *timing-constrained global routing with buffered Steiner trees*. Our tool BonnRouteBuffer is part of the BonnTools suite [KRV07][Hel+11] developed at the Research Institute for Discrete Mathematics, University of Bonn, in cooperation with IBM.

BonnRouteBuffer can be used in different parts of a physical design flow. In early planning phases (floor planning), top-level nets have to be connected and buffered. Many blockages make both global routing and buffering difficult. The blockage structure of an example instance (U6 in Table 8.1) is shown in Figure 8.1.
After initial placement optimization with respect to linear delays, BonnRouteBuffer can be used to manage the transition from an unbuffered netlist with optimized virtual timing to a buffered one. This step requires buffering of all nets.

In Section 8.1 we summarize how BonnRouteBuffer is composed of the building blocks developed in preceding chapters. In this section we also describe how to deal with slew effects that do not fit into the resource sharing model directly.

In Section 8.3 we compare BonnRouteBuffer with an industrial buffering algorithm that achieves congestion-awareness by buffering global wires. Our experiments show that BonnRouteBuffer is superior to the industrial algorithm with respect to timing while other metrics such as routability and power consumption are comparable.

8.1 Overview

8.1.1 BonnRouteBuffer as Part of BonnRouteGlobal

Starting point for BonnRouteBuffer is the timing-constrained global routing framework presented in Chapter 3 that we implemented as extension to BonnRouteGlobal [Mül09] [Ahr+15]. Per default we use 25 resource sharing iterations (p=25 in Algorithm 4 on Page 39) and a congestion target of 95%. If we manage to meet all constraints modeled within BonnRouteGlobal, we will still have routing resources left. These remaining routing resources are important to compensate for inaccuracies of the global routing model and to enable us to re-buffer the most critical nets or nets with electrical violations by a congestion-unaware buffering algorithm in later parts of physical design.

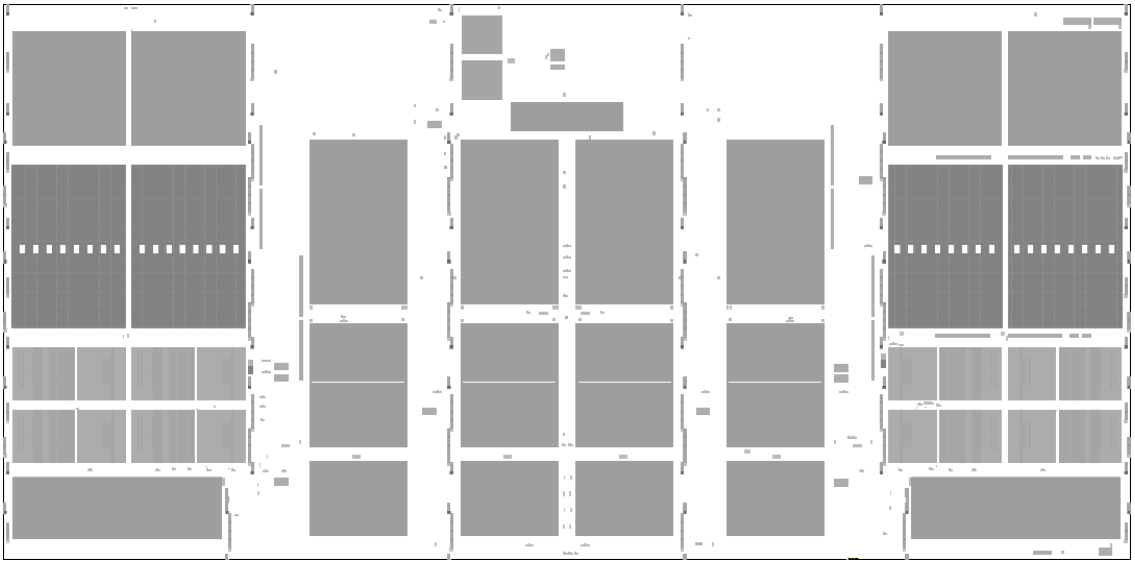


Figure 8.1: Blockage structure of design U6.

The input netlist consists of all nets present after removing all repeaters (even the parity inverters). As global routing graph we construct the coarse grid graph G_{coarse} (Section 2.4.3) with a default tile size of 70 detailed routing tracks. This graph is used during Steiner tree construction. For repeater insertion we use the fine graph G_{fine} (Section 7.1.1). As delay model we use the *Elmore Delay Model with Slew Propagation* described in Section 7.1.3.

As *lower* bound for source delays we use the delay $d_{\text{source}}(\min\{\text{cap}(L) : l \in L\})$ along a source gate driving the smallest repeater. This is indeed a lower bound if the input capacitance of each sink pin does not fall below the input capacitance of the smallest repeater (which is usually a correct assumption). As lower bound for the delay along a wire we use linear delays on a straight path on an optimum layer and with optimum wire code (see Section 6.5). The delay-per-length parameters have been computed by an almost optimum buffering of a long repeater chain using a power-time tradeoff parameter of $\xi_{\text{power}} = 0.8$ per default (see Section 7.2.2). These wire delays are no strict lower bounds on wire delays in a buffered netlist in general but turn out to be quite optimistic.

As *upper* bounds for gate delays we use the delay $d_{\text{source}}(\text{caplim}(s))$ through the driving gate in a solution that just obeys the capacitance limit at the source. This is an upper bound in an electrically correct solution. As upper bound for wire delays we use the same strategy as for virtual buffering described in Section 6.5. In addition, we estimate the delay impact of capacitances on side branches inside a Steiner tree for a net N as $b \cdot (|N| - 2)$. Here, b is the bifurcation delay penalty introduced in Chapter 5. Bartoschek [Bar14] (Section 4.1.6.) describes how to obtain the value for b as the maximum delay impact of a default capacitance on a long repeater chain. Note that the number of bifurcations on a source-sink path inside a topology for a net N is upper-bounded by $|N| - 2$.

8.1.2 Block Solver for Arrival Time Customers

The block solver for arrival time customers uses 3 iterations of Newton's method described in Section 3.6.3. Re-computation of all arrival times is repeated $n = 15$ times (Line (8) in Algorithm 4 on Page 39).

8.1.3 Block Solver for Net Customers

The most complex part of BonnRouteBuffer is the block solver for net customers.

Initial topology computation. Let N be a net. We start by clustering N as in Section 6.5 and connecting all clusters by a topology computed with the bicriteria approximation algorithm of Theorem 5.8 together with optimizations described in Section 5.5. The only difference to Section 6.5 is that we cluster sinks with the same polarity only, and choose a positive bifurcation delay penalty b.
In the default setting we choose $\epsilon = 0.1$ within the bicriteria algorithm. To compute initial topologies for the algorithm of Theorem 5.8 we have to distinguish if we are in the reach aware mode or not. If reach aware routing is active, we use the algorithm by Bihler [Bih15]. As reach length we use the distance between two repeaters in a long repeater chain computed with power time trade-off parameter $\xi_{\text{power}} = 0$. If reach aware routing is inactive, we compute an approximately shortest topology as in Section 5.6.

Topology embedding. Topologies are embedded by the embedding algorithm of Theorem 6.2 including all speed-up techniques described in Section 6.3. Per default we use an embedding tolerance of tol = 5 tiles. If reach aware routing is enabled, we optimize Steiner trees by the strategy of Section 6.4.

Buffering. The embedded Steiner trees serve as initial Steiner trees for the algorithm of Section 7.3. The default setting uses the fast mode with parameters $\gamma_{\text{cost}} = 1.1$, $\gamma_{\text{cap}} = 1$, $\gamma_{\text{slewlim}} = 1$, $k_{\text{internal}} = 5$, and $k_{\text{final}} = 1$. Alternatively, the algorithm can be configured to use the basic variant or a hybrid mode that uses the basic version for all critical nets and the fast mode or the remaining nets. To obtain a further speed-up we avoid buffering of instances for which the initial Steiner tree without repeaters meets all polarity and capacitance constraints, and for which all slews stay below the slew target.

8.1.4 Slew Updates

By definition of the *Min-Max Resource Sharing Problem* (Section 3.2), resource consumptions of every block have to be independent of solutions for other customers. In our case, consumptions from timing resources by buffered Steiner trees depend on the slew at input pins of the source gate and hence on solutions for net customers of the preceding nets. Instead of working with fixed input slews and taking inaccuracies of the timing model into account, we update input slews for root circuits of succeeding instances after each Steiner tree computation as follows. In the beginning, we set all input slews to the slew target for the buffering mode corresponding to the lowest available routing layer, the default wire code, and the currently chosen value for ξ_{power} (0.8 in the default setting). Let g be a gate with output pin s and having pin t as one of its input pins. After having computed a buffered Steiner tree (A_t, κ_t) for the net N_t containing t, we evaluate the slew by propagation along (A_t, κ_t) using a current input slew at the source gate at the root of N_t. This slew serves as the new input slew during computation of a solution for the net customer of the net with root s.

Although this approach is heuristic, it leads to a better accuracy of the timing model and to better results.

8.2 Layer and Wire Code Assignments

Except for later steps, the IBM optimization flow does not maintain global wires. Instead, for each net, a short rectilinear Steiner tree is computed and evaluated using the properties of the layer and wire code to which the net is assigned. The assignment of nets to layers (and wire codes) is called *layer assignment*. Industrial layer assignment tools such as CATALYST [Wei+13] and BonnLayerOpt iteratively assign the most critical nets to higher layers and to wire codes mapping these layers to wider widths and spacings as long as their timing improves and wiring congestion does not arise. Details on CATALYST can be found in the paper of Wei et al. [Wei+13]. BonnLayerOpt is based on a *Time-Cost-Tradeoff* formulation, see [Hel08], Chapter 6. Recall that we have already used layer assignment algorithms in the reference run for the experiments in Section 6.5.

To perform the layer assignment step after BonnRouteBuffer we use BonnLayerOpt that is part of the BonnTools suite. BonnLayerOpt makes use of a (usually timing-unaware) global router (e. g. BonnRouteGlobal) that re-routes all nets for which the assignment has changed. During such a re-route, the global router avoids wiring planes below the assigned layer and chooses wire widths and spacings according to the assigned wire code. This way, the global router serves as congestion estimate.

By running such a layer assignment algorithm afterwards, we achieve that BonnRouteBuffer outputs layer and wire code assignments instead of concrete global wires. Instead of making the assignment based on short rectilinear Steiner trees, we can use the RC-aware mode of BonnRouteGlobal [Hel+17] and compute Steiner trees trading-off congestion for Elmore delays.

There are alternative approaches to output layer and wire code assignments at the end of BonnRouteBuffer. The easiest possibility is to assign each net of the netlist after buffering to the highest layer z for which at least a fraction of τ of the global wires for this net use layers on plane z and above. A similar strategy can be used to obtain a wire code assignment. The most serious drawback of this approach is that it is not clear how to choose the parameter τ. If we choose small values for τ, we might create congestion while too large values lead to timing degradations.

A better method would be to compute layer and wire code assignments already during resource sharing. This would require to replace net customers by customers consisting of pairs (net, assignment). Solutions for the new type of customers are Steiner trees that do not use wires in x- or y-direction on a layer below the assigned one. The wire code assignment specifies widths and spacings of these wires. To be consistent with timing engines that compute delays based on 2-dimensional Steiner trees by assuming timing functions corresponding to the lowest assigned layer and the assigned wire code (cf. Section 7.1.3), Steiner trees using edges on layers higher than the assigned one should not see the timing benefit for doing so.
This approach would perfectly fit into the timing-constrained global routing framework. Since there are usually only constantly many possible assignments, a naive block solver would be to compute almost-cheapest Steiner trees for each assignment and select the pair (Steiner tree, assignment) with minimum cost.
Implementation of such an approach would be a useful future project. Finding a block solver for the new type of customers that is faster in practice an still admits an approximation ratio is an interesting open problem.

8.3 Experimental Results

We ran the default version of BonnRouteBuffer on 12 unbuffered 14 nm designs. The instances were provided by IBM. They do not contain initial plane- and wire code-assignments and do not contain repeaters except for the parity inverters. We compare our tool with BuffOpt, the default buffering algorithm in the IBM physical design flow. BuffOpt achieves congestion-awareness by following the output of a global router. In a first step, the IBM global router ROUGHROUTE computes a (timing-unaware) global routing and outputs unbuffered Steiner trees for each net. Each of these Steiner trees is then buffered sequentially by dynamic programming using a highly optimized implementation of the algorithm of Lillis et al. [LCL96] including optimizations and speed-up techniques described in Section 7.2.1. The global routing is updated incrementally during buffering. Placement-awareness during the BuffOpt flow is achieved by running placement legalization at several intermediate steps. In addition, BuffOpt performs local gate sizing and Vt-optimization.

We ran two different post-optimization flows. The first flow performs placement legalization and BonnLayerOpt only. In a second flow we run global gate-sizing and Vt-optimization, placement legalization, and BonnLayerOpt.

The results are shown in Table 8.1. Rows entitled **w/o gate sizing** show results of BuffOpt and BonnRouteBuffer with the first post-optimization flow. Rows entitled **with gate sizing** show results of buffering and the second post-optimization flow.

We compare results by worst slack (**wsl**), sum of negative slacks (**sns**), number of inserted repeaters (# **rpt**), power consumption (**static power** and **dynamic power**), routing overflow (**ol**), wire length (**wl**), number of vias (**vias**), and the amount of electrical violations (**slew vio** and **load vio**). We compare running times for the pure buffering calls (**wall time buffering**) and for the whole flow (**wall time total**). All reported timing values and electrical violations were computed with IBM EinsTimer based on the netlist at the very end. For each net, EinsTimer computes a 2-dimensional timing-unaware Steiner tree and propagates delays, slews, and capacitances along it assuming delay parameters of the assigned layer and the assigned wire code. EinsTimer uses the *Elmore Delay Model with Slew Propagation*. Static and dynamic power is computed with IBM EinsPower. Routing overload is determined by a final call to the timing-unaware version of BonnRouteGlobal, respecting all computed assignments. The reported running times exclude running times for design loading and for computation of quality metrics.

Table 8.1 shows that BonnRouteBuffer did a better job from a timing point of view. On U1, U2, U3, U4, U5, U6, U9, U11, and U12 the worst slack improvements are significant with 50 ps and more for both post-optimization flows. Among all instances, U4 and U6 have the largest chip image and the most critical timing. Here, the solution computed by BuffOpt has a timing far away from optimum. On instance U10 the BuffOpt-based flow including gate sizing and Vt-assignment achieved a better timing than the corresponding BonnRouteBuffer-based flow but on the cost of a higher power consumption, worse routability, and more electrical violations.

With respect to power consumptions none of the buffering tools dominates the other one. On units U1, U4, and U6, BonnRouteBuffer consumed more power. This effect can be explained by the exponential cost functions used in the resource sharing algorithm: Prices for the critical timing resources dominate prices for the less critical power resource and hence large power consumptions are tolerated to achieve small delays. On these units,

Unit (#nets, cycle time)	Experiment	Postopt	wsl [ps]	sns [ns]	# rpt	area	static power [mW]	dynamic power [mW]	ol	wl [m]	vias [k]	slew vio [ps]	load vio [fF]	wall time buffering [h:m:s]	wall time total [h:m:s]
U1	BuffOpt	w/o gate sizing	-366	-831	29 105	642 374	27.5	27.3	137	3.64	491	43 281	224	0:25:24	1:00:53
(21 385,	BonnRouteBuffer	w/o gate sizing	-102	-239	15 302	923 478	37.4	31.0	7	3.54	374	81 113	244	0:27:35	1:14:13
240 ps)	BuffOpt	with gate sizing	-217	-425	29 105	654 351	34.9	27.7	102	3.64	480	25 443	465	0:25:24	1:03:47
	Bonnroutebuffer	with gate sizing	-112	-225	15 302	790 770	41.7	28.3	6	3.54	370	71 641	94	0:27:35	1:15:45
U2	BuffOpt	w/o gate sizing	-187	-222	10 412	341 483	22.8	10.6	0	0.40	256	10 487	0	0:15:19	0:19:08
(25 044,	BonnRouteBuffer	w/o gate sizing	-91	-149	5 260	351 833	20.9	11.0	0	0.40	227	4 038	0	0:03:54	0:07:30
184 ps)	BuffOpt	with gate sizing	-126	-141	10 412	338 468	24.0	10.5	0	0.40	256	7 482	0	0:15:19	0:19:45
	Bonnroutebuffer	with gate sizing	-62	-84	5 260	325 602	19.9	10.5	0	0.40	230	1 244	0	0:03:54	0:08:40
U3	BuffOpt	w/o gate sizing	-190	-48	14 603	379 066	12.9	8.9	0	0.64	305	184	4	0:09:01	0:12:20
(32 442,	BonnRouteBuffer	w/o gate sizing	-113	-24	6 808	423 062	13.9	9.8	0	0.60	270	258	0	0:04:23	0:07:26
184 ps)	BuffOpt	with gate sizing	-161	-25	14 603	372 914	10.1	8.8	0	0.64	305	356	0	0:09:01	0:13:11
	Bonnroutebuffer	with gate sizing	-70	-23	6 808	341 446	5.4	8.2	0	0.61	282	2 628	0	0:04:23	0:08:58
U4	BuffOpt	w/o gate sizing	-984	-32 498	32 845	4 404 323	42.2	103.6	693	17.92	900	19 157 031	21 955	0:44:12	2:25:49
(37 370,	BonnRouteBuffer	w/o gate sizing	-229	-6 118	65 376	6 290 657	119.3	139.0	362	17.62	1 419	1 566 067	3 002	1:10:01	1:59:43
250 ps)	BuffOpt	with gate sizing	-984	-32 404	32 845	4 403 702	42.8	103.5	468	17.96	888	19 038 134	21 033	0:44:12	2:55:02
	Bonnroutebuffer	with gate sizing	-229	-6 174	65 376	6 292 440	119.0	138.6	327	17.63	1 423	1 570 497	3 063	1:10:01	2:09:17
U5	BuffOpt	w/o gate sizing	-297	-57	13 448	330 210	12.5	7.3	0	0.48	361	940	0	0:11:15	0:14:36
(37 917,	BonnRouteBuffer	w/o gate sizing	-175	-41	5 525	351 521	10.5	8.2	0	0.49	327	1 771	13	0:04:47	0:08:24
184 ps)	BuffOpt	with gate sizing	-199	-41	13 448	318 526	8.4	6.7	0	0.49	362	679	0	0:11:15	0:16:18
	Bonnroutebuffer	with gate sizing	-129	-38	5 525	296 971	6.2	6.8	0	0.49	337	5 432	0	0:04:47	0:10:28
U6	BuffOpt	w/o gate sizing	-821	-4 474	157 957	13 793 175	59.8	228.6	105	37.11	2 293	1 835 626	5 375	2:35:35	2:50:01
(38 066,	BonnRouteBuffer	w/o gate sizing	-182	-1 457	75 014	15 800 355	152.1	230.1	5	35.29	1 414	9 037 388	22 409	1:58:50	2:37:12
250 ps)	BuffOpt	with gate sizing	-821	-4 025	157 957	13 871 591	70.1	229.7	97	37.10	2 279	1 696 970	5 369	2:35:35	3:06:09
	Bonnroutebuffer	with gate sizing	-182	-1 458	75 014	15 800 456	152.0	229.9	7	35.29	1 414	9 037 100	22 381	1:58:50	2:44:07
U7	BuffOpt	w/o gate sizing	-135	-138	21 521	572 226	33.0	13.1	0	0.96	501	367	7	0:13:18	0:18:54
(44 423,	BonnRouteBuffer	w/o gate sizing	-97	-87	12 030	647 049	34.5	14.7	0	0.99	453	857	0	0:07:10	0:12:54
184 ps)	BuffOpt	with gate sizing	-77	-57	21 521	574 285	30.8	13.0	0	0.96	499	390	0	0:13:18	0:20:31
	Bonnroutebuffer	with gate sizing	-66	-59	12 030	561 504	25.0	13.2	0	1.00	462	1 170	0	0:07:10	0:14:16
U8	BuffOpt	w/o gate sizing	-149	-78	27 298	634 202	17.7	14.7	0	1.15	532	563	25	0:17:50	0:23:13
(49 192,	BonnRouteBuffer	w/o gate sizing	-174	-134	17 682	736 019	16.0	16.4	0	1.11	474	266	0	0:07:11	0:11:56
264 ps)	BuffOpt	with gate sizing	-113	-37	27 298	617 292	15.0	14.3	0	1.15	534	707	0	0:17:50	0:24:51
	Bonnroutebuffer	with gate sizing	-97	-67	17 682	599 321	12.1	14.4	0	1.12	488	1 632	3	0:07:11	0:14:19
U9	BuffOpt	w/o gate sizing	-174	-151	40 734	998 787	26.0	24.8	0	1.49	978	281	0	0:26:43	0:34:56
(93 783,	BonnRouteBuffer	w/o gate sizing	-125	-225	22 416	1 097 076	20.1	26.8	0	1.45	862	220	0	0:12:01	0:19:11
264 ps)	BuffOpt	with gate sizing	-115	-43	40 734	983 069	18.3	23.9	0	1.49	978	353	0	0:26:43	0:38:17
	Bonnroutebuffer	with gate sizing	-62	-55	22 416	940 349	13.9	23.4	0	1.44	880	2 557	0	0:12:01	0:23:32
U10	BuffOpt	w/o gate sizing	-323	-1 039	38 427	1 224 151	40.4	13.4	589	2.47	1 450	2 245	0	0:29:21	0:38:04
(141 175,	BonnRouteBuffer	w/o gate sizing	-308	-1 107	18 389	1 273 621	25.6	14.3	265	2.37	1 345	914	6	0:14:27	0:22:30
240 ps)	BuffOpt	with gate sizing	-199	-636	38 427	1 261 029	51.7	13.9	546	2.48	1 445	2 405	0	0:29:21	0:41:54
	Bonnroutebuffer	with gate sizing	-215	-609	18 389	1 185 880	44.4	13.3	234	2.36	1 344	526	0	0:14:27	0:24:10
U11	BuffOpt	w/o gate sizing	-379	-1 995	121 017	2 635 955	100.7	51.6	5	1.82	584	3 815	6	1:15:55	1:34:28
(156 464,	BonnRouteBuffer	w/o gate sizing	-209	-1 495	63 233	2 767 163	70.8	54.5	95	5.38	2 065	30 072	0	0:27:24	0:59:22
264 ps)	BuffOpt	with gate sizing	-192	-1 019	121 017	2 630 576	129.2	51.4	2	1.24	405	3 399	0	1:15:55	1:40:49
	Bonnroutebuffer	with gate sizing	-124	-800	63 233	2 455 688	113.3	49.9	0	1.74	524	22 872	0	0:27:24	0:56:04
U12	BuffOpt	w/o gate sizing	-509	-727	131 544	4 632 645	116.8	143.8	219	9.93	4 075	62 892	19	1:56:56	2:51:45
(361 665,	BonnRouteBuffer	w/o gate sizing	-355	-545	82 410	5 208 123	92.5	153.9	109	9.54	3 662	284 718	205	0:56:14	3:09:21
184 ps)	BuffOpt	with gate sizing	-343	-337	131 544	4 524 806	92.8	139.9	208	9.94	4 092	66 906	7	1:56:56	3:13:20
	Bonnroutebuffer	with gate sizing	-296	-325	82 410	4 393 723	66.6	132.6	128	9.52	3 814	243 635	66	0:56:14	2:29:02

Table 8.1: Comparison between BonnRouteBuffer and the industrial buffering algorithm BuffOpt.

gate sizing and Vt-optimization could not help to achieve a significant power reduction. Further post-processing with optimization tools that directly optimize objectives modeled by dominated resources such as the power resource in later parts of physical design is important.

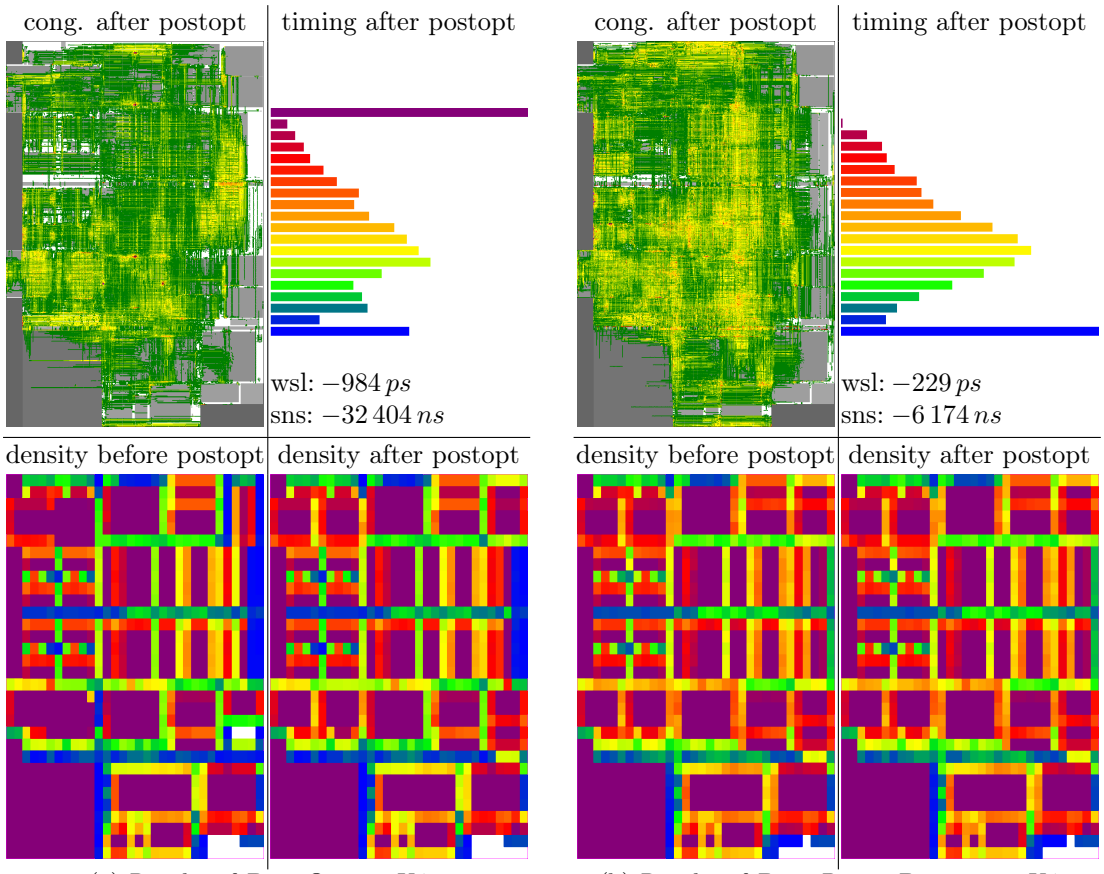
On units U9, U10, U11, and U12, the solution computed by BonnRouteBuffer consumes less power than the solution of BuffOpt and on U2, U3, U5, U7, and U8, power consumptions do not differ much between the tools. After the application of gate sizing and Vt-optimization, power consumptions on all these 9 units are smaller with the BonnRouteBuffer-based flow.

On all instances except for U4, BuffOpt inserts more repeaters than BonnRouteBuffer and seems to prefer the smallest repeaters. This approach has both advantages and disadvantages for later optimization steps. By inserting many repeaters, smaller nets arise and the probability that the design's timing engine chooses unfavorable Steiner trees for computation of delays and electrical violations decreases. The drawback of insertion of many repeaters is that subsequent optimization tools have less flexibility. Power reductions achieved by gate sizing and Vt-optimization are smaller when run on the output of BuffOpt while timing improvement are comparable. During global routing connecting the many repeaters inserted by BuffOpt require to insert many vias.

On U1, U5, U6, U7, U11, and U12, BonnRouteBuffer produces more slew violations than BuffOpt. On U6, the number of load violations is much larger, too. There are several reasons for electrical violations. Sometimes, BonnLayerOpt cannot assign nets to higher layers due to congestion issues and larger slew degradations on lower layers create violations. On some nets, unfavorable routing topologies of the timing-unaware 2-dimensional Steiner trees computed by IBM EinsTimer induce slew violations that are not present in the timing-driven Steiner trees computed by BonnRouteBuffer. The largest amount of electrical violations is present in timing-uncritical nets. For these nets, the Steiner trees given as input to the buffering algorithm from Chapter 7 use the lowest layers. Some of these Steiner trees are not reach-aware and the buffering algorithm can not avoid electrical violations. Activating the reach-aware mode as described in Section 6.4 can certainly reduce these problems but cannot completely avoid them. The main reason for this is that the simple reach-aware model from Section 6.4 does not take pin capacitances into account which have a significant impact on large instances such as U6. There is an ongoing project by a master student who works at the Research Institute for Discrete Mathematics, University of Bonn, under my co-supervision. By using a multi-label approach he re-embeds topologies for such instances, taking also placement-, slew-, and capacitance constraints into account. With this project we can hope for better solutions with less electrical violations in future versions of BonnRouteBuffer.

Since BonnRouteBuffer is run with 16 threads, running times are smaller than for BuffOpt which performs repeater insertion sequentially. The only units where BonnRouteBuffer is still slower are U1 and U4.

Figures 8.2 and 8.3 show congestion, timing, and placement density of U4 and U8 respectively. Recall from Section 6.5 that in a congestion plot each edge of the global routing graph is colored according to the fraction to which the corresponding congestion resource is used and the timing histograms show the slack distribution of all gates, where each gate is represented by its worst slack. Congestion plots and slack histograms shown in Figures 8.2 and 8.3 visualize the state at the end of the flows including gate sizing and Vt-assignment. The pictures at the bottom are placement density plots. Here, each



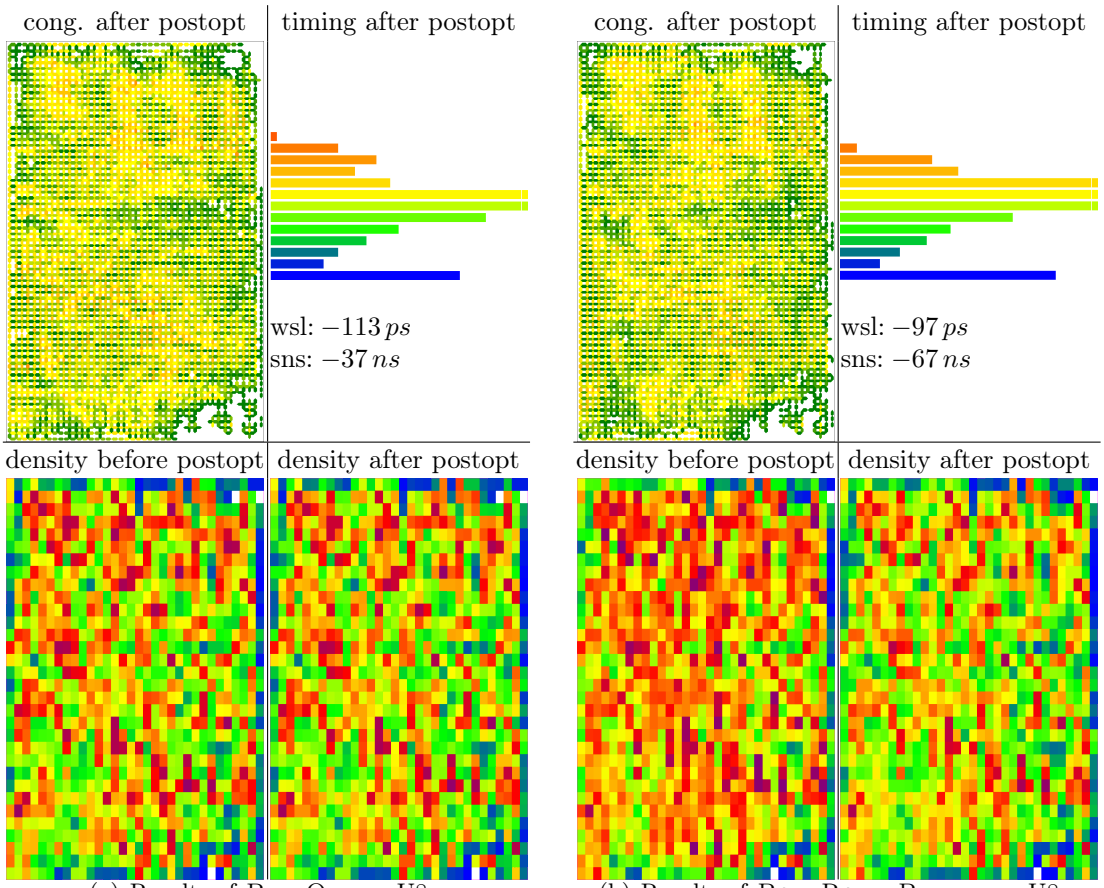
(a) Results of BuffOpt on U4.

(b) Results of BonnRouteBuffer on U4.

Figure 8.2: Congestion, timing, and placement density of U4. BonnRouteBuffer has a larger though feasible area consumption but achieves a much better timing. The congestion plot for BuffOpt shows local hot-spots.

placement bin is shown in a color corresponding to its density. White and blue represent bins with small density (below 10%), yellow show a moderate usage of around 50%, and the colors orange and red indicate that a placement bin is used by almost its full amount. The purple bins are either used by 100% or even more. Example of bins that are used by exactly 100% are all bins in Figure 8.2 completely covered by a placement blockage. Overfull bins can only occur before placement legalization. In Figures 8.2 and 8.3 we show density plots directly after buffering (left) and at the end of the flow (right).

Figure 8.2 shows that BonnRouteBuffer consumes more area than BuffOpt. Gate sizing could not reduce that amount. By paying the price of an increased area and power consumption, we could obtain a much better timing. Figure 8.3 shows an example of a unit for which gate sizing *could* reduce the amount of occupied area in the BonnRouteBuffer-based flow. In contrast to the situation after buffering, density plots on the right hand side of Figures 8.3(a) and 8.3(b) show that the BonnRouteBuffer-based flow consumes less area than the BuffOpt-based flow where gate sizing could not lead to a significant reduction of area consumption. From a timing point of view, none of the solution is really better. BonnRouteBuffer achieves a better worst slack while the sum of negative slacks is better with BuffOpt.



(a) Results of BuffOpt on U8. (b) Results of BonnRouteBuffer on U8.

Figure 8.3: Congestion, timing, and placement density of U8. The solution of BonnRouteBuffer is overpowered and consomes more area than necessary. Gate sizing can drastically decreases the area consumption.

8.3.1 Comparison with the IBM Physical Design Flow

The results of Section 8.3 suggest that BonnRouteBuffer is well-suited as global buffering tool in early phases of a chip design flow: On most instances the combination of BonnRouteBuffer and the post-optimization flow including gate sizing and Vt-optimization achieves better timing than the industrial buffering algorithm BuffOpt and even uses less power and area.

In this section we compare BonnRouteBuffer with the IBM physical design flow. This design flow uses many tools for timing optimization and power reduction and we cannot hope that our simple BonnRouteBuffer-based optimization flows from the previous section achieve equally good results on general instances.

We run a design similar to unit U6 directly within the IBM environment. This design consists of repeater trees only and most optimization tools contained in the design flow that are missing in our simple flows are not needed here. We demonstrate that on such an instance a BonnRouteBuffer-based flow achieves even better results than the complex and well-tuned design flow.

The design flow achieves the following result:

Worst slack:	-45 ps	Routing Overload:	320
Sum of negative slacks:	-0.3 ns	Wire length:	37.75 m
Number of repeaters:	158 398	Number of vias:	2 482 k
Area:	12 093 604	Slew violations:	9 193 612 ps
Static power:	59.4 mW	Load violations:	1 425 160 fF
Dynamic power:	587.0 mW		

The BonnRouteBuffer-based flow from Section 8.3 that includes gate sizing and Vt-optimization could compute a solution with smaller static and dynamic power consumption and less routing overflow:

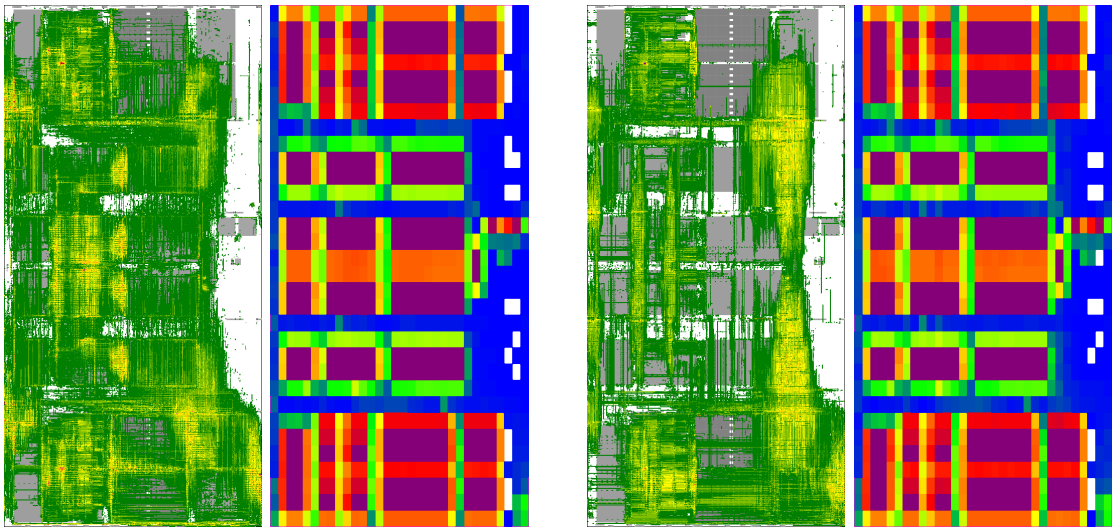
Worst slack:	-68 ps	Routing Overload:	88
Sum of negative slacks:	-0.7 ns	Wire length:	29.05 m
Number of repeaters:	105 283	Number of vias:	1 650 k
Area:	13 157 757	Slew violations:	3 881 047 ps
Static power:	26.8 mW	Load violations:	619 483 fF
Dynamic power:	550.4 mW		

The solution computed by the design flow is almost timing-clean and BonnRouteBuffer achieved a worst slack that is larger by roughly 20 ps smaller. The output of the design flow is more difficult to route and BonnRouteGlobal needed to make detours to avoid congestion which results in larger net length and more used vias. Both solutions produce a huge amount of electrical violations in timing-uncritical nets.

We optimized the output of the BonnRouteBuffer-based flow further by re-buffering the 0.5% most timing-critical nets with the *fast buffering algorithm* by Bartoschek et al. [Bar+09][Bar14] (see Section 7.2.2). In this algorithm we used the configuration $\xi_{\text{power}} = \xi_{\text{buf}} = 1$ that optimizes timing only and ignores the power consumption. After re-buffering we legalized the placement, re-computed layer assignments with BonnLayerOpt, and re-ran gate sizing and Vt-assignment. We obtain the following result:

Worst slack:	-47 ps	Routing Overload:	81
Sum of negative slacks:	-0.2 ns	Wire length:	29.06 m
Number of repeaters:	105 289	Number of vias:	1 657 k
Area:	13 119 012	Slew violations:	3 879 779 ps
Static power:	23.2 mW	Load violations:	619 380 fF
Dynamic power:	548.9 mW		

With this additional post-optimization we could achieve an equally good timing as the IBM design flow. The impact on the other quality metrics is negligible and hence, the final result of the BonnRouteBuffer-based flow is better than the result of the IBM design flow with respect to power consumption, routability, and electrical violations. Congestion and density plots of the two results are shown in Figure 8.4.



(a) IBM physical design flow.

(b) BonnRouteBuffer-based flow that includes gate sizing, Vt-optimization, and re-buffering of the 0.5% most timing-critical nets.

Figure 8.4: Congestion plots and density plots of the instance from Section 8.3.1.

8.4 Conclusions and Future Work

With BonnRouteBuffer we have developed a theoretically-founded tool that combines global routing with buffering. These central problems of any physical design flow have been solved separately in the past. The experimental results presented in this chapter already look quite promising: On most units, one single application of BonnRouteBuffer is sufficient to build fast repeater trees for most repeater tree instances without creating routing hot-spots and without creating placement instances that cannot legalized afterwards. Later optimization steps can now concentrate on detailed optimization such as eliminating electrical violations, optimizing the most critical paths, and decreasing the power consumption in uncritical parts of the chip.

Existing buffering tools such as BuffOpt sometimes fail to accomplish the task of global buffering. Often, many iterations of timing optimization tools are necessary to compute globally good solutions. Most repeater trees built in early iterations are ripped-up and replaced during later steps. This leads to unnecessarily large running times. Simulating global optimization with iteration of tools for detailed optimization is not always a good idea.

Although BonnRouteBuffer already produces good results, there are still many features missing that need to be implemented and are subject to ongoing or future research.

- Computation of the Steiner tree that is passed as input to buffering should be aware of placement congestion. The buffering algorithm should always have a chance to avoid placing repeaters inside placement hot-spots.
- Buffered Steiner trees for the most critical instances should be computed by a block solver that does not separate Steiner tree computation and buffer insertion. Recall that we presented such an algorithm for the basic version of the Elmore delay model in Chapter 4.
- Lower and upper bounds for arrival times are currently estimates based on linear

delays. Bartoschek [Bar14] (page 46) showed that there is a good correlation between linear delays and delays after buffering, but there are many outliers. Using better bounds would improve results and would lead to faster convergence of arrival times during the resource sharing part of BonnRouteBuffer.

- Resource prices within the first resource sharing iterations are often not good enough. It can happen that long connections of eventually critical nets are routed on the lowest layers because prices of the corresponding timing resources have not increased enough yet. For such Steiner trees it is not necessary to spend the running time for computing an actual buffering solution.

Another interesting open problem is the interaction between buffering and gate sizing. On the one hand, tools for gate sizing need to get completely buffered nets as input to perform their task successfully. On the other hand, sizes of the logic gates have a big impact on buffering. Often, insertion of additional repeaters close to the source pin is necessary if the source gate can only drive small capacitances. In many of these cases, re-sizing the source gate leads to better solutions than inserting the additional repeaters. The ambitious task of solving gate sizing, buffering, and global routing simultaneously should get high priority in future research. Except for the work of Alpert et al. [Alp+04] who developed an algorithm for simultaneous driver sizing and buffering in which delay effects of re-sizing the source gate are modeled as a pre-computed delay penalty, not much is known about this problem. A promising approach would be to combine the algorithm by Schorr [Sch15] for resource-sharing-based gate sizing and Vt-optimization with the resource-sharing-based timing-constrained global routing and buffering within BonnRouteBuffer.

Bibliography

[Ahr+15] Markus Ahrens, Michael Gester, Niko Klewinghaus, Dirk Müller, Sven Peyer, Christian Schulte, and Gustavo Téllez (2015). Detailed Routing Algorithms for Advanced Technology Nodes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.4, pp. 563–576.

[Alb+02] Christoph Albrecht, Andrew B Kahng, Ion Mandoiu, and Alexander Zelikovsky (2002). Floorplan Evaluation with Timing-Driven Global Wireplanning, Pin Assignment and Buffer/Wire Sizing. *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*. IEEE, p. 580.

[AD97] Charles J Alpert and Anirudh Devgan (1997). Wire Segmenting for Improved Buffer Insertion. *Proceedings of the 34th annual Design Automation Conference*. ACM, pp. 588–593.

[ADQ99] Charles J Alpert, Anirudh Devgan, and Stephen T Quay (1999). Buffer Insertion With Accurate Gate and Interconnect Delay Computation. *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. ACM, pp. 479–484.

[AHQ04] Charles J Alpert, Miloš Hrkić, and Stephen T Quay (2004). A Fast Algorithm for iIdentifying Good Buffer Insertion Candidate Locations. *Proceedings of the 2004 International Symposium on Physical Design*. ACM, pp. 47–52.

[Alp+95] Charles J Alpert, Te C Hu, J H Huang, Andrew B Kahng, and David R Karger (1995). Prim-Dijkstra Tradeoffs for Improved Performance-Driven Routing Tree Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.7, pp. 890–896.

[Alp+04] Charles Alpert, Chris Chu, Gopal Gandham, Miloš Hrkić, Jiang Hu, Chandramouli Kashyap, and Stephen Quay (2004). Simultaneous Driver Sizing and Buffer Insertion Using a Delay Penalty Estimation Technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.1, pp. 136–141.

[Aro98] Sanjeev Arora (1998). Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems. *Journal of the ACM (JACM)* 45.5, pp. 753–782.

[Bar14] Christoph Bartoschek (2014). Fast Repeater Tree Construction. *PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Bar+10] Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen (2010). The Repeater Tree Construction Problem. *Information Processing Letters* 110.24, pp. 1079–1083.

[Bar+09] Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen (2009). Fast Buffering for Optimizing Worst Slack and Resource Consumption in Repeater Trees. *Proceedings of the 2009 International Symposium on Physical Design*. ACM, pp. 43–50.

[Bel58] Richard Bellman (1958). On a Routing Problem. *Quarterly of Applied Mathematics* 16, pp. 87–90.

[Bih15] Tilmann Bihler (2015). Rektilineare Steinerbäume mit längen- und richtungsbeschränkenden Blockaden. *Bachelor's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[BZ15] Marcus Brazil and Martin Zachariasen (2015). *Optimal Interconnection Trees in the Plane: Theory, Algorithms and Applications.* Springer Publishing Company, Incorporated.

[Byr+13] Jarosław Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanita (2013). Steiner Tree Approximation via Iterative Randomized Rounding. *Journal of the ACM (JACM)* 60.1, p. 6.

[CM99] Chung-Ping Chen and Noel Menezes (1999). Noise-aware Repeater Insertion and Wire Sizing for On-chip Interconnect using Hierarchical Moment-Matching. *Proceedings of the 36th annual ACM/IEEE Design Automation Conference.* ACM, pp. 502–506.

[CW08] Chris Chu and Yiu-Chung Wong (2008). FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.1, pp. 70–83.

[Chu+05] Julia Chuzhoy, Anupam Gupta, Joseph Naor, and Amitabh Sinha (2005). On the Approximability of Network Design Problems. *ACM Transactions on Algorithms (TALG).*

[CLZ93] Jason Cong, Kwok-Shing Leung, and Dian Zhou (1993). Performance-Driven Interconnect Design Based on Distributed RC Delay Model. *Proceedings of the 30th Conference on Design Automation.* IEEE, pp. 606–611.

[Con+92] Jingsheng Cong, Andrew B Kahng, Gabriel Robins, Majid Sarrafzadeh, and Chak-Kuen Wong (1992). Provably Good Performance-Driven Global Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11.6, pp. 739–752.

[CL94] Javier Córdova and Yann-hang Lee (1994). A Heuristic Algorithm for the Rectilinear Steiner Arborescence Problem. *Engineering Optimization.*

[CW03] John F Croix and D F Wong (2003). Blade and Razor: Cell and Interconnect Delay Analysis Using Current-Based Models. *Proceedings of the 40th Design Automation Conference.* IEEE, pp. 386–389.

[Dij59] Edsger W Dijksta (1959). A Note on Two Problems in Connexion with Graphs. *Numerische mathematik* 1.1, pp. 269–271.

[ES15] Michael Elkin and Shay Solomon (2015). Steiner Shallow-Light Trees are Exponentially Lighter than Spanning Ones. *SIAM Journal on Computing* 44.4, pp. 996–1025.

[Elm48] William C Elmore (1948). The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics* 19.1, pp. 55–63.

[Fen+06] Zhe Feng, Yu Hu, Tong Jing, Xianlong Hong, Xiaodong Hu, and Guiying Yan (2006). An $\mathcal{O}(n \log n)$ Algorithm for Obstacle-Avoiding Routing Tree Construction in the λ-Geometry Plane. *Proceedings of the 2006 International Symposium on Physical Design.* ACM, pp. 48–55.

[For56] Lester R Ford (1956). Network Flow Theory. *DTIC Document.*

[FT87] Michael L Fredman and Robert E Tarjan (1987). Fibonacci Heaps and their Uses in Improved Network Optimization Algorithms. *Journal of the ACM (JACM)* 34.3, pp. 596–615.

[GJ77] Michael R Garey and David S. Johnson (1977). The Rectilinear Steiner Tree Problem is NP-Complete. *SIAM Journal on Applied Mathematics* 32.4, pp. 826–834.

[GJ79] Michael R Garey and David S Johnson (1979). Computers and Intractability: a Guide to the Theory of NP-Completeness. *San Francisco, LA: Freeman.*

[GK07] Naveen Garg and Jochen Koenemann (2007). Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. *SIAM Journal on Computing* 37.2, pp. 630–652.

[Ges+13] Michael Gester, Dirk Müller, Tim Nieberg, Christian Panten, Christian Schulte, and Jens Vygen (2013). BonnRoute: Algorithms and Data Structures for Fast and Good VLSI Routing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18.2, p. 32.

[Goe+12] Michel X Goemans, Neil Olver, Thomas Rothvoß, and Rico Zenklusen (2012). Matroids and Integrality Gaps for Hypergraphic Steiner Tree Relaxations. *Proceedings of the forty-fourth annual ACM symposium on Theory of computing.* ACM, pp. 1161–1176.

[GH05] Andrew V Goldberg and Chris Harrelson (2005). Computing the Shortest Path: A^* Search Meets Graph Theory. *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics, pp. 156–165.

[GK94] Michael D Grigoriadis and Leonid G Khachiyan (1994). Fast Approximation Schemes for Convex Programs with many Blocks and Coupling Constraints. *SIAM Journal on Optimization* 4.1, pp. 86–107.

[Häh15] Nicolai Hähnle (2015). Time-Cost Tradeoff and Steiner Tree Packing with Multiplicative Weights. *Technical report no. 1511115, Research Institute for Discrete Mathematics, University of Bonn.*

[Hel08] Stephan Held (2008). Timing Closure in Chip Design. *PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[HHV15] Stephan Held, Stefan Hougardy, and Jens Vygen (2015). Chip Design. *Princeton Companion to Applied Mathematics (N. Higham, ed.)* Princeton University Press.

[Hel+11] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen (2011). Combinatorial Optimization in VLSI Design. *Combinatorial Optimization-Methods and Applications* 31, pp. 33–96.

[Hel+17] Stephan Held, Dirk Müller, Daniel Rotter, Rudolf Scheifele, Vera Traub, and Jens Vygen (2017). Global Routing with Timing Constraints. submitted.

[Hel+15] Stephan Held, Dirk Müller, Daniel Rotter, Vera Traub, and Jens Vygen (2015). Global Routing with Inherent Static Timing Constraints. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design.* IEEE Press, pp. 102–109.

[HR13] Stephan Held and Daniel Rotter (2013). Shallow-Light Steiner Arborescences with Vertex Delays. *Integer Programming and Combinatorial Optimization.* Springer, pp. 229–241.

[HS14] Stephan Held and Sophie T Spirkl (2014). A Fast Algorithm for Rectilinear Steiner Trees with Length Restrictions on Obstacles. *Proceedings of the 2014 on International Symposium on Physical Design.* ACM, pp. 37–44.

[Hen16] Dorothee Henke (2016). Pfadsuche im Detailed Routing. *Bachelor's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[HSC82] Robert B Hitchcock, Gordon L Smith, and David D Cheng (1982). Timing Analysis of Computer Hardware. *IBM journal of Research and Development* 26.1, pp. 100–105.

[HS75] Dan Hoey and Michael I Shamos (1975). Closest-Point Problems. *Proceedings of the 17th IEEE Annual Symposium on Foundations of Computer Science.* Vol. 26, pp. 151–162.

[Hon+97] Xianlong Hong, Tianxiong Xue, Jin Huang, Chung-Kuan Cheng, and Ernest S Kuh (1997). TIGER: an Efficient Timing-Driven Global Router for Gate Array and Standard Cell Layout Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.11, pp. 1323–1331.

[HL02] Miloš Hrkić and John Lillis (2002). S-Tree: a Technique for Buffered Routing Tree Synthesis. *Proceedings of the 39th Annual Design Automation Conference.* ACM, pp. 578–583.

[Hu+03] Jiang Hu, Charles J Alpert, Stephen T Quay, and Gopal Gandham (2003). Buffer Insertion with Adaptive Blockage Avoidance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22.4, pp. 492–498.

[HS02] Jiang Hu and Sachin S Sapatnekar (2002). A Timing-Constrained Simultaneous Global Routing Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.9, pp. 1025–1036.

[Hu+07] Shiyan Hu, Charles J Alpert, Jiang Hu, Shrirang K Karandikar, Zhuo Li, Weiping Shi, and Chin N Sze (2007). Fast Algorithms for Slew-Constrained Minimum Cost Buffering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.11, pp. 2009–2022.

[HLA09] Shiyan Hu, Zhuo Li, and Charles J Alpert (2009). A Fully Polynomial Time Approximation Scheme for Timing Driven Minimum Cost Buffer Insertion. *Proceedings of the 46th Annual Design Automation Conference.* ACM, pp. 424–429.

[Hua+93] Jin Huang, Xian-Long Hong, Chung-Kuan Cheng, and Ernest S Kuh (1993). An Efficient Timing-Driven Global Routing Algorithm. *Proceedings of the 30th Conference on Design Automation.* ACM, pp. 596–600.

[Huf52] David A Huffman (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40.9, pp. 1098–1101.

[Hwa76] Frank K Hwang (1976). On Steiner Minimal Trees with Rectilinear Distance. *SIAM journal on Applied Mathematics* 30.1, pp. 104–114.

[JZ08] Klaus Jansen and Hu Zhang (2008). Approximation algorithms for general packing problems and their application to the multicast congestion problem. *Mathematical Programming* 114.1, pp. 183–206.

[Kas+04] Chandramouli V Kashyap, Charles J Alpert, Frank Liu, and Anirudh Devgan (2004). Closed Form Expressions for Extending Step Delay and Slew Metrics to Ramp Inputs for RC Trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.4, pp. 509–516.

[KRY95] Samir Khuller, Balaji Raghavachari, and Neal Young (1995). Balancing Minimum Spanning Trees and Shortest-Path Trees. *Algorithmica* 14.4, pp. 305–321.

[Kie16] Annika K Kiefner (2016). Minimizing path lengths in rectilinear Steiner minimum trees with fixed topology. *Operations Research Letters* 44, pp. 835–838.

[KRV07] Bernhard Korte, Dieter Rautenbach, and Jens Vygen (2007). BonnTools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip. *Proceedings of the IEEE* 95.3, pp. 555–572.

[KV12] Bernhard Korte and Jens Vygen (2012). *Combinatorial Optimization: Theory and Algorithms. 5th edition.* Springer.

[Kra49] Leon G Kraft (1949). A Device for Quanitizing, Grouping and Coding Amplitude Modulated Pulses. MA thesis. MIT, Cambridge.

[LS06] Zhuo Li and Weiping Shi (2006). An $\mathcal{O}(b \cdot n^2)$ Time Algorithm for Optimal Buffer Insertion with b Buffer Types. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.3, pp. 484–489.

[LZS12] Zhuo Li, Ying Zhou, and Weiping Shi (2012). $\mathcal{O}(mn)$ Time Algorithm for Optimal Buffer Insertion of Nets With m Sinks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.3, pp. 437–441.

[LCL96] John Lillis, Chung-Kuan Cheng, and Ting-Ting Y Lin (1996). Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model. *IEEE Journal of Solid-State Circuits* 31.3, pp. 437–447.

[Lin+08] Chung-Wei Lin, Szu-Yu Chen, Chi-Feng Li, Yao-Wen Chang, and Chia-Lin Yang (2008). Obstacle-Avoiding Rectilinear Steiner Tree Construction Based on Spanning Graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.4, pp. 643–653.

[Liu+09] Chih-Hung Liu, Shih-Yi Yuan, Sy-Yen Kuo, and Szu-Chi Wang (2009). High-Performance Obstacle-Avoiding Rectilinear Steiner Tree Construction. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.3, p. 45.

[LZM08] Jieyi Long, Hai Zhou, and Seda Ogrenci Memik (2008). An $\mathcal{O}(n \log n)$ Edge-Based Algorithm for Obstacle-Avoiding Rectilinear Steiner Tree Construction. *Proceedings of the 2008 International Symposium on Physical Design*. ACM, pp. 126–133.

[LR00] Bing Lu and Lu Ruan (2000). Polynomial Time Approximation Scheme for the Rectilinear Steiner Arborescence Problem. *Journal of Combinatorial Optimization* 4.3, pp. 357–363.

[Maß15] Jens Maßberg (2015). The rectilinear Steiner tree problem with given topology and length restrictions. *International Computing and Combinatorics Conference*. Springer, pp. 445–456.

[MV08] Jens Maßberg and Jens Vygen (2008). Approximation Algorithms for a Facility Location Problem with Service Capacities. *ACM Transactions on Algorithms (TALG)* 4.4, p. 50.

[Meh88] Kurt Mehlhorn (1988). A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Information Processing Letters* 27.3, pp. 125–128.

[MMP00] Adam Meyerson, Kamesh Munagala, and Serge Plotkin (2000). Cost-Distance: Two Metric Network Design. *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. IEEE, pp. 624–630.

[Moo59] Edward F Moore (1959). The Shortest Path Through a Maze. *Proceedings of the International Symposium on the Theory of Switching, Part II*. Harward University Press, pp. 285–292.

[Mül06] Dirk Müller (2006). Optimizing Yield in Global Routing. *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE, pp. 480–486.

[Mül09] Dirk Müller (2009). Fast Resource Sharing in VLSI Design. *PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[MRV11] Dirk Müller, Klaus Radke, and Jens Vygen (2011). Faster min–max resource sharing in theory and practice. *Mathematical Programming Computation* 3.1, pp. 1–35.

[MP03] Matthias Müller-Hannemann and Sven Peyer (2003). Approximation of Rectilinear Steiner Trees with Length Restrictions on Obstacles. *Workshop on Algorithms and Data Structures.* Springer, pp. 207–218.

[Nic66] T Alastair J Nicholson (1966). Finding the shortest route between two points in a network. *The computer journal* 9.3, pp. 275–280.

[OC96] Takumi Okamoto and Jason Cong (1996). Interconnect Layout Optimization by Simultaneous Steiner Tree Construction and Buffer Insertion. *Proceedings of the Fifth ACM/SIGDA Physical Design Workshop.* Citeseer, pp. 1–6.

[Per16] Rodion Permin (2016). A Near-Optimum Algorithm for Cost-Based Buffering. *Master's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Pri57] Robert C Prim (1957). Shortest connection networks and some generalizations. *Bell Labs Technical Journal* 36.6, pp. 1389–1401.

[RT87] Prabhakar Raghavan and Clark D Tompson (1987). Randomized Rounding: a Technique for Provably Good Algorithms and Algorithmic Proofs. *Combinatorica* 7.4, pp. 365–374.

[Rao+92] Sailesh K Rao, P Sadayappan, Frank K Hwang, and Peter W Shor (1992). The Rectilinear Steiner Arborescence Problem. *Algorithmica* 7.1-6, pp. 277–288.

[RP94] Curtis L Ratzlaff and Lawrence T Pillage (1994). RICE: Rapid Interconnect Circuit Evaluation using AWE. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.6, pp. 763–776.

[Roc16] Benjamin M Rockel (2016). Optimale Shallow-Light-Steiner-Arboreszenzen. *Bachelor's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Rom15] Daniel Romen (2015). Cost-Based Buffering for Multiple Resources. *Master's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Rot12] Daniel Rotter (2012). Light and Fast Repeater Tree Topologies. *Master's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Sac15] Pietro Saccardi (2015). Global routing with exact pin positions. *Master's thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Sam+15] Radhamanjari Samanta, Adil I Erzin, Soumyendu Raha, Yuriy V Shamardin, Ivan I Takhonov, and Vyacheslav V Zalyubovskiy (2015). A provably tight delay-driven concurrently congestion mitigating global routing algorithm. *Applied Mathematics and Computation* 255, pp. 92–104.

[Sap04] Sachin Sapatnekar (2004). *Timing.* Springer Science & Business Media.

[Sch14] Rudolf Scheifele (2014). Steiner trees with bounded RC-delay. *Proceedings of the International Workshop on Approximation and Online Algorithms (WAOA).* Springer, pp. 224–235.

[Sch15] Ulrike Schorr (2015). Algorithms for Circuit Sizing in VLSI Design. *PhD thesis, Research Institute for Discrete Mathematics, University of Bonn, Germany.*

[Sch09] Werner Schwärzler (2009). On the Complexity of the Planar Edge-Disjoint Paths Problem with Terminals on the Outer Boundary. *Combinatorica* 29.1, pp. 121–126.

[SL05] Weiping Shi and Zhuo Li (2005). A Fast Algorithm for Optimal Buffer Insertion. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.6, pp. 879–891.

[SLA04] Weiping Shi, Zhuo Li, and Charles J Alpert (2004). Complexity Analysis and Speedup Techniques for Optimal Buffer Insertion with Minimum Cost. *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. IEEE, pp. 609–614.

[SS05] Weiping Shi and Chen Su (2005). The Rectilinear Steiner Arborescence Problem is NP-Complete. *SIAM Journal on Computing* 35.3, pp. 729–740.

[Tov84] Craig A Tovey (1984). A Simplified NP-Complete Satisfiability Problem. *Discrete Applied Mathematics* 8.1, pp. 85–89.

[Van90] Lukas P P P Van Ginneken (1990). Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay. *Proceedings of the IEEE International Symposium on Circuits and Systems*. IEEE, pp. 865–868.

[Vyg04] Jens Vygen (2004). *Near-Optimum Global Routing with Coupling, Delay Bounds, and Power Consumption*. Springer.

[Vyg16] Jens Vygen (2016). Chip Design. *Lecture Notes. Preliminary version.*

[Wei+13] Yaoguang Wei, Zhuo Li, Cliff Sze, Shiyan Hu, Charles J Alpert, and Sachin S Sapatnekar (2013). CATALYST: Planning Layer Directives for Effective Design Closure. *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. IEEE, pp. 1873–1878.

[Yan+06] Jin-Tai Yan, Yen-Hsiang Chen, Chia-Fang Lee, and Ming-Ching Huang (2006). Multilevel Timing-Constrained Full-Chip Routing in Hierarchical Quad-Grid Model. *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*.

[YL04] Jin-Tai Yan and Shun-Hua Lin (2004). Timing-Constrained Congestion-Driven Global Routing. *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*. IEEE, pp. 683–686.