Interconnect Optimization in Chip Design

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Benjamin Marc Rockel-Wolff

aus

Essen

Bonn, Juni 2024

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

Gutachter / Betreuer: Herr Professor Dr. Stephan Held

Gutachter: Herr Professor Dr. Jens Vygen

Tag der Promotion: 30.07.2024

Erscheinungsjahr: 2024

Acknowledgements

This thesis is the result of many years of work. Throughout these years I have been supported by many great people and I would like to take this opportunity to thank them.

First I want to thank my advisor Prof. Dr. Stephan Held for his guidance, support, collaboration and for his feedback. I want to thank Prof. Dr. Jens Vygen for his feedback and ideas in the many jour fixes and seminars.

I had a great time at the Institute for Discrete Mathematics and want to thank Prof. Dr. Bernhard Korte for creating such excellent working conditions and for setting up the cooperation with IBM that made this all possible.

Many thanks go to my current and former colleagues at the Institute: To Dr. Daniel Rotter for giving me such a warm welcome, when I started out as a student, Dr. Siad Daboul for mentoring me in the first years and the coffee rounds. A special thank-you goes to Benjamin Ihme, whom I was mentoring in turn almost during my whole time as a PhD student. I want to thank the whole coffee crew (past and present) Dr Siad Daboul, Dr. Pietro Saccardi, Dr. Tilmann Bihler, Dr. Benjamin Klotz, Stefan Rabenstein and Martin Drees. The coffee rounds were always a highlight of my days.

Among the IBM employees that I worked with, I especially want to thank Harald Folberth for providing many hard instances to improve on and Alex Suess for the excellent cooperation. Furthermore, I want to thank the BonnCell team, in particular Tobias Werner and Prof. Dr. Stefan Hougardy for the great time I had during the short period in which I supported them.

I want to thank all the people who proofread preliminary versions of this thesis: Martin Drees, Fine Foos, Benjamin Ihme, Benjamin Kästner, my mother Andra and my brother Thomas.

A great thank-you goes to my wife Sveni, my family and friends for their support during the whole period of writing this thesis. I would not have been able to finish this without you.

Contents

Introduction

Today's world is driven by the increasing computing power of computer chips like the one shown in Figure 1. More and more parallel cores and components are added to today's chips, from specialized cryptography units to AI-accelerators.

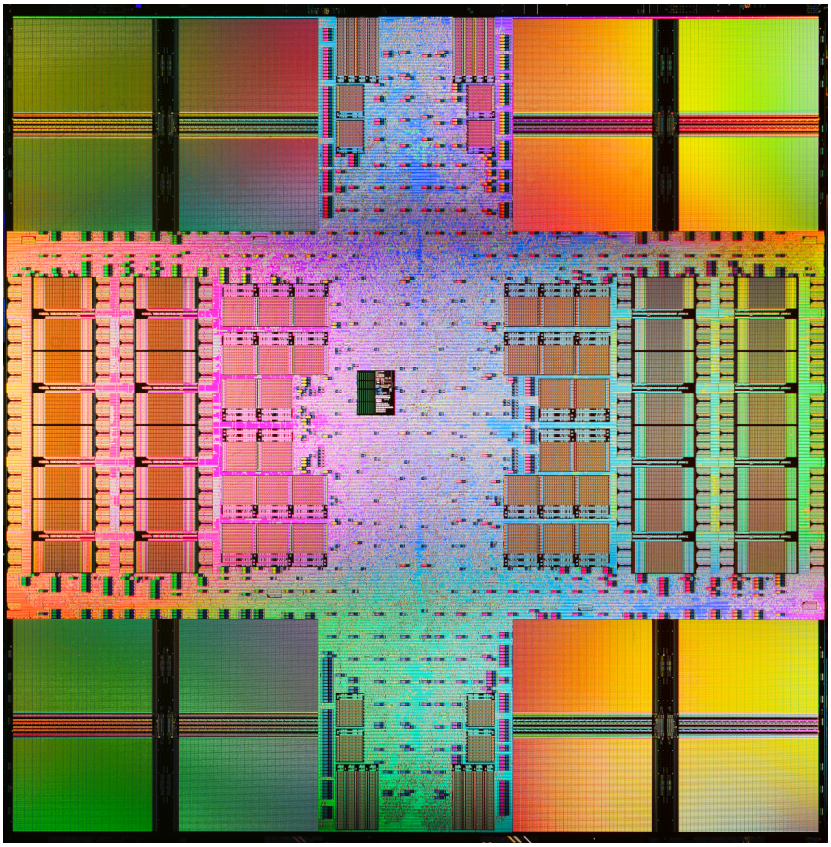
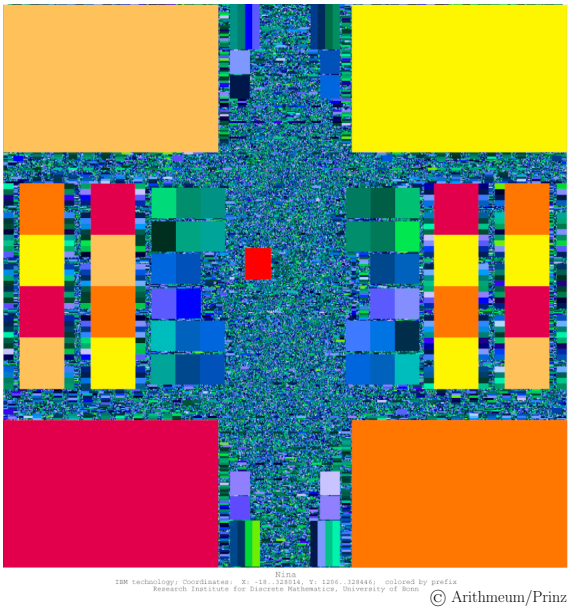
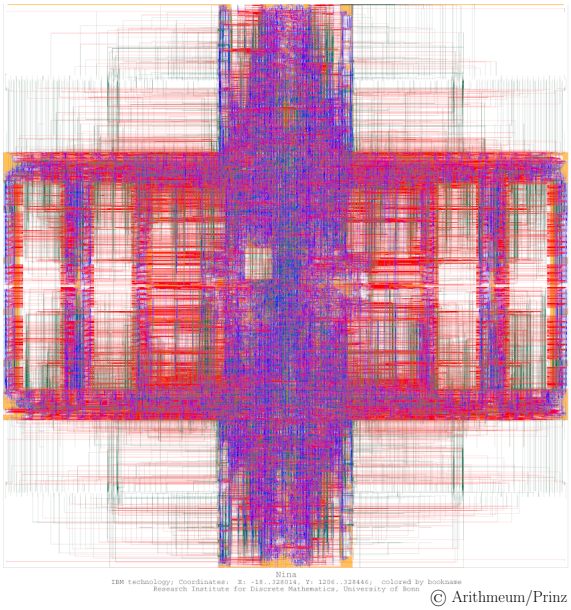


Figure 1: A photography of the PC layer of the chip Nina, published in "Mathematik und Ästhetik des Chipdesigns" [Hou+19].

To make this possible, feature sizes had to decrease significantly. It has also become impossible to design a state-of-the-art computer chip without electronic design automation (EDA) tools.



(a) The placement of the chip Nina, published in "Mathematik und Ästhetik des Chipdesigns" [Hou+19].



(b) The routing of the chip Nina, published in "Mathematik und Ästhetik des Chipdesigns" [Hou+19].

Figure 2: The colored data of the placement and the routing of the chip Nina that is also shown in Figure 1.

The BonnTools are a set of EDA tools that are developed by the Research Institute for Discrete Mathematics at the University of Bonn in collaboration with IBM. They deal with physical design. In physical design, millions of logic gates that together form the chip's logical functions need to be placed on the chip (see Figure 2a) and the connections transferring the signals between these gates (interconnect) have to be planned (see Figure 2b). The connections can in total make up several meters of wiring on a chip the size of a fingernail. Examples of the placement and routing of a chip can be seen in Figure 2.

There are the clear physical constraints. For example that the wiring should not contain short circuits and obeys more complex rules imposed by the manufacturing process. There are also optimization targets, like the cycle time and the power usage. Nonetheless, the cycle time is usually treated as a constraint. If the chip is specified to run at 4GHz for example, it must complete a cycle in 250ps. To put this into a relation: Light travels roughly 7.5cm in vacuum in this time. Our test chips range from 0.04mm to 5.4mm from one side to the other (see Chapter 6).

To meet the cycle time, we need to optimize the interconnect between gates with respect to timing, but we also need to keep power usage, netlength (total length of the wiring) usage and other objectives in mind. The interconnect distributes the signal from a gate's output, called source, to the inputs, called sinks, of subsequent gates. It consists of wiring and of additional gates, called repeaters. An example of interconnect with repeaters is depicted in Figure 3. A repeater implements a logical identity (buffer)

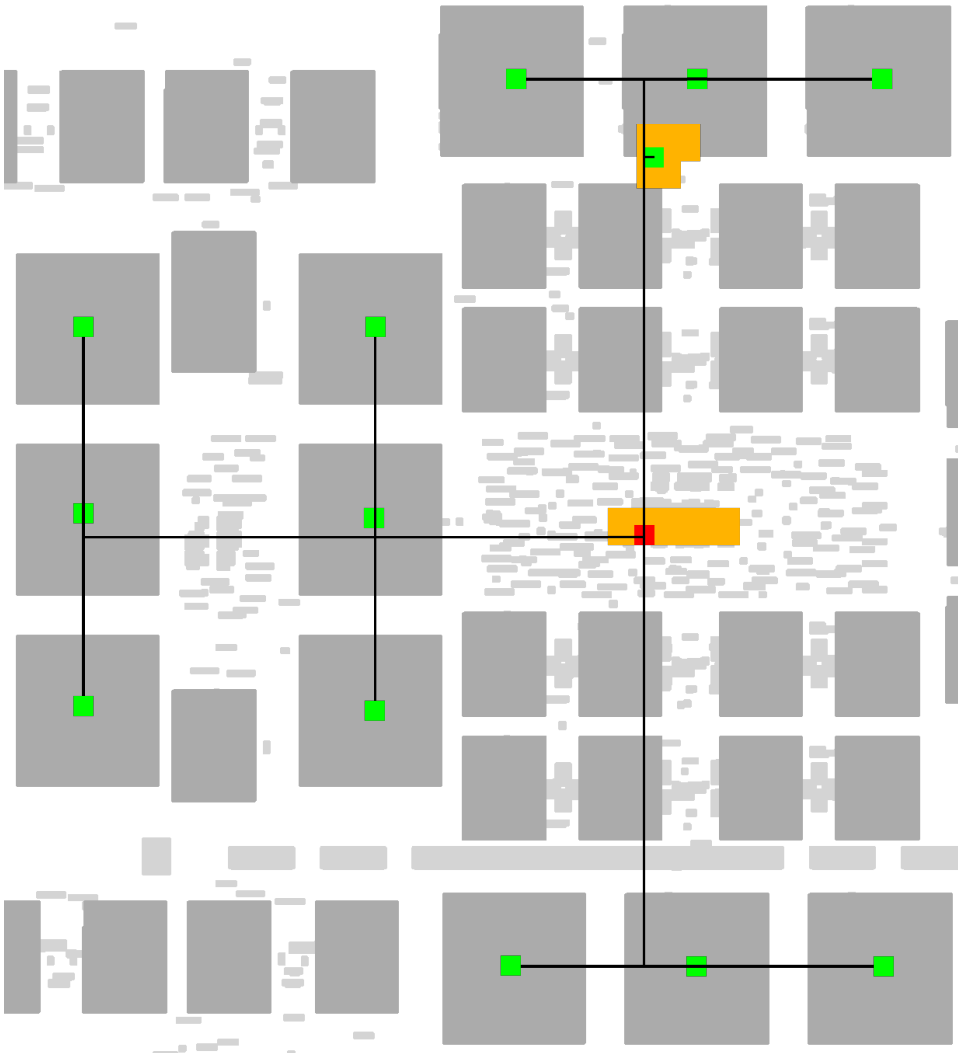


Figure 3: A buffered route (see Chapter 1.1.5) distributing a signal from the source in the center (red) to sinks (green) on some of the blockages. Blockages are shown in gray. Repeaters are orange, the wiring is black. The buffered route has been computed using our algorithm. This plot has been modified to make the repeaters and terminals visible, while still showing the whole instance. The solution contains 8 repeaters, but some of them are hidden beneath others in this picture.

or inversion (inverter) function. It shields away its downstream capacitance from its upstream wiring and strengthens the signal. The process of adding repeaters is called buffering. It is one of the hardest problems in chip design, because it combines discrete decisions with non-linear signal delays. No exact algorithm is known for the general buffering problem.

The delay through a wiring segment is roughly proportional to the wire's resistance and its capacitance with respect to other wires. We can change these properties by selecting

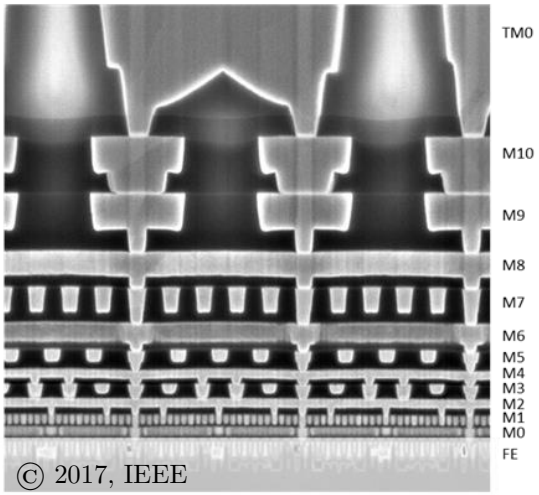


Figure 4: A cross-section of a metal stack in 10nm technology by Intel [Aut+17]. The wiring layers are numbered M1 – M10. They are connected by via layers. The upper layers feature both higher and wider wires.

different widths, spacings and materials. Higher widths lead to lower resistances, but higher capacitances. Higher spacing decreases the capacitance. Typically, chips have wider widths and spacings on higher layers, providing lower signal delays. A cross-section of a chip, showing the layers can be seen in Figure 4. The capacitance that a gate can drive is limited. So it is also required to add repeaters if the capacitance becomes too large.

With decreasing feature sizes, the influence of the wire resistance has increased more and more. With that, buffering has become a significant part of timing optimization. As resistance and capacitance of a wire grow about linear in their length, wire delay grows roughly quadratically. The delay through an optimally buffered long connection only grows approximately linear [Bar+10] in its length. Additionally, buffers can speed up timing critical connections by shielding away the capacitance of uncritical connections.

Because buffering is so important, it has been widely studied and many algorithms have been proposed and used in practice. From algorithms to limit the load at each source ([BCD89], [Tou90]) over van Ginneken's dynamic program [Van90] that inserts buffers (of a single type) into given wiring with respect to timing and its derivatives, to global solutions like BonnRouteBuffer [Dab+23]. They add aspects like choosing from multiple buffers and also inverters [LCL96a], selecting wire types and layers for the wiring [Zho+99] and trading off resource usage globally [Dab+23]. An extensive review of the literature can be found in Chapter 2.1.

Despite the extensive research on buffering, even state-of-the-art algorithms often leave a few instances optimized insufficiently. In these cases designers buffer them by hand. Figure 3 shows our solution to an instance that had been buffered by a designer for the chip release. Another problem is that it is hard to judge if a buffering is good or how it could be improved. Our goal is to solve these problems.

In this thesis, we aim to provide a flexible formulation of the buffering problem that allows to capture today's most important aspects of the buffering problem, and provide an optimum but exponential time algorithm for this formulation. We created a framework that can be used to solve buffering problems with varying levels of accuracy. From an almost optimum at a high running time — for example for benchmarking — to a reasonably fast, but more inaccurate variant that can be used in a high effort mode for very timing critical instances in real world applications as part of the BonnTools.

In Chapter 1 we present some foundations. We start by presenting the mathematical objects that describe the chips and tasks that have to be performed during physical design. Then we give an introduction to (static) timing analysis for integrated circuits. Finally, we give a quick introduction to path searches, Steiner trees and resource sharing, which will occur multiple times in this thesis.

Then we take an in-depth look at the buffering problem in Chapter 2. We review the existing literature and previous work. Then we give an overview over the constraints and objectives of the buffering problem and how they can be taken into account. Finally, we present our new formulation of the buffering problem, enhancing [Rot17]. Similar to previous works it can not place repeaters or wires at arbitrary positions, but it covers a wider range of objectives and constraints than its predecessors. To our best knowledge, it is the first formulation that can accurately optimize with respect to realistic delay models with slew propagation.

Once we have presented our problem formulation, we propose our new algorithm in Chapter 3. We show that it can be used for finding approximate and optimum solutions to our problem. Afterwards, we present two extensions to our algorithm that are joint work with Benjamin Ihme.

In Chapter 4 we present speed up techniques that we developed for our algorithm. Among these is a new timing based iterative clustering heuristic that uses a non-timing aware clustering algorithm as a black box.

As the black box clustering algorithm, we use a new 3-approximation algorithm for the Capacitated Tree Covering Problem with Edge Loads, which we present in Chapter 5. A short version of this chapter has been accepted at the SWAT 2024 conference [Roc24].

Finally, in Chapter 6, we will test our new algorithm and speedup techniques on practical instances. First, we run a comparison of the base implementation of our algorithm (benchmark setting) that can give us an approximation guarantee to versions with speedup techniques that do not give us any guarantee. These experiments are conducted on small instances, because the base version is too slow for larger instances. We show that some of the techniques have only a small negative impact on the result, but reduce the running time significantly on these instances. In a second set of experiments, we then test our faster version in practical settings and show that it can be useful for hard cases.

1 Preliminaries

Before we can start to deal with the buffering problem, we need to lay some foundations. This chapter aims at providing basic knowledge on the problems and constraints that we encounter, as well as concepts and ideas that are used during this thesis. Furthermore, we set up the basic notation that is used during this thesis for the different problems.

We start off by giving an introduction to VLSI design (Very Large Scale Integration), based on the lecture notes of the "Chip Design" lecture at the University of Bonn by Stephan Held and Jens Vygen [HV22]. First, we explain the basics of integrated circuits. Then we present the basic data structures used during the physical design process. From there, we set up the main steps of physical design and explain which constraints have to be satisfied. Finally, we give the basic formulation of the buffering problem, which is the main problem discussed in this thesis.

In the second section, we give an introduction to timing analysis, based on the book "Timing" by Sachin Sapatnekar [Sap04]. As our main goal will be to optimize interconnect with respect to timing, timing analysis naturally plays an important role, both for the modelling of our problems and for the solution. We start by explaining how circuits are modelled for the purpose of fast timing analysis and then go from a simple simulation of the interconnect wiring to deriving two commonly used delay computation methods: The Elmore delay and the asymptotic waveform evaluation.

In the last section of this chapter, we present basic problems of combinatorial optimization that play an important role for our approach to the buffering problem. We start off with the basics of path search. Then we inspect different Steiner tree problems and finally, we take a quick look at the min-max resource sharing problem, because it can be used to derive a global formulation of many of the problems that have to be solved during VLSI design.

1.1 VLSI Design

The design of a computer chip is split into several steps. After specification and high-level-design the logic of the chip is defined in a hardware description language. Logic synthesis transforms the high-level description of the chip's logic into small Boolean functions like NAND or NOR, so-called gates, and a specification how these gates have to be connected. The physical design task then takes this set of logical gates and the description how these gates have to be connected, called a netlist, as input. It aims at selecting physical implementations of these gates from a so-called library, positions for all of them and interconnect wiring, such that this can be physically realized, does not consume too much power and meets the cycle time of the chip. Afterwards, verification checks that all goals are met in simulations. If there are still errors present, we go back to a previous step and fix these errors. Finally, test versions of the chip are produced and checked again.

We will focus on the physical design. In order to be able to algorithmically handle this process or parts of it, we require precise language and modelling. This section will establish the most important concepts, models and subtasks required to understand this thesis.

We will start off by giving a short introduction to the basic building blocks of computer chips, transistors, and the so called leaf-cells that are built from them. We continue by giving the basic definitions of a chip image, a netlist and a library. Then we will explain the basic notions of placement and routing. These are the tasks to find positions and interconnect wiring. And finally we will take a look at buffering, which is part of the effort made to meet the cycle time.

This section is based on and uses the notation of the lecture notes on "Chip Design" by Jens Vygen and Stephan Held [HV22].

1.1.1 Transistors and Leaf Cells

Modern computer chips consist of millions of transistors. A transistor is a semiconductor that, given a voltage between source and drain, lets an electrical current flow from its drain to its source contact, only if there is a high potential at its gate contact. This type of transistor is called an n-transistor. A p-transistor will let the current flow only if there is no potential at the gate contact. An example is shown in Figure 1.1.

A field effect transistor (FET) is built by doping the silicon crystal with atoms that have either an additional electron (n-type region) or a missing electron (p-type region). For an n-transistor, source and drain are n-type regions that are isolated by a p-type region. The p-type region is enclosed by a thin insulation and the gate, as shown in Figure 1.1. If a strong enough positive electric field is induced by the gate, an n-channel will open between source and drain. Otherwise, the additional electrons in the n-regions will fill the holes left in the p-region and the material will not be conductive. For p-type transistors, n- and p-regions are exchanged.

Figure 1.1 shows a FinFET transistor. The Fin in FinFET means that the p- and n- regions are organized such that the gate may surround them on multiple sides. This

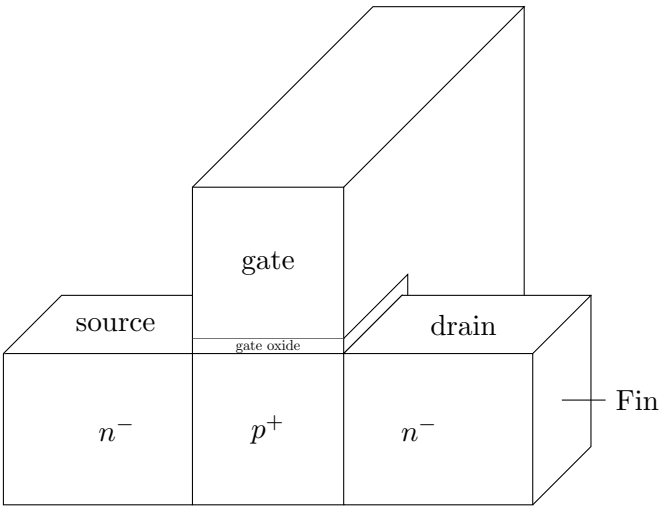


Figure 1.1: Cross-section of an n-transistor in FinFET technology.

technology has been used for the 14-nm and 7-nm technology nodes. Older technologies used planar transistors, where the gate would contact the substrate only from the top. For newer technologies, gate-all-around concepts are explored that increase the contact area even further [Kam22].

A chip's base layer is manufactured in rows with a p- and an n-stack in each row, allowing to place p- and n-transistors respectively. Each stack may have multiple fins, such that a transistor can use multiple fins [Cre19]. A piece of such a circuit row is shown in Figure 1.2.

Creating a state-of-the-art chip on transistor level is not feasible. This is why small logic functions are pre-built in a separate step called leaf cell design and can then be used to realize these logic functions. Often these small logic cells are called logic gates. Modern chips are built in Complementary Metal-Oxide-Semiconductor (CMOS) technology. This means that the gates are realized with complimentary p- and n- transistors such that for each output, a power-output channel is open if the output has the value 1, and a ground-output channel is open if the output has value 0. Figure 1.3 shows the diagram of a CMOS inverter.

1.1.2 Chip Image, Netlist and Library

We first give the definition of a *chip image*, which is a data structure capturing the boundary conditions of the chip. The chip image [HV22] is a tuple

$$\mathcal{I} = (A(\mathcal{I}), B(\mathcal{I}), P(\mathcal{I}), (p(t))_{t \in P(\mathcal{I})}).$$

Here $A(\mathcal{I}) = [x_{min}, x_{max}] \times [y_{min}, y_{max}] \times \{0, \ldots, z_{max}\}$ denotes the chip area. The third dimension denotes the layer, where layer 0 is the placement layer and the others are called routing layers and will contain wires. $B(\mathcal{I})$ is the set of blockages, which are rectangular subsets of the chip area on a single layer. $P(\mathcal{I})$ denotes the input and output pins of the chip and $p(t) \in A(\mathcal{I})$ denotes the position of the pin $t \in P(\mathcal{I})$.

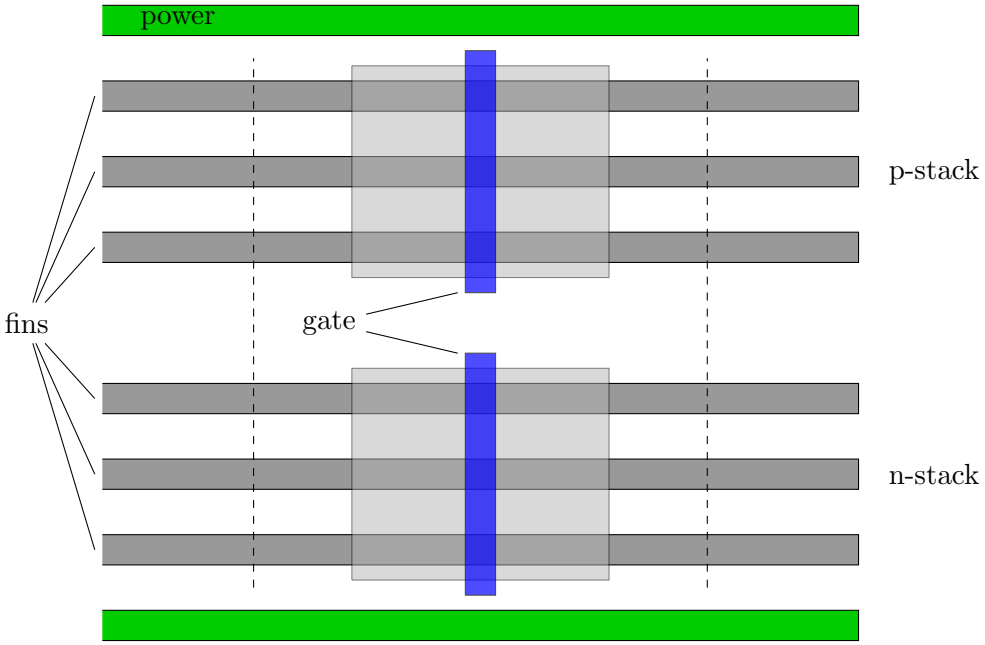


Figure 1.2: A small piece of a circuit row in FinFET technology.

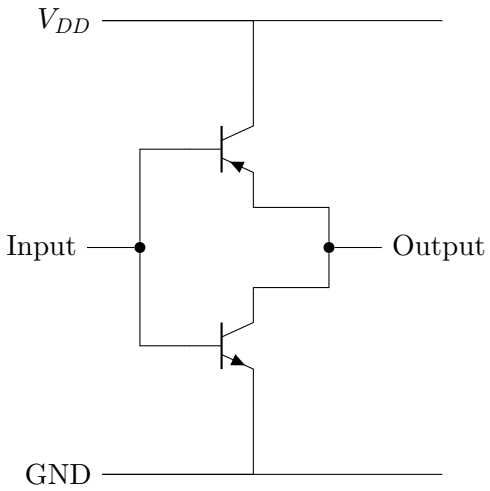


Figure 1.3: An inverter in CMOS technology. The p-transistor on the top will connect the output to V_{DD} only if a positive potential is applied to the input, while the n-transistor on the bottom will connect the output to ground only if no potential is applied to the input.

The logic of a chip is realized by logical elements, so-called circuits, that are connected by nets. Circuits are predesigned realizations of logical functions (logic gates) like NAND or NOR. However, circuits may also encode more complex functions like an adder circuit, or even a larger piece of the chips logic like a floating point unit or an IO-controller. These large functions are called macros and are designed using the same techniques as described here. There is a third class of circuits that do not encode logical functions: Memory elements like latches or flip-flops are able to store single bits and carry them over to the next cycle. The logic of the chip without memory elements is called combinational logic and is acyclic. The nets carry the input signals of a chip and the output signals of circuits to inputs of other circuits or outputs of the chip.

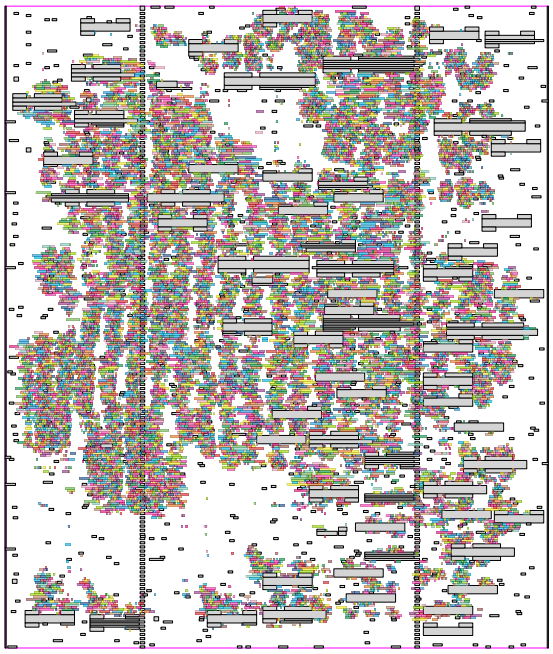
Formally, we store this information in a *netlist*

$$(\mathcal{C}, P = P_{in} \dot\cup P_{out}, \gamma, \mathcal{N}).$$

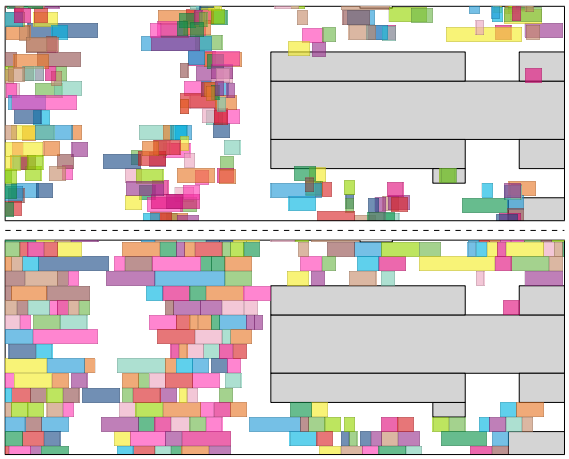
$\mathcal{C}$ is the set of circuits. P is the set of input/output pins, where P_{in} stores the inputs, and P_{out} stores the outputs. $\gamma : P \to \mathcal{C} \cup \{\square\}$ maps the pins to the circuits, or to $\square$ if the pin is an input/output of the chip. The pins belonging to the chip are called primary inputs/outputs. $\mathcal{N} \subset 2^P$ contains the nets. The pins N of a net are called terminals. A net contains exactly one source pin that is either a primary input or an output of a gate. Additionally, it contains at least one sink pin that is either a primary output or an input of a gate.

To each gate, we need to associate the information, how it is built. This information is stored in a so called *book* b. It need not be complete during each step of the design process. But we will assume that it always contains a (rectangular) outline $A(b) = [0, \mathit{width}(b)] \times [0, \mathit{height}(b)] \subset \mathbb{R}^2$, called its shape, the pins of the book $P(b)$ togther with positions for its pins $p : P(b) \to \mathbb{R}^2_{\geq 0} \times \{1, \ldots, z_{max}\}$. The x- and y- coordinates of the pins are relative to the lower left corner of its shape. Additionally, information on the book's power consumption and on its timing properties are stored. For macros, there is typically only one book, but standard gates like NANDs or inverters usually have different possible *sizes* and *V_t-levels*, represented by different books. The V_t level refers to a gates switching voltage. A lower switching voltage will decrease the delay through a gate, but increase its power consumption. For increasing sizes, the gate is built of larger or multiple parallel transistors that can drive a higher load. This usually increases the area of its shape and its power consumption, but decreases the delay for larger loads. The process of choosing the sizes for the gates is called *gate sizing*.

The set of all books L is called *library*. During physical design, we will always have a technology mapping (β, π), where $\beta : \mathcal{C} \to L \cup \{\mathcal{I}\}$ maps the circuits to books or the chip $\mathcal{I}$ itself and $\pi : P \to \{I\} \cup \bigcup_{b \in L} P(b)$ is a one to one mapping of the pins of each gate to the pins of the circuits book and the primary pins to $\mathcal{I}$. We require that β and π are consistent with the netlist and logic of the chip. This means that $\beta(C)$ is a book realizing the logical function of $C \in \mathcal{C}$ and $\pi(t)$ must map primary inputs and outputs to $\mathcal{I}$ and outputs of circuits to outputs of the book, as well as sink pins to inputs of the book.



(a) A legal placement of a chip. The gray boxes are blockages, the coloured boxes are the placed gates.



(b) A small region of the chip in Figure 1.4a. On the top we see a state after global placement and before legalization, on the bottom we see the state after legalization.

Figure 1.4: A placement of a chip. Once complete, and once a small subregion before and after legalization.

1.1.3 Placement

The task of placement is to find positions and orientations for each circuit in the chip area, such that the design rules are satisfied. The most basic rule is that the outlines of the circuits should not overlap. The rules can be more complicated than that. One important example is that the substrates for the transistors are usually organized in rows, such that on one row, only p-transistors can be placed and on another row only n-transistors can be placed (see Figure 1.2). The circuits must be placed aligning with the circuit rows.

Let us formally define a placement. We will ignore orientations here, as we do not need them for the core of this thesis. A *placement* of the circuits $\mathcal{C}$ is a mapping $(x, y) : \mathcal{C} \to \mathbb{R}^2$. The placement shape of $C \in \mathcal{C}$ is then the rectangle resulting from moving the lower left corner of $A(\beta(C))$ to the position $(x(C), y(C))$. The position of a pin $t \in P$ is then $(x(\gamma(t)), y(\gamma(t))) + p(\pi(t))$.

We call a placement *legal* if it obeys the design rules. In particular, we require that the placement shapes of all circuits lie completely within the chip area, they do not intersect with each other, and they do not intersect with blockages. For an example see Figure 1.4.

Additionally, being legal with respect to the design rules is not sufficient. A placement must allow a wiring to exist and since the delay increases with the distance, it also heavily influences the timing of the chip.

There are two main placement steps. Global placement starts with no given placement and tries to spread the circuits across the chip such that the density of circuits is not too large and an estimate of the length of the wiring and the future timing is minimized. Legalization starts with a given, possibly non-legal placement and tries to move the circuits to legal positions, often while minimizing the total squared movement. The picture in Figure 1.4b shows a small region of a chip after global placement in the top half and after subsequent legalization in the bottom half.

1.1.4 Routing

When all circuits are placed, we can compute a routing. The task here is to find wires for each net that connect all the pins of the net. Let us view this more formally. A routing of net $\{r\}\dot{\cup}T = N \in \mathcal{N}$ with source r and sinks T is an arborescence A, rooted at r and with leaves T, together with positions $pos : V(A) \to A(\mathcal{I})$ and a set of rectangles S, called shapes, that represent the wires' outlines in the plane. All edges must correspond to either horizontal or vertical lines or connect two points with the same x and y coordinate on subsequent layers. This means that for $(v, w) \in E(A)$, $pos(v)$ and $pos(w)$ should differ in exactly one coordinate. Edges with both endpoints on the same routing layer are called wires, while edges with endpoints on different layers are called vias. The direction of a wire is horizontal if the x-coordinate changes and vertical if the y-coordinate changes. Examples of wires and their shapes can be seen in Figure 1.5b. Each routing layer has a preferred direction in which the wires are supposed to go. If a wire's direction is orthogonal to the layer's preferred direction, we call it a jog. However, we will not allow jogs in this thesis.

In reality, a wire is a cuboidal piece of metal with the wire's edge centered in its cross-section and enclosed by it. Its height is fixed for each layer, so we may represent it by a rectangle on the wire's layer (see also Figure 4). The length of the rectangle in the wire's direction is called the wire's length, while the length in the orthogonal direction is called its width.

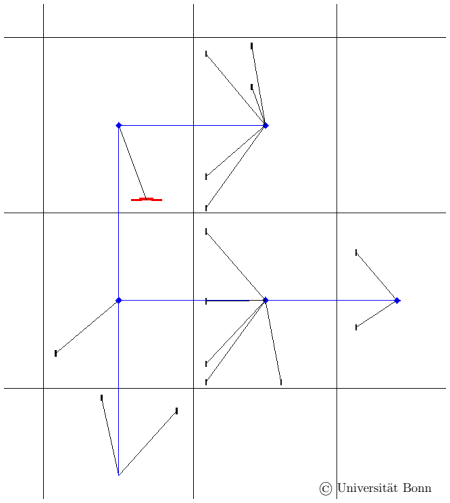
Usually there is a discrete set of allowed widths for the wires, together with a minimum required spacing to parallel wires on the same layer. These sets of width and spacing combinations are called *wire types*. With increasing width the capacitance of a wire grows, but its resistance shrinks. With increasing spacing the capacitance shrinks.

Vias can be represented by two rectangles: A top shape on the upper layer, and a bottom shape on the lower layer.

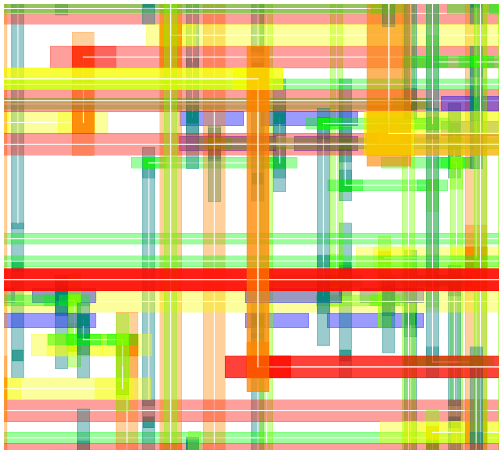
S contains exactly the rectangles belonging to the wires and vias.

A legal routing of the chip is a routing for all nets that does not violate any design rules. These incorporate, at least, that no shapes of routes for two different nets may intersect, but there may also be more complicated rules like a minimum length of wires.

Because routing is so computationally complex, it is often split into two steps: Global routing and Detailed routing.



(a) A global route (in blue). The root is shown in red, the sinks in black. Black lines connect the terminals and the tile center they are projected to. Blue vertices are Steiner points of the global route.



(b) A small region of a chip with detailed routing. The dark blue rectangles are pins, the remaining rectangles are the shapes of wires and vias. The colours show the layers. More blue means lower, more red means higher. The white lines are the edges in the arborescence belonging to a route. Power connections are left out for clarity.

Figure 1.5: A global route and a small region of the chip with detailed wires.

For global routing, we partition the chip into rectangular tiles. Then we construct the global routing graph. We insert a vertex in the center of each tile. Then we connect vertices of neighbouring tiles in preferred direction by an edge. Furthermore, we add edges between the vertices of tiles with the same x and y coordinates on subsequent layers. Now we assign a capacity to each edge, based on the number of detailed edges that may cross the tile border belonging to that edge. We project each pin to the vertex of the tile it is contained in. Then we can compute a routing in the global routing graph connecting the projected pins. Instead of requiring that the routes do not intersect, we require that not too many routes use the same global routing edge. This can be done by computing an estimate of how much of the capacity of a global routing edge is used by each route and minimizing the overusage. An example of a global route is shown in Figure 1.5a.

Detailed routing then tries to find routes that obey all the design rules (see Figure 1.5b). It uses the global routes as a basis to reduce the search space.

1.1.5 Buffering

Placing the circuits and connecting them by wires will not be sufficient to meet the cycle times on modern chips. We will informally define the delay from one pin of a chip to a pin that lies downstream as the time it takes for a signal originating at the first pin to reach the second pin. We will discuss timing and delay in more detail in the second section of this chapter. The delay through a gate depends roughly on the total capacitance of the wiring connected to its output pin. The delay through a wire segment depends on the resistance of the segment, its capacitance and the downstream capacitance at the end of the segment. Since resistance and capacitance of a wire grows roughly linearly with its length, the delay from a source of a net to a sink of the same net depends roughly quadratically on the length of the wiring in between. Furthermore, it also depends on the capacitance of wiring that connects other sinks in the same net to the source. To meet the cycle times, it is necessary to speed up connections by inserting *repeaters*. Repeaters are logical identity functions, called *buffers*, or negation functions, called *inverters*. If inverters are inserted, we also need to make sure that the logic does not change. A repeater shields away the capacitance of its downstream wiring from the wiring before the repeater. By inserting a repeater on a side branch of a path, we can shield away the capacitance of the side branch, thus reducing the influence of the side branch on the delay through our path. By inserting repeaters on a path, we can reduce the quadratic dependence on the length of the path roughly to a linear dependence. An example for the development of the delay per length with and without a repeater is depicted in Figure 1.6. Historically, only buffers were used as repeaters. This is why this process of inserting repeaters is called *buffering*. This process changes the netlist.

Let us now give a basic formulation of the buffering problem. We will spend the whole chapter 2 of this thesis to discuss how to model the buffering problem such that we incorporate as many constraints as possible, while still being able to solve it.

We are given a net $\{r\} \dot{\cup} T := N \in \mathcal{N}$ with source r and sinks T. Additionally, we are given polarities $pol : T \to \{ident, invert\}$ and a repeater library L consisting of buffers

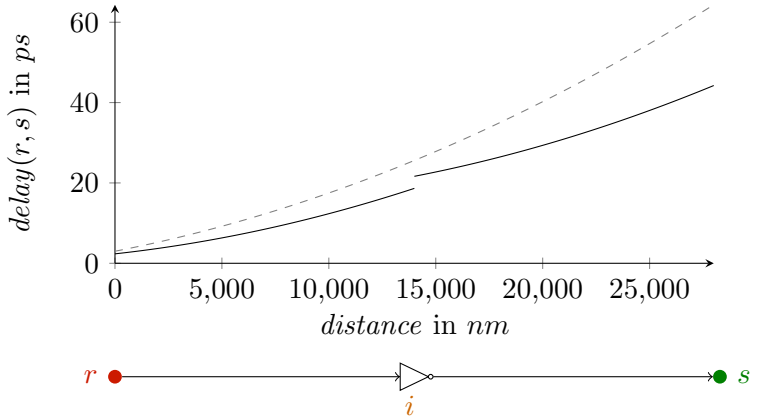


Figure 1.6: A path with a repeater in the center. The graph shows the delay to the point on the path at the x-coordinate. The solid graph is the delay with the inverter, the dashed line without the repeater.

and inverters. We want to construct a buffered route (see Figure 1.7). Its base is a repeater tree that encodes the changes to the netlist, that we need to make. It stores the nets as an arborescence, where each non-leaf vertex and its children make up a net. The repeaters are stored information on the vertices. In order to keep the logic correct, we must make sure that the number of inverters on each path from the source to a sink with *invert* polarity is odd. An example is depicted in Figure 1.7a. The buffered route then also adds a route for each new net and is shown in Figure 1.7b. Formally we define repeater tree and buffered route as follows:

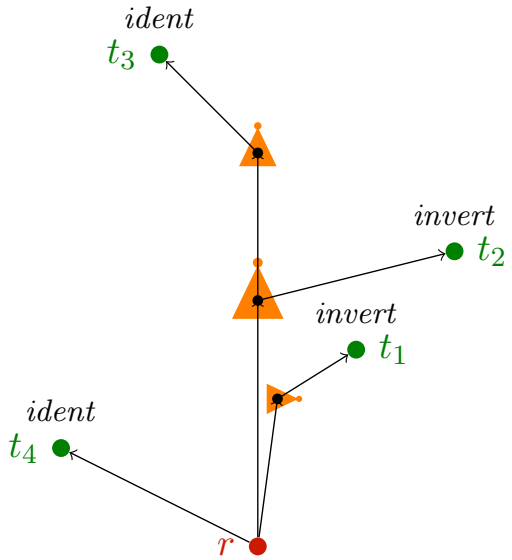
Definition 1.1.1. *A* repeater tree *for* (r, T, pol) *is an arborescence* A*, rooted at* r *with leaves* T *and positions* $pos : V(A) \setminus (\{r\} \cup T) \to \mathbb{R}^2$*, together with a mapping* $b : V(A) \to L \cup \{\square\}$*, such that* $b(r) = b(t) = \square$ *for each* $t \in T$ *and* $b(v) \in L$ *for* $v \in V(A) \setminus (\{r\} \cup T)$ *and we have*

$$\left|\{v \in V(A_{[r,t]}) \mid b(v) \text{ inverter}\}\right| \text{ odd iff } pol(t) = invert.$$

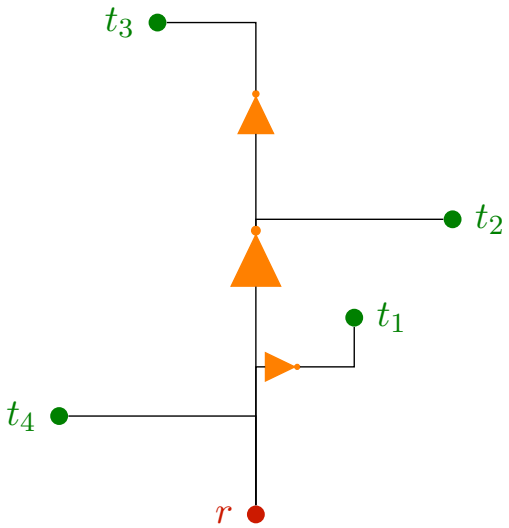
Denote by v_{source} *the source pin and by* v_{sink} *the sink pin of* $b(v)$ *for* $v \in V(A) \setminus \{r\} \cup T$*. For simplicity, we use the convention that* $r_{source} := r$ *and* $t_{sink} := t$ *for* $t \in T$*.*

A buffered route *is a repeater tree* (A, b) *together with a route for each net* $N_v := \{v_{source}\} \cup \{w_{sink} \mid (v, w) \in E(A)\}$ *for* $v \in V(A) \setminus T$*.*

The task of buffering is to find buffered routes for each net in the netlist, such that the cycle times are met and the placement and routes are legal. This task is extremely complex. So usually the constraints are relaxed.



(a) A repeater tree. The polarities are shown as black text. The repeaters are drawn in orange at their respective node. The sizes indicate different repeater sizes.



(b) A buffered route based on the repeater tree from Figure 1.7a. The black lines show the projection of the routes into the plane. Polarities and the arborescence are left out for clarity.

Figure 1.7: A repeater tree together with a buffered route. The root is red, the sinks are green and the repeaters are orange.

1.2 Timing Analysis

In this section, we will deal with the question of how to compute the delay through the combinational logic of the chip. The modelling choices for the buffering problem heavily rely on properties of the selected method of timing computation. This is usually not part of the curriculum of the studies of mathematics. This is why we provide the basics of timing computation here. For further reading we refer to the book "Timing" by Sapatnekar [Sap04], on which this section is based.

We will start by showing how to model circuits and how to describe them by a system of algebraic and differential equations. This system can be used to model the voltages at every pin of the chip over time. However, doing this is computationally expensive and requires knowledge of the whole chip. So it is not suitable for use in optimization algorithms.

We will make the assumption that each stage of the logic can be considered separately. A stage consists of a logic gate together with the subsequent wiring. Knowing the curve of voltage over time at the inputs of the logic gate, we can simulate the voltage curves at the input pins of subsequent logic gates. The curve of voltage over time is also called waveform. Simulating the waveforms is still too slow. So we restrict to two features of the waveform that have proven to be very helpful in timing analysis. The 50% voltage pass and the slew, which is a measure for the slope of the curve. Then we will derive the two most commonly used delay computation methods: The Elmore delay and the AWE technique.

Finally, we will define the timing graph and show how to use it to compute the timing of the chip.

1.2.1 Modeling Circuits

Suppose we are given the schematic of an electrical circuit, like the one in Figure 1.9b. We want to model the circuit as a directed graph with an electrical device associated to each edge, as described in [Sap04, Chapter 2.2]. This can be achieved by first inserting one node n_0 for ground. Then we insert nodes at each electrical node and between electrical devices on the same branch. Finally, we add edges between the enclosing nodes of each electrical device, directed in the same direction as the electrical flow. Note that a voltage or current source counts as an electrical device as well. An example of a circuit with its model is depicted in Figure 1.9 b and c.

Formally, a model of a circuit is a directed graph $G = (V, E)$ with a ground node $n_0 \in V$, together with electrical devices D and a mapping of the edges to the devices $d : E \to D$. We are interested in the potentials $\pi : V \to \mathbb{R}$ on the nodes, where we fix $\pi(n_0) := 0$, as well as the voltages $v : E \to \mathbb{R}$ and currents $I : E \to \mathbb{R}_{\geq 0}$ on the edges.

They are subject to Kirchhoff's laws. Kirchhoff's current law (KCL) states that the sum of the outgoing current of each node must equal the sum of the incoming current:

$$\sum_{e \in \delta^-(n)} I(e) = \sum_{e \in \delta^+(n)} I(e) \text{ for each node } n \in V.$$

Kirchhoff's voltage law (KVL) states that the voltage around an undirected cycle in G must be 0. This means, if we walk around a cycle, adding up voltages of edges that we use in their direction and subtracting voltages of edges that we use in reverse direction, this sum must be 0. However, this law is implied by the definition of voltages as the difference of the potentials at their endpoints:

$$v((p,q)) = \pi(p) - \pi(q) \text{ for each edge } (p,q) \in E.$$

Note that we may set $\pi(n_0) := 0$ without violating Kichhoff's voltage law and that the current equation for n_0 is implied by the other KCL equations. So we may remove n_0 and its KCL equation.

The relation between voltages and currents is then given by the device equations, which may be nonlinear or even differential equations. We will provide the most important device equations for our purpose.

The simplest device that we will see is a resistor. Let $e \in E$ be an edge such that $d(e)$ is a resistor with resistance R. Then $v(e) = R \cdot I(e)$. Another simple device is a constant voltage source. If $d(e)$ is a constant voltage source providing a voltage w, then $v(e) = w$.

Finally, we need the device equation of a capacitor. This will be a differential equation, as the capacitor has to be charged and thus is time-dependent. If $d(e)$ is a capacitor with capacitance C, the equation becomes: $I(e,t) = C \cdot \frac{dv(e,t)}{dt}$.

Suppose now that we had a circuit that consists of a constant voltage source at the input and some capacitors and resistors. Then we have everything required to numerically compute the voltage curves on every edge. Indeed, we now have everything that we need to describe the response of interconnect wiring to a step input. This type of timing model that we describe here is called voltage source model. The source can also be a current source. In this case, we speak of current source models.

1.2.2 Modeling Interconnect Wires

The three important characteristics of interconnect wires that have an impact on the waveform are the resistance, the self inductance and capacitance to ground and to other wires. These three properties are typically referred to as RLC (derived from their physical symbols). However, for our purposes, we can ignore the self inductance. The resistance depends on the wire's length, and its cross-section. For now, we can assume the cross-section to be constant. This means that the resistance depends mainly on the length of the wire. The capacitance of a wire mainly depends on the surface area directly facing the surface of wires of other nets (coupling capacitance) and ground (ground capacitance). We will make another simplification and assume that there are evenly spread wires with some given density placed around our wires. This way, we can get an estimate for the capacitance that mainly depends on the length, the spreading, and the width and height of the wire. The latter three can be considered to be constant for now. So the wire's capacitance also mainly depends on its length.

Optimally, we would then model a wire by infinitesimally small segments of resistors and at each node an infinitesimally small capacitor to ground. This is not feasible for

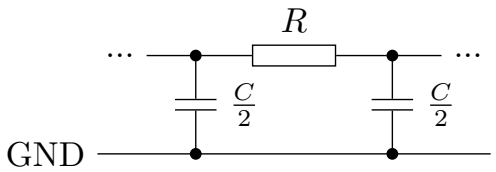


Figure 1.8: A π model for a wiring segment with resistance R and capacitance C.

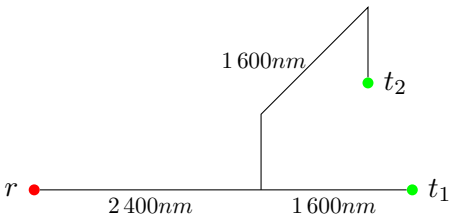
repeated computations. So, we instead approximate our wire by first cutting it into segments of some fixed finite length. Let C be the capacitance of this segment, and R be its resistance. Each of these wiring segments will be modeled by two nodes connected by an edge. At each node, we insert a capacitor to ground with capacitance $\frac{C}{2}$ and we connect the two nodes by a resistor with resistance R. This is called a π model [Sap04, Chapter 3.2] and can be seen in figure 1.8. If we represent a whole net like this, we call the representation an RC-tree. An example can be seen in figure 1.9b. If we merge parallel capacitors and resistors in a row, we can make two observations for the resulting RC-tree. First, each edge that is not incident to ground, belongs to a resistor. Second, each node that is not incident to the power source is incident to exactly one capacitor. This leads us to the definition of a canonical RC-tree, which will make subsequent notation much easier.

Definition 1.2.1. *Let $N := \{r\} \cup T$ be a net, A an arborescence that represents its wiring, $cap : E(A) \cup T : \mathbb{R}_{\geq 0}$ be the capacitance of the edges and sinks and $res : E(A) \cup \{r\} \to \mathbb{R}_{\geq 0}$ the resistance of the edges and the root.*

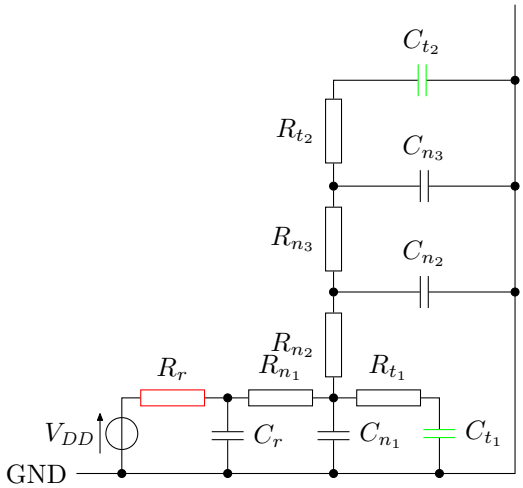
We define the canonical RC-tree *G for A by the following construction: Start with a copy of A and add nodes n_0 and n_{pwr} together with edges $e_{pwr} = (n_0, n_{pwr})$ and (n_{pwr}, r). Then add edges (v, n_0) for $v \in V(A)$. For each node $v \in V(A)$ denote by e_v the unique incoming edge of v in G and by c_v the edge from v to n_0. Set $d(e_{pwr})$ to be the power source and for $v \in V(A)$ let $d(e_v)$ be a resistor with resistance $R_v := res(e_v)$ and let $d(c_v)$ be a capacitor with capacitance $C_v := cap(v) + \frac{cap(e_v)}{2}$ if $v \in T$ and $C_v := \frac{1}{2} \cdot \sum_{e \in \delta_A(v)} cap(v)$ otherwise.*

1.2.3 Computing a Voltage Curve

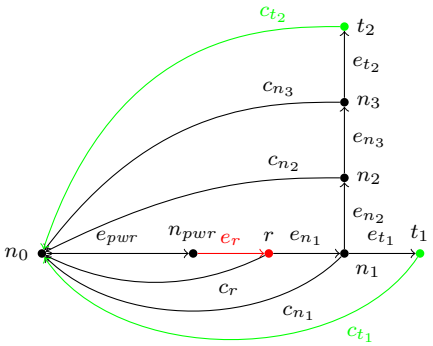
As an example, let us numerically approximate the voltage curve at the sinks for a step input to the wiring given in Figure 1.9a using the techniques from [Sap04, Chapters 2.2, 2.5]. We model the driver as a constant voltage source and a resistor. Furthermore, the capacitance of the vias in our example is negligible, so we only model them as resistors. We end up with the circuit diagram that is shown in Figure 1.9b. We create the canonical RC-Tree G that is shown in Figure 1.9c. Setting $X := V(G) \setminus \{n_0, n_{pwr}\}$, we end up



(a) An example wiring for a small net.



(b) The circuit diagram of an RC-tree modelling the wiring in Figure 1.9a with a root resistance. Parallel capacitances with no separating resistor are joined to a single capacitor. Via capacitance is negligible, so we only model the resistance.



(c) The canonical RC-tree for the wiring in Figure 1.9a and the graph model arising from the diagram in Figure 1.9b.

Figure 1.9: Example of a wiring, the circuit diagram belonging to its canonical RC-tree and the canonical RC-tree.

with the following set of equations, describing our system:

$$\begin{aligned}
\sum_{e\in\delta^-(n)} I(e,t) &= \sum_{e\in\delta^+(n)} I(e,t) && \text{for } n \in V(G)\setminus\{n_0\} && \text{(KCL)}\\
v(e_n,t) &= \pi(p,t) - \pi(n,t) && \text{for } (p,n) = e_n \text{ with } n \in X &&\\
v(c_n,t) &= \pi(n,t) && \text{for } (n,n_0) = c_n \text{ with } n \in X && \text{(KVL)}\\
\pi(n_{pwr},t) &= V_{DD} && && \text{(VDD)}\\
v(e_n,t) &= R_n \cdot I(e_n,t) && \text{for } n \in X && \text{(resistors)}\\
I(c_n,t) &= C_n \cdot \dot{v}(e_n,t) && \text{for } n \in X && \text{(capacitors)}
\end{aligned}$$

Now we want to transform our system to reduce the number of variables and equations. First, we will replace all voltages by the potentials using the KVL equations. This changes the equations for the resistors and capacitors:

$$\begin{aligned}
\pi(p,t) - \pi(n,t) &= R_n \cdot I(e_n,t) && \text{for } e_n = (p,n) \text{ with } n \in X && \text{(resistors)}\\
I(c_n,t) &= C_n \cdot \dot{\pi}(n,t) && \text{for } c_n = (n,n_0) \text{ with } n \in X && \text{(capacitors)}
\end{aligned}$$

The voltages do no longer occur in other equations than the KVL equations. So we can remove the voltages and KVL equations from our system and compute the voltages on demand, when we know the potentials. Next, we rewrite the resistor equations as

$$I(e_n,t) = \frac{\pi(p,t) - \pi(n,t)}{R_n} \text{ for } e_n = (p,n) \text{ with } n \in X.$$

We use this to replace all currents of edges belonging to resistors in the KCL equations. Then, we remove the resistor equations and currents for edges belonging to resistors from our system. We can again compute them on demand, when we know the potentials.

Finally, we need to deal with the differential equations for the capacitors. Let $c_n = (n,n_0)$ with $n \in X$. We want to approximate the solution using the forward Euler method [Sap04, Chapter 2.5]. This method examines the Taylor expansion of $\pi(n,t)$ around some known point a

$$\pi(n,a+h) = \sum_{k=0}^{\infty} \frac{h^k}{k!} \left.\frac{d^k\pi(n,t)}{dt^k}\right|_a.$$

The equation for our capacitor gives us the first derivative of $\pi(n,t)$. So we truncate the Taylor expansion after the first derivative to approximate the value at the point $a+h$ for some step $h > 0$:

$$\pi(n,a+h) \approx \pi(n,a) + h\dot{\pi}(n,a) = \pi(n,a) + \frac{h}{C_n} I(c_n,a).$$

The right-hand side here is a constant that depends on the state of the system at a previous point in time. This means that, if we know the values of the system at one point in time, we can approximate its value at a small timestep in the future using this

equation. We will choose a timestep $h > 0$ and set $\tau_k := k \cdot h$ for $k \in \mathbb{N}$, and replace the equations for the capacitors at timestep $k > 0$ using the forward Euler method:

$$\pi(n, \tau_k) = \pi(n, \tau_{k-1}) + \frac{h}{C_i} I(c_n, \tau_{k-1}) \text{ for } c_n = (n, n_0) \text{ with } n \in X$$

Let us write the whole system of linear equations at time step $k > 0$:

$$\begin{pmatrix}
-\frac{1}{R_r} & \frac{1}{R_r} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{R_r} & -\frac{1}{R_r} - \frac{1}{R_{n_1}} & \frac{1}{R_{n_1}} & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{R_{n_1}} & -\frac{1}{R_{n_1}} - \frac{1}{R_{t_1}} - \frac{1}{R_{n_2}} & \frac{1}{R_{t_1}} & \frac{1}{R_{n_2}} & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{R_{t_1}} & -\frac{1}{R_{t_1}} & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{R_{n_2}} & 0 & -\frac{1}{R_{n_2}} - \frac{1}{R_{n_3}} & \frac{1}{R_{n_3}} & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{R_{n_3}} & -\frac{1}{R_{n_3}} - \frac{1}{R_{t_2}} & \frac{1}{R_{t_2}} & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{R_{t_2}} & -\frac{1}{R_{t_2}} & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
\pi(n_{pwr}, \tau_k) \\
\pi(r, \tau_k) \\
\pi(n_1, \tau_k) \\
\pi(t_1, \tau_k) \\
\pi(n_2, \tau_k) \\
\pi(n_3, \tau_k) \\
\pi(t_2, \tau_k) \\
I(e_{pwr}, \tau_k) \\
I(c_r, \tau_k) \\
I(c_{n_1}, \tau_k) \\
I(c_{t_1}, \tau_k) \\
I(c_{n_2}, \tau_k) \\
I(c_{n_3}, \tau_k) \\
I(c_{t_2}, \tau_k)
\end{pmatrix} =
\begin{pmatrix}
0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ V_{DD} \\
\pi(r, \tau_{k-1}) + \frac{h}{C_r} I(c_r, \tau_{k-1}) \\
\pi(n_1, \tau_{k-1}) + \frac{h}{C_{n_1}} I(c_{n_1}, \tau_{k-1}) \\
\pi(t_1, \tau_{k-1}) + \frac{h}{C_{t_1}} I(c_{t_1}, \tau_{k-1}) \\
\pi(n_2, \tau_{k-1}) + \frac{h}{C_{n_2}} I(c_{n_2}, \tau_{k-1}) \\
\pi(n_3, \tau_{k-1}) + \frac{h}{C_{n_3}} I(e_{n_3}, \tau_{k-1}) \\
\pi(t_2, \tau_{k-1}) + \frac{h}{C_{t_2}} I(e_{t_2}, \tau_{k-1})
\end{pmatrix}$$

For this system, we do not even need to invert the matrix, as we can compute the potentials directly from the previous values, then put them into the KCL equations and compute the current values.

It remains to find an initial state. We assume that the system starts in the resting state at time τ_0. This means all voltages across the capacitors are 0. The corresponding solution is $\pi(n_{pwr}, \tau_0) = V_{DD}$ and $\pi(n, \tau_0) = 0$ for all $n \neq n_{pwr}$. Then $I(e_{pwr}, \tau_0) = I(c_r, \tau_0) = \frac{V_{DD}}{R_r}$ and $I(e, \tau_0) = 0$ for all other edges e.

Now, we insert values and simulate our system. The following values are realistic for short uncritical nets that are driven by a small inverter. We assume a source resistance of $3.1k\Omega$ and a sink capacitance of $0.9fF$ for t_1 and $1.2fF$ for t_2. Furthermore, we assume that a via has a resistance of 30Ω. Wires have a resistance per length of $0.15\frac{\Omega}{nm}$ and a capacitance per length of $0.24\frac{aF}{nm}$. V_{DD} is at $1V$. Then we choose a step length of $0.001ps$ to simulate our system until both sinks reach 0.93% V_{DD}. The resulting curve of t_1 is depicted in figure 1.10. The techniques used here are a very simple version of what is known as SPICE simulation [Nag75].

1.2.4 Computing Delay

Let us now take a look at the resulting waveform if we instead approximate a rising voltage at the input by a piecewise linear function and run our simulation on that. The result is shown in Figure 1.11.

We consider a node in our model to be carrying a logical 1 if the voltage is above 50% of V_{DD} and a 0 if the voltage is below that. Then we define the arrival time of a signal at a node as the time, when 50% V_{DD} is passed at that node. The delay from one node to a subsequent node is the difference between their arrival times. It should be noted that with this definition, the delay can actually be negative.

For a reasonably good approximation of the delay curves, we also need to know how fast the voltage at a node is rising. This is captured by the slew, which is defined as

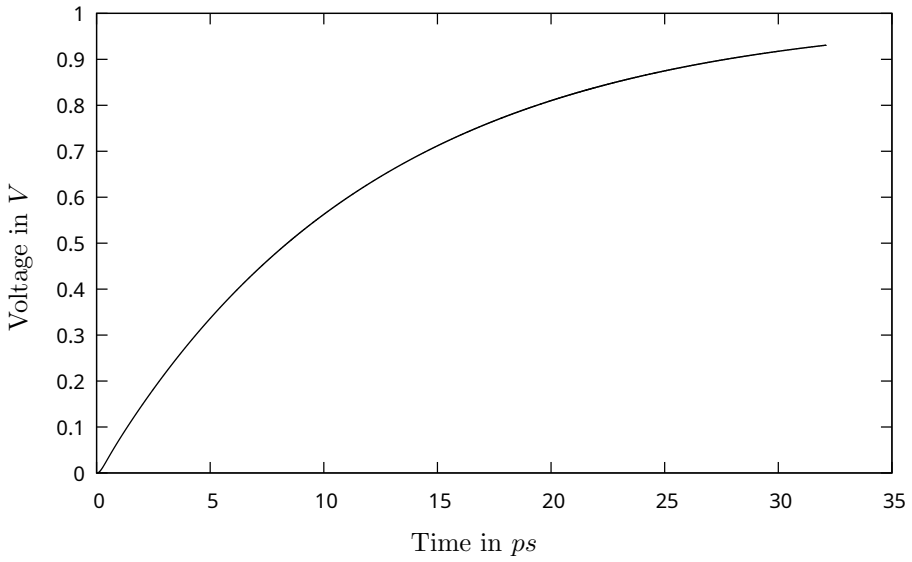


Figure 1.10: The waveform at sink t_1 of the net from Figure 1.9 for a step input.

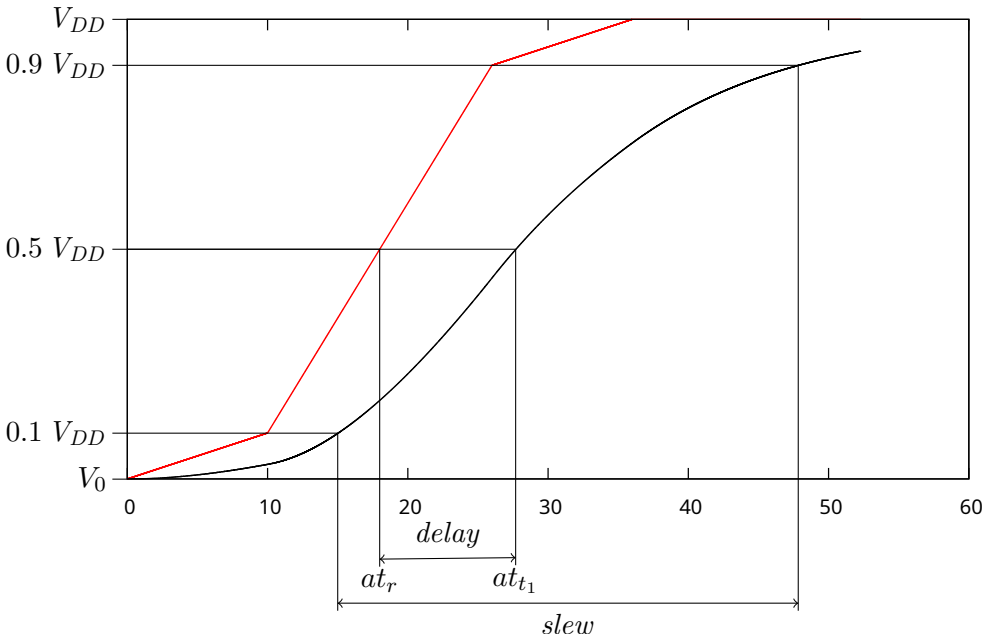


Figure 1.11: The waveform at t_1 (black) for a piecewise linear input voltage (red).

the time it takes for the voltage to rise from 10% V_{DD} to 90% V_{DD} on a rising flank or reverse on a falling flank.

It should be noted here that, while these are common definitions of delay and slew, the percentages in the definitions may vary depending on the work. For example the slew is sometimes defined as the time it takes from 20% V_{DD} to 80% V_{DD}.

To approximate the delay for a step input, we will be using a technique called model order reduction [Sap04, Chapter 3.5]. Suppose, we want to compute the waveform $v(t)$, which is equal to the potential curve $\pi(t)$ (because we set $\pi(n_0) := 0$) at some node n of our model. The overall idea is to transform our system to the frequency-domain using a Laplace transform. There, we will be able to write the transfer function of the potential as a power series, which we can approximate by a Padé-approximant [Bak+96].

Let us first recall the Laplace transform in a very naive definition: For a function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}$, we denote by $\mathcal{L}f : \mathbb{C} \to \mathbb{C}$ its Laplace transform

$$\mathcal{L}f(s) = \int_0^\infty f(t)e^{-st}dt.$$

Then we speak of f being in the time domain and $\mathcal{L}\,f$ being in the frequency domain. It should be noted that the Laplace transform does not necessarily exist for all functions and all values $s \in \mathbb{C}$. However, for the step response of an RC-tree everything works out nicely, so we will assume that all functions are well-behaved.

The transfer function of our potential in the frequency domain is the ratio (in the frequency domain) of the potential to the input waveform. If we are given a function $p : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ that describes the waveform of our voltage source, then the transfer function for our potential $\pi(n, t)$ is a function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that

$$\mathcal{L}\,\pi(s) = \mathcal{L}\,f(s) \cdot \mathcal{L}\,p(s).$$

We transform f to the frequency domain and denote $F(s) := \mathcal{L}\,f$. Then we rewrite F as a power series and approximate that by a using a $[L/M]$-Padé-approximant for natural numbers $L \geq 0$ and $M \geq 1$:

Replacing the exponential function in the definition of the Laplace transform using its series characterization (which is equivalent to doing a Taylor expansion about $s = 0$), we get

$$F(s) = \int_0^\infty f(t)e^{-st}dt = \int_0^\infty f(t)\sum_{i=0}^{\infty}(-1)^i\frac{s^i t^i}{i!}dt = \sum_{i=0}^{\infty} s^i \cdot \frac{(-1)^i}{i!}\int_0^\infty t^i f(t)dt.$$

We define the moments of f as

$$m_i := \frac{(-1)^i}{i!}\int_0^\infty t^i f(t)dt \text{ for } i \in \mathbb{N}.$$

Now, we want to find polynomials u of order L and q of order M with $q(0) = 1$, such that the first $N := L + M + 1$ moments of F and $\frac{u(s)}{q(s)}$ are the same. This means that, if

we write the power series $q(s) \cdot F(s) =: \sum_{i=0}^{\infty} z_i s^i$, we have

$$\sum_{i=0}^{N} z_i s^i = u(s).$$

The function $R(s) := \frac{u(s)}{q(s)}$ is our $[L/M]$-Padé-approximant. Let us now apply this scheme to find a first delay model.

Elmore Delay

While Elmore [Elm48] arrived at his formula by a different path, the Elmore delay [Elm48] can be derived from a first order-approximation to the unit step response. We will mostly follow [Sap04, Chapters 3.4, 3.5].

Let $H : \mathbb{R} \to \mathbb{R}$ be the unit step. That is $H(t) = 1$ for $t > 0$ and $H(t) = 0$ for $t \leq 0$. Then we have

$$\pi(n,t) = \int_0^t \dot{\pi}(n,\tau)d\tau = \int_0^\infty \dot{\pi}(n,\tau)H(t-\tau)d\tau = \dot{\pi}(n,\cdot) * H(t)$$

and in particular $\mathcal{L}\,\pi = \mathcal{L}[\dot{\pi} * H] = \mathcal{L}\,\dot{\pi} \cdot \mathcal{L}\,H$. So our transfer function is the time derivative of π with $\mathcal{L}H(s) = \frac{1}{s}$. We start by computing the $[0/1]$-Padé-approximant, setting $p(s) := a_0$ and $q(s) := 1 + b_1 s$. By setting

$$m_0 + (m_1 + b_1 m_0)s = a_0$$

and matching the coefficients, we get the system

$$\begin{aligned} m_0 &= a_0 \\ m_1 + b_1 m_0 &= 0 \end{aligned}$$

Solving it yields

$$R(s) = \frac{p(s)}{q(s)} = \frac{m_0}{1 - \frac{m_1}{m_0}s}.$$

Now we normalize V_{DD} to 1 to simplify the notation. We know that asymptotically, each node must reach a voltage of 1. This means that we have the boundary condition $\lim_{t\to\infty} \pi(t) = 1$. We use the final value theorem to compute

$$1 = \lim_{t\to\infty} \pi(t) = \lim_{s\to 0} s\,\mathcal{L}\,\pi(s) = \lim_{s\to 0} s\frac{1}{s}\sum_{i=0}^{\infty} m_i s^i = m_0.$$

So our approximation simplifies to

$$R(s) = \frac{1}{1 - m_1 s}.$$

We use this to reconstruct our approximated voltage in the frequency domain:

$$\mathcal{L}\,\tilde{\pi}(s) = \frac{1}{s}R(s) = \frac{1}{s - m_1 s^2}.$$

From this, we can retrieve our approximation to the voltage in the time domain.

$$\tilde{\pi}(t) = 1 - e^{\frac{t}{m_1}}.$$

As we are interested in the delay and slew, we now find the value that $t \in \mathbb{R}_{\geq 0}$ has to take for the voltage to be at value $x \in [0, 1)$:

$$x = 1 - e^{\frac{t}{m_1}} \iff t = \log(1 - x)m_1$$

So the delay is $-\ln(2)m_1$ and the slew is $-(\ln(10) - \ln(\frac{10}{9}))m_1 = -\ln(9)m_1$. This means that the delay and slew to a node are essentially given by its negative first moment

$$-m_1 = \int_0^\infty t\dot{\pi}(t)\,dt = \int_0^\infty t\frac{d}{dt}(\pi(t)-1)\,dt = t\,[\pi(t) - 1]_0^\infty - \int_0^\infty \pi(t)-1\,dt = \int_0^\infty 1-\pi(t)\,dt.$$

Let us now go back to our equation system describing the RC-tree. At each node n that is not incident to the power source, the potential of this node $\pi(n)$ is described by the differential equation of the capacitor C_n and the KCL equation with inserted resistor equations. Denote by $par(n)$ the unique predecessor of n. The equations at n are

$$\begin{aligned} I(c_n, t) &= C_n\dot{\pi}(n, t) \\ I(c_n, t) &= \frac{1}{R_n}(\pi(par(n), t) - 1 - \pi(n, t) + 1) + \sum_{(n,q)\in E(G)} \frac{1}{R_q}(\pi(q, t) - 1 - \pi(n, t) + 1) \end{aligned}$$

Combining both equations for each node, the resulting system can be written as

$$R(\Pi(t) - 1) = -C\dot{\Pi}(t)$$

where C is the diagonal matrix containing all capacitances, R contains all the resistance factors and Π is the vector of potentials for each node that is not incident to the power source. We can leave n_{pwr} out, because we are considering the step input and hence $\pi(n_{pwr}) - 1 = 0$.

For RC-trees R is invertible. We make use of this fact to rewrite

$$R(\Pi(t) - \mathbf{1}) = -C\dot{\Pi}(t) \iff \mathbf{1} - \Pi(t) = R^{-\mathbf{1}}C\dot{\Pi}(t).$$

Now, we can use this and the fact that $\Pi(0) = 0$ and $\lim_{t\to 0}\Pi(t) = \mathbf{1}$ to compute the first moments of all nodes at once

$$-M_1 = \int_0^\infty \mathbf{1} - \Pi(t)\,dt = \int_0^\infty R^{-1}C\dot{\Pi}(t)\,dt = R^{-1}C\mathbf{1},$$

where M_1 is the vector of first moments. Since the resistor edges form a tree for an RC-tree, we can give an explicit expression of the moments at each node, which is known as the Elmore delay. Let $downcap(n)$ be the sum of all downstream capacitances of node n and P_n be the unique path from r to n in G. Then

$$m_1(n) = \sum_{q \in V(P_n)} R_p \cdot downcap(q).$$

We can even define the Elmore delay directly for routed nets. If we are given an arborescence A rooted at r and with leaves T that represents a routing of the net $\{r\} \cup T$ with edge lengths $l : E(A) \to \mathbb{R}_{\geq} 0$, a root resistance res_{root}, wire resistance per length res_{wire}, wire capacitance per length cap_{wire} and sink capacitances cap_t for $t \in T$, we can recursively define the downstream capacitance of each vertex $v \in V(A)$ as

$$downcap(v) := \begin{cases} cap_v, & \text{if } v \in T \\ \sum\limits_{e=(v,w) \in \delta_A^-(v)} downcap(w) + cap_{wire} \cdot l(e) & \text{otherwise.} \end{cases}$$

And subsequently the delay to vertex $v \in V(A)$ as

$$delay(v) := \ln 2 \cdot \left[res_{root} \cdot downcap(r) + \sum_{e=(v,w) \in E(A_{[r,v]})} res_{wire} \cdot l(e) \cdot \left(downcap(w) + \frac{cap_{wire} \cdot l(e)}{2} \right) \right].$$

Asymptotic Waveform Evaluation

For more accurate timing results, we can use Asymptotic Waveform Evaluation (AWE) [PR90]. It is a more general technique for model order reduction that includes the Elmore delay as a special case. There are many numerical concerns that have to be taken care of in order to practically apply it, but we will only review the basic technique here, as described in [Sap04, Chapter 3.5].

Assume now a more general voltage source given by the function $p : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Similarly to the previous section, we can write our system of inequalities as

$$R(\Pi(t) - p(t)\mathbf{1}) = -C\dot{\Pi}(t).$$

We want to transform this system to the frequency domain. The Laplace transform of the RHS is given as

$$\mathcal{L}[C\dot{\Pi}](s) = \int_0^\infty C\dot{\Pi}(t)e^{-st}dt = \left[C\Pi(t)e^{-st}\right]_0^\infty - \int_0^\infty -sC\Pi(t)e^{-st}dt = sC\mathcal{L}\Pi(s).$$

So our system becomes

$$R(\mathcal{L}\,\Pi(s) - \mathcal{L}\,p(s)\mathbf{1}) = -sC\,\mathcal{L}\,\Pi(s) \iff (R + sC)\,\mathcal{L}\,\Pi(s) = R\,\mathcal{L}\,p(s)\mathbf{1}.$$

So if f is the transfer function for our power source and we set $F := \mathcal{L} f$, we can instead solve

$$(R + sC)F(s) = R\mathbf{1} = R\,\mathcal{L}\,\delta(s)\mathbf{1},$$

which corresponds to a system excited by the unit impulse $\delta(t)$ (, because $\mathcal{L}\,\delta(s) = 1$). If we multiply the solution by $\mathcal{L}\,p$, we retrieve a solution to our original system.

We replace F by its power series

$$(R + sC)\sum_{i=0}^{\infty} M_i s^i = R\mathbf{1}$$

and match the powers to find that (for invertible R)

$$\begin{aligned} M_0 &= \mathbf{1} \\ RM_i &= -CM_{i-1} \qquad\qquad \text{for } i = 1, \ldots, \infty. \end{aligned}$$

In other words, we can compute the 0-th moment by $M_0 = \mathbf{1}$ and then iteratively compute the remaining moments for $i \geq 1$ by $M_i = -R^{-1}CM_{i-1}$. We want to compute a $[q-1/q]$-Padé-approximant for F, where typically $q \leq 4$. So we need to compute the first $2q-1$ moments. To find the voltages at node n, let $m_0, \ldots, m_{2q-1}$ be its moments. We want to find coefficients $a_0, \ldots, a_{q-1}$ and $b_1, \ldots, b_q$, such that

$$\frac{\sum_{i=0}^{q-1} a_i s^i}{1 + \sum_{i=1}^{q} b_i s^i} = \sum_{i=0}^{2q-1} m_i s^i,$$

which we can do by solving the linear system

$$\sum_{i=1}^{q} m_{k-i} b_i = m_k \text{ for } k = q, \ldots, 2q-1$$

and then by inserting the values for the b_i (for $i = 1, \ldots, q$) to compute

$$a_k = m_k + \sum_{i=1}^{k} m_{k-i} b_i \text{ for } k = 1, \ldots, q-1.$$

Finally, we can transform our approximant back to the time domain by applying the inverse Laplace transform. This way, we retrieve the transfer function of the potential at our node. The waveform is the convolution of the waveform at the source and the transfer function.

Gate Delay

To this point, we have always modeled gates by a resistor. However, most gates will show nonlinear behaviour. The problem is that the fast delay computation methods we have seen rely on the linearity of the systems. So the key to keeping a fast evaluation, while improving the accuracy of the gate delay, is to separate the nonlinearities from the linearities. As a first step, we will assume that for gates with more than one input, only one of them is switching.

Then there are multiple methods to achieve the separation. The simplest and fastest one is to neglect the effect of the wiring resistance on the gate delay [Sap04, Chapter 4.3]. Instead, we assume that only the input waveform or slew and the capacitance of the wiring and sinks have an influence on the gate delay. This way, we can simulate the behaviour of the gate at a set of sampling points using SPICE [Nag75]. Then we store them in a table and approximate them by piecewise linear functions. To reduce the memory overhead, the delay of a gate can be computed by a fitted curve. This improves the accuracy significantly, in particular when the model resistance of the gate is large compared to the wiring resistance. These approximations of the gate delay are usually shipped with a library and are referred to as *timing rules*.

If the wiring resistance is larger, resistive shielding has to be taken into account [Sap04, Chapter 4.4]. This describes the effect that the gate delay behaves as if it were driving a lower capacitance, if the wiring resistance is high. The resistance is shielding away part of the wiring (and sink) capacitance from the gate.

To take this effect into account, we approximate our wiring by a simple π-model consisting of a capacitance C_{near} close to the voltage source and a capacitance C_{far} further away from the voltage source, separated by some resistance R and driven by an impulse voltage source. This can be achieved by computing the first three moments of the current through an impulse voltage source applied to our original RC-tree and matching them to the moments of the reduced model. A piecewise linear interpolation can then be used to compute the gate delay for given input slew, R, C_{near} and C_{far}.

However, a 4-dimensional table is very memory consuming. Often another reduction is applied to compute an effective capacitance C_{eff} from this simple π-model that can be used with 2-dimensional lookup tables or fitted curves.

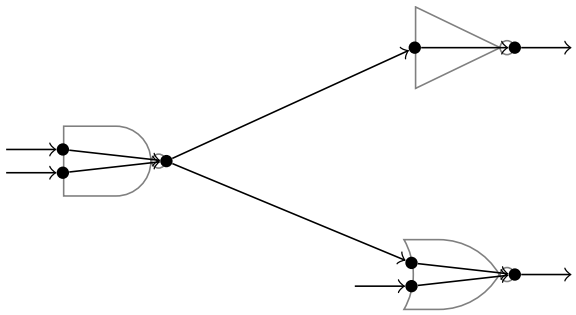


Figure 1.12: An example for a small part of the coarse timing graph.

1.2.5 Timing Propagation

Now that we know how to compute the timing of a combinational stage, we can review the timing properties of a chip. We stick to the notation and base our explanation on the book "Chip Design" by Stephan Held and Jens Vygen [HV22].

We start by defining the coarse timing graph of a chip. It has a vertex for each pin. Then it has edges from each input of a combinational gate to the gate's outputs and for each net, it has edges from all sources to all sinks of the net. A small part of a timing graph is shown in Figure 1.12.

Definition 1.2.2. *Let $(\mathcal{C}, P = P_{source} \dot\cup P_{sink}, \gamma, \mathcal{N})$ be the netlist of a chip. The* coarse timing graph *is the graph $G_{coarse} = (V, E)$ with*

- $V = P$ *and*
- $E = \{(v,w) \in P_{source} \times P_{sink} \mid v, w \in N\ \} \cup$ (Net edges)
  $\{(v,w) \in P_{sink} \times P_{source} \mid \gamma(v) = \gamma(w) = C \in \mathcal{C}$ *combinational* $\}$ (Gate edges)

For most of the gates the flank of the waveform, that is whether we have a rising or falling signal, also changes the delay through that gate. Gates, like inverters for example, may also change a rising signal at the input to a falling signal at the output or vice versa. We denote by $\Psi := \{rise, fall\}^2$ the set of possible transitions. Then we denote by $\Psi_{id} := \{(rise, rise), (fall, fall)\}$ the set of identical transitions and by $\Psi_{inv} := \{(rise, fall), (fall, rise)\}$ the set of inverting transitions.

Definition 1.2.3. *Let C be a gate of the chip implementing the logical function $b : \{0,1\}^k \to \{0,1\}$ for some $k \in \mathbb{N}$. Let $e \in E(G_{coarse})$ be a gate edge in the coarse timing graph, belonging to the $i-th$ input of C, for $i \in [k]$. We call $u, v \in \{0,1\}^k$ an i-flipping pair, if $u_j = v_j$ for $i \neq j \in [k]$ and $u_i = 1 - v_j$.*

- *The edge e is* identical *if there is an i-flipping pair u, v such that $b(u) = u_i$ and $b(v) = v_i$.*
- *The edge e is* inverting *if there is an i-flipping pair u, v such that $b(u) = v_i$ and $b(v) = u_i$.*

Note that an edge can have both properties at the same time.

Furthermore for different clock-phases, the arrival times or required arrival times may change. So we encode this information in the fine timing graph.

Definition 1.2.4. *Let G_{coarse} be a coarse timing graph and Φ the set of available phases. Let $e \in E(G_{coarse})$ be an edge of the coarse timing graph. We denote by Ψ_e the set of allowed transitions of e. If e is a net edge, we have $\Psi_e := \Psi_{id}$. If e is a gate edge, we have $\Psi_{id} \subseteq \Psi_e$ iff e is identical and $\Psi_{inv} \subseteq \Psi_e$ iff e is inverting.*

Then we define the fine timing graph G_{fine} *as*

- $V(G_{fine}) = V(G_{coarse}) \times \{rise, fall\} \times \Phi$ *and*
- $E(G_{fine}) = \big\{((v, \psi, \phi), (w, \psi', \phi)) \mid (v, w) \in E(G_{coarse}), (\psi, \psi') \in \Psi_{(v,w)}, \phi \in \Phi\big\}$

Assume that we are given a capacitance function for the nets $cap : \mathcal{N} \to \mathbb{R}_{\geq 0}$ and two timing functions: One for the net edges of the coarse timing graph $timing_{net,e} : \mathbb{R}_{\geq 0} \to \mathbb{R}^2_{\geq 0}$ mapping an input slew to a delay and output slew and one for the gate edges $timing_{gate,e} : \mathbb{R}^2_{\geq 0} \times \Psi_e \to \mathbb{R}^2_{\geq 0}$ mapping a capacitance and an input slew for a given transition to a delay and an output slew.

An example for such timing functions would be an effective capacitance computation for cap, an AWE computation for $timing_{net,e}$ and a fitted curve for $timing_{gate,e}$.

For a pin $p \in P$, denote by $N(p) \in \mathcal{N}$ the unique net that this pin belongs to. Then we define a timing function $timing : E(G_{fine}) \times \mathbb{R}_{\geq 0} \to \mathbb{R}^2_{\geq 0}$ mapping an edge and input slew to a delay and output slew for the edges of the fine timing graph. For $e := ((v, \psi, \phi), (w, \psi', \phi)) \in E(G_{fine})$ with $\hat{e} = (v, w) \in E(G_{coarse})$, we set

$$timing(e, s) := \begin{cases} timing_{net,\hat{e}}(s), & \text{if } \hat{e} \text{ net edge,} \\ timing_{gate,\hat{e}}(cap(N(w)), s, (\psi, \psi')), & \text{if } \hat{e} \text{ gate edge.} \end{cases}$$

Now we want to compute arrival times $at : V(G_{fine}) \to \mathbb{R}_{\geq 0}$, slews $slew : V(G_{fine}) \to \mathbb{R}_{\geq 0}$ and required arrival times $rat : V(G_{fine}) \to \mathbb{R}_{\geq 0}$ for the signals at each node in the fine timing graph. For nodes without incoming edges (nodes belonging to primary inputs and sources of memory elements), the arrival times and slews are given and for nodes without outgoing edges (nodes belonging to primary outputs and sinks of memory elements), the required arrival times are given. Then we will propagate these values through the fine timing graph. However, for nodes with multiple incoming edges (nodes belonging to sources of gates with multiple inputs) it is not obvious, how to define the arrival time and slew.

Suppose we knew a constant that bounds the impact of a small slew change on the delay through all edges. Then there is a way to merge the slews and arrival times, such that the arrival times are upper bounds on the real arrival times.

Definition 1.2.5. *Let $\nu \in \mathbb{R}_{\geq 0}$. Let $(a, s), (a', s') \in \mathbb{R}^2_{\geq 0}$ be pairs of arrival times and slews. We say that (a, s) ν-dominates (a', s') if*

$$a - a' \geq \nu \max(0, s - s').$$

For a set $S \subset \mathbb{R}^2_{\geq 0}$ of arrival time-slew pairs, we define their ν-supremum as $\sup_\nu(S) := (\hat{a}, \hat{s})$*, where*

$$\hat{a} := \sup_{(a,s)\in S} a \quad \textit{and} \quad \hat{s} := \sup_{(a,s)\in S} \left[s + \frac{a - \hat{a}}{\nu} \right].$$

So let us assume that we have a ν for a vertex $v \in V(G_{\mathit{fine}})$, such that the downstream delay from v increases by at most $\nu\epsilon$, if we increase the slew by ϵ at v for any $\epsilon > 0$. Then picking an arrival time-slew pair at v that ν-dominates all incoming signals will lead to higher arrival times downstream than any of the incoming arrival time-slew pairs.

Now we can define what it means to propagate the arrival times and slews through the timing graph. Let ν_v for $v \in V(G_{\mathit{fine}})$ be constants that bound the total downstream delay impact of a slew change at v. We traverse the nodes $w \in V(G_{\mathit{fine}})$ that have incoming edges in topological order and set $\mathit{at}(w) := a$ and $\mathit{slew}(w) := s$, where

$$(a, s) := \sup_{\nu_v} \left\{ (\mathit{at}(v), 0) + \mathit{timing}((v, w), \mathit{slew}(v)) \mid (v, w) \in \delta^-(w) \right\}.$$

For the required arrival times, we then traverse the nodes $v \in V(G_{\mathit{fine}})$ that have outgoing edges in reverse topological order and set

$$\mathit{rat}(v) := \min_{e=(v,w)\in\delta^+(v)} \left[\mathit{rat}(w) - d_e - \nu_v \cdot \max(0, s_e - \mathit{slew}(w)) \right],$$

where $(d_e, s_e) := \mathit{timing}((v, w), \mathit{slew}(v))$, for $e \in \delta^-(v)$.

Vygen [Vyg06] proved that indeed, we can find such constants ν_v, such that all arrival times are upper bounds on the real arrival times.

1.3 Path Search, Steiner Trees and Resource Sharing

In this section, we discuss the basic concepts and algorithms of combinatorial optimization that the following work is based on. It does not aim at providing the full theory, but rather tries to provide the ideas that are used later on. We will use the notation of the book "Combinatorial Optimization" by Korte and Vygen [KV18].

We start by giving the basic ideas of shortest path searches. Here, we will focus on Dijkstra's algorithm [Dij59], which is the simplest version of a type of dynamic program that we will see multiple times in this thesis.

Then we will state the Steiner tree problem and give basic properties of Steiner trees, as well as some algorithms for the Steiner tree problem. We will also review some variations of the Steiner tree problem that are important later on. Most formulations of the buffering problem are generalizations of the Steiner tree problem. We will use concepts from Steiner tree computation, as well as Steiner trees themselves.

Finally, we will discuss the resource sharing problem which can be solved with an algorithm by Müller, Radke und Vygen [MRV11]. The problem of finding buffered routes for each net, such that the timing constraints are met on each path and some density constraints on the placement and routing are met can be formulated as a resource sharing problem [Hel+17]. This leads to a Lagrangian relaxation formulation of the buffering problem, which we also will consider later [Dab+23].

1.3.1 Path Search

Finding shortest paths is an essential subtask in many applications. In the shortest path problem, we are given a directed graph G with edge costs $c : E(G) \to \mathbb{R}$ and two vertices $s, t \in V(G)$. The task is to find an s-t-path P in G that minimizes the cost of the path $\mathit{length}(P) := \sum_{e \in E(G)} c(e)$.

This problem is NP-hard for arbitrary costs [KV18], but can be solved efficiently with Dijktstra's algorithm [Dij59] if the edge costs are nonnegative. During the algorithm, we maintain a distance $\mathit{dist} : V(G) \to \mathbb{R}_{\geq 0}$ to each vertex and a predecessor $p : V(G) \to V(G)$ of each vertex. In the beginning each vertex is its own predecessor: $p(v) = v$ for $v \in V(G)$. We initialize $\mathit{dist}(s) := 0$ and $\mathit{dist}(v) := \infty$ for $v \in V(G) \setminus \{s\}$. Then we put all vertices $v \in V(G)$ into a heap Q, with $\mathit{dist}(v)$ as a key. Furthermore, we keep a set of permanent vertices P that is empty in the beginning. While there are vertices in Q, we extract the minimum vertex v from Q and add it to P. If $v = t$, we are done. We reconstruct the path using the predecessor function and return it. Otherwise, we check for all outgoing edges $(v, w) = e \in \delta^-(v)$, with w not permanent, if $\mathit{dist}(v) + c(e) < \mathit{dist}(w)$. If this is the case, we set $\mathit{dist}(w) := \mathit{dist}(v) + c(e)$ and store v as the predecessor of w. Otherwise, we do nothing. If Q is empty and we did not find t, we return that there is no s-t-path. The algorithm is written down as pseudocode in Algorithm 1.

Theorem 1.3.1 (Dijkstra 1959 [Dij59],Fredman and Tarjan 1987 [FT87])**.** *Dijkstra's algorithm is correct and can be implemented to run in* $\mathcal{O}(m + n \log n)$.

Algorithm 1: Dijktra's algorithm. [Dij59]

Input : An instance (G, c, s, t) of the shortest path problem with nonnegative edge cost.

Output: A shortest s-t-path in G if there is one, the information that there is no s-t-path otherwise.

```
1  dist(s) := 0, dist(v) := ∞ for v ≠ s, p(v) := v for v ∈ V(G), P := ∅;
2  Create heap Q with V(G) as elements and dist as keys;
3  while Q ≠ ∅ do
4      Let v be the minimum element of Q;
5      Remove v from Q and add it to P;
6      if v = t then
7          return Recursively reconstructed s-t-path;
8      for (v, w) = e ∈ E(G) with w ∉ P do
9          if dist(v) + c(e) < dist(w) then
10             p(w) := v;
11             dist(w) := dist(v) + c(e);
12             Decrease key of w in Q to dist(w);
13 return There is no s-t-path;
```

This is not the end of the story yet. While we are not able to reduce the running time guarantee of Dijkstra's algorithm in general, we can improve its practical running time. The basic version of Dijktstra's algorithm presented in Algorithm 1 first finds all vertices that have a lower distance to s than t and only then it returns t. The idea to improving the practical running time is to direct the search towards t [HNR68]. We achieve this by changing the objective function in such a way that shortest paths remain shortest, but their length decreases compared to non-shortest paths.

Definition 1.3.2. *Let G be a directed graph with edge cost $c : E(G) \to \mathbb{R}$. A function $\pi : V(G) \to \mathbb{R}$ is a* feasible potential*, if*

$$c_\pi(e) := \pi(w) + c(e) - \pi(v) \geq 0 \text{ for each } (v, w) = e \in E(G).$$

We call c_π the reduced cost *of π.*

Such a feasible potential exists if and only if the edge costs are conservative. Conservative edge costs are costs such that each cycle has nonnegative total cost.

Let P be a s-t-path and π be a feasible potential. Then the reduced cost of this path,

$$\sum_{e \in E(P)} c_\pi(e) = \pi(t) - \pi(s) + \sum_{e \in E(P)} c(e),$$

does not depend on the potential of the inner vertices of the path.

As the cost of every s-t-path is shifted by the same constant, shortest paths remain shortest and since the reduced costs are nonnegative, we can use Dijkstra's algorithm on an instance with reduced costs to find a shortest path.

Suppose now that π is the minimum distance from t to every vertex. Then the distance from s to any vertex v with respect to the reduced costs is 0 if and only if v lies on a shortest s-t-path. This is why Dijktstra's algorithm potentially requires fewer iterations. Of course computing this potential is as hard as computing the s-t-path, but for nonnegative costs, a lower bound on the distances to t can be much easier to compute, but still reduce the number of iterations significantly. In geometric settings, often the geometric distance is used as such a lower bound.

1.3.2 Steiner Trees

The task of finding a route for a net with minimum amount of wiring is the search for a minimum Steiner tree. Naturally, Steiner trees appear in almost every aspect of physical design. We start with the minimum Steiner tree problem in graphs.

Apart from that there are many variants of the minimum Steiner tree problem, for example the minimum Steiner tree problem in metric spaces. The rectilinear Steiner tree problem, where the metric space is $(\mathbb{R}^2, l_1)$, is the most important special case for VLSI design. We will later see that it can be reduced to the Steiner tree problem in graphs. Most variants of the minimum Steiner tree problem, like the rectilinear version [GJ77] and the version in graphs [Kar72], are NP-hard.

Steiner Trees in Graphs

We are given a connected graph G with nonnegative edge cost $c : E(G) \to \mathbb{R}_{\geq 0}$ and a subset of the vertices $T \subset V(G)$, called *terminals*. The task is to find a tree S in G with $T \subset V(S)$ that minimizes the total cost

$$c(S) := \sum_{e \in E(S)} c(e).$$

A 2-approximation to the minimum Steiner tree problem in graphs [KMB81] can be computed by first computing the metric closure of the terminals. Then we compute a minimum spanning tree (MST) in the metric closure. Finally, we take the union of the shortest paths belonging to the edges in the metric closure and remove cycles.

It is easy to verify that the resulting tree is a 2-approximation. The first observation we need to make is that the resulting tree has cost at most that of a MST in the metric closure. Now assume, we had a minimum Steiner tree. If we double all edges, the resulting graph is Eulerian. We take an Eulerian tour through this graph. Let $t_1, \ldots, t_k$ for $k := |T|$ be the order, in which the terminals are visited for the first time. The path between consecutive terminals in that order is at most as long as a shortest path and all of them sum up to at most the cost of the Steiner tree with the doubled edges, which is twice the cost of the minimum Steiner tree. The path in the metric closure of the terminals visiting the terminals in the order of the tour is a spanning tree in the metric

closure, so it costs at least as much as a minimum spanning tree in the metric closure. Hence, the MST in the metric closure of the terminals can cost at most as much as twice the cost of a minimum Steiner tree.

As a next step, we want to solve the minimum Steiner tree problem in graphs optimally. This can be done with the Dijkstra-Steiner algorithm [HSV17], an extension to Dijkstra's algorithm. Similarly to Dijkstra's algorithm, we can direct our search with an extension to feasible potentials.

Definition 1.3.3 (Hougardy, Silvanus and Vygen 2017 [HSV17]). *Let (G, c, T) be an instance of the minimum Steiner tree problem and $r \in T$. A function $\mathcal{L} : V(G) \times 2^T \to \mathbb{R}_{\geq 0}$ is a* valid lower bound *to r, if $\mathcal{L}(r, \{r\}) = 0$ and*

$$\mathcal{L}(v, I) \leq \mathcal{L}(w, I') + smt((I \setminus I') \cup \{v, w\}),$$

for $v, w \in V(G)$ and $\{r\} \subseteq I' \subseteq I \subseteq T$, where $smt(J)$ denotes the cost of a minimum Steiner tree in G for terminals $J \subset V(G)$.

The algorithm proceeds similar to Dijkstra's algorithm. We first choose a target $r \in R$. Then we traverse the graph from the other terminals towards the target. Instead of storing the distance from the source to each vertex, we store for each subset of the non-target terminals $I \subseteq T \setminus \{r\}$ and vertex $v \in V(G)$ the minimum known length of a Steiner tree for the terminal set $I \cup \{v\}$. We call a pair $(v, I) \subset V(G) \times 2^T$ a label. In the beginning $\mathit{length}(t, \{t\}) := 0$ for each $t \in T \setminus \{r\}$. This time, the domain of *length* has an exponential size, so we only store the values for labels that have been assigned a finite value. Again, we store a set of permanent labels P that is empty in the beginning. We store a set of non-permanent labels in a heap Q, that contains the labels $(t, \{t\})$ for $t \in T \setminus \{r\}$ in the beginning. Furthermore, we store a set of predecessor labels for each label. We start off with $p(t, \{t\}) := \square$ for $t \in T \setminus \{r\}$.

While Q is not empty, we pick a label $l := (v, I)$ from Q that minimizes $\mathit{length}(v, I) + \mathcal{L}(v, T \setminus I)$. We remove it from Q and add it to P. If l already belongs to a minimum Steiner tree on T (so $v = r$ and $I = T \setminus \{r\}$), we reconstruct the Steiner tree using our predecessor function p. Otherwise, we check for each edge $e = \{v, w\} \in E(G)$, if (w, I) is already contained in $P \cup Q$. If not, or otherwise if $\mathit{length}(v, I) + c(e) < \mathit{length}(w, I)$, we update $\mathit{length}(w, I) := \mathit{length}(v, I) + c(e)$ and set $p(w, I) := \{l\}$. Additionally, we now try to merge the Steiner tree belonging to l with Steiner trees that belong to other permanent labels at v and that are connecting terminal sets which are disjoint to I. For each permanent label $(v, J) \in P$ with $J \cap I = \emptyset$, we check if $(v, I \cup J)$ is not in Q yet, or otherwise, if $\mathit{length}(v, I) + \mathit{length}(v, J) < \mathit{length}(v, I \cup J)$. If this is the case, we update $\mathit{length}(v, I \cup J) := \mathit{length}(v, I) + \mathit{length}(v, J)$ and set $p(v, I \cup J) := \{(v, I), (v, J)\}$ to store both labels as predecessors.

If Q is empty, there was no Steiner tree for T and we return this information instead. The pseudocode of the algorithm is shown in Algorithm 2.

Theorem 1.3.4 (Hougardy, Silvanus and Vygen 2017 [HSV17]). *The Dijkstra-Steiner algorithm is correct and can be implemented to run in $\mathcal{O}(3^k n + 2^k(n \log n + m) + 2^k n f_{\mathcal{L}})$, where $n := |V(G)|$, $m := |E(G)|$, $k := |T|$ and $f_{\mathcal{L}}$ is an upper bound on the time required to evaluate $\mathcal{L}$.*

Algorithm 2: The Dijkstra-Steiner algorithm. [HSV17]

Input : An instance (G, c, T) of the minimum Steiner tree problem in graphs with $r \in T$ and a valid lower bound $\mathcal{L}$ to r.

Output: A minimum Steiner tree.

```
1  length(t, {t}) := 0 and p(t, {t}) := ∅ for t ∈ T \ {r};
2  P := ∅;
3  Insert (t, {t}) for t ∈ T \ {r} into Q;
4  while Q ≠ ∅ do
5      Take (v, I) ∈ Q minimizing length(v, I) + L(v, T \ I);
6      Remove (v, I) from Q and add it to P;
7      if v = r and I = R \ {r} then
8          return Recursively reconstructed minimum Steiner tree;
9      for {v, w} = e ∈ E(G) with (w, I) ∉ P do
10         if (w, I) ∉ Q or length(v, I) + c(e) < length(w, I) then
11             p(w, I) := {(v, I)};
12             length(w, I) := length(v, I) + c(e);
13             Add (w, I) to Q, if it is not contained yet;
14     for (v, J) ∈ P with I and J disjoint do
15         if (v, I ∪ J) ∉ Q or length(v, I) + length(v, J) < length(v, I ∪ J) then
16             p(v, I ∪ J) := {(v, I), (v, J)};
17             length(v, I ∪ J) := length(v, I) + length(v, J);
18             Add (w, I ∪ J) to Q, if it is not contained yet;
19 return There is no Steiner tree;
```

Rectilinear Steiner Trees

In the rectilinear Steiner tree problem, we are given a set of terminals T together with positions $\bar{p} : T \to \mathbb{R}^2$ of the terminals. Our goal is to find a tree S with $T \subset V(S)$ and positions $p := V(S) \to \mathbb{R}^2$, with $p(t) = \bar{p}(t)$ for $t \in T$ that minimizes

$$\mathit{length}(S) := \sum_{\{v,w\} \in E(S)} \|p(v) - p(w)\|_{l_1}.$$

In this case a minimum spanning tree on the terminals (in the complete graph on the terminals with edge cost given by the l_1-distance between the endpoints) is even a 1.5-approximation [Hwa76]. Furthermore, there are heuristics that solve almost all VLSI-instances of Steiner trees within a few percent of optimality in practice. For optimum solutions, there is a lookup table approach, which is very efficient for $|T| \leq 9$ [CW07]. There is an optimal solver called GeoSteiner [Juh+18] that also solves most instances with $|T| \leq 200$ in reasonable time.

An idea that we will borrow from rectilinear Steiner trees is that the problem can be reduced to the Steiner tree problem in graphs. This can be achieved with the Hanan grid [Han66], which is a grid graph containing the terminals.

Definition 1.3.5. *Let $(T, \bar{p})$ be an instance of the rectilinear Steiner tree problem. Let $X := \{\bar{p}_x(t) | t \in T\}$ be the x-coordinates and $Y := \{\bar{p}_y(t) \mid t \in T\}$ be the y-coordinates of the terminals. We define the* Hanan graph *of $(T, \bar{p})$ as $H := (V(H), E(H))$ with*

$$V(G) := X \times Y \text{ and}$$
$$E(G) := \{\{(x, y), (x', y')\} \mid \text{exactly one of } x = x' \text{ or } y = y'\}.$$

We define the edge costs $c : E(H) \to \mathbb{R}_{\geq 0}$ as $c(e) = \|v - w\|_{l_1}$ for $\{v, w\} = e \in E(H)$. We call the instance $(H, c, \bar{p}(T))$ of the Steiner tree problem in graphs the Hanan graph instance *of $(T, \bar{p})$.*

Let $(H, c, \bar{p})$ be the Hanan graph instance for $(T, \bar{p})$. Note that this instance might have fewer terminals than $|T|$. Suppose, we have a Steiner tree S_H for the graph instance. Then we can construct a rectilinear Steiner tree S with the same total cost by adding T to S_H and adding the edges $\{t, \bar{p}(t)\}$ for $t \in T$. The positions are given by $p(t) := \bar{p}(t)$ for $t \in T$ and $p((x, y)) := (x, y)$ for $(x, y) \in S_H$.

Theorem 1.3.6 (Hanan 1966 [Han66])**.** *Let $(T, \bar{p})$ be an instance of the rectilinear Steiner tree problem and $(H, c, \bar{p}(T))$ be its Hanan graph instance. The optimum values of both instances are equal.*

This allows us to use the Dijkstra-Steiner algorithm to solve the rectilinear Steiner tree problem.

Linear Delay Steiner Arborescences

Steiner trees are often used as a starting point or an estimate for routing and buffering. But in the basic form, timing properties of the Steiner trees are not part of the problem.

We want to use Steiner trees as a basis for decisions during buffering. To achieve this, we need to estimate the delay after buffering. For long evenly spaced repeater chains, the delay grows roughly linear in the distance to the root of the repeater tree. Each side branch will also increase the delay on a path. These considerations lead to the following model [Bar+10].

We are given a net $N := \{r\} \cup T$ with source r and sinks T, as well as positions $\bar{p} : N \to \mathbb{R}^2$. A *Steiner arborescence* for $(N, \bar{p})$ is an arborescence A, rooted at r with leaves T together with positions $p : V(A) \to \mathbb{R}^2$ such that $p(t) = \bar{p}(t)$ for $t \in N$ and

- r has outdegree 1 and
- each Steiner vertex $v \in V(A) \setminus N$ has outdegree 2.

Now we can define a linear delay model for a Steiner arborescence A, p. We are additionally given a delay per length $w \in \mathbb{R}_{>0}$ and a branch delay $b \geq 0$. Then the linear delay to the vertices $v \in V(A)$ is defined as

$$\mathit{delay}_{A,p}(v) := b \cdot (|V(A_{[r,v]})| - 2) + w \cdot \sum_{(v,w) \in E(A_{[r,v]})} \|p(v) - p(w)\|_{l_1}.$$

This way, we have a measure for the delay that includes the distance in the tree from the root to each vertex and also the number of branches on a path.

In the delay-bounded Steiner arborescence problem (DBSA), we are given a net $N := \{r\} \dot{\cup} T$, with root r and sinks T, positions $\bar{p} : N \to \mathbb{R}^2$, a wire delay $w > 0$, branch delay $b \geq 0$ and required arrival times $\mathit{rat} : T \to \mathbb{R}_{\geq 0}$. Our goal is to find a Steiner arborescence A, p minimizing

$$\mathit{length}(A, p) := \sum_{(v,w) \in E(A)} \|p(v) - p(w)\|_{l_1},$$

such that the worst slack is nonnegative

$$\mathit{wsl}(A, p) := \min_{t \in T} \left[\mathit{rat}(t) - \mathit{delay}_{A,p}(t) \right].$$

There is no constant factor approximation algorithm for the DBSA problem, unless $P = NP$ [HR13]. So finding a good DBSA is hard.

For checking feasibility of an instance, we can find a Steiner arbeorescence that maximizes the worst slack by using Huffman coding [Huf52] [HR13]. We first build up the arborescence A. In the beginning, we add N to A. We always maintain a set of active vertices H and a slack bound $\sigma : H \to \mathbb{R}$ for the active vertices. We initialize $H := T$ and set the slack bound

$$\sigma(t) := \mathit{rat}(t) - \|p(t) - p(r)\|_{l_1} \text{ for } t \in T.$$

While $|H| \geq 2$, we pick the two vertices s_1, s_2 from H with maximum slack bound. We add a vertex v to A with edges (v, s_1) and (v, s_2). Then we add v to H and set $\sigma(v) := \min\{\sigma(s_1), \sigma(s_2)\} - b$. Finally, we remove s_1 and s_2 from H.

When $|H| = 1$, let $\{s\} := H$. We add the edge (r, s) to A and set $p(t) := \bar{p}(t)$ for $t \in N$ and $p(v) := \bar{p}(r)$ for $v \in V(A) \setminus N$. Then we return the Steiner arborescence A, p.

If we are willing to accept a small violation of the required arrival times, we can find good solutions. There is a bicriteria approximation algorithm by Held and Rotter [HR13] that starts with a rectilinear Steiner tree S for N and finds for $\epsilon > 0$ a Steiner arborescence A, p with $\mathit{delay}_{A,p}(t) \leq (1 + \epsilon) \cdot \mathit{rat}(t) + 2b$ for $t \in T$ that has cost at most $\left(1 + \frac{2}{\epsilon}\right) \cdot \mathit{length}(S) + \frac{4b|N|}{\epsilon}$.

For a given Steiner arborescence A without positions, Held and Rockel-Wolff [HR18] presented a Min-Cost-Flow formulation that can be used to find positions p in time $\mathcal{O}(|T|^2 \log |T|)$ such that the delay bounds are met and the length is minimized, if such positions exist. This can be used in a branch-and-bound approach to find a minimum length DBSA.

For many applications it is also interesting to look at the Lagrangian relaxation version of this problem. Instead of required arrival times, we are given delay weights $\lambda_t \in \mathbb{R}_{\geq 0}$ for $t \in T$ and look for a Steiner arborescence A, p that minimizes netlength plus the weighted sum of delays

$$\mathit{length}(A, p) + \sum_{t \in T} \lambda_t \, \mathit{delay}_{A,p}(t).$$

For this problem, the bicriteria approximation algorithm, with well-chosen ϵ, serves as a constant factor approximation [Rot17]. The Minimum-Cost-Flow formulation can also be adapted to find optimum positions for a given arborescence in this case. So we can again use a branch-and-bound algorithm to solve this problem optimally (in exponential worst case running time).

1.3.3 Resource Sharing

For many of the problems we encounter in VLSI design, it is already NP-hard to find solutions to single instances, like a minimum Steiner tree as a route for a net. Additionally, the solutions of single instances are typically not independent. Two wires can not occupy the same space and the timing properties of a net influence the timing requirements downstream and upstream of that net. Even if we had a Steiner tree oracle, packing Steiner trees is still NP-hard [Kar72] (even in grid graphs [KV84]).

One approach to overcome this obstacle in practice is to use Lagrangian relaxation formulations and move the constraints into the objective function. For timing constrained routing and buffering, it can be shown that we can formulate the problem as a resource sharing problem [Dab+23].

In the Min-Max resource sharing problem, we are given a finite set of resources $\mathcal{R}$ and customers $\mathcal{N}$, as well as an approximation factor $\sigma \geq 1$. For each customer $C \in \mathcal{N}$, we have a compact set of feasible solutions $\mathcal{B}_C \subseteq \mathbb{R}^{\mathcal{R}}_{\geq 0}$ and an oracle function $f_C : \mathbb{R}^{\mathcal{R}}_{\geq 0} \to \mathcal{B}_C$ that maps a price vector $p \in \mathbb{R}^{\mathcal{R}}_{\geq 0}$ to a solution with $p^T f_C(p) \leq \sigma \inf_{b \in \mathcal{B}_C} p^T b$.

Our goal is to find for each $C \in \mathcal{N}$ a b_C such that the b_C minimize

$$\max_{r \in \mathcal{R}} \sum_{C \in \mathcal{N}} (b_c)_r.$$

Since we will only consider an approximation algorithm to the resource sharing problem, we can drop the compactness constraint for the solution sets. We can find a $\sigma(1+\omega)$-approximation for $\omega > 0$ using an algorithm by Müller, Radke and Vygen [MRV11]. Its running time has been improved by Blankenburg [Bla22] to $\mathcal{O}(\theta \log |\mathcal{R}|(|\mathcal{N}| + |\mathcal{R}|)\omega^{-2})$, where θ denotes the time needed for an oracle call.

Now let us give an overview over different objectives that we might want to optimize during buffering and routing and how to incorporate them in a resource sharing problem. We will only give the core ideas. In practice a lot of work is spent to accurately model all important aspects of the objectives.

Routing and placement have the simplest models. They provide a polynomial number of fixed constraints. For routing, the edges of the global routing graph are the resources. We assign as capacity to an edge half the area of each tile it belongs to. The area consumed by blockages is considered as fixed usage. Each net is a customer, using the area that is blocked by its wires. That is, the wires width and spacing and possibly some estimate of the additional blocked area that is required to obey all design rules. Similarly, for placement, some tiling of the placement area serves as resources and the placement area of each tile is used as a capacity. Blocked placement area is considered as constant usage. For our purposes, we also consider the non-buffering gates as blockages. The nets are the customers and use the area that is blocked by repeaters in the tiles the repeaters are placed in. To arrive at a suitable formulation for the resource sharing problem, we divide the usages by the capacities.

Power and netlength are both objectives that we want to minimize. For them, we use an estimate for the final values as a budget. The main problem is to find a budget that is as low as possible, but achievable. Then we divide the usages by the budgets [Dab+23].

Timing is more complicated, because the timing on each path is relevant. The natural formulation is to have a resource for each path from primary input to primary output in the fine timing graph. The difference between required arrival time at its endpoint and arrival time at its start is its capacity. Each edge in the timing graph is a customer, using its delay on paths that it is contained in. However, there are exponentially many of these paths. So this is not feasible. Instead, arrival time customers can be used. They are a method, proposed by Held et al. [Hel+17] that allows us to define a timing resource for each edge in the fine timing graph. With these resources, the cycle time is met, if and only if the timing resource usage of each edge is at most 1.

2 The Buffering Problem

The buffering problem in its basic formulation in 1.1.5 is too vague to produce good results in practice and has search space that is too large to be tractable.[1]

This is why many different formulations have been proposed, ranging from buffering a given planar Steiner tree to models on given routes or on grids. They incorporate different objectives and respect different constraints.

The purpose of this chapter is to discuss different formulations and their drawbacks and present a flexible model framework for buffering based on [Rot17] and [Roc18] that is tractable, but allows a wide variety of objectives to be incorporated.

The most important questions, that a formulation of the buffering problem needs to answer, are the following. How do we model the placement and routing space? How do we model the shapes and connection of repeaters? How do we model the timing, in particular, how accurate can we make the timing model? Which additional objectives do we want to cover?

We will start by reviewing the literature on buffering problems. Then we will summarize different possibilities to answer the above questions. Finally, we present the model framework that we are proposing.

2.1 Previous work

Buffering started out as a process to reduce the load on gates with high fanout, because gate delay was the limiting factor. This is, why it is often referred to as the fanout reduction problem or the fanout tree problem in the older literature [BCD89][Tou90].

With ever decreasing feature sizes, the influence of wire resistance became more relevant and buffering became more important and more complicated. With this change, buffering algorithms became more and more sophisticated. Probably the most important work in the field is the dynamic program by van Ginneken [Van90], which is the basis of many other algorithms. In its first version, it was only able to handle a library with a single buffer. It inserts buffers into a given 2D-route such that the Elmore delay is minimized in time $\mathcal{O}(n^2)$, where n is the number of possible buffer positions on that

[1] For Elmore delay, and arbitrary repeater positions, we do not know if it is in NP. The obvious encoding of a solution by the repeater positions may contain square roots. Thus, a checking oracle would have to answer the question if a sum of square roots is below a given number.

route. This version of the buffering problem, where the route is fixed and we try to insert repeaters, is often referred to as the buffer insertion problem.

Van Ginneken's dynamic program was extended by Lillis et al. [LCL96a] to consider a larger library L that may also contain inverters in $\mathcal{O}(|L|^2n^2)$. Their algorithm is also able to consider wire sizing. Alpert and Devgan [AD97] presented a way to subdivide long edges of the route for good timing properties.

Later on Shi and Li [SL05] were able to reduce the running time of the dynamic program to $\mathcal{O}(|L|^2 n \log n)$ by a clever candidate tree data structure. They also introduced the idea of pruning dominated candidates. Li et al. [LZS12] were able to reduce the running time even further to $\mathcal{O}(|L|^2 n + |L||T|n)$.

Since the Elmore delay does not consider resistive shielding and hence may lead to excessive use of buffers, Alpert et al. [ADQ99] combined van Ginneken's algorithm with a bottom up moment and C_{eff} computation to allow for AWE-computations. Gao and Wong [GW01] extended this idea even further by presenting a technique for single sink nets that allows to use SPICE simulation in the algorithm.

The buffer insertion problem for minimizing slack, required arrival time or delay allows for these optimal polynomial time algorithms. The cost based variant on the other hand, where we want to insert buffers, minimizing a cost function, like power, subject to delay constraints, is NP-complete [SLA04].

For the cost based variant, a pseudo-polynomial algorithm was given in [LCL96a] and a FPTAS, minimizing simple costs per buffer was developed by Hu et al. [HLA09]. The FPTAS was then extended by Romen [Rom15] to 3D-routes and more general cost functions, which allowed him to incorporate routing costs as well. Furthermore, Permin [Per16] gave a bicriteria-approximation algorithm that is also able to include slew limits.

The work that we considered so far only inserts repeaters into a given route. Computing good interconnect wiring with respect to timing also became more important with the rising wire resistance. Boese et al. [Boe+94] showed that finding Steiner trees that minimize the maximum Elmore delay or the sum of Elmore delays to the sinks is already NP-hard. They were able to show that there are instances of the min-max problem such that the Hanan grid does not contain an optimal solution. However, they were also able to show that the Hanan grid does always contain an optimal solution to the sum-version and gave an optimal algorithm for this version. Lillis et al. [Lil+96] provided the P-tree algorithm that computes a route minimizing the maximum Elmore delay for a subset of the topologies that arises from a fixed permutation of the sinks. Scheifele [Sch17] gave an approximation algorithm for both variants that has been improved by Glubrecht [Glu23]. For the slack minimization variant, a heuristic has been proposed by Lin et al. [LCL11] that also takes blockages into account.

Of course computing a route and buffering are not independent of each other, so performing the tasks one after the other does limit the solution space. Multiple approaches for achieving a combination of both have been explored. We will split these algorithms into four categories.

We start with those that fix the topology, and perform routing and buffering at the same time. For paths, Zhou et al. [Zho+99] gave an algorithm that is able to consider blockages, congestion costs and perform layer assignment. Hu et al. [Hu+03] gave an

extension to van Ginneken's algorithm that is able to adapt the routing to blockage positions in a 2D route without incorporating all blockage corners into a Hanan grid. An algorithm that works in a global routing graph and is also able to avoid blockages, consider placement costs and incorporate slew limits has been presented by Natura [Nat17].

Next, we consider algorithms that heuristically build up the topology. Okamoto and Cong [OC96] presented a heuristic that tries to maximize the slack by using the A-tree algorithm for Steiner trees as a basis and simultaneously inserting buffers for a library consisting of only one buffer.

Starting from a given 2D route computed with a linear delay model, the heuristic by Bartoschek et al. [Bar+09] allows changes to the topology by inserting two parallel wires and merging at later nodes. Furthermore, it dynamically computes buffer positions based on optimal buffer distances on a long wire and it allows more accurate delay models than Elmore delay. It has been extended to work on 3D-routes [Rom15] and with a more sophisticated cost function [Rot17].

Then there are algorithms which explore a restricted subspace of the topologies. Lillis et al. [LCL96b] used their P-tree algorithm in combination with a simultaneous buffer insertion dynamic program to minimize cost subject to delay constraints, exploring the topologies with fixed permutation. Salek et al. [SLP98] gave an algorithm that explores the space of LT-trees (of type I) [Tou90] for a fixed permutation, where a buffer is placed at each vertex and the wiring of each stage is a P-tree. LT-trees are trees, where each vertex must have at least two children and at most one child may be an inner vertex. Then they used local changes of the permutation to iteratively improve the solution in [SLP99].

Finally, there are algorithms that explore all topologies as well. The algorithm of Cong and Yuan [CY00] constructs a tree, maximizing the required arrival time for Elmore delay. Their algorithm is meant to work with fixed sets of buffer locations, where a yes/no decision can be made for placing the buffer. They always pick the subtree maximizing the required arrival time, and extend it by an edge and directly merge with all other solutions at that vertex. Hrkić and Lillis [HL02] iteratively extend all trees connecting a given sink set by all buffered paths to the vertices in the grid graph, pruning those labels that can be proved to be redundant by an estimate of the upstream delay. Then they merge these extended subtrees to subtrees spanning more sinks.

Rockel-Wolff [Roc18] then presented an algorithm based on the Dijkstra-Steiner algorithm that combines the advantages of both algorithms. It proceeds in a Dijkstra like order and introduces a lower bound function that can be used to guide the search. Furthermore, it extends the delay model to also incorporate a discrete slew function and it is able to work in general graphs and perform layer and wire assignment.

All these algorithms only work on single instances. But on a chip, the instances are not really separate from each other, but rather compete for the same resources. Thus, it is important to also take a global view on buffering. This has been done by Lin and Marek-Sadovska [LM91] for the fanout problem. They partition all cells into critical and uncritical and then insert buffer trees into the fanout of the critical cells. As a recovering step, they then remove buffers from uncritical fanouts, as long as no timing constraints

are violated. Hu et al. [Hu+18] tackle this problem via a new metric, the pFOM, which is the p-norm of the deviation from a slack target. It generalizes the total negative slack. They also provide a variant of the van Ginneken algorithm that optimizes pFOM.

While these two approaches heuristically incorporate global competition for the resources, BonnRouteBuffer, which was first presented by Rotter [Rot17] is a tool that uses a resource sharing formulation as a basis for buffering. In theory, it can use any of the above algorithms that allow minimizing a cost function incorporating placement, routing and timing costs. In practice, it first computes a 2D topology, then computes a global route with the same topology for a linear timing model and finally uses an improved version of the algorithm from [Bar+09] and [Rom15] to compute a buffering based on this. For hard cases, it instead uses the routing and buffering oracle from [Nat17]. Its practical implementation has been improved to be competitive to industrial tools by Daboul [Dab21]. The overall approach is captured in [Dab+23].

2.2 Search Space

Before we present our model framework, we want to take a look at the search space. Recall the notion of a repeater tree and buffered route from 1.1.1. Our goal is to find such a buffered route for each net, such that all timing, placement and routing constraints are met, and the power consumption is as small as possible. This task includes selecting the topology, finding repeater positions and the nets they belong to, selecting the repeaters computing all the routes. The search space is too large to be able to find a solution for each net in reasonable running time.

This is why we and all the works we have cited in the previous section, will restrict the search space. This section will show options for restrictions of the search space and give some motivation why we are choosing the ones we use.

2.2.1 Routing, Placement and Repeaters

For very simple timing models and no repeaters, like a linear timing model ([HR18], [HR13]) or the Elmore delay ([Boe+94]), the routing space (placement is not necessary in this case) is often modelled by the rectilinear plane. If we have repeaters, then the placement grid gives us a discrete set of possible repeater positions. A natural representation for possible wiring between these positions are edges. So we end up with a graph as our representation of the routing and placement space. By adding or removing positions or edges, we can scale the accuracy with which we represent our search space.

Graph based models usually assume that the repeaters are points, in the sense that x- and y- coordinates of the repeater, and both the input and output pins are the same. We will call this type of repeater model *point repeaters*.

All the graph models we are covering assign x/y-coordinates to each vertex. Based on this, we will define a graph model.

Definition 2.2.1. *A* graph model *is a graph G together with positions $p : V(G) \to \mathbb{R}^2$, such that $p(v)$ and $p(w)$ differ in at most one coordinate if $(v, w) \in E(G)$.*

The base $\mathcal{B}(G,p)$ *of a graph model* (G,p) *is the graph that arises from* G *by contracting all sets of vertices* $S \subseteq V(G)$ *with* $p(v) = p(w)$ *for each pair* $v, w \in S$ *and then removing duplicate edges.*
A graph model (G,p) *is called* flat *if* $\mathcal{B}(G,p) = G$.

If a model is flat, that does not mean that it can not model different layers. A typical strategy is to assign a pair of x- and y-layers. But we can also model the layers directly.

Definition 2.2.2. *A* layered graph model *is a graph model* (G,p) *together with a layer for each vertex* $\text{layer} : V(G) \to [z_{max}]$.

An easy way of creating a layered graph model is to start with its base. Then, we copy the vertices of the base for each layer and add edges. If we want to adhere to the preferred directions of the layers, the edges in x-direction of the base would be added on the layers with preferred x-direction and the edges in y-direction on the layers with preferred y-direction. We may leave out blocked edges, or add a penalty in an objective function if they are used. Additionally, edges between two copies of the same vertex on subsequent layers are added to model vias.

Models that have a grid graph as a base are very common ([Hu+03], [CY00], [HL02], [Roc18], [Dab+23]). An example is the global routing graph, or a Hanan graph like the one presented in Chapter 4.2. Simpler models use a Steiner tree or a linear delay Steiner arborescence as a base, possibly with subdivisions on long edges ([Van90], [Bar+09]). All these models are used both in a flat or in a layered variant. Some algorithms also use a global or even detailed route as a base, again with subdivisions on long edges.

In a graph model, there are two ways to model repeaters. We can model a repeater as an attribute to vertices or we can model it as an edge. For flat models, it is usually enough to model repeaters as attributes at vertices. For layered models however, modelling repeaters as edges has the advantage that the repeater-edge can connect the layer of the input pin to the layer of the output pin of the repeater. This way, we are able to accurately model that we may have to use vias to connect to the pins of the repeater and that they may not lie on the same layer. In flat models with repeaters as attributes and layers chosen per net, we can make up for that by including the layer-changes in the objective, when adding a repeater. In layered models, we can do the same, but we will have trouble modelling a layer-change at a repeater correctly.

We can also model repeaters by edges in flat models. Using a repeater edge corresponds to placing the repeater somewhere on that edge in this case. This has the advantage that it does not restrict repeater placement to positions at vertices. Transferring this to layered models is possible, but if we want to model layer changes accurately, we would have to add the cross product of layer changes and repeaters as edges. Furthermore, for these models that allow a continuous placement of the repeaters, we need to make sure that our objective function is convex in these positions, or at least has a unique minimum. Otherwise, it is hard to find the global optimum.

For the placement, it is sufficient to look at the base of the graph model. We can require the repeaters to fit at exactly the position that they are placed at. This may be too restrictive in cases, where the graph does not contain every possible placement

position. So we can require the repeater to exactly fit at a position that is close to the position that we place it on, or we can use a density based approach. A density based approach can either check if the repeater fits into a placement tile, or in a resource sharing formulation, can include the placement cost of the repeater into the objective.

Finally, we need to take a closer look at solutions of the buffering problem in a graph model. For a buffering problem on a net $N := \{r\} \cup T$ in a graph model (G, p) with a set of vertex attributes X_V and edge attributes X_E, a solution consists of an arborescence A, with an embedding $\kappa : A \to G$ of A into G and maps $b : V(A) \to X_V$ and $w : E(A) \to X_E$.

Definition 2.2.3. *Let A be an arborescence. Let $\mathcal{A}$ be the graph that arises from A by replacing each maximal path such that all its inner vertices are of degree 2 by an edge. We call $\mathcal{A}$ the* topology *of A.*

Some models, in particular those that are based on a route or Steiner tree also restrict the topology. Either to a single topology, or a class of topologies.

2.2.2 Objectives

The reason why we are inserting repeaters is that we want to improve timing properties of our interconnect. However, repeaters also consume additional resources, like placement space and power. This is why in the literature different types of objectives have been considered, with more and more aspects being taken into account. Examples of objectives that have been considered, range from the maximum slack in Elmore delay in [Van90]

$$\max_{A \text{ buffered route}} \quad \min_{t \in T} rat(t) - delay_{Elmore,A}(r,t) - delay(r, downcap(r))$$

to the minimization of a weighted sum of delay costs, routing congestion costs, power consumption costs and placement density costs in a Lagrangian relaxation formulation in [Dab+23]

$$\begin{aligned}\min_{A \text{ buffered route}} \quad & \sum_{t \in T} \lambda_t \cdot \left[delay_{RC,A}(r,t) + delay(r, downcap(r)) \right] \\ & + \sum_{v \in V(A) \setminus T} \sum_{e \in R_v} \lambda_e \cdot length(e) \\ & + \sum_{\substack{v \in V(A) \\ b(a) \neq \square}} \lambda_{power}\, power(b(a)) + \lambda_{bin(v)} \frac{\text{area}(b(v))}{\text{area}(bin(v))}.\end{aligned}$$

Here the λ-values are the factors from the Lagrangian relaxation, R_v denotes the route with source $v \in V(A) \setminus T$ and $bin(v)$ refers to the placement bin of v.

We will now take a look on how these objectives can be included in a model of the buffering problem.

Timing

For timing models, the main question is, how accurate we can make it and still have an algorithm with practical running times. In general, the more accurate, the longer the timing evaluation takes.

We split our timing models into two categories. Stage based models are models that require to time a whole stage (see Chapter 1.2), or at least consider the whole wiring to compute the delay. An example would be AWE. Piece wise models can compute the delay segment by segment.

For the stage based models there is not a lot to discuss. They are more accurate, but much slower, since we have to evaluate the timing for a whole net or even a complete subsolution each time we want to know timing values.

Piece wise models comprise linear delay models and the Elmore delay, as well as extensions of it. They can be computed by traversing the solution in reverse topological order. We will use the more accurate Elmore delay. There, we only need to know the downstream capacitance at each vertex. Hence, it can easily be used in a dynamic program. A major deficiency is that it does not consider input slews. Furthermore, it may be too pessimistic in many cases, such that we insert too many repeaters. This can degrade other objectives, like power and placement or routing congestion.

To mitigate these problems we augment it by a slew computation for wires. Furthermore, we use timing rules that are given as input to approximate the gate delay, like we described in Chapter 1.2.4. We use the simple variant that ignores the effect of the wiring resistance instead of C_{eff} computations, because they require knowledge of the upstream net.

Timing rules come with ranges for slews and downstream capacitances, in which the behaviour is modeled accurately. Gates may also show unwanted behaviour outside these ranges. Values outside these ranges are treated as timing violations and we can assume the presence of upper bounds. Sometimes, constraints like slew- or capacitance- limits are relaxed a bit, such that we are able to find a solution for infeasible instances. In this case violations of the original constraints are moved into the objective.

A problem with using slew in this kind of model is that we have to know the slew at the inputs of the segments. This can be dealt with by restricting slew and capacitance to ranges, where the slew function is invertible. Then we can compute the slew from the output to the input.

Additionally, we assume that there is a lower bound on the delay through a repeater. This is the case in practice, because repeaters always have at least a small output wiring segment that has to be charged.

These considerations lead to the definition of an RC-timing model that we will be using as input:

Definition 2.2.4. *Given a graph model G with a library L and a net $N := \{r\} \cup T$, a* RC-timing model *on (G, L, N) is a tuple $(delay, cap, slew, caplim, S, s_r, (S_t)_{t \in T})$, consisting of*

- *a delay function $delay : E(G) \cup L \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \mathbb{R}_{\geq 0}$, computing for each (routing) edge or repeater $a \in E(G) \cup L$ the delay $delay(a, c, s)$ through a for the downstream capacitance $c \in \mathbb{R}_{\geq 0}$ and the input slew $s \in \mathbb{R}_{\geq 0}$, such that*
  - *delay is nondecreasing in c and s and*
  - *each repeater $b \in L$ has a base delay $C_b > 0$ such that $delay(b, \cdot, \cdot) \geq C_b$,*
- *a capacitance function $cap : E(G) \cup L \to \mathbb{R}_{\geq 0}$, mapping each edge $e \in E(G)$ to its capacitance and each repeater $b \in L$ to its input pin capacitance,*
- *a slew function $slew : E(G) \cup L \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \mathbb{R}_{\geq 0}$, computing for each edge or repeater $a \in E(G) \cup L$ the output slew $slew(a, c, s)$ of a if the downstream capacitance is $c \in \mathbb{R}_{\geq 0}$ and the input slew is $s \in \mathbb{R}_{\geq 0}$, such that*
  - *$slew(a, c, \cdot)$ is strictly increasing and continuous for all $c \in \mathbb{R}_{\geq 0}$,*
  - *$slew(a, \cdot, s)$ is nondecreasing for all $s \in \mathbb{R}_{\geq 0}$,*
- *a capacitance limit $caplim : \{r\} \cup L \to \mathbb{R}_{\geq 0}$ mapping the root r and each repeater in L to an upper bound on the capacitance they can drive,*
- *an upper bound on the allowed slews $S \in \mathbb{R}_{\geq 0}$,*
- *an output slew function $s_r : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ for the source, computing the output slew of the source $s_r(c)$ for a given downstream capacitance $c \in \mathbb{R}_{\geq 0}$ that is also nondecreasing,*
- *an upper bound $S_t > 0$ on the slew at each sink $t \in T$*

Other Objectives

Apart from timing, we might want to model some other objectives. The most common objective is power. Power consumption depends on the repeaters we are using and the netlength. Static consumption may be modeled as a constant, while dynamic power consumption may depend on the wiring driven by the repeater. In both cases, we can include it in the repeater cost.

Routing resources like netlength or usage of global routing edges have a limited meaning in graph models that are based on routes or Steiner trees, in particular if the topology is fixed. There, we may restrict to a penalty for the layers and wire widths we are using. For grid graph based models, in particular for the global routing graph, it makes more sense to add the netlength or edge prices into our objective.

Of course, we can also add placement costs or area consumption into our objective cost. This can again be modeled as part of the repeater cost.

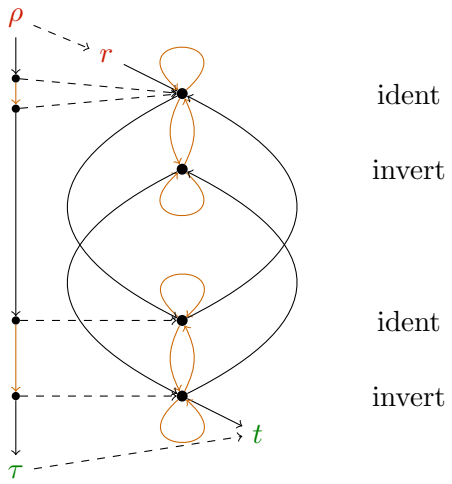


Figure 2.1: A mapped arborescence on the left in the graph that we will construct in Section 2.3.2 on the right. Repeater edges are orange and wiring edges are black, the mapping is shown with dashed edges.

2.3 New Framework

In this section, we present the basic framework for solving buffering problems that we are proposing. The high level idea of our framework is to reduce formulations with a wide variety of objective functions and graph models to a search for an arborescence with a mapping into a graph. It is centered around an abstract combinatorial problem that we call Timing-Driven Min-Cost Mapped Arborescence Problem or TMCMAP in short. A mapped arborescence is defined as follows.

Definition 2.3.1. *For a directed graph H and a net $\{r\} \cup T \subset V(H)$, a* mapped arborescence*, is an Arborescence A, together with a mapping κ of A into H that identifies r with the root of A and the leaves with T. Formally, $\kappa : V(A) \cup E(A) \to V(H) \cup E(H)$, with $\kappa(V(A)) \subseteq V(H)$ and $\kappa(E(A)) \subseteq E(H)$ and for each $(v, w) \in E(A)$, we have $\kappa((v, w)) \in E(H)$ that starts in $\kappa(v)$ and ends in $\kappa(w)$. Note that we allow to map to vertices and edges multiple times, except for the root and leaves, which are mapped exactly to the source r and sinks T.*

An example of a mapped arborescence in the practical setting of Section 2.3.2 can be seen in Figure 2.1. Separating the arborescence and the graph and relating them with a mapping will allow us to represent both wires (black) and repeaters (orange) as edges in our graph.

The exact formulation of the TMCMAP will be given in Subsection 2.3.1. It is quite complex, but we can solve it optimally and it is very flexible. This allows us to capture almost all of the aspects of buffering that we have considered so far.

It is an extension of the models presented in ([Rot17], [Dab+23], [Roc18]). The idea of turning a buffering problem into a search for a mapped arborescence was used in [Rot17] and [Dab+23]. This formulation requires to explicitly consider polarities during

the search. It is able to minimize a cost function that takes the form of a weighted sum and only propagates slew limits to avoid slew violations. Furthermore, it requires a topology to be given. We use some ideas from [Roc18]. There, polarities are no longer modeled explicitly, but moved into the graph. The algorithm is able to enumerate all topologies and minimizes either a weighted sum of delays or the maximum delay. It uses slews in the delay functions, but requires the slew to be discrete. This turns out to be a severe restriction in practice.

In our framework, we will be able to use continuous slews with mild but realistic assumptions on the slew. We split the objective into a multidimensional objective space that represents the different subobjectives with an objective function that gives us values in this objective space and an evaluation operator that gives us the single value that is to be minimized. This objective function and evaluation operator capture the essence of the different objective functions. This way, we are able to minimize all kinds of combinations of weighted sums and weighted maxima. An example objective function is the maximum delay plus weighted power. Another one is a weighted sum objective from a Lagrangian relaxation like the one presented in [Dab+23].

Nonetheless, the TMCMAP can be solved optimally, or by returning early, with an approximation guarantee. We will show this in Chapter 3. Both variants require exponential worst case running time.

Here, we will first present the formulation of the TMCMAP. Then we show how we can construct an instance of it for graph models and a certain set of objective functions. We will see that this construction works for most formulations and practical objectives considered in Section 2.2.

2.3.1 Problem Formulation

We are given a directed graph H and a net $N := \{r\} \cup T$ with $N \subseteq V(H)$. For the sinks $t \in T$, we are given slew limits $S_t > 0$. Additionally, we are given a global slew limit $S > 0$ and an objective space $\Omega := \mathbb{R}^d_{\geq 0}$ for some $d \in \mathbb{N}$. The entries of the objective space will contain our different objectives, like the delay for each branch or the power consumption. On this objective space, we have an evaluation operator $\chi : \Omega \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that is

- *sublinear*: for all $\omega, \nu \in \Omega$, $\alpha > 0$
  - $\chi(\alpha\omega) \leq \alpha\, \chi(\omega)$,
  - $\chi(\omega + \nu) \leq \chi(\omega) + \chi(\nu)$ and
- *nondecreasing*: $\chi(\omega) \leq \chi(\nu)$ for all $\omega, \nu \in \Omega$ with $\omega \leq \nu$ entrywise.

The evaluation operator is supposed to condense our separate objectives into a single objective cost that we are optimizing.

Finally, we are given some functions for the edges that dictate the behaviour of the objective and timing values.

The first type are limits and bounds. A slew lower bound for each edge that might be influenced by the capacitance $\mathit{minslew} : E(H) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ (since we are going

to propagate the slew from the sinks to the source, we need lower bounds instead of upper bounds), as well as a constant capacitance limit from above for each edge $caplim : E(H) \to \mathbb{R}_{\geq 0}$. The second type are the timing functions $slew^{-1} : E(H) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and $cap : E(H) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ that dictate how the slew and capacitance change from the head of an edge to its tail.

For these functions, we require that for each edge $e \in E(H)$, the following hold:

- Either $cap(e, c) \geq c$ for each capacitance $c \in \mathbb{R}_{\geq 0}$ or $cap(e, \cdot)$ is constant. We call edges with the latter property *constant edges*.
- $slew^{-1}(e, c, \cdot)$ is strictly increasing and surjective (onto $\mathbb{R}_{\geq 0}$) for each capacitance $c \in [0, caplim(e)]$ and $slew^{-1}(e, \cdot, s)$ is nonincreasing for each slew $s \in \mathbb{R}_{\geq 0}$.
- $slew^{-1}(e, c, s) \leq s$ for constant edges e and all $c, s \in \mathbb{R}_{\geq 0}$.

Note that we are defining $slew^{-1}$ and cap outside the limits as well. This simplifies the notation. Instead of making the functions undefined for some values, we will require the objective cost to be infinite outside these ranges (see Definition 2.3.3). That allows us to define values at all vertices of a mapped arborescence and avoid case distinctions.

Given a mapped arborescence (A, κ) in H. Denote by A_v the subarborescence of A, rooted in $v \in V(A)$. If I is the leaveset of A (then $\kappa(I) = T$) denote by I_v the leaves of A_v and by $T_{A,v} := \kappa(I_v)$ the sinks connected by A_v.

For slews at the sinks $(s_t)_{t \in T}$ and a function $\mathrm{obj} : E(H) \times 2^T \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \Omega$, we recursively define objective values $\omega_A : V(A) \to \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$ for the vertices $v \in V(A)$:

$$\omega_A(v)_o := \begin{cases} 0 & \text{if } \kappa(v) \in T \\ \sum\limits_{(v,w) \in \delta^+(v)} \omega_A(w)_o + \mathrm{obj}(\kappa((v,w)), T_{A,w}, \omega_A(w)_c, \omega_A(w)_s) & \text{otherwise,} \end{cases}$$

$$\omega_A(v)_c := \begin{cases} 0 & \text{if } \kappa(v) \in T \\ \sum\limits_{(v,w) \in \delta^+(v)} cap(\kappa((v,w)), \omega(w)_c) & \text{otherwise,} \end{cases}$$

$$\omega_A(v)_s := \begin{cases} s_t & \text{if } \kappa(v) = t \in T \\ \min\limits_{(v,w) \in \delta^+(v)} slew^{-1}(\kappa((v,w)), \omega(w)_c, \omega(w)_s) & \text{otherwise.} \end{cases}$$

Note that this means that for each $v \in V(A)$, $\omega_A(v)_o$ is also given by

$$\omega_A(v)_o = \sum_{(v,w) \in E(A_v)} \mathrm{obj}(\kappa((v,w)), T_{A,w}, \omega_A(w)_c, \omega_A(w)_s).$$

For the TMCMAP, we require obj to be a feasible objective function. Before we define what a feasible objective function is, we introduce a preobjective that does not know about the sink sets. It will be helpful in the construction of new feasible objective functions.

Definition 2.3.2. *A function* $f : E(H) \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \Omega$ *is a* preobjective *for the evaluation operator* χ *if for each edge* $e \in E(H)$*, we have*

- $f(e, \cdot, s)$ *is nondecreasing in each entry and* $s \in \mathbb{R}_{\geq 0}$*.*
- $\chi(f(e, c, s)) = \infty$ *if* $s < \mathit{minslew}(e, c)$*,* $\mathit{slew}^{-1}(e, c, s) > S$ *or* $c > \mathit{caplim}(e)$
- $f(e, c, \cdot)$ *nondecreasing in each entry on the interval* $[\mathit{minslew}(e, c), \infty)$ *for all* $c \in \mathbb{R}_{\geq 0}$*.*

It is called increasing *if there is a constant* $C > 0$*, such that* $\chi(\omega + f(e, c, s)) > \chi(\omega) + C$ *for all constant edges* e *and* $c, s \in \mathbb{R}_{\geq 0}$*,* $\omega \in \Omega$*.*

Using this preobjective, we can now define a feasible objective function.

Definition 2.3.3. *A function* $\mathrm{obj} : E(H) \times 2^T \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \Omega$ *is a* feasible objective function *if for each* $\emptyset \neq I \subseteq T$*,*

- *the function* $\mathrm{obj}(\cdot, I, \cdot, \cdot)$ *is a preobjective and*
- *there is a constant* $C > 0$*, such that* $\chi(\omega + \mathrm{obj}(e, I, c, s)) > \chi(\omega) + C$ *for all constant edges* e *and* $c, s \in \mathbb{R}_{\geq 0}$*,* $\omega \in \Omega(\mathrm{obj}, I)$*,*

where $\Omega(\mathrm{obj}, I)$ *is defined as follows. For* $I \subseteq T$*, we denote by*

$$\mathrm{imsupp}_I(\mathrm{obj}) := \{i \in [d] \mid \mathrm{obj}(e, J, c, s)_i \neq 0 \text{ for any } e \in E(H), J \subseteq I \text{ and } c, s \in \mathbb{R}_{\geq 0}\}$$

the support of the image of obj *for all subsets of* I*. Then*

$$\Omega(\mathrm{obj}, I) := \{\omega \in \Omega \mid \omega_i = 0 \text{ for all } i \in [d] \setminus \mathrm{imsupp}_I(\mathrm{obj})\}$$

is the set of objective values that are only positive on entries in $\mathrm{imsupp}_I(\mathrm{obj})$*.*

Remark. *The second condition is not exactly the same as the condition for increasing preobjectives. Instead, we only require the objective cost to increase on the objective values that could arise as the objective values of a subarborescence connecting the sinks in* $I \subseteq T$*.*

Remark. *There are a lot of different objects that belong to the objective. For clarity, we will always refer to* Ω *as the objective space, to elements* $\omega \in \Omega$ *as objective values, to evaluated objective values* $\chi(\omega)$ *as objective cost and to feasible objective functions* obj *by at least objective function.*

We are looking for a mapped arborescence (A, κ) for the net N and slews at the sinks $s_t \in [0, S_t]$ for each $t \in T$ that minimize $\chi(\omega_A(\kappa^{-1}(r))_o)$. A solution is infeasible if $\chi(\omega_A(v)) = \infty$ for any vertex $v \in V(A)$. Let us summarize this:

Definition 2.3.4 (The Timing-Driven Min-Cost Mapped Arborescence Problem)**.** *We are given a directed graph H, a net $N := \{r\} \cup T$ in H, sink slew limits $(S_t)_{t\in T}$, a global slew limit $S > 0$, slew lower bounds minslew : $E(H) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, capacitance limits caplim : $E(H) \to \mathbb{R}_{\geq 0}$, a slew inverse slew^{-1} : $E(H) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, a capacitance function cap : $E(H) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, an objective space $\Omega := \mathbb{R}^d_{\geq 0}$ for some $d \in \mathbb{N}$, an evaluation operator $\chi : \Omega \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ and a feasible objective function* obj : $E(H) \times 2^T \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \Omega$.

Then we want to find a mapped arborescence (A, κ) for the net N and slews at the sinks $s_t \in [0, S_t]$ for each $t \in T$ that minimize $\chi(\omega_A(\kappa^{-1}(r))_o)$, such that for all vertices $v \in V(A)$, we have $\chi(\omega_A(v)) \neq \infty$.

2.3.2 Problem Construction

The problem formulation is very abstract. However, in order to encode a buffering problem, we need to construct our graph and timing functions in a suitable way. Let a graph G be given that models our routing space. An example is the global routing graph. Let the net $N := \{r\} \cup T$ with $N \subseteq V(G)$ be given with polarities $pol : N \to \{ident, invert\}$. Let L be a repeater library and let an RC-timing model $(delay_G, cap_G, slew_G, caplim_G, S, s_r, (S_t)_{t\in T})$ on (G, L, N) be given.

We will construct an instance of the TMCMAP in such a way that each edge in a solution will correspond to either inserting a wire segment or a repeater.

First we construct the graph H. The first step is to duplicate G. The two copies will be denoted G_{ident} and G_{invert}. Edges within one of these two graphs that belong to an edge in G correspond to the wiring segment they represent in G. For each vertex v and repeater $b \in L$, we add edges $(v_{ident}, v_{ident})_b$ and $(v_{invert}, v_{invert})_b$ if b is a buffer and edges $(v_{ident}, v_{invert})_b$ and $(v_{invert}, v_{ident})_b$ if b is an inverter. Then we add a vertex $\bar{r}$ with an edge $(\bar{r}, r_{ident})$ and vertices $\bar{t}$ with edges $(t_{pol(t)}, \bar{t})$ for each sink $t \in T$. An example for this construction is shown in Figure 2.2.

Now each mapped Steiner arborescence for $\bar{r}$ and $\bar{T}$ in H can be transformed into a unique buffered route, by inserting a repeater, whenever a repeater edge is used and replacing wiring edges in H by their corresponding edges in G. Vice versa, we cannot create a unique Steiner arborescence from a buffered route. However, the possible Steiner arborescences differ only in the order, in which multiple branches at the same node are merged. This poses no problem, as the order of the sum at a node does not influence the objective value.

Remark. *If G is a layered model, we can connect start and end-layer of a repeater to improve the accuracy.*

Then we can use the timing functions of our RC-timing model to define timing functions and an increasing preobjective for the TMCMAP that, given an edge $e \in E(H)$, compute the new values at the tail of that edge. For all edges $e \in E(H)$, and $c, s \in \mathbb{R}_{\geq 0}$, we define the timing functions as follows:

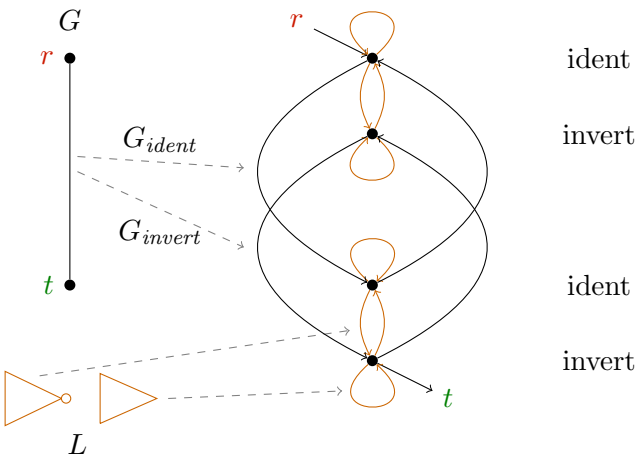


Figure 2.2: The construction of H on the right from G on the left and a simple library L consisting of one repeater and one inverter. (Some dashed edges to the repeater edges are left out for simplicity)

We start with the slew lower bound. We set $minslew(e, c) := a$ for the minimum $a \geq 0$ such that a value σ exists with $slew_G(e, c, \sigma) = a$. Then we set $delay(e, c, s) := \infty$ and $slew^{-1}(e, c, s) := s$ if $s < minslew(e, c)$.

Let now $s \geq minslew(e, c)$. If e belongs to a repeater or the source $b \in L \cup \{r\}$, let σ be the slew, such that $slew_G(b, c, \sigma) = s$. We set

$$\begin{aligned} cap(e, c) &:= cap_G(b) \\ slew^{-1}(e, c, s) &:= \sigma \\ delay(e, c, s) &:= \begin{cases} delay_G(b, c, \sigma), & \text{if } \sigma \leq S \\ \infty & \text{otherwise} \end{cases} \end{aligned}$$

and if e belongs to a wire edge $\hat{e} \in E(G)$, let σ be the slew, such that $slew^{-1}(\hat{e}, c, \sigma) = s$. We set

$$\begin{aligned} cap(e, c) &:= c + cap_G(\hat{e}) \\ slew^{-1}(e, c, s) &:= \sigma \\ delay(e, c, s) &:= \begin{cases} delay_G(\hat{e}, c, \sigma), & \text{if } \sigma \leq S \\ \infty & \text{otherwise} \end{cases} \end{aligned}$$

If e belongs to a sink $t \in T$, we set

$$\begin{aligned} cap(e, c) &:= c + cap_G(t) \\ slew^{-1}(e, c, s) &:= s \\ delay(e, c, s) &:= 0 \end{aligned}$$

And if e belongs to the root, we set $cap(e, c) := c$ and $slew^{-1}(e, c, s) := s_r(c)$. Then we check if $s_r(c) \leq s$. If this is the case, we set $delay(e, c, s) := delay_G(r, c, s_r)$ and otherwise $delay(e, c, s) := \infty$.

Now only the capacitance limit is missing. We set $caplim_{max} := \max_{b \in L \cup \{r\}} caplim_G(b)$ and for all $e \in E(H)$, we set $caplim(e) := caplim_{max}$ if e is a wiring edge and $caplim(e) := caplim_G(b)$ if e is an edge for $b \in L \cup \{r\}$.

Our algorithm will only be guaranteed to return if there is a solution to our instance or if we give it an upper bound on the objective cost. We can always make the instance feasible or decide a-priori that it is infeasible. First, we check that there is a path from the source to each sink. If not, there is no buffered route that connects our net and the instance is infeasible. Otherwise, we compute a tree in H that connects the source to each sink, for example by combining the previously found paths. Then we compute all timing values for the corresponding buffered route by using our RC-timing model. We increase the limits *caplim* and $S, (S_t)_{t \in T}$ to the values assumed in this buffered route and store the old limits as $caplim_{old}$ and $\hat{S}, (\hat{S}_t)_{t \in T}$. Finally, we add a penalty as an additional objective: $\text{penalty}(e, c, s) := \max\{0, c - caplim_{old}(e)\} + \max\{0, s - \hat{S}\}$ for edges $e \in E(H)$ belonging to wires or repeaters and $\text{penalty}(e_t, c, s) := \max\{0, s - \hat{S}_t\}$ for edges $e_t \in E(H)$ belonging to sinks $t \in T$.

This way, the resulting instance will be feasible and we can derive an upper bound on its value. Of course, we can use other initial solutions like a previously present buffered route or one computed by another algorithm as the initial solution that we base our changes on.

2.3.3 Evaluation and Objective Functions

We have prepared timing functions for the TMCMAP. For the objective, we need a feasible objective function and evaluation operator. We will first prove for some general operations that they can be used to construct evaluation operators and feasible objective functions. Then we show some examples how we can use that to build up specific formulations.

Evaluation Operators

As a start, we make the following observation.

Observation. *For $\Omega := \mathbb{R}_{\geq 0}$, the identity function is an evaluation operator.*

Now, we will prove that we can join multiple arbitrary evaluation functions on projections of Ω to some entries. First, we show, that we can join them by a (weighted) maximum operator.

Proposition 2.3.5. *Let $q \in \mathbb{N}$ and $d_i \in \mathbb{N}$ for $i \in [q]$. Let $\Omega := \Omega_1 \times \ldots \times \Omega_q$ with $\Omega_i = \mathbb{R}_{\geq 0}^{d_i}$ and an evaluation operator $\chi_i : \Omega_i \to \mathbb{R}_{\geq 0}$ for each $i \in [q]$. Denote by p_i the projection of Ω onto Ω_i. Then $\chi : \Omega \to \mathbb{R}_{\geq 0}$ given by*

$$\chi(\omega) := \max_{i=1,\ldots,q} \lambda_i \, \chi_i(p_i(\omega)),$$

for scaling factors $\lambda_i > 0$ for $i \in [q]$, is an evaluation operator.

Proof. Let $\omega, \nu \in \Omega$. First, we show subadditivity. We know that for each $i \in [q]$, we have

$$\chi_i(p_i(\omega + \nu)) = \chi_i(p_i(\omega) + p_i(\nu))) \leq \chi_i(p_i(\omega)) + \chi_i(p_i(\nu)).$$

Using this and the fact that $\lambda_i > 0$ for $i \in [q]$, we see that

$$\chi(\omega + \nu) = \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega + \nu)) \leq \max_{i=1,\dots,q} \lambda_i \left[\chi_i(p_i(\omega)) + \chi_i(p_i(\nu))\right].$$

By the subadditivity of the maximum operator, we conclude

$$\chi(\omega + \nu) \leq \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega)) + \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\nu)) = \chi(\omega) + \chi(\nu).$$

Similarly, for $\alpha > 0$, we see that

$$\chi(\alpha\omega) = \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\alpha\omega)) = \max_{i=1,\dots,q} \lambda_i \, \chi_i(\alpha p_i(\omega)) \leq \max_{i=1,\dots,q} \alpha\lambda_i \, \chi_i(p_i(\omega)) = \alpha \, \chi(\omega).$$

To show that it is nondecreasing, also assume that $\omega \leq \nu$. For each $i \in [q]$, we have $\chi_i(p_i(\omega)) \leq \chi_i(p_i(\nu))$. So each single element is increasing. Hence,

$$\chi(\omega) = \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega)) \leq \max_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\nu)) = \chi(\nu).$$

□

The same holds for weighted sums.

Proposition 2.3.6. *Let $q \in \mathbb{N}$ and $d_i \in \mathbb{N}$ for $i \in [q]$. Let $\Omega := \Omega_1 \times \ldots \times \Omega_q$ with $\Omega_i = \mathbb{R}^{d_i}_{\geq 0}$ and an evaluation operator $\chi_i : \Omega_i \to \mathbb{R}_{\geq 0}$ for each $i \in [q]$. Denote by p_i the projection of Ω onto Ω_i. Then $\chi : \Omega \to \mathbb{R}_{\geq 0}$ given by*

$$\chi(\omega) := \sum_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega)),$$

for scaling factors $\lambda_i > 0$ for $i \in [q]$, is an evaluation operator.

Proof. We proceed analogously to the proof of Proposition 2.3.5. Let $\omega, \nu \in \Omega$, $\alpha > 0$. By our assumption each χ_i is subadditive for $i \in [q]$. Thus,

$$\chi(\omega + \nu) = \sum_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega + \nu)) \leq \sum_{i=1,\dots,q} \lambda_i \left[\chi_i(p_i(\omega)) + \chi_i(p_i(\nu))\right] = \chi(\omega) + \chi(\nu)$$

and

$$\chi(\alpha\omega) = \sum_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\alpha\omega)) = \sum_{i=1,\dots,q} \lambda_i \, \chi_i(\alpha p_i(\omega)) \leq \sum_{i=1,\dots,q} \alpha\lambda_i \, \chi_i(p_i(\omega)) = \alpha \, \chi(\omega).$$

Now assume that $\omega \leq \nu$. Then

$$\chi(\omega) = \sum_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\omega)) \leq \sum_{i=1,\dots,q} \lambda_i \, \chi_i(p_i(\nu)) = \chi(\nu).$$

□

These propositions allow us to successively build up evaluation operators. With matching objective functions, we can model all the objectives we have mentioned.

Objective Functions

Similar to the evaluation operator, a preobjective can be used as a feasible objective function for an identity evaluation operator:

Observation. *Let f be an increasing preobjective for an evaluation operator $\chi : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. Then* $\mathrm{obj}(e, I, c, s) := f(e, c, s)$ *is a feasible objective function.*

Let us recall that $\mathrm{imsupp}_I(\mathrm{obj})$ denotes the support of the image of obj for all subsets of I (including I), and that $\Omega(\mathrm{obj}, I) \subseteq \Omega$ denotes the set of objective values that are nonzero only on $\mathrm{imsupp}_I(\mathrm{obj})$ (for Ω objective space, obj objective function and $I \subseteq T$).

We want to prove the analogue to Propositions 2.3.5 and 2.3.6 for feasible objective functions:

Proposition 2.3.7. *Let Ω_1, Ω_2 with evaluation operators χ_1 and χ_2, as well as feasible objective functions* obj_1 *and* obj_2*. Then* $\mathrm{obj} := (\mathrm{obj}_1, \mathrm{obj}_2)$ *is a feasible objective function for the evaluation operators $\chi_+, \chi_{\max} : \Omega_1, \Omega_2 \to \mathbb{R}_{\geq 0}$, where we define*

$$\chi_+((\omega_1, \omega_2)) := \lambda_1 \chi_1(\omega_1) + \lambda_2 \chi_2(\omega_2)$$

and

$$\chi_{\max}((\omega_1, \omega_2)) := \max\{\lambda_1 \chi_1(\omega_1), \lambda_2 \chi_2(\omega_2)\},$$

for any $\lambda_1, \lambda_2 > 0$.

Proof. We check Definition 2.3.3. Let $\emptyset \neq I \subset T$. We need to show that $\mathrm{obj}(\cdot, I, \cdot, \cdot)$ is a preobjective. Both entrywise conditions clearly still hold. Furthermore, if one of the entries has value ∞, then our new evaluation function has value ∞.

Now we check the second condition that is required for feasible objective functions. Let $\omega_1 \in \Omega_1(\mathrm{obj}_1, I)$, $\omega_2 \in \Omega_2(\mathrm{obj}_2, I)$, C_1, C_2 be the constants from the second condition for obj_1 and obj_2 and $e \in E(H)$ a constant edge, $c, s \in \mathbb{R}_{\geq 0}$. Then

$$\begin{aligned}\chi_+((\omega_1, \omega_2) + \mathrm{obj}(e, I, c, s)) &= \lambda_1 \chi_1(\omega_1 + \mathrm{obj}_1(e, I, c, s)) + \lambda_2 \chi_2(\omega_2 + \mathrm{obj}_2(e, I, c, s)) \\ &> \lambda_1(C_1 + \chi_1(\omega_1)) + \lambda_2(C_2 + \chi_2(\omega_2)) = \lambda_1 C_1 + \lambda_2 C_2 + \chi_+((\omega_1, \omega_2))\end{aligned}$$

and

$$\begin{aligned}&\chi_{max}((\omega_1, \omega_2) + \mathrm{obj}(e, I, c, s)) \\ =& \max\{\lambda_1 \chi_1(\omega_1 + \mathrm{obj}_1(e, I, c, s)), \lambda_2 \chi_2(\omega_2 + \mathrm{obj}_2(e, I, c, s))\} \\ >& \max\{\lambda_1(C_1 + \chi_1(\omega_1)), \lambda_2(C_2 + \chi_2(\omega_2))\} \\ \geq& \min\{\lambda_1 C_1, \lambda_2 C_2\} + \chi_{max}((\omega_1, \omega_2)).\end{aligned}$$

This shows that the second condition holds as well. □

Since timing is one of our main goals, let us note here that the *delay*-function from Section 2.3.2 is an increasing preobjective, when we use the identity function as evaluation operator. The most important objectives include the delay on each source-sink-path, either as a weighted sum, or as a maximum. These objectives are defined by their branch based structure. We capture this in the definition of branch based objectives:

Definition 2.3.8. *A* branch based objective*, is an objective space of the form* $\Omega := \mathbb{R}^T_{\geq 0}$ *together with an objective function* $\mathrm{obj} : E(H) \times 2^T \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \Omega$ *of the form*

$$\mathrm{obj}(e, I, c, s)_t := \begin{cases} f(e, c, s) & \text{if } t \in I \\ 0 & \text{otherwise,} \end{cases}$$

where $f : E(H) \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \mathbb{R}_{\geq 0}$ *is an increasing preobjective for the identity evaluation operator.*

Indeed, with both a weighted sum or weighted maximum, a branch based objective leads to a feasible objective function.

Proposition 2.3.9. *Let* Ω *and* obj *be a branch based objective. Then it is a feasible objective function for both* $\chi_+(\omega) := \sum_{t\in T} \lambda_t \omega_t$ *and* $\chi_{\max} := \max_{t\in T} \lambda_t \omega_t$ *for any* $\lambda_t > 0$ *for* $t \in T$*.*

Proof. As before, let $\emptyset \neq I \subset T$. Clearly $\mathrm{obj}(\cdot, I, \cdot, \cdot)$ is a preobjective, because it is a preobjective in each entry and both sum and maximum are ∞ if one entry is ∞.

For the second condition, let f be the increasing preobjective of obj, and C the constant by which it increases. Furthermore, let $e \in E(H)$ be constant and $c, s \in \mathbb{R}_{\geq 0}$. Let $\omega \in \Omega(\mathrm{obj}, I)$. Then

$$\begin{aligned} \chi_+(\omega + \mathrm{obj}(e, I, c, s)) &= \sum_{t\in T} \lambda_t \omega_t + \sum_{t\in I} \lambda_t f(e, c, s) \\ &> \sum_{t\in T} \lambda_t \omega_t + \sum_{t\in I} \lambda_t C = \chi_+(\omega) + \sum_{t\in I} \lambda_t C. \end{aligned}$$

For the maximum note that $\max_{t\in T\setminus I} \lambda_t \omega_t = 0$, because $\mathrm{imsupp}_I(\mathrm{obj}) = I$ and so $\omega_t = 0$ for $t \in T \setminus I$. Then

$$\begin{aligned} \chi_{\max}(\omega + \mathrm{obj}(e, I, c, s)) &= \max\{\max_{t\in T\setminus I} \lambda_t \omega_t, \max_{t\in I} \lambda_t(\omega_t + f(e, c, s))\} \\ &> \max\{\max_{t\in T\setminus I} \lambda_t \omega_t, \max_{t\in I} \lambda_t(\omega_t + C)\} \geq \chi_{\max}(\omega) + \min_{t\in I} \lambda_t C. \end{aligned}$$

This finishes the proof. □

Now we have everything we need to build up the objectives that we listed in Section 2.2. So let us give two examples of objective functions. The first one minimizes the maximum delay to a sink:

Example 1. *Let delay as in Section 2.3.2. We set* $\Omega := \mathbb{R}^T_{\geq 0}$ *and store in each entry the delay to the respective sink. We set as objective function* $\mathrm{obj}(e, I, c, s) := d \in \Omega$*, where*

$$d_t := \begin{cases} \mathit{delay}(e, c, s), & \text{if } t \in I, \\ 0 & \text{otherwise.} \end{cases}$$

In order to evaluate to the maximum, we set $\chi(\omega) := \max_{t\in T} \omega_t$*, which is an evaluation function by Proposition 2.3.5. Together this is a feasible objective by Proposition 2.3.9.*

The maximum of delays can be used to optimize the worst slack as well. Suppose we have required arrival times $rat(t, s)$ for each sink $t \in T$ and input slew $s \in [0, S_t]$. Then we compute a constant $M \in \mathbb{R}_{\geq 0}$ such that $rat(\cdot, \cdot) \geq M$. The delay through the sink edges e_t from Subsection 2.3.2 then changes to $delay(e_t, \cdot, s) := M - rat(t, s)$.

The second example minimizes the weighted sum of delays and power:

Example 2. *Let delay as in Section 2.3.2. Let additionally a power function power* : $E(H) \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ *be given that produces a power consumption value from an edge and is nondecreasing in the capacitance and has a static component on constant edges (repeaters). Let a power weight* $\gamma > 0$ *be given. We set* $\Omega_P := \mathbb{R}_{\geq 0}$ *and* $\mathrm{obj}_P(e, I, c, s) := power(e, c)$.

Furthermore, we set $\Omega_D := \mathbb{R}_{\geq 0}^T$ *with* obj_D *as in Example 1 and assume, we have weights* $\lambda_t > 0$ *for* $t \in T$. *We can use Proposition 2.3.6 to construct an evaluation operator* $\chi(\omega) := \sum_{t \in T} \lambda_t \omega_t + \gamma \omega_P$. *By Propositions 2.3.9 and 2.3.7, we know that* $(\mathrm{obj}_D, \mathrm{obj}_P)$ *is a feasible objective function.*

Similarly, we can add placement costs, netlength costs or routing costs. Even though the latter two are not necessarily preobjectives by themselves, everything works out if they are combined with other objectives.

Tree monotonicity

Before we leave the problem-specific objective and evaluation functions and treat them as abstract objects, we need to examine one very important property that some, but not all of them have. If we take a mapped arborescence and replace a subarborescence by a different one with a smaller objective cost, then this smaller cost propagates through the rest of the original arborescence. A picture illustrating this idea is shown in Figure 2.3.

Definition 2.3.10. *A feasible objective function* $\mathrm{obj} : E(H) \times 2^T \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \Omega$ *for an evaluation operator* χ *has the* tree-monotonicity property, *if the following holds: Let* $v \in V(H)$ *and* $\emptyset \neq J \subseteq T$. *Let* (A, κ) *be a mapped arborescence for* $\{x\} \cup J$, $y \in V(A)$ *and* $D \subseteq \delta_A^+(y)$. *Let* $I := \bigcup_{(y,w) \in D} T_{A,w} \subseteq J$ *and* (B, ϕ) *be a mapped arborescence on* $\{\kappa(y)\} \cup I$. *Denote by* $A_{y,D}$ *the subarborescence of* A *that arises from* A_y *by removing the branches belonging to the edges in* $\delta_A^+(y) \setminus D$ *and by* $\omega_{A_{y,D}} := \sum_{e=(y,w) \in D} \omega_A(w)_o + \mathrm{obj}(e, \omega_A(w)_c, \omega_A(w)_s)$. *If* $\chi(\omega_B(y)) \leq \chi(\omega_{A_{y,D}})$, *then*

$$\chi\left(\omega_B(y) + \sum_{(v,w) \in E(A) \setminus E(A_{y,D})} \mathrm{obj}(\kappa((v,w)), T_w, \omega_A(w)_c, \omega_A(w)_s)\right)$$
$$\leq \chi\left(\omega_{A_{y,D}} + \sum_{(v,w) \in E(A) \setminus E(A_{y,D})} \mathrm{obj}(\kappa((v,w)), T_w, \omega_A(w)_c, \omega_A(w)_s)\right) = \chi(\omega(x))$$

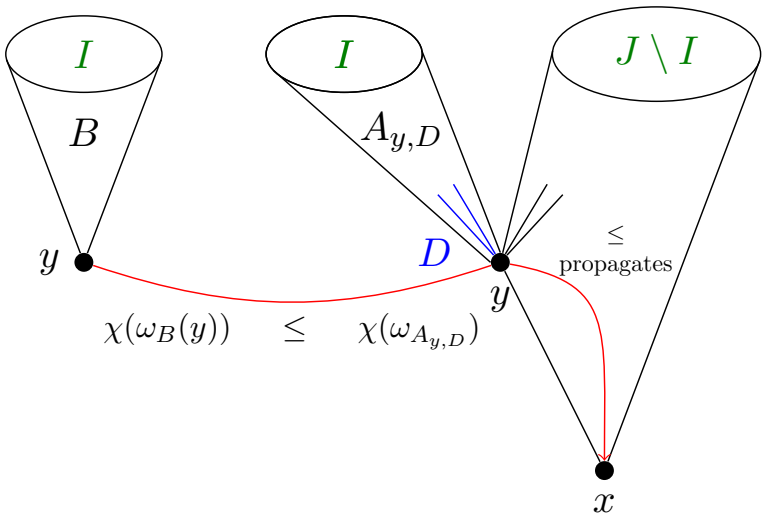


Figure 2.3: An illustration of the tree-monotonicity property (Definition 2.3.10). If the value at the vertex y in the arborescence B is lower than the value of the subarborescence $A_{y,D}$ of A at y, then this property "propagates" through A.

This property will prove to be useful in our algorithm. There we will split up the objective into tree-monotone parts. Then we will use the tree-monotonicity of these parts to define a dominance relation. For practical applications, we need to know which objectives have the tree-monotonicity property. First we show that branch based objectives with our previously used evaluation functions have the tree-monotonicity property. Note that the conditions in Lemma 2.3.11 are the two relevant properties of maximum and sum that we need. It is helpful to imagine $\chi(\omega)$ as $\max_{t\in T} \lambda_t\omega_t$, χ_I as $\max_{t\in I} \lambda_t\omega_t$ and $\mathcal{M}_I$ as max.

Lemma 2.3.11. *Let Ω, obj be a branch based objective and χ an evaluation operator that additionally fulfills the following. For each $I \subseteq T$ there are evaluation operators $\chi_I : \mathbb{R}^I_{\geq 0} \to \mathbb{R}_{\geq 0}$, $\chi_{\bar{I}} : \mathbb{R}^{T\setminus I}_{\geq 0} \to \mathbb{R}_{\geq 0}$ and $\mathcal{M}_I : \mathbb{R}^2_{\geq 0} \to \mathbb{R}_{\geq 0}$, such that $\mathcal{M}_I(\chi_I, \chi_{\bar{I}}) = \chi$, $\mathcal{M}_I(x, 0) = x$ for each $x \in \mathbb{R}_{\geq 0}$ and $\chi_I(\omega_I + C \cdot \mathbf{1}_I) = \chi_I(\omega_I) + \chi_I(C \cdot \mathbf{1}_I)$. Then obj and χ have the tree-monotonicity property.*

Proof. Let (A, κ) and (B, κ), $x \in E(H), y \in V(A), D \subseteq \delta^+_A(y)$ and I as in definition 2.3.10. To shorten the notation, set

$$\omega_{A\setminus A_{y,D}} := \sum_{(v,w)\in E(A)\setminus E(A_{y,D})} \mathrm{obj}(\kappa((v,w)), T_w, \omega_A(w)_c, \omega_A(w)_s).$$

Since A is an arborescence, we have for each $v \in (V(A) \setminus V(A_{y,D})) \cup \{y\}$, that either $I \subseteq T_{A,v}$ or $I \cap T_v = \emptyset$. Hence, $(\omega_{A\setminus A_{y,D}})_t = (\omega_{A\setminus A_{y,D}})_u$ for all $t, u \in I$. So we set $\omega_{A,I} := (\omega_{A\setminus A_{y,D}})_t$ for any $t \in I$. Furthermore, $(\omega_{A_{y,D}})_t = (\omega_B(y))_t = 0$ for all $t \in T \setminus I$. Then we have

$$\begin{aligned}\chi_I((\omega_B(y))_I) = \mathcal{M}_I(\chi_I(\omega_B(y)_I), 0) &= \chi(\omega_B(y)) \\ &\leq \chi(\omega_{A_{y,D}}) = \mathcal{M}_I(\chi_I((\omega_{A_{y,D}})_I), 0) = \chi_I((\omega_{A_{y,D}})_I)\end{aligned}$$

and hence

$$\chi_I(\omega_B(y)_I + \omega_{A,I} \cdot \mathbf{1}_I) = \chi_I(\omega_B(y)_I) + \chi_I(\omega_{A,I} \cdot \mathbf{1}_I)$$
$$\leq \chi_I((\omega_{A_{y,D}})_I) + \chi_I(\omega_{A,I} \cdot \mathbf{1}_I) = \chi_I((\omega_{A_{y,D}})_I + \omega_{A,I} \cdot \mathbf{1}_I).$$

We conclude

$$\chi(\omega_B(y) + \omega_{A\setminus A_{y,D}}) = \mathcal{M}_I\left(\chi_I(\omega_B(y)_I + \omega_{A,I} \cdot \mathbf{1}_I)\ ,\ \chi_{\bar{I}}((\omega_{A\setminus A_{y,D}})_{\bar{I}})\right)$$
$$\leq \mathcal{M}_I\left(\chi_I((\omega_{A_{y,D}})_I + \omega_{A,I} \cdot \mathbf{1}_I)\ ,\ \chi_{\bar{I}}((\omega_{A\setminus A_{y,D}})_{\bar{I}})\right) = \chi(\omega_A(x)).$$

□

Since both, weighted maximum and sum function, have the required properties, we can conclude, that together with branch based objectives, they have the tree-monotonicity property.

Corollary 2.3.12. *A branch based objective together with a maximum or weighted sum as evaluation operator has the tree-monotonicity property.*

If the evaluation consists only of a weighted sum, then any feasible objective function has the tree-monotonicity property.

Proposition 2.3.13. *Let $q \in \mathbb{N}$ and $\Omega := \mathbb{R}^q_{\geq 0} \times \mathbb{R}^T_{\geq 0}$ with a feasible objective function* $\mathrm{obj} : E(H) \times 2^T \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \Omega$. *Let $\chi : \Omega \to \mathbb{R}_{\geq 0}$ be a weighted sum for scaling factors $\lambda_i > 0$ for $i \in [q]$, as in Proposition 2.3.6. Then they have the tree-monotonicity property.*

Proof. Let (A, κ) and (B, κ), $x \in E(H), y \in V(A), D \subseteq \delta^+_A(y)$ and I as in definition 2.3.10. Set $\omega_{A\setminus A_{y,D}}$ as in the proof of Lemma 2.3.11. In this case, we know that

$$\chi(\omega_B(y) + \omega_{A\setminus A_{y,D}}) = \chi(\omega_B(y)) + \chi(\omega_{A\setminus A_{y,D}}) \leq \chi(\omega_{A_{y,D}}) + \chi(\omega_{A\setminus A_{y,D}}) = \chi(\omega_A(x)).$$

□

Unfortunately, not all objective/evaluation combinations have the tree-monotonicity property. Consider the example shown in Figure 2.4. Our evaluation operator computes the maximum of the delays to the sinks t_1 and t_2 and adds to this value the penalty. Then the subsolutions at y have values 2<3, but the values at x are 5>4.

In cases like this, it is helpful to instead split the objective into its tree-monotone components:

Definition 2.3.14. *Let $q \in \mathbb{N}$ and $\Omega := \mathbb{R}^q_{\geq 0}$ be an objective space with* obj *a feasible objective function for an evaluation operator χ. A* tree monotone objective decomposition *of these three is a decomposition $\Omega_1 \times \ldots \times \Omega_p := \Omega$ with projections* $\mathrm{proj}_i : \Omega \to \Omega_i$, $(\mathrm{obj}_1, \ldots, \mathrm{obj}_p) := \mathrm{obj}$ *with evaluation operators $\chi_1, \ldots, \chi_p$ and $\mathcal{M} : \mathbb{R}^p_{\geq 0} \to \mathbb{R}_{\geq 0}$ for $p \leq q$, such that for all $i \in [p]$ we have:*

- obj_i *is the projection of* obj *onto the subspace* Ω_i $(\mathrm{proj}_i(\mathrm{obj}) = \mathrm{obj}_i(\mathrm{proj}_i))$,

- $\chi_i : \Omega_i \to \mathbb{R}_{\geq 0}$,
- $\Omega_i, \chi_i, \mathrm{obj}_i$ *have the tree-monotonicity property*

and we have that

$$\mathcal{M}(\chi_1(\mathrm{proj}_1(\omega)), \ldots, \chi_p(\mathrm{proj}_p(\omega))) = \chi(\omega)$$

for all $\omega \in \Omega$.

Such a decomposition always exists.

Remark. *For* $\omega, \mathrm{obj}, \chi$ *each individual entry of* obj *together with the identity function is tree-monotone. Thus decomposing* obj *into its entries with identity functions as evaluation operators and setting* $\mathcal{M} = \chi$ *is a tree monotone decomposition.*

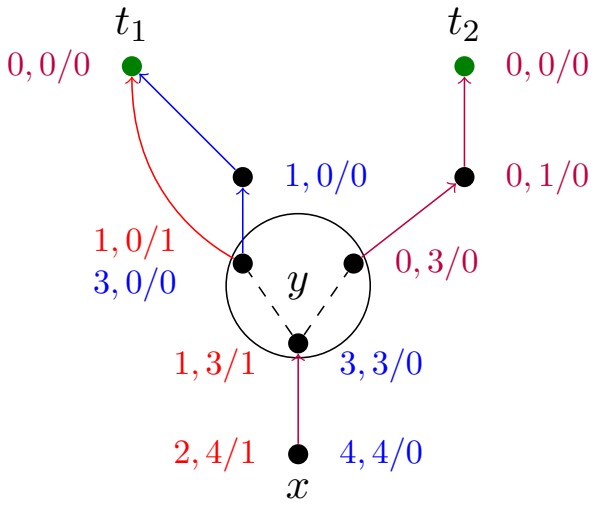


Figure 2.4: Two solutions with their respective objective values, one in red and one in blue. Edges and values that belong to both solutions are shown in purple. The objective is <delay to t_1>,<delay to t_2>/<penalty>.

3 Optimum Algorithm for the Timing-Driven Min-Cost Mapped Arborescence Problem

In this chapter, we will describe the algorithm to compute an optimum solution to the TMCMAP. To this end, let an instance $(H, N = \{r\} \cup T, (S_t)_{t \in T}, S, \Omega, \chi, \text{obj})$ be an instance of TMCMAP (see Chapter 2.3.1, Definition 2.3.4).

Additionally, we are given a tree monotone objective decomposition of our objective and evaluation operator (see Definitions 2.3.14 and 2.3.10).

We will use a bottom up dynamic program that starts with a discrete set of slews at each sink. Though our algorithm is not guaranteed to find the optimum solution, when guessing the sink slews of a single optimum solution, we can prove that it finds an optimum solution if we start with a special constructed finite set of slews at every sink.

The algorithm extends the Dijkstra-Steiner algorithm [HSV17] for Steiner trees (a description can be found in Chapter 1.3.2). Like the Dijkstra-Steiner algorithm, our algorithm builds up a solution in a bottom up order from the sinks to the source. It iteratively extends subarborescences by an edge, or joins two subarborescences. The three main advantages it has over previously published buffering algorithms (without fixed topology) are the following.

First, we make use of a feasible lower bound function which is an extension to the feasible lower bounds in the Dijkstra-Steiner algorithm [HSV17]. This function guides the search and allows us to arrive at an optimum solution faster in practice. While the approach by Cong and Yuan [CY00] uses a similar enumeration order as the Dijkstra-Steiner algorithm, it is missing the concept of lower bounds and an optimality proof.

The approach by Hrkić and Lillis [HL02] uses lower bounds to prune partial solutions that can not lead to optimum solutions, but it has an ineffective enumeration order. For $k = 1, \ldots, |T|$, they first enumerate all partial solutions with sink sets of size k, before they merge partial solutions. We enumerate partial solutions in order of their (effective) objective cost. Thus, a promising partial solution with large sink set can be completed to a solution before other partial solutions that connect smaller sink sets.

And third, we can use the input slew in our timing function and prove an approximation guarantee based on the used slew accuracy and even compute optimum solutions.

To our knowledge, there is no other algorithm that can do this. The algorithms by Rotter [Rot17], Romen [Rom15] and Permin [Per16] only propagate slews as an upper bound that is used to avoid slew violations. The algorithms by Hrkić and Lillis [HL02] and Cong and Yuan [CY00] do not consider slews at all.

In [Roc18] a prototype of our algorithm was presented, however, it had many deficiencies. The most important improvement we make is that we use continuous slews, where the prototype assumed a slew discretization that does not exist in practice. Assuming the slew only takes discrete values always leads to problems. Enabling us to use continuous slews will take most of the work in this chapter.

Furthermore, we are now able to handle much more general objective functions, compared to only delay. Even the delay is more accurate. We can now optimize both the rise and fall transition, which have different delays through inverters.

We start by giving the description of the algorithm in Section 3.1, where we require that we are additionally given slews at the sinks. In Section 3.2, we will then show how to construct slews at the sinks from arbitrary, finite sets of sink slews and sink slews of an optimum solution. We show that with a set of slews that was constructed by our procedure, the algorithm can find an optimum solution. Then, in Section 3.3, we will show how we can generate finite sets of sink slews without knowing an optimum solution and slightly modify the algorithm, such that we can use it to find approximately optimum and optimum solutions. The key idea is that we want the algorithm to produce optimum solutions with slightly overestimated slews. By comparing to the solutions from Section 3.2, we can show that the modified version finds such a solution.

Finally, we will outline two extensions to the algorithm in Section 3.4. They are joint work with Benjamin Ihme [Ihm23]. The first extension incorporates higher order delay models in the algorithm by repeated calibration. The second extension allows to optimally choose repeater positions on edges under a pure RC-delay model. For the second change, I provided the initial idea and recursive formula, as well as key ideas for the proof of correctness.

3.1 Algorithm Description

We start by giving the basic description of the algorithm. This version will also be used in the correctness proof in Section 3.2, but we will change it slightly for the practical version in Section 3.3.

Since we are building up a solution from the sinks, but have only upper bounds for the slews at the sinks, we need to guess our actual slew values there. For now, we assume that we are given a set of slews Σ_t at each sink t. For the correctness we will have to assume that these slew sets contain slews of optimum solutions at the sinks. In a later section, we will show how to get rid of this assumption, by allowing small errors.

During our algorithm, we will store our subarborescences indirectly via labels. We define a label as a tuple $l := (v, e, P, \omega, I)$. Here, $v \in V(H)$ is a vertex, $e \in E(H) \cup \{\square\}$ is either an edge in H or $\square$ if we did not use an edge, P is a set of up to two predecessor labels, $\omega \in \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$ is our objective value with capacitance and slew and $I \subseteq T$ contains

(a) A new label k is created from l by extending the corresponding arborescence by an edge from v to w.

(b) A new label m is created by joining the arborescences corresponding to the labels l and k at the vertex v.

Figure 3.1: Illustration of propagate and merge.

the sinks that are connected by the subarborescence. We will denote the objective value with capacitance and slew of our label by $\omega(l) := \omega$.

We have two main operations that create new labels. The propagate step allows us to create a new subarborescence that extends our current subarborescence by an edge. The merge step allows us to create a new subarborescence by merging two subarborescences with disjoint sink sets at the same vertex. An illustration of these operations can be seen in Figure 3.1. Their pseudocode is shown in Algorithm 3.

The algorithm will always pick the best label that has not been marked as permanent, mark it as permanent and then use the propagate and merge functions to create new labels that extend the current solution.

We can guide the search by a feasible lower bound function that extends feasible potentials for Dijkstra's algorithm, or feasible lower bounds for the Dijkstra-Steiner algorithm [HSV17]. The idea is that it gives us a lower bound on what it would cost to complete a given subarborescence to a solution. For a feasible potential, we need to make sure that the reduced cost does not become negative. We need to fulfill a similar constraint here.

Algorithm 3: The two main operations of the algorithm

1 **Function** `propagate`($l = (w, e_l, P_l, \omega, I)$, $e = (v, w) \in E(G)$, *Non-Permanent labels Q, Labels L*)**:**
2 $\quad k := (v, e, \{l\}, (\omega_o + \mathrm{obj}(e, I, \omega_c, \omega_s), \mathit{cap}(e, \omega_c), \mathit{slew}^{-1}(e, \omega_c, \omega_s)), I)$;
3 $\quad$ **if** $\chi(\omega_o(k)) < \infty$ **then**
4 $\quad\quad$ `insertLabel`(*k, Q, L*);

5 **Function** `merge`($l = (v, e_l, P_l, \omega, I)$, $k = (v, e_k, P_k, \nu, J)$, *Non-Permanent labels Q, Labels L*)**:**
6 $\quad m := (v, \square, \{l, k\}, (\omega_o + \nu_o, \omega_c + \nu_c, \min(\omega_s, \nu_s)), I \cup J)$;
7 $\quad$ **if** $\chi(\omega_o(m)) < \infty$ **then**
8 $\quad\quad$ `insertLabel`(*m, Q, L*);

The algorithm will always pick the best label that has not been marked as permanent, mark it as permanent and then use the propagate and merge functions to create new labels that extend the current solution.

We can guide the search by a feasible lower bound function that extends feasible potentials for Dijkstra's algorithm, or feasible lower bounds for the Dijkstra-Steiner algorithm [HSV17]. The idea is that it gives us a lower bound on what it would cost to complete a given subarborescence to a solution. For a feasible potential, we need to make sure that the reduced cost does not become negative. We need to fulfill a similar constraint here.

For a (tree-monotone) component $i \in [\iota]$ of the objective, let $\mathrm{OPT}_i(w, S_w, c_w, J + (v, \omega))$ denote the minimum objective cost for obj_i, χ_i of a new instance $Inst = (G, w, T')$ of our problem. This instance uses $J \cup \{v\} = T'$ as sinks, w as source, and adds the additional restriction that each solution (A, κ) fulfills $\mathrm{proj}_i(\omega_A(v)_o) = \mathrm{proj}_i(\omega_o)$, $\omega_A(w)_c \leq c_w$ and $\omega_A(w)_s \geq S_w$. Let then

$$\begin{aligned}\mathrm{opt}_i(w, S_w, c_w, J + (v, \omega)) := \\ \{\nu \in \Omega_i \mid \mathrm{proj}_i(\omega_o) + \nu \text{ is the objective value of a solution of } Inst, \\ \text{with } \chi_i(\mathrm{proj}_i(\omega_o) + \nu) = \mathrm{OPT}_i(w, S_w, c_w, J + (v, \omega))\}.\end{aligned}$$

Definition 3.1.1. *A function* $\mathfrak{Lb} : V(G) \times \mathbb{R}_{\geq 0}^{\{c,s\}} \times 2^T \to \Omega$ *is a* feasible lower bound *if* $\mathfrak{Lb}(r, \cdot, \cdot, \cdot) = 0$ *and for each component* $i \in \iota$ *of the objective and* $\nu \in \mathrm{opt}_i(w, S_w, c_w, J \setminus J' + (v, \omega))$*, we have*

$$\chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega_c, \omega_s), J))) \leq \chi_i\left(\mathrm{proj}_i(\mathfrak{Lb}(w, (c_w, S_w), J')) + \nu\right)$$

for all $v, w \in V(G)$, $\emptyset \subsetneq J' \subsetneq J \subseteq T$, $\omega \in \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$,$(c_w, S_w) \in \mathbb{R}_{\geq 0}^{\{c,s\}}$.

Additionally, we require that $\mathrm{proj}_i(\mathfrak{Lb})$ *is consistent with the tree monotonicity property of* χ_i *in the following sense: For* $\omega, \nu \in \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$ *and* $c', s' \in \mathbb{R}_{\geq 0}$ *with* $\chi_i(\mathrm{proj}_i(\omega_o)) \leq \chi_i(\mathrm{proj}_i(\nu_o))$, $\omega_c, \nu_c \leq c'$, $\omega_s, \nu_s \geq s'$ *and both belong to arborescences connecting* $v \in V(H)$ *to* $\emptyset \neq I \subseteq T$ *we have*

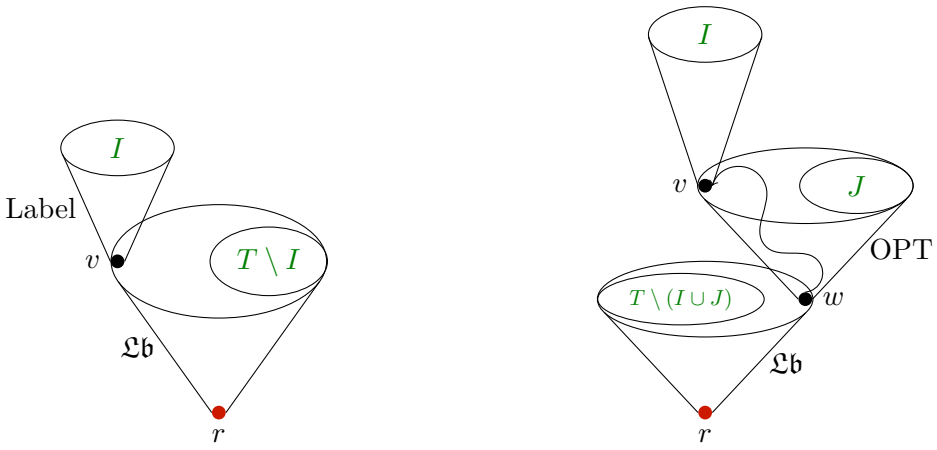
$$\chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (c', s'), T \setminus I + \omega_o)) \leq \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I) + \nu_o)).$$

A picture of the feasibility constraint of feasible lower bounds can be found in Figure 3.2.

Note that $\mathfrak{Lb} \equiv 0$ is a trivial feasible lower bound. Given a feasible lower bound $\mathfrak{Lb}$, we can now define the effective objective of a label $l = (v, e, P, \omega, I)$ as $\omega_{\mathit{eff}}(l) := \chi(\omega(l)_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I))$.

Two essential concepts that decrease the running time of our algorithm are the equivalence and the dominance of labels.

Definition 3.1.2. *Let* $l = (v, e_l, P_l, \omega, I)$ *and* $k = (w, e_k, P_k, \nu, J)$ *be two labels. They are* equivalent *if* $v = w$, $I = J$ *and* $\chi_i(\omega) = \chi_i(\nu)$ *for all components* $i \in [\iota]$ *of the objective. And* l dominates k*, if* $v = w$, $I = J$ *and* $\omega_s \geq \nu_s$ *and* $\omega_c \leq \nu_c$ *and* $\chi_i(\mathrm{proj}_i(\omega_o)) \leq \chi_i(\mathrm{proj}_i(\nu_o))$ *for all components* $i \in [\iota]$ *of the objective and this last inequality is strict for at least one value of* i*.*



(a) $\mathfrak{Lb}$ is a lower bound on the cost of an extension of a label to a complete solution.

(b) In the right-hand side of the feasibility constraint, we replace the lower bound to one point in our search space by a lower bound to another point and an optimum connection between the two points.

Figure 3.2: For a feasible lower bound, we require that the difference between the lower bounds on two different points in our search space does not exceed the cost of an optimum connection between these points. This is an extension to feasible potentials, where we want that the difference between the potentials of two vertices does not exceed the length of a shortest path between them.

We will only insert labels if they are not dominated and there is no equivalent label already present, this is captured in the insert function in algorithm 4.

Remark. *While we could include equivalent labels in the dominance relation, differentiating between both will be helpful in our proofs.*

Algorithm 4: The function for inserting labels.

```
1 Function insertLabel(Label l, Non-Permanent labels Q, Labels L):
2     if there is no label in L that is equivalent to l or dominates l then
3         L := L ∪ {l};
4         Insert l into Q;
5         Remove all non-permanent labels from L that are dominated by l;
```

Now, we have all tools required to describe the algorithm. We start by inserting labels $(t, \square, \emptyset, (0, 0, s), \{t\})$ for each $s \in \Sigma_t$ into the set of labels L and the set of non-permanent labels Q. As long, as Q is not empty, we pick a label $l = (v, e, P, \omega, I)$ from $Q \cap L$ that lexicographically minimizes $(\omega_{\mathit{eff}}(l), \omega_c, -\omega_s)$, remove it from Q and mark it as permanent. If $v = r$ and $I = T$, we are done and recursively reconstruct the arborescence from l. Otherwise, we remove all non-permanent labels from L that are

dominated by l. We propagate l along all incoming edges $\delta^-(v)$ of v in G and merge l with all permanent labels with vertex v and sink set disjoint from I. If Q is empty and we did not find a solution, we report that there is no solution. The pseudocode is written down in algorithm 5.

Algorithm 5: Our algorithm for optimum nonlinear delay Steiner arborescences.

Input : An instance of the TMCMAP, a lower bound function $\mathfrak{Lb}$, and slew sets $(\Sigma_t)_{t\in T}$.

Output: (A, κ) minimizing $\chi(\omega_A(\kappa^-1(s)))$, or the information that there is no solution.

```
1  L := Q := {(t, □, ∅, (0, 0, s), {t}) | t ∈ T, s ∈ Σ_t};
2  while Q ≠ ∅ do
3      Choose l = (v, e, P, ω, I) ∈ Q ∩ L lexicographically minimizing
         (ω_eff(l), ω_c, −ω_s);
4      Remove l from Q and mark it permanent;
5      if v = r and I = T then
6          Recursively construct (A, κ) from l;
7          return (A, κ);
8      Remove all labels except permanent labels from L that are dominated by l;
9      for k = (v, e', P', ω', J) ∈ L permanent, with I and J disjoint do
10         merge(l,k,Q,L);
11     for (w, v) = e ∈ δ⁻_G(v) do
12         propagate(l,e,Q,l);
13 Report that there is no solution.
```

3.2 Correctness Knowing Optimum Slews

In order to prove that the algorithm can find an optimum solution, we need to show that there are slew sets $(\Sigma_t)_{t\in T}$ at the sinks, such that the following holds. The algorithm constructs a label that encodes an optimum solution and the algorithm does not remove all of these labels via the dominance relation. And we need to show that the returned label encodes an optimum solution.

The majority of this section will deal with these first two parts of finding such slew sets. At first glance, one would assume that it is sufficient to start with an optimum solution and the slew values it attains at the sinks. This would be true, if the objective were a weighted sum. However, in general the algorithm can not recover this optimum solution (or one with the same slew values).

To understand this, we take a look at Figure 3.3, where we try to optimize the maximum delay to the sinks t_1 and t_2. We start out with the same label at t_1. It is propagated

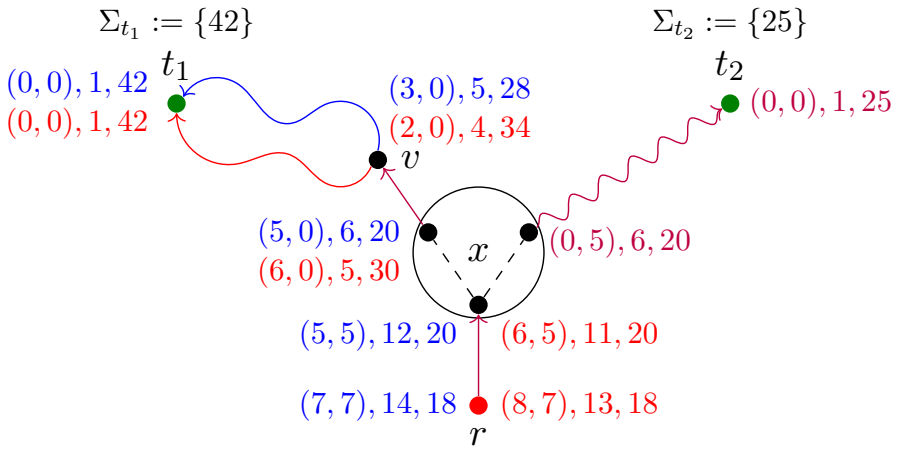


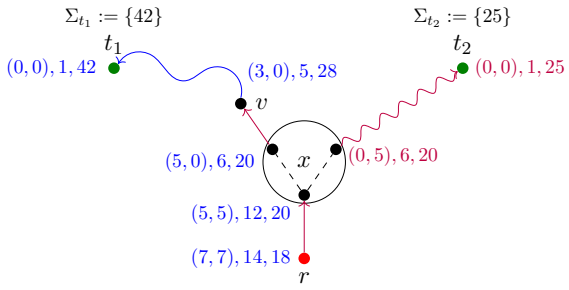
Figure 3.3: An example, where the red label at v dominates the blue label at v, but red leads to an overall worse solution. The objective is the delay to t_1 and t_2 respectively. They are evaluated by a maximum function. The numbers show: (delay to t_1, delay to t_2), capacitance, slew.

once through the blue path and once through the red path. We end up with the red and blue labels at v. Here, the red label dominates the blue label. We should note that the red label has a much higher slew than the blue label. When we now propagate both labels through the same edge (x, v), we are confronted with our main dilemma. A higher slew is both better and worse in some sense. A higher slew is better in the sense, that it is less restricted during backwards propagation, because the difference to slew lower bounds is higher. However, a lower slew leads to less delay. After merging both labels with the purple label that was propagated from t_2 and has the same slew as the blue label, both labels end up with the same slew. We propagate to the root and end up with a value of 8 for the red label and 7 for the optimum blue label. The algorithm would return the red label, because we pruned the blue label that belongs to an optimum solution at v.

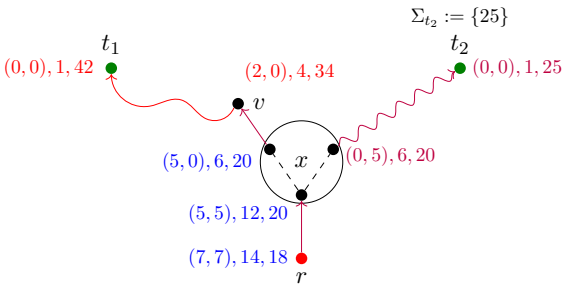
Note that for this to happen, we do need multiple sinks and it does not work for weighted sums. It may seem like that when looking at the example, but in these cases, we could prove that the blue solution was not optimum. This will become clear, when reading the proof of Lemma 3.2.6.

What happened here? While the value of the maximum is optimum, this does not mean that each branch is optimum under this condition. It can happen that a branch of the solution, that does not attain the maximum, would be represented by a dominated label. A dominating label may now have a lower objective value at this point in the solution, but have a much higher slew. This higher slew may lead to a higher objective value at the point, where the branch connects to the remaining solution. This difference can be high enough to increase the objective cost at the source.

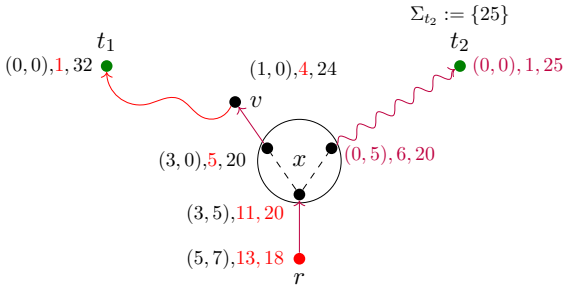
This seems counterintuitive. Why should we use a dominance relation that removes optimum solutions in favour of suboptimum ones? The answer here is that it only



(a) We start with the initial optimum solution in blue from Figure 3.3.



(b) The blue dominated label at v is formally replaced by the dominating red label.



(c) The values of the labels are updated to create new labels.

Figure 3.4: Constructing new optimum solution.

removes the label, but we can still use the dominating label to construct an optimum solution. This new solution will have different slews at the sinks and the labels will have different values. Let us show this on the same example as before.

Take a look at Figure 3.4. We construct a new optimum solution, by starting with the blue optimum solution in Figure 3.4a. As a first step, we formally replace the blue dominated label at v by the dominating red label in Figure 3.4b. By doing this, we change the structure of the associated mapped arborescence. This means that we are replacing the blue v-t_1-path by the red one. Afterwards, we update the values of the labels, by first recomputing the capacitances, then computing new slews from the source to the sinks (we make use of the fact that $slew^{-1}$ is invertible) and finally computing new objective values, using obj. We end up with a new optimum solution that has different sink slews in Figure 3.4c. In particular, these may not be part of our initial sink sets. Here 32 was not in Σ_{t_1}. Note that our new solution has lower delay to t_1 than the old one. This process is formalized in Lemma 3.2.6.

The problem here is that the algorithm can only find this new optimum solution, if the new slews at the sinks are in the initial sink sets. So we will show that we can construct finite sets of slews at the sinks, such that this is always possible. The key idea is to iteratively construct new optimum solutions using dominating labels. Each new solution will improve in some entry, until we find a solution such that none of its entries can be improved.

Unfortunately, we need to know an optimum solution to find this set. In Section 3.3 we will then show how to work around this problem.

Let us start by showing that we can encode each (partial) solution with labels. Recall from Chapter 2.3.1 that a solution consists of a mapped arborescence and a slew value for each sink. Also, we defined the objective values for a mapped arborescence together with a slew value for each sink.

Now, we want to show that if the sink slews of a solution are elements of the sink slew sets $(\Sigma_t)_{t\in T}$ that are given to the algorithm as input, then we can construct a label that represents this solution and has the same objective values. In short, we can represent this solution by labels. Note that this means only that we can construct this label, not that the algorithm can find it, nor that the algorithm does not prune it.

Proposition 3.2.1. *Let A, κ be a mapped arborescence for H, N and $s_t \in [0, S_t]$ for $t \in T$. If $s_t \in \Sigma_t$ for all $t \in T$, then we can construct a label l_v for each vertex $v \in V(A)$, by only using initial labels from the algorithm and propagate and merge operations, such that l_v encodes the subarborescence $A_{v,\kappa}$, rooted at v with objective $\omega(l) = \omega_A(v)$, where $\omega_A(v)$ is the objective value at v for the sink slews $(s_t)_{t\in T}$.*

Proof. We prove this by induction. For each $t \in T$ this certainly holds, as the label $l_t(t, \square, \emptyset, (0, 0, s_t), \{t\})$ is in the initial labels, encodes A_t, κ and $(0, 0, s_t) = \omega_A(t)$.

Now let $v \in V(A)$ and assume that for each outgoing edge $(v, w) \in \delta^+(v)$, there is a label l_w that encodes A_w, κ with $\omega(l_w) = \omega_A(w)$. Then we first create the labels $l_e := (\kappa(v), \kappa(e), \{l_w\}, \omega^e, T_{A,w})$ for each $e = (v, w) \in \delta^+(v)$, where $\omega^e := (\omega(l_w)_o + \mathrm{obj}(\kappa(e), T_{A,w}, \omega(l_w)_c, \omega(l_w)_s), \mathit{cap}(\kappa(e), \omega(l_w)_c), \mathit{slew}^{-1}(\kappa(e), \omega(l_w)_c, \omega(l_w)_s))$. These are exactly the labels resulting from propagating the existing labels l_w along the edges e.

Then we choose an order of the outgoing edges $\{e_1, \ldots, e_k\} := \delta^+(v)$. We set $(v, w_i) := e_i$ and create the merge labels $l_2 := (v, \square, \{l_{e_1}, l_{e_2}\}, (\omega_o^{e_1} + \omega_o^{e_2}, \omega_c^{e_1} + \omega_c^{e_2}, \min\{\omega_s^{e_1}, \omega_s^{e_2}\}, I_2)$, with $I_2 := T_{A,w_1} \cup T_{A,w_2}$ and

$$l_i := (v, \square, \{l_{i-1}, l_{e_i}\}, (\omega(l_{i-1})_o + \omega_o^{e_i}, \omega(l_{i-1})_c + \omega_c^{e_i}, \min\{\omega(l_{i-1})_s, \omega_s^{e_i}\}, I_i)$$

with $I_i := I_{i-1} \cup T_{A,w_i}$ for $i = 3, \ldots, k$. Then $I_k = T_{A,v}$ and $l_k =: l_v$ encodes A_v, κ.

We have

$$\begin{aligned}
\omega(l_v)_o &= \sum_{e=(v,w)\in\delta^+(v)} \omega(l_w)_o + \mathrm{obj}(\kappa(e), T_{A,w}, \omega(l_w)_c, \omega(l_w)_s) && = \omega_A(v)_o,\\
\omega(l_v)_c &= \sum_{e=(v,w)\in\delta^+(v)} \mathit{cap}(\kappa(e), \omega(l_w)_c) && = \omega_A(v)_c \text{ and}\\
\omega(l_v)_s &= \min_{e=(v,w)\in\delta^+(v)} \mathit{slew}^{-1}(\kappa(e), \omega(l_w)_c, \omega(l_w)_s) && = \omega_A(v)_s.
\end{aligned}$$

□

This shows that we can, in theory, explore the whole solution space (generated by the input slews) by repeatedly applying propagate and merge functions. We notice that a label and its predecessors form an arborescence:

Definition 3.2.2. *The* label arborescence *of $l = (v, e, P, \omega, I)$ is the arborescence A_l constructed by starting with l as the root and recursively adding the label arborescences A_p and edges (l, p) to the graph for all $p \in P$. We call a label $a \in V(A_l) \setminus \{l\}$ an* ancestor *of l.*

We will say that a label *exists* if it is either one of the initial labels

$$\{(t, \square, \emptyset, (0, 0, s), \{t\}) \mid t \in T, s \in \Sigma_t\}$$

or it can be created from the initial labels by a finite sequence of propagates and merges.

Before we construct the sink slew sets, let us finish the easy part. If the algorithm creates a label that encodes a complete optimum solution in some iteration, then it also returns an optimum solution. We take a closer look at the order, in which the algorithm explores the labels. We start by proving the first invariant.

Lemma 3.2.3. *During the algorithm, the minimum effective objective of the labels in Q in line 3 never decreases.*

Proof. First, we prove that the lowest effective objective is assumed at an initial label:

Claim (1)**.** *For any $t \in I \subseteq T$, and any label $k = (v, e, P, \omega, I)$ that originates from $l = (t, \square, \emptyset, (0, 0, s), \{t\})$ ($l \in A_k$) with $s \in \Sigma_t$, it holds that*

$$\omega_{\mathit{eff}}(l) \leq \omega_{\mathit{eff}}(k).$$

We start by observing that $\omega_{\mathit{eff}}((t, \square, \emptyset, (0, 0, s), \{t\})) = \chi(\mathfrak{Lb}(t, (0, s), T \setminus \{t\}))$. By the definition of feasible lower bounds, we know for each component $i \in [\iota]$ of the objective and $\nu \in \mathrm{opt}_i(v, \omega_s, \omega_c, I \setminus \{t\} + (t, (0, 0, s)))$ that

$$\chi_i(\mathrm{proj}_i(\mathfrak{Lb}(t, (0, s), T \setminus \{t\}))) \leq \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I)) + \nu).$$

Since k cannot be better than the optimum, we have $\chi_i(\nu) \leq \chi_i(\mathrm{proj}_i(\omega_o))$ and hence

$$\begin{aligned}\chi(\mathfrak{Lb}(t, (0, s_t), T \setminus \{t\})) &= \underset{i \in [\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(t, (0, s_t), T \setminus \{t\}))) \\ &\leq \underset{i \in [\iota]}{\mathcal{M}} \chi_i\left(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I) + \omega_o)\right) = \omega_{\mathit{eff}}(k).\end{aligned}$$

This proves Claim (1).

Second, we prove that the effective objective cannot decrease due to a merge.

Claim (2)**.** *Let the label m be created from $l = (v, e, P, \omega, I)$ by a merge with $k = (v, f, P', \mu, J)$ in line 10. Then $\omega_{\mathit{eff}}(m) \geq \omega_{\mathit{eff}}(l)$.*

For each component $i \in [\iota]$ of the objective, let $\nu \in \mathrm{opt}_i(v, \omega(m)_s, \omega(m)_c, J + (v, \omega))$. For the effective objective, we get

$$\begin{aligned}\omega_{\mathit{eff}}(l) = \chi(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I)) &= \underset{i \in [\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I))) \\ \leq \underset{i \in [\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega(m)_c, \omega(m)_s), T \setminus (I \cup J)) + \nu + \mathrm{proj}_i(\omega_o))).\end{aligned}$$

However, we know that

$$\chi_i(\nu + \mathrm{proj}_i(\omega_o)) = \mathrm{OPT}_i(v, \omega(m)_s, \omega(m)_c, J + (v, \omega)) \leq \chi_i(\mathrm{proj}_i(\omega(m)_o)),$$

because $\chi_i(\mathrm{proj}_i(\omega(m)_o))$ cannot be better than the optimum. By tree monotonicity, we get that

$$\omega_{\mathit{eff}}(l) \leq \underset{i\in[\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(v, (\omega(m)_c, \omega(m)_s), T \setminus (I \cup J)) + \omega(m)_o)) = \omega_{\mathit{eff}}(m).$$

This proves Claim (2).

Third, the effective objective cannot decrease due to a propagate.

Claim (3)**.** *Let the label $p = (w, f, \{l\}, \mu, I)$ be created from $l = (v, e, P, \omega, I)$ by propagating l through $f \in E(H)$ in line* 12*. Then $\omega_{\mathit{eff}}(p) \geq \omega_{\mathit{eff}}(l)$.*

For each component $i \in [\iota]$ of the objective, let $\nu \in \mathrm{opt}_i(w, \mu_s, \mu_c, \emptyset + (v, \omega))$. We have

$$\begin{aligned}\omega_{\mathit{eff}}(l) &= \chi(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I)) = \underset{i\in[\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I))) \\ &\leq \underset{i\in[\iota]}{\mathcal{M}} \chi_i(\mathrm{proj}_i(\mathfrak{Lb}(w, (\mu_c, \mu_s), T \setminus I)) + \nu + \mathrm{proj}_i(\omega_o)). \qquad (3.1)\end{aligned}$$

The optimum value in the last sum is at most the increase of the objective value in the propagate step. This means that $\chi_i(\mathrm{proj}_i(\omega_o) + \nu) = \mathrm{OPT}(w, \mu_s, \mu_c, \emptyset + (v, \omega)) \leq \chi_i(\mathrm{proj}_i(\mu_o))$. By inserting into the above inequality, we again get $\omega_{\mathit{eff}}(l) \leq \omega_{\mathit{eff}}(p)$. This proves Claim (3).

By Claim (1), the minimum effective objective is assumed in the first iteration. Now consider an iteration of the algorithm. When we remove l from Q in line 4, the minimum cannot decrease, because we picked a minimum label from Q in line 3. All labels that are added to Q in this iteration are created either in line 10 by a merge step or by a propagate step in line 12. For the merged labels Claim (2) shows that this cannot decrease the minimum (below $\omega_{\mathit{eff}}(l)$). For the propagate labels Claim (3) shows that this cannot decrease the minimum (below $\omega_{\mathit{eff}}(l)$). So the minimum has not decreased. This proves the first invariant that the minimum effective objective in Q never decreases. □

With this result, we can prove our second invariant:

Lemma 3.2.4. *Let $\mathcal{L}$ be the set of all labels that are added to L and never removed during the algorithm. In each iteration the following holds: Each label $l \in \mathcal{L}$ with*

$$\omega_{\mathit{eff}}(l) < M := \max_{l' \in L \text{ permanent}} \omega_{\mathit{eff}}(l'),$$

is also in L and permanent.

Proof. First, we observe that if a label is in $\mathcal{L}$, then its ancestors are in $\mathcal{L}$ as well. A label is only added to L if it is an initial label or all its parents are in L and permanent. Since the algorithm does not remove permanent labels and initial labels are never dominated, they have to be in $\mathcal{L}$ as well.

In the beginning, all initial labels are in L and Q.

For any initial label $i = (t, \square, \emptyset, (0, 0, s), \{t\})$ and any non-initial label $k = (v, e, P, \omega, I)$ derived from i, we have $\omega_{\mathit{eff}}(i) \leq \omega_{\mathit{eff}}(k)$. By the lexicographic ordering, the first chosen label i minimizes $\omega_{\mathit{eff}}(i)$ among the initial labels (and thus among all labels). Thus, the claim holds after the first iteration.

Now suppose the invariant holds up to iteration n, but it does not hold in iteration $n+1$. This means that after making some label $m = (v, e, P, \omega, I)$ permanent in line 4 of the algorithm, there is label $k \in \mathcal{L}$ that is not permanent and maybe not in L, but $\omega_{\mathit{eff}}(k) < M := \omega_{\mathit{eff}}(m)$.

This implies that there is a label $l \in L$, with $\omega_{\mathit{eff}}(l) < M$ that is not permanent, such that all its predecessors are in L and permanent or it is an initial label: If k does not fulfill this, we take a look at its label arborescence A_k. We know that all ancestors a of k are in $\mathcal{L}$ as well and $M > \omega_{\mathit{eff}}(k) \geq \omega_{\mathit{eff}}(a)$ by Lemma 3.2.3. In particular, we know that the leaves of A_k were in L and cannot be dominated. So we start with $a = k$. As long as a is not an initial label or has a predecessor p that is not permanent, we replace a by p. We either end up at a non-permanent initial label, which fulfills our criteria, or at another label, which is not permanent but all its predecessors are permanent. Let the resulting label be l.

Then merge and propagate have been called on its predecessors and `insertLabel` has been called for l. Since $l \in \mathcal{L}$, l or an equivalent label must have been added to L and Q and was also not removed from L. As it has not been chosen from Q, because it is not permanent, it must hold that

$$\omega_{\mathit{eff}}(l) \geq \min_{l' \in Q} \omega_{\mathit{eff}}(l') = \omega_{\mathit{eff}}(m) > \omega_{\mathit{eff}}(l).$$

This is a contradiction. Thus, the invariant must hold in iteration $n+1$ as well. By induction, the invariant holds in each iteration. □

This invariant means in particular that, when the algorithm returns a solution, it has minimum value among the solutions that it is able to find.

Corollary 3.2.5. *If Algorithm 5 returns a solution and the set $\mathcal{L}$ of labels that are added and never removed contains a label that encodes an optimum solution, then the returned solution is optimum.*

Proof. When the first label at r with complete sink set is made permanent, it must have minimum objective cost among all labels at r with complete sink set by Lemma 3.2.4. As $\mathcal{L}$ contains a label that encodes an optimum solution, this minimum must be the value of an optimum solution. So the returned solution must be optimum. □

So now we know that the algorithm is correct under two assumptions. First, its runtime is finite and second, during the runtime, we do not remove all optimum solutions.

For both these assumptions we will show that they are true because of our choice of the dominance relation. On the one hand, it will guarantee that we can find a finite set of starting slews and only need to consider a finite number of labels before we exceed the value of an optimum solution. On the other hand, it will guarantee that not all optimum solutions are pruned.

This is due to its main property. In essence, it says that if we have a solution to our problem (optimum or not), and a label representing a subarborescence of that solution is dominated, then we can use the dominating label to construct a better solution.

Lemma 3.2.6. *Let (A, κ) be a solution to our problem with initial slew values $S = \{\omega_A(\kappa^{-1}(t))_s | t \in T\}$ and let $\alpha := \omega_A(\kappa^{-1}(r))$. Let l be a label that belongs to a subarborescence of A and is dominated by or equivalent to a label k. Let (B, ϕ) be the solution that arises from (A, κ) by replacing the subarborescence corresponding to l by the subarborescence corresponding to k. Then there are slew values $S' = \{s'_t | t \in T\}$ with an updating procedure that computes S' and only depends on (B, ϕ) and α_s, such that starting with $\omega_B(\phi^{-1}(t))_s := s'_t$ we have for $\beta := \omega_B(\phi^{-1}(r)))$ that*

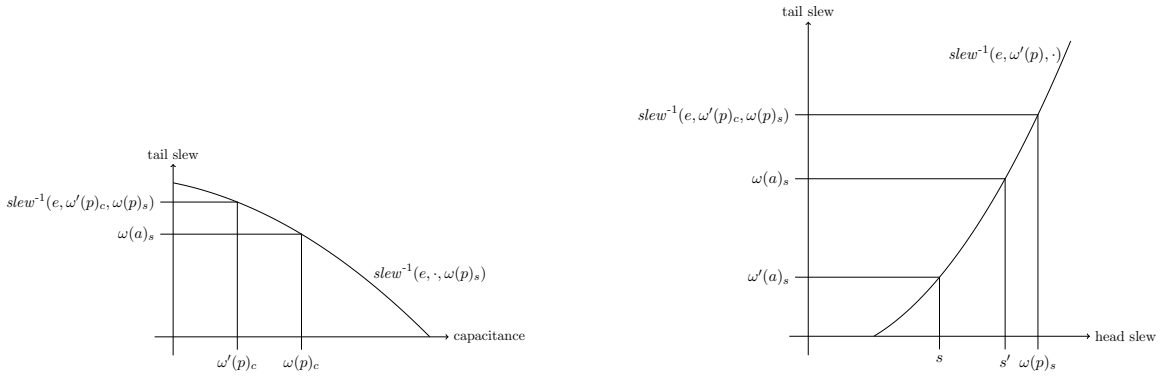
- *$\alpha_s = \beta_s$ and $\chi(\beta_o) \leq \chi(\alpha_o)$,*
- *$\omega_B(v)_s \leq \omega_A(v)_s$ and $\omega_B(v)_c \leq \omega_B(v)_c$ for each vertex $v \in V(A) \cap V(B)$ and*
- *$\chi_i(\text{proj}_i(\omega_B(v)_o)) \leq \chi_i(\text{proj}_i(\omega_A(v)_o))$ for each component $i \in [\iota]$ of the objective and each vertex $v \in V(A) \cap V(B)$.*

Proof. Let m be a label representing (A, κ) with its label arborescence A_m that contains l. Our goal is to construct a new label arborescence by replacing the label arborescence of l by the one belonging to k and then updating only the objective values.

So let A'_m be the arborescence that arises from A_m by replacing the subarborescence rooted at l by the label arborescence of k. A'_m does not necessarily belong to a sequence of propagates and merges yet. For each label a of A'_m, we denote by $\omega'(a)$ the objective value of a after updating all values.

We start by updating all the capacitances. To do this, we traverse A'_m in reverse topological order. For each label $a = (v, e, P, \omega, I)$. If k is not an ancestor of a, then we do nothing, because none of its ancestors have changed and the capacitance value is still correct. In particular, we know that $\omega'(a)_c = \omega'(a)$. For $a = k$, we set $\omega'(k)_c := \omega(k)_c \leq \omega(l)_c$. The final case is that k is an ancestor of a. Assume that for all parents $p \in P$, we already have $\omega'(p)_c \leq \omega(p)_c$. If a is a merge label, let $P = \{p_1, p_2\}$ and set $\omega'(a)_c := \omega'(p_1)_c + \omega'(p_2)_c \leq \omega(a)_c$. Otherwise, a is a propagate label and we denote $P =: \{p\}$. Set $\omega'(p)_c := cap(e, \omega'(p)_c) \leq cap(e, \omega(p)_c) = \omega(a)_c$, because $cap(e, \cdot)$ is either constant or nondecreasing. By induction, we now have that $\omega'(a)_c \leq \omega(a)_c$ for each label a of A'_m.

Next, we update the slews. We start by setting $\omega'(m)_s := \beta_s := \alpha_s$. Then we traverse A'_m in topological order. Let $a = (v, e, P, \omega, I)$ be the current label and assume that $\omega'(a)_s \leq \omega(a)_s$. If a is created by a merge step, we denote the predecessors by $P =: \{p_1, p_2\}$ and set $\omega'(p_1) := \omega'(p_2) := \omega'(a)$. Since $\omega(a) = \min\{\omega(p_1)_s, \omega(p_2)_s\}$,



(a) The $slew^{-1}$ function is nonincreasing in the capacitance for fixed slew $\omega(p)_s$ at the head of the edge e.

(b) The slew function is strictly increasing (on $[minslew(e, \omega(p')_c), \infty)$) in its slew entry for fixed downstream capacitance $\omega(p')_c$.

Figure 3.5: Using the properties of the slew function, we first establish the order of the slew values at the tail in (a). Then we use them to find the order of the head slew values in (b).

we know that also $\omega'(p) \geq \omega(p)_s$ for $p \in P$. If the label was created by a propagate step, let $P := \{p\}$. We note that $slew^{-1}(e, \omega'(p)_c, \cdot)$ is invertible and surjective onto $\mathbb{R}_{\geq 0}$ on the interval $[minslew(e, \omega'(p)_c), \infty)$. So we let $s \in \mathbb{R}_{\geq 0}$ be the unique value such that $slew^{-1}(e, \omega'(p)_c, s) = \omega'(a)_s$. Let furthermore s' be the value such that $slew^{-1}(e, \omega'(p)_c, s') = \omega(a)_s$ and recall that $slew^{-1}(e, \omega(p)_c, \omega(p)_s) = \omega(a)_s$. Then we know

$$slew^{-1}(e, \omega'(p)_c, s') = \omega(a)_s = slew^{-1}(e, \omega(p)_c, \omega(p)_s) \leq slew^{-1}(e, \omega'(p)_c, \omega(p)_s)$$

and since $slew^{-1}$ is strictly increasing in its slew entry, we know that $s' \leq \omega(p)_s$. Similarly, we use the fact that

$$slew^{-1}(e, \omega'(p)_c, s) = \omega'(a)_s \leq \omega(a)_s = slew^{-1}(e, \omega'(p)_c, s')$$

to see that $s \leq s'$. Hence, $\omega'(p)_s = s \leq \omega(p)_s$. Figure 3.5 visualizes the relation of the slew values. By induction, we again see that $\omega'(a)_s \leq \omega(a)_s$ for all a in A_m. Furthermore, we note that by this definition, no slew can be below the slewlimit of its nodes incoming edge.

Then finally, we need to update the objective values. We traverse A'_m in reverse topological order and compute the objective values as normal. Let $a = (v, e, P, \omega, I)$ be the current label. If a is an initial label, we set $\omega'(a)_o := 0$. If a is a propagate label, let p be the predecessor of a. We set $\omega'(a)_o := \omega'(p)_o + \text{obj}(e, I, \omega'(p)_c, \omega'(p)_s)$. Note that $\text{obj}(e, I, \omega'(p)_c, \omega'(p)_s) \leq \text{obj}(e, I, \omega(p)_c, \omega(p)_s)$. If a is a merge label, let its predecessors be $\{p_1, p_2\} := P$. We set $\omega'(a)_o := \omega'(p_1)_o + \omega'(p_2)_o$.

Now we set $S' := \{s'_t := \omega'(l_t)_s \mid l_t \text{ leaf of } A'_m \text{ with vertex } t\}$. Then the root of A'_m encodes (B, ϕ) and for the initial slew values S', we have $\omega_B(v) = \omega'(a)$ if $a \in V(A'_m)$ encodes the subarborescence of (B, ϕ) rooted in $v \in V(B)$.

Now let $v \in V(A)$ be the vertex that belongs to l and k and let $D \subseteq \delta_A^+(v)$ be the set of outgoing edges at v that is connected by the subarborescence encoded by l. Denote by $\bar{D} := \delta_A^+(v) \setminus D$ the remaining edges. Then $\bar{D} \subset \delta_B^+(v)$. Denote by $D' := \delta_B^+(v) \setminus \bar{D}$.

We have

$$\begin{aligned}
\beta_o = & \sum_{e=(x,y)\in E(B)} \text{obj}(\phi(e), T_{B,y}, \omega_B(y)_c, \omega_B(y)_s) \\
= & \sum_{(x,y)\in E(B_{v,D'})} \text{obj}(\phi((v,w)), T_{B,w}, \omega_B(w)_c, \omega_B(w)_s) \\
& + \sum_{e=(v,w)\in E(B)\setminus E(B_{v,D'})} \text{obj}(\phi(e), T_{B,y}, \omega_B(y)_c, \omega_B(y)_s) \\
\leq & \ \omega(k)_o + \sum_{e=(v,w)\in E(A)\setminus E(A_{v,D'})} \text{obj}(\phi(e), T_{B,y}, \omega_B(y)_c, \omega_B(y)_s) \qquad .
\end{aligned}$$

But since k dominates l or is equivalent to it, we know for each of the components $i \in [\iota]$ of the objective that $\chi_i(\text{proj}_i(\omega(k)_o)) \leq \chi_i(\text{proj}_i(\omega(l)_o))$. Hence, by using that χ_i is nondecreasing and has the tree-monotonicity property,

$$\chi_i(\text{proj}_i(\beta_o)) \leq \chi_i \left(\text{proj}_i(\omega(k)_o) + \sum_{e=(x,y)\in E(A)\setminus E(A_{v,D'})} \text{obj}_i(\phi(e), T_{B,y}, \omega_B(y)_c, \omega_B(y)_s) \right)$$

$$\leq \chi_i \left(\text{proj}_i(\omega(l)_o) + \sum_{e=(x,y)\in E(A)\setminus E(A_{v,D'})} \text{obj}_i(\phi(e), T_{B,y}, \omega_B(y)_c, \omega_B(y)_s) \right) = \chi_i(\text{proj}_i(\alpha_o)).$$

Thus

$$(\chi_1(\text{proj}_1(\beta_o)), \ldots, \chi_{|\mathcal{TM}|}(\text{proj}_{|\mathcal{TM}|}(\beta_o))) \leq (\chi_1(\text{proj}_1(\alpha_o)), \ldots, \chi_{|\mathcal{TM}|}(\text{proj}_{|\mathcal{TM}|}(\alpha_o))))$$

elementwise. Since $\mathcal{M}$ is nondecreasing, we conclude

$$\begin{aligned}
\chi(\beta_o)) &= \mathcal{M}((\chi_1(\text{proj}_1(\beta_o)), \ldots, \chi_{|\mathcal{TM}|}(\text{proj}_{|\mathcal{TM}|}(\beta_o)))) \\
&\leq \mathcal{M}((\chi_1(\text{proj}_1(\alpha_o)), \ldots, \chi_{|\mathcal{TM}|}(\text{proj}_{|\mathcal{TM}|}(\alpha_o)))) = \chi(\alpha_o).
\end{aligned}$$

□

Remark. *Recall, that we said in the beginning of this section, that for weighted sum objectives we do not remove labels that belong to optimum solutions. If our evaluation operator is a weighted sum, the above inequalities regarding the objective costs become strict. So we construct a strictly better solution in this case, which cannot happen for optimum solutions.*

There are some key implications from this Lemma. The first one is that when we construct a new solution with this procedure, we do not change the slew at the root.

But the slews at the sinks only depend on the arborescence and the slew at the root. We will use this to construct the slew sets $(\Sigma_t)_{t\in T}$. If we start from an optimum solution, only the arborescence will matter. The second one is that we can neglect optimum solutions that have a label which is dominated by one of its ancestors.

Proposition 3.2.7. *Let l belong to any optimum solution. If A_l contains a label k that is dominated by or equivalent to one of its ancestors $a \in A_k$, we can construct slews S' and an optimum solution encoded by a label l' such that no label in $A_{l'}$ is dominated by or equivalent to one of its ancestors.*

Proof. Since they both a and k connect the same sink set, there is a path from k to a with no branches on it. We can replace k by a and thus remove this path. By Lemma 3.2.6, we can retrieve a new optimum solution this way. But since we removed the path, its label arborescence contains less labels. We should note here for later that in this case the objective value at k/a has decreased in at least one of the tree monotone components and the others did not increase.

We iterate this process as long as there are still labels that are dominated by or equivalent to their ancestors in our label arborescence. Since we reduce the size of the label arborescence each time, this must terminate. For the resulting l, we now know that A_l contains no labels that are dominated by or equivalent to one of their ancestors. □

The algorithm does not consider labels with ancestors that are dominated by their ancestors, but the proposition tells us that this is not a problem. However, this restriction of not having an ancestor that is dominated by or equivalent to any of its ancestors limits the depth of arborescences that we need to consider.

Proposition 3.2.8. *Let* OPT *be the objective cost of an optimum solution. Let $\mathcal{L}$ denote the subset of the labels that can be generated from the sink slews $(\Sigma_t)_{t\in T}$ that only contains labels with objective cost at most* OPT *and that do not have an ancestor that is dominated by or equivalent to one of its ancestors. The depth of any arborescence encoded in $\mathcal{L}$ is at most $n \cdot (\lceil \frac{\mathrm{OPT}}{C} \rceil + k)$, where $n := |V(H)|, k := |T|$ and C is the constant from the definition of a feasible objective function(2.3.3).*

Proof. Let $n := |V(H)|$ denote the number of vertices in H, $k := |T|$ the number of sinks. Let l be a label with no dominated ancestor, no two equivalent ancestors and $\chi(\omega(l)) \leq \mathrm{OPT}$. The label arborescence of l contains at most $k-1$ labels that arise from a merge operation, because after $k-1$ merges, all sinks are connected by the corresponding mapped arborescence.

Now consider a path P in A_l from l to a leaf (an initial label). First note that by a merge the objective value (and cost) and capacitance do not decrease and the slew does not increase. The path can be partitioned into $m \leq k$ subpaths $P_1, \ldots, P_m$, each ending in either a merge label or an initial label and all other labels of the subpaths are propagate labels. Let P_i be one of those paths. Assume that $|V(P_i)| > n$ and take a subpath P' of P_i of length $n+1$. The corresponding walk in H must contain a loop, because H has n vertices and the walk visits $n+1$ vertices.

Let us examine a subsequence L of P' that defines a loop. Let x be the start of L and y be the end. Then x and y store the same vertex and sink set I. Assume that none of the edges on the loop was constant. By the definitions of obj, *cap*, *slew*$^{-1}$, we have $\omega(x)_o \geq \omega(y)_o$, $\omega(x)_c \geq \omega(y)_c$ and $\omega(x)_s \leq \omega(y)_s$. This means that either x and y are equivalent or x is dominated by y. This is a contradiction. So we know that the loop must contain a constant edge. This implies that $\chi(\omega(x)_o) > \chi(\omega(y)_o) + C$, where C is the constant from the definition of a feasible objective function (Definition 2.3.3). This works, because our path is part of an arborescence connecting the root to the sinks. So in particular, for all $z \in L$, $\omega(z)_o \in \Omega(\text{obj}, I)$ (see Definition 2.3.3).

Since any subpath of length $n+1$ must contain a loop, we know that each of the paths P_i for $i = 1, \ldots, m$ must contain at least $\lfloor \frac{V(P_i)}{n} \rfloor$ loops (n, because the ends of subpaths are allowed to overlap). We also know that the whole path P can contain at most $\lceil \frac{\text{OPT}}{C} \rceil$ loops, because $\chi(\omega(l)) \leq \text{OPT}$. So the length of P is bounded by

$$|V(P)| \leq \left\lceil \frac{\text{OPT}}{C} \right\rceil \cdot n + (n-1) \cdot m + m \leq \left\lceil \frac{\text{OPT}}{C} \right\rceil \cdot n + (n-1) \cdot k + k = n \cdot \left(\left\lceil \frac{\text{OPT}}{C} \right\rceil + k \right).$$

This means that the depth of a label arborescence belonging to such a label is bounded by $n \cdot (\lceil \frac{\text{OPT}}{C} \rceil + k)$ and hence the depth of the encoded mapped arborescence is bounded by the same number as well. □

This depth limit also implies that the number of arborescences that we need to consider is finite. Now we have all that we need, in order to construct our slew set.

Lemma 3.2.9. *For each sink $t \in T$ let $U_t \subset [0, S_t]$ be a finite set of slews. And let o be an optimum solution with sink slews $s_t \in [0, S_t]$ for each $t \in T$ (that has a representation as a label such that no ancestor is dominated by or equivalent to any of its ancestors). There is a finite set of slews $(\Sigma_t)_{t \in T}$ for each sink with $U_t \subseteq \Sigma_t$ such that the following holds:*

Let $\mathcal{L}$ be the set of labels from Proposition 3.2.8. We recursively define the sets O_i for $i \in \mathbb{N}$. We set $O_0 := \{o\}$. For $i > 0$, O_i is the set of optimum solutions that can be derived by the following procedure: If $l \in \mathcal{L}$ represents an optimum solution $o' \in O_{i-1}$ and a label $m \in A_l$ is dominated by another label $k \in \mathcal{L}$, then we replace m by k as in Lemma 3.2.6 to retrieve a new optimum solution and take all shortened versions as in 3.2.7. Then all optimum solutions in $\bigcup_{i \in \mathbb{N}} O_i$ are represented by a label in $\mathcal{L}$.

Proof. We iteratively construct the slew sets by always adding missing slews. We start with the slew sets $\Sigma_t^{(0)} := U_t \cup \{s_t\}$ for $t \in T$. Let $\mathcal{L}_0$ be the set of labels from Proposition 3.2.8 for the starting slews $(\Sigma_t^{(0)})_{t \in T}$. Furthermore, we set $O_0' := \{o\}$. By construction $\mathcal{L}_0$ contains O_0'. Furthermore, $\mathcal{L}_0$ is finite. This is a direct consequence of Proposition 3.2.8 and the fact that $\Sigma_t^{(0)}$ is finite for each $t \in T$. The idea now is to always add missing slews. But since all new slews are generated by the updating procedure from Lemma 3.2.6, the slew at the root does not change. Since we can only represent a finite number of arborescences and the sink slews from the updating procedure only depend on the slew at r and arborescence, we end up with a finite set of slews. In detail:

Let $\Sigma_t^{(i)}, \mathcal{L}_i, O'_i$ be already constructed for some $i \in \mathbb{N}$ and all $t \in T$. Let $\tilde{O}$ contain all solutions that can be generated from solutions in O'_i by replacing a label that is dominated by another one in $\mathcal{L}_i$. Then all solutions in $\tilde{O}$ have the same slew at r as o. Let P_t for $t \in T$ be all slews that solutions from $\tilde{O}$ have at sink t. We set $O'_{i+1} := O'_i \cup \tilde{O}$, $\Sigma_t^{(i+1)} := \Sigma_t^{(i)} \cup P_t$ and $\mathcal{L}_{i+1}$ to be the set from 3.2.8 for sink slews $(\Sigma_t^{(i+1)})_{t\in T}$. Then all solutions from O'_{i+1} are represented in $\mathcal{L}_{i+1}$ by construction and all of them have the same slew at r as o.

We set $O' := \bigcup_{i\in\mathbb{N}} O'_i$ and $\Sigma_t := \bigcup_{i\in\mathbb{N}} \Sigma_t^i$ for $t \in T$. Σ_t consists exactly of the union of U_t and the slews of solutions in O' at $t \in T$.

All solutions in O' have the same slew at r as o. Furthermore, all solutions in O' are representable by a label that has no ancestors that are equivalent to or dominated by any of their ancestors. By Proposition 3.2.8 this means that there are only a finite number of arborescences in O'. As all elements of O' (except for o) have been constructed by the replacement procedure from Lemma 3.2.6, we can conclude that the $(\Sigma_t)_t \in T$ are finite. Thus, O' is finite as well.

This finiteness, implies that there is a number $k \in \mathbb{N}$, such that $O' = \bigcup_{i=0}^{k} O_i$, $\Sigma_t = \bigcup_{i=0}^{k} \Sigma_t^i$ for $t \in T$ and $\mathcal{L}_k$ is the set from Proposition 3.2.8 for the starting slews Σ_t. Finally, this means that $\bigcup_{i\in\mathbb{N}} O_i \subseteq O'$. □

We have constructed sink slews that are finite and thus generate a finite set of labels that are relevant for us. Furthermore, we know that the labels generated by this set of slews contain an optimum solution such that we do not leave our set of labels if we apply repeated replacement operations. With this basis, we will be able to show that the algorithm does not remove all optimum solutions.

Lemma 3.2.10. *For each sink $t \in T$ let $U_t \subset [0, S_t]$ be a finite set of slews. And let o be an optimum solution. Let $(\Sigma_t)_{t\in T}$ be the sink slews generated by Lemma 3.2.9. Let $\mathcal{L}$ denote the set of labels generated from $(\Sigma_t)_{t\in T}$ that are not dominated by, or equivalent to one of their ancestors. There is a label $l \in \mathcal{L}$ that belongs to an optimum solution and none of its ancestors (nor itself) is dominated by another label in $\mathcal{L}$.*

Proof. Let $l \in \mathcal{L}$ be a label that represents o. First, we note that for $k \in V(A_l)$, we have $\chi(\omega(k)_o) \leq \mathrm{OPT}$, because it belongs to an optimum solution. Additionally, k is never dominated by a label with value greater than OPT. So we may restrict now to $\mathcal{L}_{\mathrm{OPT}} := \{l \in \mathcal{L} \mid \chi(\omega(l)_o) \leq \mathrm{OPT}\}$. This set is finite by Proposition 3.2.8 and the fact, that the $(\Sigma_t)_{t\in T}$ are finite.

Now consider the label arborescence A_l. Let A'_l be a subarborescence of A_l that is rooted at l, such that none of its labels is dominated in $\mathcal{L}_{\mathrm{OPT}}$ and none of its labels can become dominated in $\mathcal{L}_{\mathrm{OPT}}$ by an exchange operation as in the proof of Lemma 3.2.6 on a label $a \in V(A_l) \setminus V(A_l)'$. Note that the label l forms such a subarborescence.

For the remainder of this proof, we will only write dominated, when we refer to dominated by a label in $\mathcal{L}_{\mathrm{OPT}}$.

We want to transform A_l (and with it A'_l) by repeated replacement operations on labels in $V(A_l) \setminus V(A_l)'$, such that in each iteration either A'_l grows, or A_l does not contain a dominated label.

If $A'_l = A_l$ or A_l contains no labels that are dominated in $\mathcal{L}_{\text{OPT}}$, we are done. Otherwise, there is at least one dominated label in $V(A_l) \setminus V(A'_l)$.

Let $l_1, \ldots, l_k$ for some $k \leq |T|$ be the labels of A'_l such that at least one predecessor is in $V(A_l) \setminus V(A'_l)$. Let $a \in V(A_l) \setminus V(A'_l)$ be a label that is dominated by another label b. If we replace a by b, as in Lemma 3.2.6, there are three observations. First, we can apply the procedure from Proposition 3.2.7 in order to get to a label that is in $\mathcal{L}_{\text{OPT}}$. This is guaranteed by Lemma 3.2.9. Second, the new objective value at a/b has decreased in at least one of the tree monotone components and the others did not increase. Technically, a/b could have been removed by the shortening procedure. Then this holds for the label that is closest to a/b on the path from l to a/b and still exists. Third, no label in A'_l has been replaced or removed, only their objective values might have changed.

The second observation implies that b cannot dominate a label in $V(A_l) \setminus V(A'_l)$ again, unless a label on the path from l to a/b is replaced.

As long as there is a child a of l_i for $i \in [k]$ that is not dominated and cannot become dominated by an exchange operation, we add a to A'_l. If there is none, but there is a dominated label in $V(A_l) \setminus V(A'_l)$, we choose one such label a with no dominated labels on the path from A'_l to a. Then we replace a by a dominating label b. We keep a set Q of all the labels with replacements that are still part of A_l. So we add the vertex $a/b \in A_l$ to Q, but remove all ancestors of a from Q (if there were any). By our observations, b can only be used again for dominance if a/b is removed from Q (not replaced). This however, means that we add a label to Q that is closer to A'_l than a/b. Furthermore, labels in A'_l can never be replaced. Thus, the vertices of A_l that are incident to A'_l will not be removed from Q and the labels that have been used for dominating them become permanently unavailable for dominance.

So each time we do one of these replacements, one label becomes temporarily unavailable for dominance. Additionally, maybe some labels become available for dominance again, but then a label in Q comes closer to A'_l.

However, the number of available labels, as well as the path length in A'_l and thus the size of Q are finite. This means that either we run out of labels that can dominate. Then we are done. Or one of the vertices of A_l that are incident to A'_l cannot become dominated by exchange operations any longer, so A'_l grows.

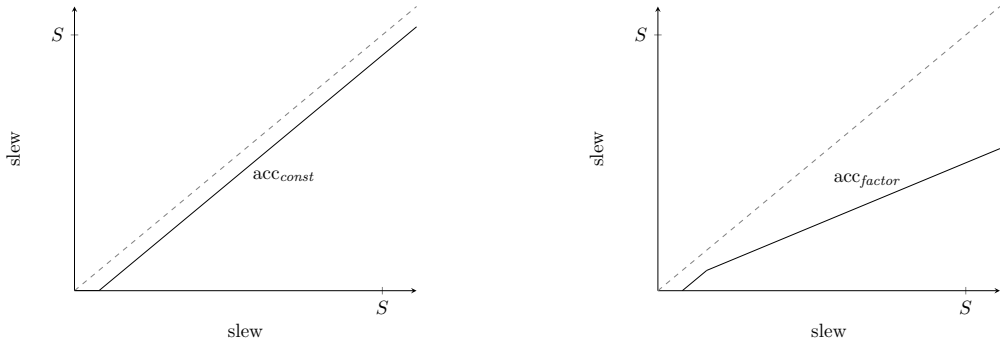
As A'_l cannot grow indefinitely, this must stop at some point. Then we have found our solution without dominated ancestors.

□

Finally, we can prove correctness in our theoretical setting.

Theorem 3.2.11. *If there is an optimum solution to the given instance, there are sink slews $(\Sigma_t)_{t \in T}$, such that Algorithm 5 returns after a finite number of iterations and the returned solution is optimum.*

Proof. Since there is an optimum solution, we can choose one and generate the slews from Lemma 3.2.9. By Proposition 3.2.8, we know that the algorithm considers only finitely many labels with value at most that of an optimum solution. Lemma 3.2.10 shows that there is at least one optimum solution that is considered by the algorithm, because none



(a) A plot of acc_{const}.

(b) A plot of acc_{factor}.

Figure 3.6: Examples of accuracy functions. The gray dashed line marks the identity function.

of the labels it consists of are dominated. By Corollary 3.2.5 this is sufficient to show optimality. □

In total, we have proved that, if we knew all the slews at the outputs for a large set of optimum solutions, we would be able to compute the optimum solution using algorithm 5. Of course, we do not know them in the beginning. We can fix this by essentially guessing the values. If we do it correctly, we can still guarantee that the solution we find is only a small fraction away from the optimum.

3.3 Guessing the Slews

The idea to getting rid of the assumption that we know all the slews of optimum solutions is to guess some slews that are close by. This way, we allow a small error in the slew. If we are able to bound the error in the objective cost induced by this slew error, we can get an approximation guarantee. This is similar to ν-dominance [Vyg06] used in the timing propagation in Chapter 1.2.5.

The proofs are easier, if we use a simple guessing strategy, like guessing slews with a constant distance in between. However, the practical running time both increases with a higher number of slew guesses and with an increasing error. This is why it can make sense to tailor the way we guess the slews to the actual objective functions. So we instead use a more abstract and flexible slew accuracy function.

Let $\text{acc} : [0, S] \to [0, S]$ be a slew accuracy function, where S is the slew upper bound from our instance. For a slew accuracy function, we require that it is continuous, strictly increasing, except on an interval $[0, s']$ for $0 < s' \leq S$, where it is constant 0, $\text{acc}(s) \leq s$ and $s - \text{acc}(s)$ is nondecreasing. An easy example of such a function is $\text{acc}_{const}(s) := \max\{0, s - \delta\}$ for some $\delta > 0$ (Figure 3.6a), or $\text{acc}_{factor}(s) := \min\{\text{acc}_{const}(s), \frac{s}{1+\gamma}\}$ for an additional $\gamma > 0$ (Figure 3.6b). Our goal will be to guess a discrete set of starting slews and modify Algorithm 5 in such a way, that it explores solutions that are optimum (in the sense that the encoded mapped arborescence is optimum), but have a wrong

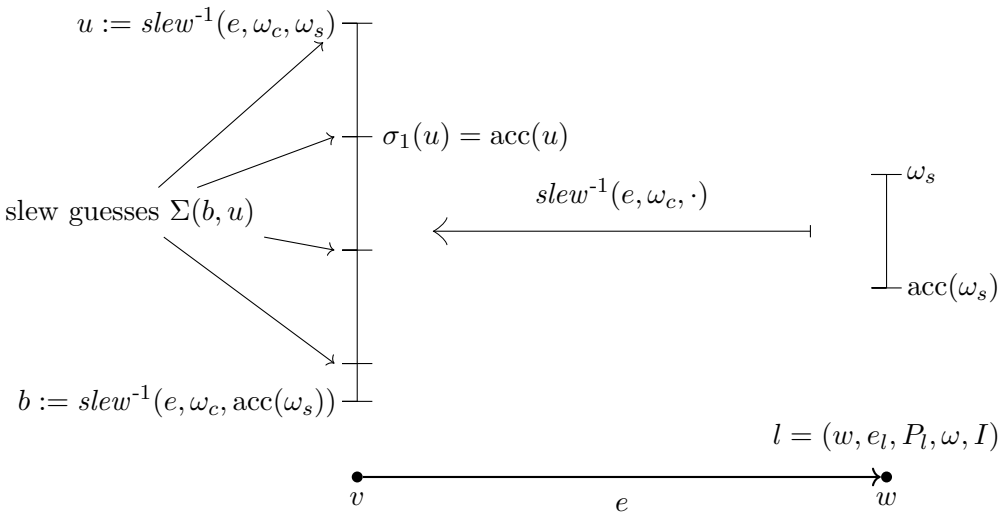


Figure 3.7: The slew interval $[\mathrm{acc}(\omega_s), \omega_s]$ for the slew ω_s on the right is mapped by the slew function to the interval $[b, u]$ on the left, when we propagate l through e. We need to guess the slews in $\Sigma(u, b)$.

slew s, such that the optimum solution has a slew $s_{opt} \in [\mathrm{acc}(s), s]$. Then we can get an approximation guarantee based on the error in the slew. In this sense, while we are using the slews we have for computing the objective values, the slew values can be considered an upper bound on the slew value that an actual solution would have.

To make this work, we guess some starting slews, but also we need to guess some slews during the propagate steps. For an upper bound $u \in [0, S]$, we define the sequence of slew guesses as $\sigma(u)_0 := u$ and $\sigma(u)_i := \mathrm{acc}(\sigma(u)_{i-1})$ for $i > 0$. We are generally only interested in the elements of the sequence that are above some lower bound. Thus, we define the set of slew guesses for the upper bound and a lower bound $l \in [0, S]$ as

$$\Sigma(l, u) := \{\min\{S, u\}\} \cup \{\sigma_i(u) \mid i \in \mathbb{N}, \sigma_i(u) > l\}.$$

Then, we define the starting slew guesses for each sink $t \in T$ as $\Sigma_t := \Sigma(0, S_t)$. Now, we know for the starting slews that for each slew $s \in [0, S_t]$, there is an element $\sigma \in \Sigma_t$ with $s \in [\mathrm{acc}(\sigma), \sigma]$.

However, this is not enough, as this property is not guaranteed to be stable under propagates: If for an edge $e \in E(H)$, a slew $s \in [0, S]$ and capacitance $c \in \mathbb{R}_{\geq 0}$, we have that

$$slew^{-1}(e, c, s) - slew^{-1}(e, c, \max\{minslew(e, c), \mathrm{acc}(s)\}) \\ > slew^{-1}(e, c, s) - \mathrm{acc}(slew^{-1}(e, c, s)),$$

we need to guess slews in the propagate step as well. We do this by creating labels with all slews in $\Sigma(slew^{-1}(e, c, \max\{minslew(e, c), \mathrm{acc}(s)\}), slew^{-1}(e, c, s))$. The new function, called `propagate+`, is written down in Algorithm 6. Figure 3.7 illustrates the slew guesses. Finally, we need to use a weaker version of dominance:

Definition 3.3.1. *Let $\alpha > 1$. A label l α-dominates a label k if l dominates k, we have for each component $i \in [\iota]$ of the objective that $\alpha \cdot \chi_i(\mathrm{proj}_i(\omega(l)) \leq \chi_i(\mathrm{proj}_i(\omega(k)))$ and one of these inequalities is strict.*

This α in the α-dominance will be our approximation guarantee. It depends on the choice of the slew accuracy function. Let $q \in \mathbb{N}$ such that $\Omega = \mathbb{R}^q_{\geq 0}$. The error we make is

$$\alpha := \max_{\substack{e \in E(H),\\ \emptyset \neq I \subseteq T,\\ i \in [q]}} \sup_{\substack{c \in [0, caplim(e)],\\ s \in [minslew(e,c), S]}} \frac{\mathrm{obj}(e, I, c, s)_i}{\mathrm{obj}(e, I, c, \max\{\mathrm{acc}(s), minslew(e, c)\})_i}.$$

If we replace dominance by α-dominance in our algorithm, it becomes an (exponential time) α-approximation algorithm.

Algorithm 6: The propagation with slew guesses.

```
1 Function propagate+(l = (w, e_l, P_l, ω, I), e = (v, w) ∈ E(G), Non-Permanent
    labels Q, Labels L):
2     u := slew^-1(e, ω_c, ω_s);
3     b := slew^-1(e, ω_c, max{acc(ω_s), minslew(e, c)});
4     for σ ∈ Σ(b, u) do
5         k := (v, e, {l}, (ω_o + obj(e, I, ω_c, ω_s), cap(e, ω_c), σ), I);
6         if ω(k) < ∞ then
7             insertLabel(k, Q, L);
```

In fact we will prove that the explored search space of our algorithm (if we run it long enough) contains a label that encodes the same structure as an optimum solution, but has slightly higher slews at each label. Let us define, what this means:

Definition 3.3.2. *Let $O := (A, \kappa, (s_t)_{t \in T})$ be an optimum solution to our problem. Let l be a label that encodes our optimum solution (produced from initial labels with starting slews $(s_t)_{t \in T}$). A* slew-approximate optimum solution *for O is a label arborescence $A_{l'}$ with a label l' such that there is a bijection $\varphi : V(A_l) \to V(A_{l'})$ with*

1) $\varphi((v, e, P, \omega, I)) = (v, e, \varphi(P), \omega', I)$,

2) $\omega_s \in [\mathrm{acc}(\omega'_s), \omega'_s]$ *and* $\omega'_s \in [minslew(e, \omega_c), S]$

3) $\omega_c = \omega'_c$ *and*

4) $\omega'_o \leq \alpha \cdot \omega_o$ *entrywise,*

for each $(v, e, P, \omega, I) \in V(A_l)$.

First, we show that we can construct a slew-approximate optimum solution for each optimum solution using only our starting slews and `propagate+` and `merge`.

Proposition 3.3.3. *Let a slew accuracy function* acc *be given. Let* $O := (A, \kappa, (s_t)_{t\in T})$ *be an optimum solution to our problem. Let* l *be a label that encodes* O*. Let* $\Sigma_t := \Sigma(0, S_t)$ *for each* $t \in T$*. We can construct a slew-approximate optimum solution* $(A_{l'}, \varphi)$ *for* O *from our initial labels* $\{(t, \square, \emptyset, (0, 0, s), \{t\}) \mid t \in T, s \in \Sigma_t\}$*, using* `propagate+` *and* `merge`*.*

Proof. We do this by traversing A_l in reverse topological order and constructing a label satisfying 1) – 4) from Definition 3.3.2 for each label in A_l,.

Let us start with the initial labels. For an initial label $a \in V(A_l)$ at sink $t \in T$, we select the lowest slew $s \in \Sigma_t$ with $s \geq \omega(a)_s$. By definition of Σ_t, we have $s = \sigma(S_t)_i$ for some $i \in \mathbb{N}$ and by selection of s, we have $\mathrm{acc}(s) = \sigma(S_t)_{i+1} < \omega(a)_s$. So we select the initial label $(t, \square, \emptyset, (0, 0, s), \{t\}) =: \varphi(a)$ as our matching label in $A_{l'}$ and it fulfills our requirements.

Now let $a \in A_l$, such that we already constructed labels meeting the requirements for the children of a. If $a = (v, e, \{p\}, \omega, I)$ resulted from a propagate step, we execute `propagate+` on $p' := \varphi(p)$.

Denote by $b := \mathit{slew}^{-1}(e, \omega_c, \max\{\mathit{minslew}(e, \omega_c), \mathrm{acc}(\omega(p')_s)\})$ the lower bound of our guessing interval and by $u := \mathit{slew}^{-1}(e, \omega_c, \omega(p')_s)$, the upper bound of our guessing interval. In `propagate+`, we create a label a_s with $\omega(a_s)_s = s$ for each slew $s \in \Gamma := \Sigma(b, u)$. We have $\omega(a_s)_c = \omega(a)_c$, because $\omega_c = \omega(p')_c$ and we propagated through the same edge. Then we select the lowest slew $s \in \Gamma$ with $s \geq \omega_s$. We know that $s \leq S$, because $\omega_s \leq S \in \Gamma$.

Since $\mathit{slew}^{-1}(e, \omega(p)_c, \cdot)$ is strictly increasing and $\max\{\mathit{minslew}(e, \omega_c), \mathrm{acc}(\omega(p')_s)\} \leq \omega(p)_s \leq \omega(p')_s$, we know that $b \leq \omega_s \leq u$. Let $i \in \mathbb{N}$ be maximum such that $s \leq \sigma(\mathit{slew}^{-1}(e, \omega_c, \omega(p')_s))_i \in \Gamma$. There are two possibilities.

If $\sigma(\mathit{slew}^{-1}(e, \omega_c, \omega(p')_s))_{i+1} \in \Gamma$, we know $\mathrm{acc}(s) \leq \sigma(\mathit{slew}^{-1}(e, \omega_c, \omega(p')_s))_{i+1} < \omega_s$ by our choice of s. Otherwise, $\mathrm{acc}(s) \leq \sigma(\mathit{slew}^{-1}(e, \omega_c, \omega(p')_s))_{i+1} \leq b \leq \omega_s$. It remains to prove that $\omega(a_s)_o \leq \alpha\omega$ entrywise. For each $i \in [q]$ with $\Omega = \mathbb{R}^q_{\geq 0}$, let $\mathrm{proj}_i : \Omega \to \mathbb{R}_{\geq 0}$ be the projection on the i-th entry of Ω. We have

$$\begin{aligned}\mathrm{proj}_i(\omega(a_s)_o) &= \mathrm{proj}_i(\omega(p'_s)_o + \mathrm{obj}(e, I, \omega(p)_c, \omega(p')_s)) \\ &\leq \mathrm{proj}_i(\alpha\omega_o + \alpha\,\mathrm{obj}(e, I, \omega_c, \omega_s)) = \alpha\,\mathrm{proj}_i(\omega_o)\end{aligned}$$

So a_s fulfills our requirements and we set $\varphi(a) := a_s$.

If $a = (v, \square, \{p_1, p_2\}, \omega, I)$ resulted from merging two labels, we construct the new label $a' =: \varphi(a)$ by merging $p'_1 := \varphi(p_1)$ and $p'_2 := \varphi(p_2)$. Since $\omega(p'_1)_c = \omega(p_1)_c$ and $\omega(p'_2)_c = \omega(p_2)_c$, we know that $\omega(a')_c = \omega(p'_1)_c + \omega(p'_2)_c = \omega(a)_c$. For the slews, we have $\mathrm{acc}(\omega(p'_i)_s) \leq \omega(p_i)_s \leq \omega(p'_i)_s$ for $i = 1, 2$. Furthermore, $\mathrm{acc}(\min\{\omega(p'_1)_s, \omega(p'_2)_s\}) = \min\{\mathrm{acc}(\omega(p'_i)_s), \mathrm{acc}(\omega(p'_i)_s)\}$. So we know that $\mathrm{acc}(\min\{\omega(p'_1)_s, \omega(p'_2)_s\}) \leq \omega(a)'_s \leq \min\{\omega(p'_1)_s, \omega(p'_2)_s\} = \omega(a')_s$. So the requirement on the slew is met as well. Finally, $\omega(a')_o = \omega(p'_1)_o + \omega(p'_2)_o \leq \alpha(\omega(p_1)_o + \omega(p_2)_o) = \alpha\omega(a)_o$.

We conclude that if we apply this process in reverse topological order, we construct a label l' with label arborescence A'_l and a bijection φ between them that has the required properties. □

This is, however, not sufficient, because the algorithm could remove all of these slew-approximate optimum solutions. By comparing to our optimum solutions, we can show that at least one of the slew-approximate optimum solutions only requires non-dominated labels.

Lemma 3.3.4. *Let a slew accuracy function* acc *be given. If we use Σ_t as above for the initial slews, replace* `propagate` *by* `propagate+`*, and dominance by α-dominance, then the set $\mathcal{L}$ of labels that are added to L by the algorithm and never removed (if we were to run it infinitely long) contains a slew-approximate optimum solution.*

Proof. We need to prove that we do not remove all slew-approximate optimum solutions when removing α-dominated labels.

Let l, l' be a pair of an optimum and a slew-approximate optimum solution with bijection φ. If there is a label $k' \in V(A_{l'})$ that is α-dominated by another label m during our algorithm, then we know that

$$\alpha\,\chi_i(\mathrm{proj}_i(\omega(m)_o)) \leq \chi_i(\mathrm{proj}_i(\omega(k')_o)) \leq \alpha\,\chi_i(\mathrm{proj}_i(\omega(\varphi^{-1}(k'))_o)) \text{ for each } i \in [\iota]$$

and one of these inequalities is strict. Furthermore, $\omega(m)_c \leq \omega(k')_c = \omega(\varphi^{-1}(k'))_c$ and $\omega(m)_s \geq \omega(k')_s \geq \omega(\varphi^{-1}(k'))_s$. In other words, $\varphi^{-1}(k')$ is dominated by m.

Now we observe, that Lemma 3.2.9 still holds with `propagate+`, because the number of arborescences is still finite. We recall, that Lemma 3.2.9 allowed us to add arbitrary finite sets of slews at each sink. Thus, we may use our slew guesses as these additional slews. Our finiteness argument in Lemma 3.2.10 still holds and we can apply it here.

This means that there is an optimum solution and a label l encoding this optimum solution, such that no ancestor of l is dominated by another label that is found by the algorithm. So the slew-approximate optimum solution that belongs to l cannot have an α-dominated ancestor. □

Finally, this means that when the algorithm returns, the returned solution cannot have a worse objective cost than our slew-approximate optimum solution.

Theorem 3.3.5. *Let a slew accuracy function* acc *be given. If we use Σ_t as above for the initial slews, replace* `propagate` *by* `propagate+`*, and dominance by α-dominance, Algorithm 5 computes an α-approximate solution if a solution exists.*

Proof. By Lemma 3.3.4, we know that the set of labels that the algorithm will create and not remove contains a slew-approximate optimum solution l'. Let l encode a corresponding optimum solution. When the algorithm returns then the value of the returned label can be at most $\chi(\omega(l'))$ by Lemma 3.2.4. We know that $\omega(l') \leq \alpha\omega(l)$ entrywise. Thus, $\chi(\omega(l')) \leq \alpha\,\chi(\omega(l))$. □

Note that this does not mean, that the algorithm returns an optimum solution with wrong slews. It can return a suboptimal solution if there is a solution that, using these overestimated slews, has a better value than the slew-approximate optimum solution.

In this case, we may be able to retrieve a better a-posteriori bound. From the returned solution, we can derive a lower bound on the optimum value (value of the returned

solution divided by the approximation guarantee). Then we can recompute the slews and objective values of our solution and compare to the lower bound.

We even guarantee that there is a slew-approximate optimum solution in our search space. If we do not return immediately, but rather let the algorithm run longer, then one of the labels at the source will belong to this slew-approximate optimum solution.

Corollary 3.3.6. *Algorithm 5 can be modified to return an optimum solution, even when we use slew guesses as described above.*

Proof. We replace `propagate` by `propagate+`, and dominance by α-dominance. This guarantees that there is a slew-approximate optimum solution. In line 7 of the algorithm, we do not return the solution, but rather store it in a set R. We denote the value of the first solution by x and stop the algorithm only if the solution in line 7 has value greater than αx or we reach line 13. When the algorithm stops, either R is empty and we know that there is no solution, or we know that R contains a slew-approximate optimum solution. So we iterate through our set R. For each solution, we compute the minimum valid slews and recompute the objective values. Then we return the solution with the minimum updated objective value. As the updated value of the slew-approximate optimum solution matches the optimum, we return an optimum solution. □

There is another insight, we can gain from the proof of Theorem 3.3.5. We know that at least one slew-approximate optimum solution will not be removed. By construction, for the slews $s_1 \leq s_2$ of all pairs of labels that are merged in this solution $s_1 \in [\text{acc}(s_2), s_2]$ and $\text{acc}(s_2) \in [\text{acc}(s_1), s_1]$. Hence, it is sufficient to only merge labels that fulfill this condition.

In practice, we can improve α by not copying the label and replacing the slew value during propagate. Instead, we compute the slew at the head that would be required to achieve the changed slew value at the tail and compute a new objective value. In this case, we can decrease α to the highest change that this would cause, which is typically lower than the original definition.

3.4 Extensions

In addition to the base algorithm, I have worked together with Benjamin Ihme, whom I (unofficially) co-supervised in his master's thesis, on two extensions or modifications of the algorithm [Ihm23]. The first one is a way to incorporate more accurate delay models by recomputing the delay of labels with a higher order delay model occasionally. The second one uses a pure RC-delay model to allow for continuous repeater placements on segments between two vertices of a graph.

It should be noted here that I only provided technical assistance on the first extension. For the second extension, I provided the initial idea and formula for the optimum delay based on the closest downstream repeater in the path variant. Additionally, we discussed some of the bigger hurdles, like the problem with concave delay functions or blockages

and came up with the idea of creating two fixed labels. The details and proofs, as well as some key components like the interval propagation have been worked out by Ihme. This is, why we only give an outline of the two extensions here and refer to [Ihm23] for further reading.

3.4.1 More Accurate Delay Models

We want to heuristically include more accurate delay models in our computations without losing the advantages of our piecewise model [Ihm23, Chapter 3.6.2]. The idea is to periodically update the delay with a more accurate delay model that may be stage based. To do this, we use our usual RC-timing model from Definition 2.2.4 as a basis. Additionally, we store the delay difference δ since our last update as part of our label. This means that, whenever we perform a propagate step, we add the delay part (see 2.3.2) to this difference, which is identical for all currently connected sinks. If we merge, we take the maximum of both differences. We choose a value Δ and whenever $\delta \geq \Delta$ for a label l after a propagate step, we reconstruct the partial solution that is represented by l and time it with a separate timing engine. Then we use the newly computed delays to the sinks to update our objective and reset $\delta := 0$.

One key problem with this is that the delay and thus the objective may be reduced during an update. It is not trivial that this does not lead to endless loops. However, Ihme was able to show that this is not the case. The idea is that at least one of the latest exact delay and the δ increases in the upstream direction.

3.4.2 Interval-Based Buffering

In our original formulation, repeaters are placed at vertices of our graph model. However, this can lead to inaccuracies. In order to avoid being too inaccurate, we need to make sure that the distance between vertices is short. This leads to the usage of many vertices, even if only few are actually used for repeater placement in an optimum solution. With the Interval-Based Buffering [Ihm23, Chapter 4], we instead place repeaters on continuous intervals between vertices. To make this possible, we need some simplifications. First, we will work on a flat model (see Definition 2.2.1), where the input graph is embedded into the rectilinear plane. Second, we will use a less accurate RC-delay without slew. This means that we model the delay through wires by the Elmore delay and each repeater is represented by a constant base delay, an internal resistance and an input pin capacitance. Our goal is now to optimize the (weighted) sum of delays from the source to the sinks.

Let us start with an instance with only one sink and a graph that is a path in order to explain the idea. Additionally, we only allow a single wire type per stage. Again we start at the sink and produce labels by iteratively extending partial solutions. However, instead of storing a fixed delay, we store a function and a legal interval in each label. The function takes as input a position in the interval. It computes the delay from our current vertex to the sink if the next downstream repeater is fixed at the given position and all other downstream repeaters are placed at optimum positions (in their respective interval) w.r.t. the input position.

Each time we propagate through an edge, we create new labels for the possibility of using the whole edge as a wire and for all available repeaters. Using the edge with a repeater means that we place a repeater anywhere on that edge.

Now let us take a look at the different situations, we may encounter, when we propagate a label l at w through an edge $e = (v, w)$ to create the new label l':

Case 1: The solution encoded by l has no downstream repeater and we want to extend it by a wire.

The legal interval is the whole edge, that is $[0, \mathit{length}(e)]$. The Elmore delay from v to the sink is a constant. So $\mathit{delay}_{l'} : [0, \mathit{length}(e)] \to \mathbb{R}_{\geq 0}$ is given by a constant $h > 0$: $\mathit{delay}_{l'}(x) := h$.

Case 2: The solution encoded by l has no downstream repeaters and we want to extend it by a repeater.

The legal interval represents the positions at which the repeater may be placed on the edge e, measured from w to v. In this case the repeater may be placed anywhere on the edge, so the legal interval is $[0, \mathit{length}(e)]$. The delay from v to the sink is a quadratic function $\mathit{delay}_{l'} : [0, \mathit{length}(e)] \to \mathbb{R}_{\geq 0}$ of the form $\mathit{delay}_{l'}(x) := fx^2 + gx + h$ for constants $f > 0$, $g, h \in \mathbb{R}$. In particular, we can compute an optimum position for the repeater on the edge. It lies either on one of the endpoints of $[0, \mathit{length}(e)]$ or at value of x, where $\mathit{delay}'_{l'}(x) = 0$. This happens at $x = \frac{-g}{2f}$.

Case 3: The solution encoded by l has a downstream repeater and we want to extend it by a wire.

Not much happens here. The legal interval is the same as the legal interval of l and only the constants of the quadratic function describing the delay may change.

Case 4: The solution encoded by l has a downstream repeater and we want to extend it by a repeater.

This is the most interesting case. Let us first assume that there is exactly one downstream repeater on the path to the sink. Let the respective edge be called e'. From Case 2 and 3, we know that the legal interval is $[0, \mathit{length}(e')]$. The delay from w to the sink is a quadratic function $\mathit{delay}_l(x_1) := f_l x_1^2 + g_l x_1 + h_l$ for $x_1 \in [0, \mathit{length}(e')]$ with constants $f_l > 0$, $g_l, h_l \in \mathbb{R}$. Now we want to place the second repeater on e. So our delay becomes a quadratic function in two variables: $\mathit{delay} : [0, \mathit{length}(e')] \times [0, \mathit{length}(e)] \to \mathbb{R}_{\geq 0}$. Let R be the resistance, B the base delay and C the input pin capacitance of the repeater that we want to insert. Let additionally r_v, c_v be the resistance of the wire connecting to the input of the repeater and r_w, c_w be the resistance of the wire connecting to its output. Denote by $C(x_1)$ the downstream capacitance at w with respect to the downstream

repeaters position x_1. Then the delay function at v is

$$\begin{aligned} \mathit{delay}(x_1, x_2) = \quad & f_l x_1^2 + g_l x_1 + h_l + x_2 r_w \cdot \left(x_2 \frac{c_w}{2} + C(x_1) \right) + B + R(x_2 c_w + C(x_1)) \\ & + (\mathit{length}(e) - x_2) r_v \cdot \left((\mathit{length}(e) - x_2) \frac{c_v}{2} + C \right) \\ = \quad & f_1 x_1^2 + f_2 x_2^2 + g_1 x_1 + g_2 x_2 + g_{12} x_1 x_2 + h, \end{aligned}$$

for constants $f_1, f_2 > 0$ and $g_1, g_2, g_{12}, h \in \mathbb{R}$. In particular, if we fix a position for x_2, this is a convex function in x_1. Similar to Case 2, we can find its global minimum at $p(x_2) = -\frac{g_1 + g_{12} x_2}{2 f_1}$. Thus, $\mathit{delay}(p(x_2), x_2)$ is a quadratic function again. We set $\mathit{delay}_{l'}(x) := \mathit{delay}(p(x), x)$. And we can show that it is again of the form $\mathit{delay}_{l'}(x) = fx^2 + gx + h$ for $f > 0$, $g, h \in \mathbb{R}$. But we need to be careful, because this function is only valid, as long as $p(x) \in [0, \mathit{length}(e')]$. This is where the legal intervals come into play. Now we need to define the legal interval of l' as the interval such that $p(x) \in [0, \mathit{length}(e')]$.

Until now, we have assumed that we only have one downstream repeater. As we can see, we have transformed our delay function in such a way, that it only depends on the closest downstream repeater. So we are in the same setting as we were with only one downstream repeater. Thus, we can use the same procedure in any case.

Now we can use this method in our dynamic program (without a dominance relation). Once we have found a permanent solution at the source, we can recursively compute the optimum positions of all repeaters. Ihme was able to show that this procedure is able to find an optimum solution. It should be noted here that this approach can only place one repeater per edge. This is still an inaccuracy. Ihme was also able to extend this procedure to multiple sinks and wire types per stage. In these cases, we need to be more careful, because it is possible that the functions become concave. This can be dealt with by creating two labels, fixing the repeater at the opposing endpoints of its legal interval. One of the two positions will be optimum, depending on the upstream tree.

4 Speedup Techniques and Heuristics

While the model and algorithm can be implemented as described in the previous chapters, the actual runtime is too high to be usable on a wide range of practical instances. In this chapter, we will describe the speedup techniques and heuristics we are using to allow solving larger instances. Most of the techniques described here will remove the approximation guarantee. However, the idea is to find ways to reduce the practical complexity in such a way that the instances can be solved in reasonable time, but the solutions remain good.

We start by presenting a way to efficiently manage the labels. The described data structure puts its emphasis on reducing the number of operations that took the most time in a first naive implementation. They were identified by multiple iterations of profiling and improvement.

Then we present a construction for a graph structure extending the one presented in [Roc18]. It allows us to model the constraints that appear in an industrial flow. Simultaneously, it tries to reduce the number of vertices without reducing the accuracy too much. Additionally, we show how we can reduce the number of vertices even further.

Next we present a simple way to compute a practical lower bound function and that it is indeed a feasible lower bound.

Sometimes it is helpful to restrict the topology. For example, we will use a fixed topology to compute a first solution. We also use it as a fallback in case we can not find an optimum solution in reasonable time. Moreover, we use its value as an upper bound during our unconstrained run. This speeds up the computation notably. We achieve this by the merge arborescence that we present in Section 4.4.

We conclude with some simpler ideas of heuristics that we have tested during the development of the algorithm and framework.

Since instances can have many sinks, and our running time is exponential in the number of sinks, we will finally present a heuristic to reduce the instance sizes. We achieve this by clustering sinks and solving the smaller instances with respect to carefully chosen virtual roots. This is embedded in a second dynamic program which uses the clustering and solving steps to build up a solution of the initial instance.

4.1 Maintaining the Labels

When running algorithm 5, we need to maintain millions of labels. We need to

- find the minimum effective delay,
- identify permanent labels,
- check for dominance,
- find labels to merge with and
- reconstruct a solution in the end.

Since these are our core routines that will be called millions of times, we need to optimize the running time of these operations, while keeping the memory usage in a feasible range. In this section, we explain how we achieve this.

We store the data of all labels in a large dynamic array and only reference them by their index throughout the rest of the implementation. This way, we have fast (for our purposes constant time) and stable access under resizes. Labels in this vector can have two states. They are *valid* as long as they are not *invalid*. When we find that a label is dominated, we do not remove it from the vector, but instead, we mark it as invalid.

For the purpose of finding the label with minimum effective delay, we use a priority queue with logarithmic time insert and logarithmic time removal of the top (minimum) element. When a label in the priority queue becomes invalid, we do not directly remove it, but rather leave it in the queue, as this has only a small effect on running time, but finding and removing them is slow.

Whenever we extract an invalid label from the priority queue, we add its index to a list of free indices instead of performing an actual delete operation. When a new label is inserted, we add it at one of the free indices. The vector is only extended if all available indices are exhausted. This helps to reduce memory usage and reallocations.

To keep the number of labels lower, we do not create labels that belong to an invalid solution and we use an upper bound. We store the value of the minimum label at the source. Before we try to insert a label, we check if its effective value is above the upper bound. If this is the case, we do not insert it.

Next, we discuss how we check for dominance. This makes up a significant amount of the running time. For each vertex and terminal set, we store one data structure for the permanent labels and one for the non-permanent labels. Note that we store them lazy, that is, we only create them if there are labels with that terminal set at the respective vertex.

For the permanent labels, we split the range of valid slews into a constant number of buckets. In each bucket, we store an array of indices of permanent labels. When a label becomes permanent, we push it to the end of the respective bucket. Note that they are automatically sorted by effective delay in this case.

The data structure for the valid non-permanent labels is more involved. Here we again split by slew buckets. We extend the ordering for the priority queue from line

3 of Algorithm 5 to a total order in the sense that either two labels are equivalent, or comparable. An example is a lexicographic order on the values of the tree-monotone components. By making sure that the label indices in each bucket are sorted by this order, we can guarantee that if a label is chosen in line 3, it is at the first position in its bucket. Thus, we get constant time removal in this case.

By splitting into the buckets, we only need to check in buckets with higher or the same slew when searching for labels that dominate a given label, and we only need to check in buckets with lower or the same slew if we want to find labels that are dominated by our current label. Furthermore, we can ignore the slew in comparisons if we know that the labels are in different buckets. Finally, we can limit the number of buckets to check, when merging.

Next, we need the following:

Proposition 4.1.1. *If $l = (v, e, P, \omega, I)$ dominates $k = (v, e', P', \nu, I)$ then $\omega_{\mathit{eff}}(l) \leq \omega_{\mathit{eff}}(k)$.*

Proof. For each tree-monotone component $i \in \mathcal{TM}$, we compute

$$\begin{aligned}
&\chi_i(\mathrm{proj}_i(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I))) \\
\leq\ &\chi_i(\mathrm{proj}_i(\omega_o + \mathfrak{Lb}(v, (\nu_c, \nu_s), T \setminus I)))) + \mathrm{OPT}(v, \nu_c, \nu_s, \emptyset + (v, \omega)) \\
=\ &\chi_i(\mathrm{proj}_i(\omega_o + \mathfrak{Lb}(v, (\nu_c, \nu_s), T \setminus I)))) \\
\leq\ &\chi_i(\mathrm{proj}_i(\nu_o + \mathfrak{Lb}(v, (\nu_c, \nu_s), T \setminus I)))),
\end{aligned}$$

where the first inequality is from the definition of feasible lower bound together with sublinearity. The equality holds, because $\nu_c \geq \omega_c$ and $\nu_s \leq \omega_s$. This means that ω already fulfills the conditions for the optimum, so no edge has to be used in an optimum solution. The second inequality follows from the inequality $\chi_i(\mathrm{proj}_i(\omega_o)) \leq \chi_i(\mathrm{proj}_i(\nu_o))$ from the definition of dominance (see Definition 3.1.2) and the condition that feasible lower bounds must be consistent with tree-monotonicity.

Hence, all tree monotone components are smaller, which implies that

$$\chi(\omega_o + \mathfrak{Lb}(v, (\omega_c, \omega_s), T \setminus I)) \leq \chi(\nu_o + \mathfrak{Lb}(v, (\nu_c, \nu_s), T \setminus I)).$$

□

Since all buckets are sorted by the effective objective, we can now find the position at which we would insert a new label by binary search. Using the extended order described earlier, we either hit an equivalent label or we find the position in which the label should be placed. Then we know that we only need to search in one direction for dominating labels and in the other for dominated labels. As an extra speedup, if we find a dominated label next to our labels position, we can replace this label to save a shift in the bucket.

The last remaining task is to find labels for merging. Since we stored the permanent labels by vertex and sink set, we can simply find the permanent labels at a vertex. Then we iterate through the sink sets that have an entry and check if they are disjoint to our current sink set.

4.2 Graph Construction

While we already presented a construction for the TMCMAP based on a graph that models the routing space, we did not specify yet how we model the routing space. This section will cover the construction we use, which is based on the Hanan grid. We will first present the basic construction and then show how we reduce the number of vertices.

4.2.1 Basic Construction

We base our graph on the Hanan grid. Since placing buffers on blockages is not possible, it may happen that the interconnect has to go around a blockage. Delay-, slew-, or capacitance constraints may make it impossible to have a wire that is long enough to go over a blockage without splitting it by a repeater. So we need to add blockage corners as well for our Hanan grid construction.

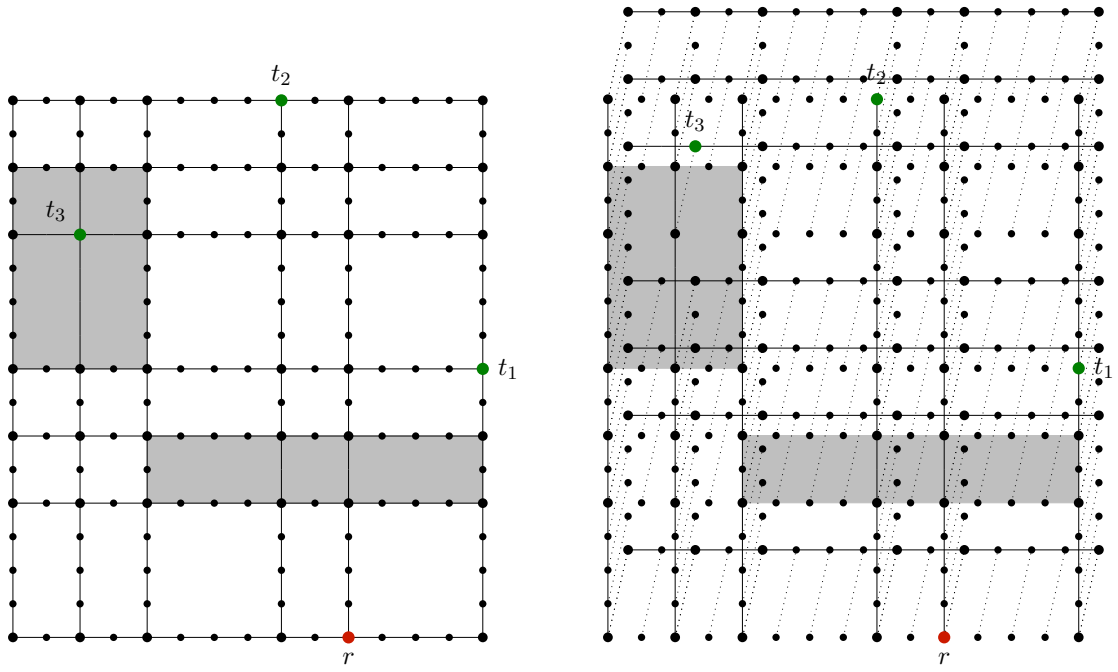
The basic idea was already used in [Roc18], but we extend it to properly handle the width/spacing assignments and present some new vertex reduction techniques.

Given a net $N := \{r\} \cup T$, and a set of blockages B, we compute the subset of blockages that intersect the bounding box of N. For simplicity, we will assume that these are all blockages in B. Then let P be the set of positions of the pins and the blockage corners. First, we construct the Hanan grid H' on the positions P. We call the vertices $V(H')$ *Hanan vertices*. Then, for a given maximum vertex distance $dist_v > 0$, we subdivide the edges in H' until no edge is longer than $dist_v$. We go through the edges in direction of increasing coordinates and always insert a vertex on that edge, when the distance to the last vertex is $dist_v$, but other ways to subdivide can be used. The resulting graph is called our *base layout* H and will later also be the base of our graph model. An example construction is shown in Figure 4.1a.

This base layout has some helpful properties. It always contains shortest (blockage avoiding) source sink paths and shortest Steiner trees on subsets of the sinks. Since no edge is longer than $dist_v$, we can place repeaters at most $dist_v$ away from their optimum positions on the grid. Additionally, all placement constraints at the vertices can be precomputed.

From this base layout, we construct a layered graph G. For this purpose, let L be the set of routing layers, and W_l for each layer $l \in L$ be the set of allowed width-spacing combinations on that layer. For each layer $l \in L$, we take a copy H_l of our base layout. We remove all edges in H_l that are not going in the preferred direction of l. Then we replace all remaining edges $e_l \in H_l$ by one edge $e_{l,w}$ for each wire type $w \in W_l$. Finally, we add edges $(v_l, v_{l'})_{w,w'}$ for all layer combinations $l, l' \in L$ with $|l - l'| = 1$ and wire type combinations $w \in W_l$, $w' \in W_{l'}$ that have a via between them. An example for the layered graph is depicted in Figure 4.1b.

This version of the graph allows wire tapering. If this is not allowed, we need to alter the construction of the graph. It is not sufficient to only allow edges of the same width and spacing as the previous one, because that would violate one of the core assumptions of the problem: The objective function only depends on the capacitance, slew, edge and



(a) The base layout arising from our blockages and net. Hanan vertices are shown slightly larger than the subdivisions.

(b) The graph arising from the base layout for two layers and only one wire code.

Figure 4.1: An example construction of our graph for the net $\{r\} \cup \{t_1, t_2, t_3\}$. Placement blockages are marked in gray.

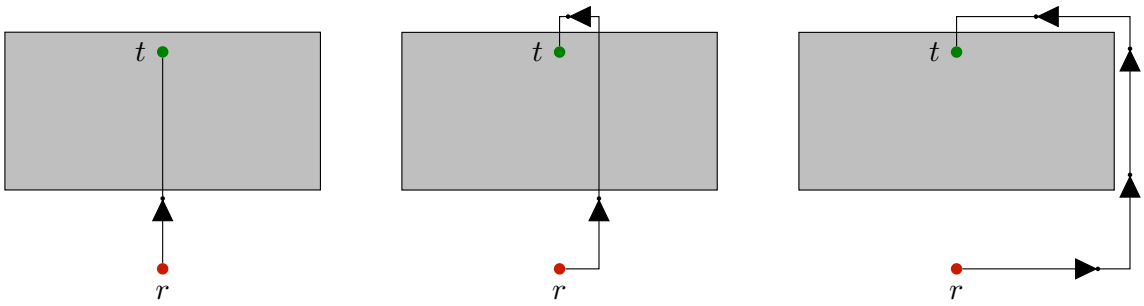
connected sinks. Or in terms of the algorithm: We are only allowed to compare labels (for dominance) if the possible propagates and merges are the same.

If we want to forbid wire tapering or subsets of the changes, we can instead create copies $H_{l,w}$ of the base layout for each $l \in L$ and $w \in W_l$. Then, as before, we remove the edges against preferred direction and add edges $(v_{l,w}, w_{l,w'})$ if $(v_{l,w}, w_{l,w}) \in H_{l,w}$ and the transition from w to w' is allowed. For vias, we add edges $(v_{l,w}, v_{l',w'})$ if $|l - l'| = 1$ and the via w to w' is allowed.

4.2.2 Vertex Reduction

The construction can have many vertices and edges, which may increase the running time to be infeasible. This is why we want to reduce the number of vertices. The layer copies are required to accurately model layer changes in the TMCMAP, but we can try to shrink the base layout.

The first step is to remove the non-Hanan vertices on blockages. Since we can not place buffers there, this has no influence on buffer placement. We have to assume that we do not lose too much by not being able to merge on non-Hanan vertices over blockages.



(a) The available repeaters are strong enough to drive a wire that crosses the blockage and directly connects to the sink.

(b) The available repeaters can drive a wire across the blockage, but the sink pin capacitance is high, or the slew limit low. A repeater has to be placed on the closest border of the blockage.

(c) The available repeaters are not able to drive a wire that crosses the whole blockage. The connection must go around the blockage.

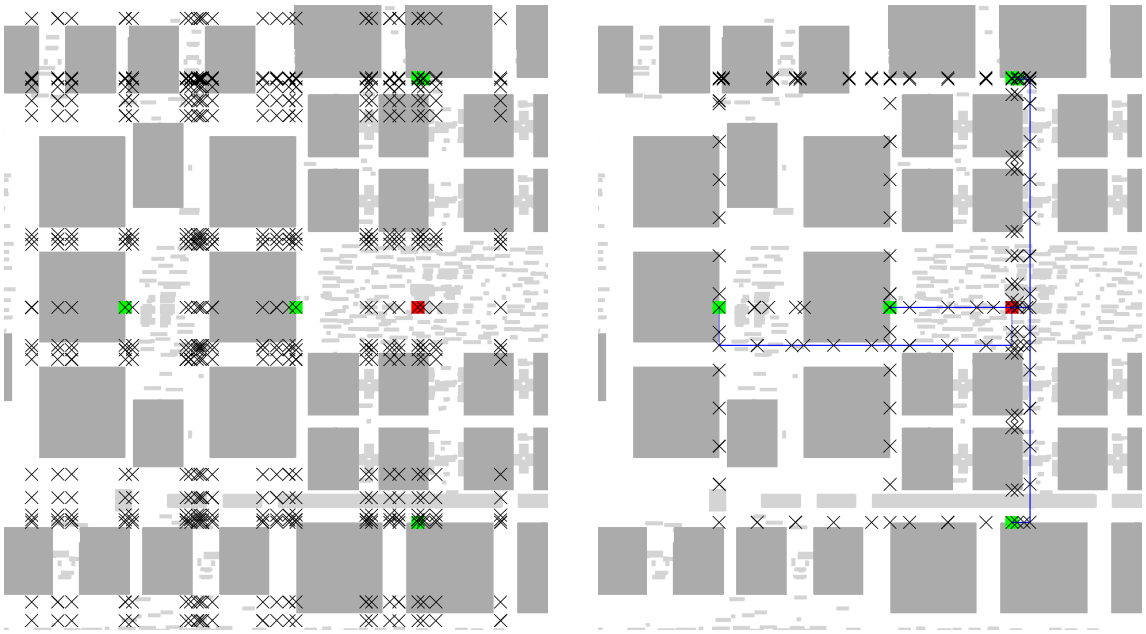
Figure 4.2: Different scenarios how a blocked sink has to be connected depending on the strength of available repeaters, size of the blockage, sink pin capacitance and slew limit.

The second step is more complicated. We want to remove points from the set P of input positions for the Hanan grid. We want to maintain the property that there is at least one shortest blockage-avoiding $r - t$ path in the grid for each sink $t \in T$. This guarantees that we keep at least an optimum (up to the buffer placement discretization) single target solution for each sink. If a sink lies on a blockage, it is unclear from which direction we should access it, judging just by the coordinates.

Take this example: We have a sink that could be reached by a straight axis parallel line from the source. It is located on a blockage, closer to the opposite of the side of the blockage that is facing the root. One possibility is that we have large enough buffers, such that we can go straight into the direction of the root from the sink. It may as well happen that we have to go into the opposite direction, because the next possible buffer location is closer and we would create a violation otherwise. This is shown in Figure 4.2.

For sinks on blockages, we instead want to have the property that there is a shortest path from the sink to each of the four projections of the sink onto the blockage borders.

In summary, we have a set of targets and a source in a Hanan grid. We want to find a shortest path tree that minimizes the number of different x-coordinates of y-directed edges and y-coordinates of x-directed edges that are used by the tree. Since this is a quite complex objective, we simplify the objective. We want to minimize the number of times that the tree is switching between x- and y-edges. When we have such a tree, we can reduce P to consist of the source, the targets, and all the points where a switch between x- and y-edges happens. An example of this reduction can be seen in Figure 4.3. The problem of finding such a tree can be modeled by the bend-avoiding shortest path tree problem.



(a) The initially computed positions for our Hanan grid, marked with black x-es. They are based on the pin positions (red and green) and the corners of blockages that intersect with the bounding box of the pins.

(b) The positions (marked with black x-es) in our Hanan graph based on the blue bend-avoiding (and blockage avoiding) shortest paths.

Figure 4.3: An example of the vertex reduction. Figure 4.3a shows the positions that we initially computed. Figure 4.3b shows the final vertex positions (with subdivisions) in our Hanan graph and the bend-avoiding shortest paths. The root is red, the sinks are green and the blockages that may be relevant for our positions are dark gray. The light gray blockages are only used for repeater placement checking.

4.2.3 The Bend-Avoiding Shortest Path Tree Problem

We define a *bendgraph* as a connected graph $G = (V_1 \cup V_2, E_1 \cup E_2 \cup E_{bend})$ with a bijection $\varphi : V_1 \to V_2$, such that $E_{bend} := \{\{v_1, v_2\} \mid v_1 \in V_1, v_2 \in V_2, \varphi(v_1) = v_2\}$ and removing E_{bend} decomposes G into two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$.

In the *Bend-Avoiding Shortest Path Tree Problem* (BASPTP), we are given a bendgraph G with a cost function $c : E_1 \cup E_2 \to \mathbb{R}_{\geq 0}$, a root $r \in V(G)$ and a set $S \subseteq V(G)$ of terminals. The task is to compute a tree T in G with $\{r\} \cup S \subseteq V(T)$ that contains a shortest r-s-path for all $s \in S$ and among these minimizes the number of bends $\text{bend}(T) := |E_{bend} \cap E(T)|$.

In our setup, the two graphs are two copies of the Hanan grid, where one only contains the x-edges and the other only contains the y-edges. The two copies of each vertex are connected by a bend edge. Unfortunately, this problem is NP-complete.

Theorem 4.2.1. *It is NP-complete to decide, given an instance of the BASPTP (G, c, r, S) and a number $k \in \mathbb{N}$, to decide if there is a BASPT with at most k bends.*

Proof. The problem is in NP, as a tree with less than k bends suffices as a certificate. We use a reduction from SAT. Given an instance of SAT with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, we define a bendgraph. We start by defining $V_1 := \{r\} \cup \{l_{x_i}, l_{\bar{x}_i} \mid i = 1, \ldots, n\}$ as the set containing r and a vertex for each literal. Then we define $V_2 := \{u_{x_i}, u_{\bar{x}_i} \mid i = 1, \ldots, n\} \cup \{C_i \mid i = 1, \ldots, m\} \cup \{c_x^i \mid x \text{ literal in } C_i, i = 1, \ldots, m\} \cup \{x_i \mid i = 1, \ldots, n\}$. We identify u_x and l_x via the bijection φ and add to V_1 and V_2 for each unmatched vertex as a pendant a vertex in the respective other set. Finally, we add edges $\{r, l_x\}$ for every literal x and edges $\{u_x, x\}, \{u_{\bar{x}}, x\}$ for each variable x. We add edges $\{C_i, c_x^i\}$ for every literal x appearing in C_i and each $i = 1, \ldots, m$. Finally, we add edges $\{u_x, c_x^i\}$ for each appearance of the literal x in a clause C_i. The cost function c is set to 1 and $S := \{x_i \mid i = 1, \ldots, n\} \cup \{C_i \mid i = 1, \ldots, m\}$.

We claim that this instance of the BASPTP has a tree with n bends if and only if there is a satisfying truth assignment to the SAT instance. An example of this construction can be seen in Figure 4.4.

Given a satisfying truth assignment, we can construct such a tree, by adding for each true variable x a path r, l_x, u_x, x and for each false variable y a path $r, l_{\bar{y}}, u_{\bar{y}}, y$. Finally, we choose for each clause C_i one fulfilling literal x and add the path u_x, c_x^i, C_i. All added paths are shortest and since we added only one bend per variable, the resulting tree has n bends.

Now assume, we are given a tree T with at most n bends. First note that all shortest r-x_i paths are of the form r, l_{x_i}, u_{x_i}, x_i or $r, l_{\bar{x}_i}, u_{\bar{x}_i}, x_i$. As there is a shortest $r - x_i$ path in T for each x_i and each of these paths introduces a bend, T must contain exactly one of u_{x_i} and $u_{\bar{x}_i}$ for each variable x_i. This yields our truth assignment. On the other hand all r-C_i paths are of the form r, l_x, u_x, c_x^i, C_i, for x a literal appearing in C_i. This means that for each C_i, there must be a path u_x, c_x^i, C_i in T for x appearing in C_i with x a true literal. Hence C_i is satisfied. □

It turns out that we cannot even get a constant factor approximation unless P=NP, because we may reduce set cover to our problem.

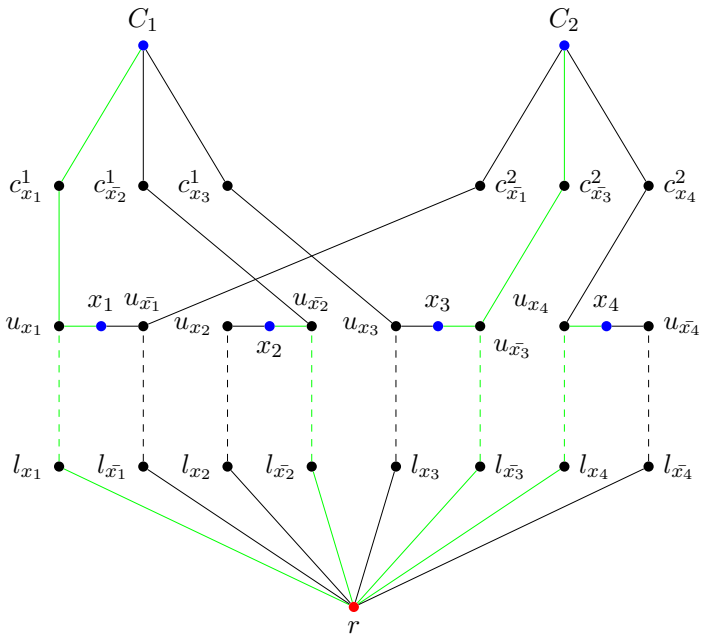


Figure 4.4: An example construction for the clauses $C_1 := \{x_1, \bar{x}_2, x_3\}, C_2 := \{\bar{x}_1, \bar{x}_3, x_4\}$ on the variables x_1, x_2, x_3, x_4. The blue vertices are the terminals. Dashed edges mark bend-edges. The green edges mark a shortest path tree with 4 bends, corresponding to a satisfying truth assignment. The complements added in the completion step are left out as a simplification.

Theorem 4.2.2. *There is no* $\ln(n) - \varepsilon$ *approximation algorithm for any* $\varepsilon > 0$ *for the BASPT problem unless P=NP.*

Proof. Given an instance of unweighted set cover with sets $\mathcal{S}$ and universe $\mathcal{U}$, we construct an instance of the BASPTP.

The bendgraph will consist of two copies V_1, V_2 of the vertices $\{r\} \cup \mathcal{S} \cup \mathcal{U}$. The bijection φ will identify the two copies of each vertex. Finally we will add edges $\{r_1, S_1\}$ for each $S \in \mathcal{S}$ and edges $\{S_2, u_2\}$ for each $u \in S$. All edges will have weight 1. The root is r_1 and the terminals will be $\{u_2 \mid u \in \mathcal{U}\}$.

Each shortest r_1-u_2-path is of the form r_1, S_1, S_2, u_2 with $u \in S \in \mathcal{S}$. Given a solution of the BASPTP with k bends, the sets corresponding to the bend-edges in this solution form a set cover of size k. On the other hand, given a set cover X of size k, we can construct a solution to the BASPTP with k bends, by choosing a tree with the bend-edges $\{S_1, S_2\}$ corresponding to the chosen sets $S \in X$, adding the edges from r_1 to S_1 for $S \in X$ and since X is a set cover, we may choose one S_u with $u \in S_u \in X$ for each $u \in U$ and then add the edges $\{S_{u_2}, u_2\}$ to complete our BASPT with exactly k bends.

Hence, we could use a $\ln(n) - \varepsilon$ approximation algorithm for the BASPTP in an approximation with the same guarantee for set cover. But such an algorithm does not exist unless $P{=}NP$. □

By this construction, we can not achieve a constant factor approximation, even if we allow the paths to be a bit longer than a shortest path. We can prove that we need to allow paths to be at least a factor 3 times longer for this construction to be insufficient.

Corollary 4.2.3. *For any $1 \leq \delta < 3$, there is no $\ln(n) - \varepsilon$ approximation algorithm for any $\varepsilon > 0$ for the BASPT problem unless P=NP, even if we allow the paths to be a factor δ longer than a shortest path.*

Proof. In the proof of the previous theorem, any r_1-u_2-path that is not of the form r_1, S_1, S_2, u_2, needs at least three edges in G_2. However, we may increase the cost of the $\{S_2, u_2\}$-edges arbitrarily. Consequently, for any such δ, we may construct an instance such that all r_1-u_2-paths, that are not more than a factor δ longer than a shortest path are again of the form r_1, S_1, S_2, u_2. □

Due to these results, we instead use a heuristic. Let G be the Hanan graph before subdividing. We can compute shortest paths from the source to all sinks with the minimum number of bends in $\mathcal{O}(n^2 \log n)$, where $n = |V(G)|$. To achieve this, we first observe that a shortest path can have length at most n, so we can have at most n bends on each of these paths. We use Dijkstra's algorithm to compute for all vertices $v \in V(G)$ and numbers of bends $b = 0, \ldots, n$ the shortest path from r to v that uses exactly b bends. Then we select at each sink the minimum b, for which the path has the same length as a shortes path without the bend-restriction.

4.3 A Feasible Lower Bound

In this section we will show how we can construct a lower bound function for the delay part of our objective if we use the Hanan graph from Section 4.2 and an RC-timing model (see Chapter 2.2.4), in which the capacitance and resistance of a wiring segment grow linearly with the length of that segment.

For our lower bound function, we extend an idea from [Roc18]. We want to make use of the fact that a buffering solution can be computed very fast, if the instance is a path with one sink at the end. Each vertex in our Hanan graph has a fixed distance to the root. So we set $dist(v) := \|p(r) - p(v)\|_{l_1}$ for all vertices $v \in V(G)$ of the (routing) graph model G. We take each distance that is occurring in the Hanan graph and create one vertex for that distance. Then we connect them in ascending order of their distance.

Indeed, we can prove that if we have a lower bound with respect to a single target, then we can construct a multi-target lower bound function from it.

Lemma 4.3.1. *Let* obj *be a branch based objective on $\Omega := \mathbb{R}^T_{\geq 0}$. Let χ be an evaluation operator on Ω.*

Let $D := \{dist(v) \mid v \in V(G)\}$ be the set of vertex distances and $f : D \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \mathbb{R}_{\geq 0}$ a function that fulfills

$$f(dist(v), c, s)) \leq f(dist(w), c', s') + \mathrm{OPT}(w, s', c', \emptyset + (v, (0, c, s))) \qquad (*)$$

for all $v, w \in V(G)$ *and* $(c,s),(c',s') \in \mathbb{R}_{\geq 0}^{\{c,s\}}$. *Define* $F : D \times \mathbb{R}_{\geq 0}^{\{c,s\}} \times I \to \Omega$ *as*

$$F_t(d,c,s,I) := \begin{cases} \min\limits_{\sigma \in \Sigma_t} f(\mathit{dist}(t),0,\sigma) & \text{if } t \in I \\ f(d,c,s) & \text{otherwise.} \end{cases}$$

Then

$$\mathfrak{Lb}(v,(c,s),J) := F(\mathit{dist}(v),c,s,I)$$

and $\mathfrak{Lb}(r,\cdot) := 0$ *is a feasible lower bound.*

Proof. Let $v, w \in V(G)$ and $\omega \in \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$, $(c',s') \in \mathbb{R}_{\geq 0}^{\{c,s\}}$, as well as $\emptyset \subsetneq J' \subsetneq J \subseteq T$. Let $\nu \in \mathrm{opt}(w,s',c',J \setminus J' + (v,\omega))$. Our goal is to show that

$$\chi(\mathfrak{Lb}(v,(\omega_c,\omega_s),J)) \leq \chi\left(\mathfrak{Lb}(w,(c',s'),J') + \nu\right).$$

First we observe that for each $t \in J'$, we have that $\nu_t \geq 0$. Then for each $t \in J \setminus J'$, we have $\nu_t \geq \min\limits_{\sigma \in \Sigma_t} \mathrm{OPT}(w,c',s',\emptyset + (t,(0,0,\sigma)))$, because any $w-t$-path in a solution ending up with objective value ν, is also considered in this optimum. With the same reason, we have for all $t \in T \setminus J$ that $\nu_t \geq \mathrm{OPT}(w,c',s',\emptyset + (v,(0,c,s))$. So, for all $t \in T$, we define

$$b_t := \begin{cases} 0 & \text{if } t \in J' \\ \min\limits_{\sigma \in \Sigma_t} \mathrm{OPT}(w,c',s',\emptyset + (t,(0,0,\sigma))) & \text{if } t \in J \setminus J', \\ \mathrm{OPT}(w,c',s',\emptyset + (v,(0,c,s)) & \text{otherwise} \end{cases}$$

and know that $b \leq \nu$ entrywise. Then we observe that

$$\mathfrak{Lb}(v,(c,s),J)_t \leq \begin{cases} \min\limits_{\sigma \in \Sigma_t} f(\mathit{dist}(t),0,\sigma) & \text{if } t \in J' \\ \min\limits_{\sigma \in \Sigma_t} f(\mathit{dist}(w),c',s') + \mathrm{OPT}(w,c',s',\emptyset + (t,(0,0,\sigma))) & \text{if } t \in J \setminus J', \\ f(\mathit{dist}(w),c',s') + \mathrm{OPT}(w,c',s',\emptyset + (v,(0,c,s)) & \text{otherwise,} \end{cases}$$

by applying inequality (*) for each entry $t \in T$. Thus, we have

$$\mathfrak{Lb}(v,(c,s),J) \leq \mathfrak{Lb}(w,c',s',J') + b \leq \mathfrak{Lb}(w,c',s',J') + \nu \text{ entrywise.}$$

We conclude

$$\chi(\mathfrak{Lb}(v,(c,s),J)) \leq \chi(\mathfrak{Lb}(w,c',s',J') + \nu).$$

□

This means that we can construct a feasible lower bound if we are able to compute a function f as in 4.3.1. So our goal will be to construct this function in such a way that we can compute it as fast as possible.

We start by building the base of our graph L. We take $V(L) := D := \{d_1,\ldots,d_k\}$ as vertex set ordered by increasing value. Then we add edges (d_i,d_j) and (d_j,d_i) for

$i = 1, \dots, k-1$ and $j = i+1, \dots, k$ to retrieve a directed path. We use this path as a base to do the same construction as for our Hanan graph, just without blockages. That is, we build a layered model from there by copying the path once for every layer. Then we replace the edges we currently have by sets of edges, one for each wire width and spacing that are allowed on the layer. Finally, we insert edges for the allowed vias. We then use this graph to apply the construction from Chapter 2.3.2 with a root $\tilde{r}$ connected to each copy of $d_1 = 0$.

With this graph, we define $f(d, c, s) := \mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (d, (0, c, s)))$ as the optimum delay of a path from $\tilde{r}$ to d that has a downstream capacitance of c at d and at most slew s.

Lemma 4.3.2. *Let $f(d, c, s) := \mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (d, (0, c, s)))$ as constructed above. Then f fulfills the inequality (*) from Lemma 4.3.1.*

Proof. Let $v, w \in V(G)$ and $\omega \in \Omega \times \mathbb{R}_{\geq 0}^{\{c,s\}}$, $(c', s') \in \mathbb{R}_{\geq 0}^{\{c,s\}}$. First, we observe that $\mathrm{OPT}_L(\mathit{dist}(w), s', c', \emptyset + (\mathit{dist}(v), (0, c, s))) \leq \mathrm{OPT}_H(w, s', c', \emptyset + (v, (0, c, s)))$. Note that we compare optima in different graphs. We can show this by taking an optimum solution in H and then projecting it into L. This means that we map vertices in H to their respective distance and edges to the corresponding edge between the two distances. We would end up with a solution in L that has the same value as the solution in H, so the optimum in L must be lower.

Now we just need to use the fact that

$$\begin{aligned}\mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (\mathit{dist}(v), (0, c, s))) \\ \leq \mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (\mathit{dist}(w), (0, c', s'))) \\ + \mathrm{OPT}_L(\mathit{dist}(w), s', c', \emptyset + (\mathit{dist}(v), (0, c, s))),\end{aligned}$$

because this combined solution on the right-hand side is one solution that is considered in the optimum on the left-hand side. Using our first inequality and the definition of f implies

$$f(\mathit{dist}(v), c, s) \leq f(\mathit{dist}(w), c', s') + \mathrm{OPT}_H(w, s', c', \emptyset + (v, (0, c, s))).$$

□

With such a function, we would have to compute the lower bound for each label by solving an instance on this path graph. This is too slow to be of help in our algorithm. So our goal is to build a table for f and compute approximate values in constant time by linearly interpolating between the entries. Then, if we are solving multiple instances, we also want to generalize the source delay and the distances. So we end up with a general approximate lower bound that we can compute during a preprocessing step.

Lemma 4.3.3. *Let* obj *be a branch based objective on $\Omega := \mathbb{R}_{\geq 0}^T$. Let χ be an evaluation operator on Ω. Let $\alpha \geq 1$ and $f : D \times \mathbb{R}_{\geq 0}^{\{c,s\}} \to \mathbb{R}_{\geq 0}$ with*

$$\begin{aligned}\mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (d, (0, c, s))) \leq f(D, c, s) \\ \leq \alpha \, \mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (d, (0, c, s))).\end{aligned}$$

If we use f to construct a lowerbound $\mathfrak{Lb}_\alpha$ as in Lemma 4.3.1 (note that this is not a feasible lower bound), then the following variant of Lemma 3.2.4 holds:

Let $\mathcal{L}$ be the set of all labels that are added to L and never removed during the algorithm. In each iteration the following holds: Each label $l \in \mathcal{L}$ with

$$\omega_{\mathit{eff}}(l) < \frac{M}{\alpha}$$

is also in L and permanent, where $M := \max_{l' \in L \text{ permanent}} \omega_{\mathit{eff}}(l')$.

Proof. Let $(v, e, P, \omega, I) = l \in \mathcal{L}$ be a label and $(w, e', P', \nu, J) = p \in V(A_l)$ an ancestor of l. Out first goal will be to prove that

$$\omega_{\mathit{eff}}(p) \leq \alpha\, \omega_{\mathit{eff}}(l).$$

For all $t \in T$, we define

$$g_t := \begin{cases} \min_{\sigma \in \Sigma_t} \mathrm{OPT}_L(r, \mathit{caplim}(r), s_r, \emptyset + (t, (0, 0, \sigma))) & \text{if } t \in T \setminus J, \\ \mathrm{OPT}_L(r, \mathit{caplim}(r), s_r, \emptyset + (w, (0, \nu_c, \nu_s)) & \text{otherwise,} \end{cases}$$

as well as

$$a_t := \begin{cases} \min_{\sigma \in \Sigma_t} \mathrm{OPT}_L(r, \mathit{caplim}(r), s_r, \emptyset + (t, (0, 0, \sigma))) & \text{if } t \in T \setminus I, \\ \mathrm{OPT}_L(r, \mathit{caplim}(r), s_r, \emptyset + (v, (0, \omega_c, \omega_s))) & \text{otherwise} \end{cases}$$

and

$$b_t := \begin{cases} 0 & \text{if } t \in T \setminus I, \\ \min_{\sigma \in \Sigma_t} \mathrm{OPT}_L(v, \omega_c, \omega_s, \emptyset + (t, (0, 0, \sigma))) & \text{if } t \in I \setminus J, \\ \mathrm{OPT}_L(v, \omega_c, \omega_s, \emptyset + (v, (0, \nu_c, \nu_s))) & \text{otherwise} \end{cases}$$

Then $a \leq \mathfrak{Lb}_\alpha(v, (\omega_c, \omega_s), I)$ by our assumption on f and $g \leq a + b$ entrywise, because every path from the combination of a and b is also considered in the optima in g. We use this to compute

$$\begin{aligned} \omega_{\mathit{eff}}(p) &= \chi(\nu_o + \mathfrak{Lb}_\alpha(w, (\nu_c, \nu_s), I)) \\ &\leq \chi(\nu_o + \alpha g) \\ &\leq \alpha\, \chi(\nu_o + a + b) \\ &\leq \alpha\, \chi(\omega_o + a) \qquad (**) \\ &\leq \alpha\, \chi(\omega_o + \mathfrak{Lb}_\alpha(v, (\omega_c, \omega_s), I)) = \alpha\, \omega_{\mathit{eff}}(l). \end{aligned}$$

For inequality (**), we use the same argument as in the proof of Lemma 4.3.1 that none of the paths from v to w, nor the paths from v to $t \in I \setminus J$ can be realized with a lower cost than the path in L. Thus, $\omega_o \geq \nu_o + b$ entrywise.

After the first iteration, our Lemma certainly holds. Now assume it holds up to iteration i, but it does not hold in iteration $i+1$. This means there is a label $l \in \mathcal{L}$ with $\frac{M_i}{\alpha} \leq \omega_{\mathit{eff}}(l) < \frac{M_{i+1}}{\alpha}$ that is not permanent or in L.

Let a be an ancestor of l. Then $\omega_{\mathit{eff}}(a) \leq \alpha\, \omega_{\mathit{eff}}(l) < M_{i+1}$. So a can not be in Q, because otherwise it would have been chosen by the algorithm. This means that none of the ancestors of l can be in Q. Since they are never removed by the algorithm and the initial labels must be in L, they must all be in L and permanent. But then $l \in Q$. So $\omega_{\mathit{eff}}(l) \geq M_{i+1} > \omega_{\mathit{eff}}(l)$, because it has not been chosen. This is a contradiction. □

This means that if we can only approximate the function for Lemma 4.3.2 by a factor α, we are still able to compute an optimum solution. We can achieve this, by waiting until the lowest valued permanent label and the highest valued label at the root differ by a factor larger than α. Then we know that the lowest valued label has minimum value. If we are guessing the slews, we also have to wait for this approximation factor to the lowest value in order to find the optimum solution.

This construction is sufficient for a per-instance lower bound table. For a general table, where we need to lower bound the source resistance and interpolate between predetermined distances, we can not guarantee the first inequality in 4.3.3. It could happen that the general distances are a slightly better discretization of the line. So we might underestimate the delay.

We are able to prove an even weaker result though, that still allows us to find optimum solutions:

Lemma 4.3.4. *Let* obj *be a branch based objective on* $\Omega := \mathbb{R}^T_{\geq 0}$. *Let* χ *be an evaluation operator on* Ω. *Let* $\alpha \geq 1$ *and* $f : D \times \mathbb{R}^{\{c,s\}}_{\geq 0} \to \mathbb{R}_{\geq 0}$ *with*

$$f(D, c, s) \leq \alpha\, \mathrm{OPT}_L(\tilde{r}, \mathit{caplim}(r), 0, \emptyset + (d, (0, c, s))).$$

If we use f *to construct a lowerbound* $\mathfrak{Lb}_\alpha$ *as in Lemma 4.3.1 (note that this is not a feasible lower bound), then the following holds:*

Let $\mathcal{L}$ *be the set of all labels that are added to* L *and never removed during the algorithm. In each iteration, if* $l \in \mathcal{L}$ *encodes a complete solution and*

$$\chi(\omega(l)_o) < \frac{M}{\alpha},$$

where $M := \max_{l' \in L \text{ permanent}} \omega_{\mathit{eff}}(l')$, *then* l *is in* L *and permanent.*

Proof. The proof is a special case of the proof of Lemma 4.3.3. Let $(r, e, P, \omega, T)l \in \mathcal{L}$ be a label that encodes a complete solution. Let $(w, e', P', \nu, J) = p \in V(A_l)$ be an ancestor of l and $\mu \in \mathrm{OPT}_H(r, \mathit{caplim}(r), s_r, J + (w, \nu))$. Then $\mathfrak{Lb}_\alpha(w, (\nu_c, \nu_s), J) \leq \alpha\mu$ entrywise. Hence,

$$\begin{aligned}\omega_{\mathit{eff}}(p) &= \chi(\nu + \mathfrak{Lb}_\alpha(w, (\nu_c, \nu_s), J)) \\ &\leq \alpha\, \chi(\nu + \mu) \\ &\leq \alpha\, \chi(\omega).\end{aligned}$$

Using the same argument as in the proof of Lemma 4.3.3, we conclude that if $M > \alpha\,\chi(\omega)$, then all ancestors of l and l itself must have been made permanent already. □

Again, we can compute the optimum solution by running the algorithm until the factor between the lowest valued and highest valued permanent solution at the root exceeds α.

It remains to find a bound on the error that is induced by an approximation. However, this depends on the exact delay model that is used and may be upper bounded numerically.

4.4 Topology Restriction

One of the key sources of possible improvements in our algorithm is that we do not fix the topology of the solutions. Yet, it might be helpful to restrict the topology sometimes to speed up the algorithm. We can use a restricted topology, if we can not compute a solution with no restrictions in reasonable time, or to provide an upper bound. Sometimes the positioning of the sinks may already dictate parts of the topology, or a designer wants to enforce certain parts of the topology.

To be able to do this, we present an efficient way to add a topology restriction to our problem. We achieve this by using merge arborescences, which are a data structure that tells us that certain sink sets have to be merged, before they may be merged with other sink sets.

Definition 4.4.1. *Let a net $\{r\} \cup T$ be given. A* merge arborescence *is an arborescence M rooted in r and with T as its leaves and bijective map $\phi : V(M) \to Img(\phi)$, such that*

- *each vertex $v \in V(M) \setminus T$ has outdegree $\delta^-(v) \geq \min(|T|, 2)$,*
- $\phi(t) = \{t\}$
- *and recursively $\phi(v) := \bigcup_{w \in \delta^-(v)} \phi(w)$ for $v \in V(M) \setminus T$*

Given a merge arborescence (M, ϕ), we apply it to the TMCMAP by only allowing solutions such that we can transform the topology of the solution to M by contracting a (unique) subset of its edges.

We can enforce this in our algorithm by choosing for each sink set $I \in 2^T$ a representant $R(I) \in V(M)$. Denote by $p(v)$ the unique parent of v for each $v \in V(M) \setminus \{r\}$. We require that $R(\phi(v)) = v$ for each $v \in V(M)$ and $\phi(R(I)) \subseteq I \subseteq \phi(p(R(I)))$ for each $I \subsetneq T$. In a practical implementation, the representants can be chosen when the sink set occurs for the first time.

Then for two labels that we want to merge, we require that their sink sets $I, J \subseteq T$ are disjoint and additionally that $p(R(I)) = p(R(J))$, which we can check in constant time.

We can even restrict the area, in which certain sink sets are allowed to move. In many cases, we can speedup the algorithm this way by not allowing labels to be created that would turn out to be suboptimal later. It may also be advantageous in some practical

use cases. If we need to connect some sinks through a small channel between two blocks for example. A designer may want to force the sinks to be joined before they pass the channel. The area restriction can be achieved, by associating it to the vertices of M. A label with sinks set I can then be not allowed to leave the area associated to $p(R(I))$.

4.5 Additional Heuristics

We can additionally speed up our computation by the simple heuristics that we explain in this section. We separate them into those that are present in our implementation and those that have not shown to be promising in our evaluations.

4.5.1 Implemented Heuristics

The heuristics that proved to be helpful both have an influence mainly on repeater placements. While working on the implementation, we have discovered two problems.

Repeater Map

The first one regards, where and how many repeaters are placed. When crossing a large blockage, the best positions for repeaters tend to be the ones just before or after the blockage. If the repeaters have to be placed, a placement cost does not really have an influence on whether a repeater is placed. This is, why it happens that multiple parallel wires end up on the same point on the other side of a blockage and all of them place repeaters there. While this is no problem in the model, a final placement must be legal. However, legalizing many repeaters from the same spot, while keeping the timing properties, is a hard problem in itself. So we either require another very specialized and possibly slow postprocessing, or we destroy the timing during legalization.

However, it is often actually possible for parallel segments to cross the blockage in different spots. We can enforce this by limiting the number of repeaters that may be placed in a certain area.

We achieve this by laying a tiling over our graph and creating a map from each vertex to each tile. Then we limit the number of repeaters per tile (we use 2 as a limit in practice). We add one entry per tile to the objective. If the upper bound is two, we only need 3 bits per entry. We store the map as an array of unsigned integers. If we view this as a contiguous range of bits, every 3 bits represent one entry. Then we can add two maps by adding the integers. Checking for invalid placements after adding two maps amounts to checking every third bit for a 1 or even faster by computing the logic AND between the integers and a fixed set of masks. The value of a repeater map is an additional 0 if it is valid and ∞ if not.

We can leave this map out of our dominance relation. In this case, it becomes a heuristic. Otherwise, we can additionally compare the entries of the map during dominance checks. Then it encodes the restriction that there must be at most two repeaters per tile.

Repeater Penalty

The second heuristic tries to discourage implausible repeater placements. Since the algorithm tries all possible repeater placements, it also tries out some that are unlikely to lead to good solutions, but are not dominated. A prime example of this is placing repeaters that have a higher input pin capacitance than the current downstream capacitance. This typically leads to the solutions assuming a high input slew at the repeater. So they are unlikely to be dominated. We discourage these solutions by adding a penalty in these cases. We can not remove them, because they might be necessary to cross large blockages. The penalty we came up with depends on the decrease in downstream capacitance Δ_c, the current vertex and the currently connected sink sets. It is computed as

$$\mathit{penalty}(\Delta_c, v, I) := \max\left\{0, B - \Delta_c \cdot \left(1 + \frac{|I|}{|T|} \cdot R \cdot \left(1 - \frac{\mathit{dist}(v)}{D}\right)\right)\right\},$$

where $B \in \mathbb{R}_{\geq 0}$ is a base penalty, R is a source resistance, used to approximate delay through the source, and D is an upper bound on the distance, for which we scale based on the resistance. We only use it if $\mathit{dist}(v) < D$. Otherwise, we use

$$\mathit{penalty}(\Delta_c, v, I) := \max\left\{0, B - \Delta_c\right\}.$$

We do not apply this penalty if the repeater is an inverter and changes the polarity from invert to ident. We have to place an inverter on the path to the root and we want the dynamic program to take the best inverter.

The reasoning behind these formulas is the following. In general, we want to discourage adding repeaters if it increases the downstream capacitance or only decreases it slightly. However, if we are close to the source we want to recognize the impact that a capacity change has on the source delay. So we scale the impact of the source delay based on the number of connected sinks and distance to the source.

4.5.2 Dropped Heuristics

The heuristics that did not prove helpful focus on the reduction of labels that are kept. We tested them in an evaluation phase and did not include them into our final implementation. The problem was that they did not decrease the running time. Some even increased the running time. And they were leading to a slight degradation in the final results.

We explain them nonetheless, so that they are not reimplemented in the same way for future research.

Dominance Cache

Since we allow placing all repeaters at almost all positions, we will be propagating through all of them at the same time. They originate from the same label and end up at the same vertex, only differing in the used repeater and guessed input slew. The idea

of the dominance cache is to eliminate those labels that are already dominated within this set of labels that are created by propagating a single label.

Let $E_1, \ldots, E_a$ for some $a \in \mathbb{N}$ be groups of parallel edges. We assume that two edges are identical if they contain the same objective function. One way to think of it is as two edges encoding the same repeater. We assume that they are maximal and distinct. Then we choose a capacitance delta and limit $\delta_c, C_{max} > 0$ and a slew delta $\delta_s > 0$. For all pairs $i, j \in \mathbb{N}$ with $\delta_c \cdot i \leq C_{max}$ and $j \leq S$ and for all edges of the same group $e \in E_k$, we compute all objective values that we could get using $\delta_c \cdot i$ and $\delta_s \cdot j$ as input. Then we compute which of them are dominated. In the end, we store for each (e, k, i, j) the input slew values that were not dominated.

In the algorithm, when we propagate through e, we find our group k by the vertex from which we are starting and the closest values for i and j. Here we search for the closest lower capacitance and the closest higher slew. Then we only create labels for the slew values that we have cached, slews that are below the covered range and the accurate value of the upper bound.

This technique of course only works if the objective function and evaluation operator allow that we can precompute the dominance.

Unfortunately the speedup gained through this technique was negligible and the result quality degraded slightly.

Capacitance Decrease Estimate

If the objective models only the maximum delay (or worst slack), then we can leave out some merges. Suppose, we want to merge two labels. One with a high value, call it l, and one with a low value, call it k. Due to the nature of the maximum function, the objective cost of the merged solution will be the higher of the two values. Furthermore, we know that a lower downstream capacitance at this merge will lead to a lower objective values upstream.

These two observations lead to the following conclusion. If we are able to reduce the capacitance on the branch of k, while keeping at least the minimum slew of both labels and not increasing the objective value of k over the value of l, then the resulting solution will be better than the one we would get by directly merging.

This leads to an idea for a speedup heuristic. We compute a table that can give us an upper bound on the delay we need to spend in order to reduce the capacitance to a certain value, while not decreasing the slew. When we want to merge two labels, we use the table to look up how much delay we need to spend, in order to decrease the capacitance of the label with the lower value. If this delay is less than the absolute difference between the two, we do not merge. Otherwise, we merge as usual.

Unfortunately, this also had only a small impact on the running time, while taking quite a while to compute itself. Also, it encourages solutions that are not good in practice. An example are solutions that have many repeaters in the same spot.

Heuristic Label Pruning

The running time depends mostly on the number of labels that are created before the algorithm finishes. However, most of the created labels never lead to good solutions.

So a natural idea is to restrict the number of labels at each vertex or in each bucket (see 4.1). We restrict the number of labels per bucket here. In order to achieve this, we have to rigorously prune labels that we assume to not be promising. We can not just take the labels with the best objective cost or effective objective, because we may only keep labels with too high capacitance this way. However, it is also not helpful to always prefer labels with low capacitance on uncritical branches, because this leads to too many repeaters being placed.

The selection criterion that we tested, was to always keep the best objective cost and best capacitance label. Then we would sort the remaining labels by objective cost times capacitance, normalized by the values of the two already selected labels, and keep the best k labels (for some $k \in \mathbb{N}$).

However, this increased the number of iterations that were required drastically. Which means that we were removing the wrong labels.

4.6 Iterative Clustering

Sometimes it is not enough to restrict the topology, but we rather want to reduce the size of the instances. In this case, we can try to cluster sinks and use the clusters as subinstances. In this section, we will describe an iterative clustering approach.

The first question that we should answer is what goals are we trying to achieve with our clustering? Ideally, we would want to partition the sinks into subsets of at most constant size and at least one of these sets should contain more than one sink. For each of these sets we want to be able to determine a new root, such that there is an optimum solution which connects each of the roots to their respective sink set.

Of course this problem is as hard as the original problem. Even for simple formulations like the linear delay Steiner arborescence problem, we can not hope to find a constant factor approximation algorithm, as we can use such a clustering algorithm to find a constant factor approximation for the linear delay Steiner arborescence problem. Since we still want to use clustering approaches, we will tackle the problem heuristically.

Intuitively, it is better to cluster sinks that are in some sense close to each other. For VLSI-design the Steiner netlength or spanning tree netlength of the sink sets is a good measure. It helps to keep netlength and routing congestion down, as well as the number or size of repeaters. This also influences power and area consumption. However, if we connect a timingwise critical sink very deep in the repeater tree, we will likely create (timing) violations. This is why we have to consider timing as well.

For this purpose, we take inspiration in a variation of Huffman coding [Huf52] that is used in [HR13]. It can be used to compute shorter solutions to the linear delay Steiner arborescence problem than standard Huffman coding. In this algorithm, we first determine an upper bound on the slack of each sink. It is defined as the required arrival

time minus the linear delay of connecting the sink directly to the root. Then we iterate the following: We find the minimum value H such that the instance remains feasible, if we reduce the slack to H on all sinks with higher slack. On the sinks with slack at least H, we compute a (almost) perfect matching with low (or minimum) weight. Then we use the normal Huffman coding step on each of these sink-pairs, except that we place the new vertices on the median of the two sinks and the root.

We want to do something similar, but instead of clustering two sinks, we will allow larger clusters. Furthermore, we do not have a fixed timing cost per merge. So we choose our cutoff value heuristically.

Let $rat(t, s)$ be the required arrival time for each sink $t \in T$ and input slew $s \in [0, S_t]$.

We define a *sink candidate* as a tuple $C := (pos(C), pol(C), c(C), s(C), rat_C)$ consisting of a point on the chip $pos(C)$, a polarity with respect to the root $pol(C)$, a capacitance $c(C) \in \mathbb{R}_{\geq 0}$, a slew limit $s(C) \in [0, S]$ and a required arrival time function $rat_C : [0, s(C)] \to \mathbb{R}$ that computes a required arrival time for an input slew. For a sink candidate C, we denote by $P(C, s)$ an estimate for the delay of an optimum buffered route from r to C with slew limit s. A *virtual sink* is a pair $(I, \mathcal{C})$ of a subset of the sinks $\emptyset \neq I \subsetneq T$ and a set $\mathcal{C}$ of sink candidates with $pos(C) = pos(C')$ for each $C, C' \in \mathcal{C}$. Now we define the slack of a sink candidate $(I, \mathcal{C})$ as

$$\mathrm{sl}(I, \mathcal{C}) := \min_{C \in \mathcal{C}} \min_{s' \in [0, s(C)]} [rat_C(s) - P(C, s')]$$

During the heuristic, we will maintain a set Q of virtual sinks. Initially, Q will consist of virtual sinks $\{\{t\}, \{C_t\}\}$ for the original sinks $t \in T$. Here C_t is the sink candidate consisting of the same attributes of the original sink. Then we iterate the following.

We choose a cutoff value for the slacks. Virtual sinks above that cutoff value will be considered *uncritical*. To the uncritical sinks, we apply a clustering algorithm that allows to limit the sizes of the clusters. Then we choose a root position for each the sets. Each cluster, together with its root will be used to construct a modified instance of the TMCMAP.

The modified instance for a cluster will be constructed as follows. We build our Hanan graph on the positions of the virtual sinks in the cluster and the root position. Then we use this to construct our graph for the TMCMAP. For each virtual sink and each of its sink candidates, we add a vertex and an edge from the corresponding vertex in our graph to that vertex. Initial labels at that specific vertex will then be generated from the values of its sink candidate. For the root, we allow connecting to the root vertex from each layer and polarity at the root position. When a label is propagated to the root, we first compute the sink candidate C that this would generate except for the *rat* function. Then the delay through the root will be $P(C, s)$, where s is the slew of the label.

We also do not stop when we find the first solution. Instead, we compute a set of non-dominated solutions. Each of these labels will form a sink candidate for the next iteration. To limit the number of candidates, we can require that the capacitance of the labels differs by at least some amount. Practical experiments also show that after a while, the capacitance reduction is only achieved by adding chains of repeaters at

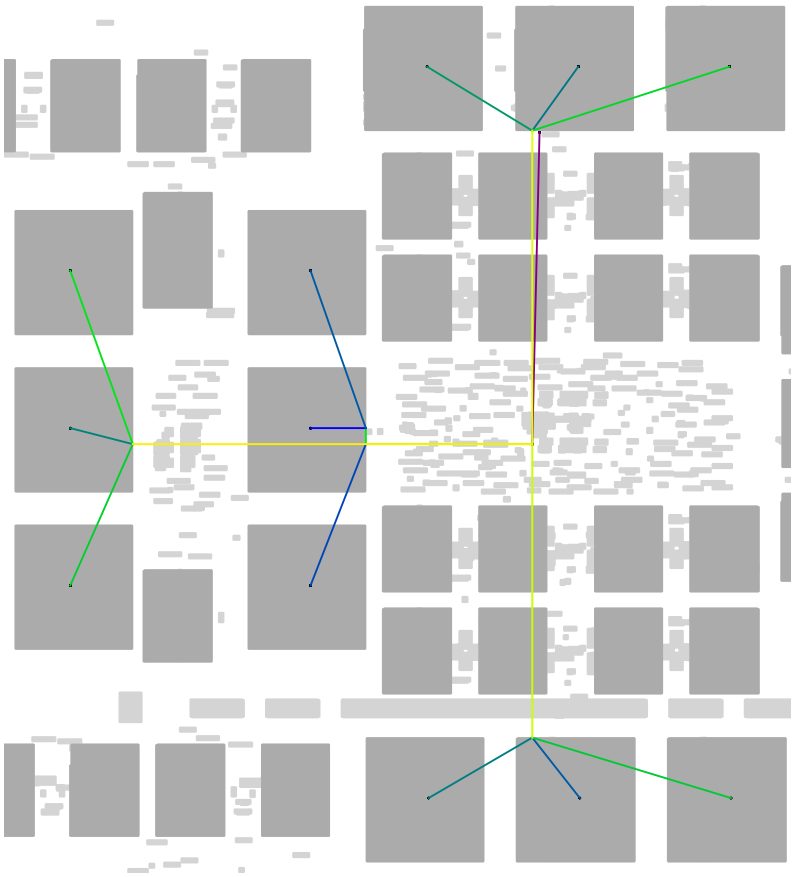


Figure 4.5: An example for an iterative clustering used to compute the buffered route in Figure 3 in the introduction. Black dots mark the sinks, the source is in the center. Each cluster sink is connected to its cluster source by an edge. The colours indicate the criticality (from uncritical blue through green, yellow, red to purple for very critical).

the virtual source. Candidates that are generated from these solutions with chains of repeaters at the virtual source do not transfer new information to the next iteration. If they are necessary, the chains would be computed in the next iteration. Furthermore, the repeaters may be spread, in the next iteration. Thus, we can stop the computation, when the delay-increase per capacitance reduction exceeds a given threshold. Finally, the roots from this iteration become virtual sinks, with additional candidates. An example of an iterative clustering computed with this process is shown in Figure 4.5.

It remains to specify, how we are choosing the cutoff value and how we compute a clustering. First we explain how we choose the cutoff value. Driven by the idea that we can afford more errors in the topology for sinks with higher slack, we choose a cutoff value $\sigma \in (0, 1)$ and consider the range of slacks $R := [\min_{t \in Q} \mathrm{sl}(q), \max_{t \in Q} \mathrm{sl}(t)]$. We want to choose sinks from the top σ-fraction of R. Sometimes, we will have outliers though. And in that case, we will likely select too many, or too few sinks. So we want to skew this cutoff a little into the direction of the median. We do this by taking the

arithmetic mean of the median value and the boundary of the σ-fraction of the interval.

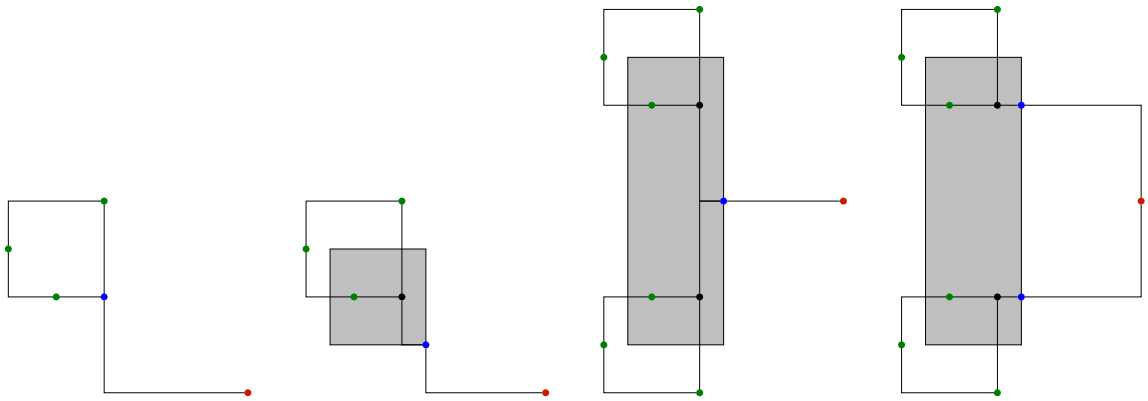
Then we need to compute a clustering. One approach to doing this, is to solve a Capacitated Tree Covering Problem with Edge Loads. This problem will be discussed in detail in Chapter 5. Here, we will require the following properties and consider the exact way, we can compute this as a black box. We want to be able to assign a load b to the sinks and an upper bound on the total load of a cluster u, as well as a cluster price γ. Our clustering black box should give us a forest containing the sinks T with trees $\{C_1, \dots, C_k\}$ that approximately minimizes $\sum_{i=1,\dots,k} \mathit{length}(C_i) + k\gamma$ such that $\mathit{length}(C_i) + |C_i|b \leq u$.

Using the sink loads and upper bound, we can limit the number of sinks in each cluster. However, if the length of a tree is small, we can expect the Hanan graph we are using to have few vertices. So we can allow instance sizes to grow in this case. We can use the parameters to allow the algorithm to increase the number of sinks per cluster slightly, if the corresponding tree has a low length. Furthermore, we can use the cluster price γ to change the maximum edge length inside the clusters. Next, we describe how we select these parameters.

Let a (weak) upper bound k on the cluster size be given, as well as a maximum distance of the vertices in the Hanan graph D. We start by computing minimum length L of a tree on all sinks and the root. Our base unit l will be the minimum of $\frac{L}{|T|}$ and $\frac{D}{2}$. To select the cluster cost γ, we search for the minimum multiple of l for $b < 1$ and $u > L + b|T|$, such that the average size of a nontrivial cluster is at least $\frac{3}{4}k$. Let L_i for $i = 1, \dots, k$ denote the maximum length of a cluster with i sinks, or γ if there is no such cluster. Then we use a binary search to find the minimum value for $b \geq l$ such that using $u := \max_{i=1,\dots,k} i \cdot b + L_i$ no cluster has more than k sinks. Finally, we add a budget of $(k+1) \cdot l$ to u to allow for trading off small amounts of netlength with additional sinks.

The last remaining question is how to choose the root positions. We want to maintain the property that the root of a cluster lies on a shortest path from each sink to the root of the whole instance, such that we do not introduce arbitrary detours. The first step is to compute the projection of the instance root to the bounding box of the cluster. If this position does not lie on a blockage, we use it. Otherwise, we need to choose a different position, because we might force solutions to the clusters subinstance to go onto the blockage with high capacitance, such that in a subsequent iteration, we need to leave the blockage again. Choosing the projection of the source onto the bounding box instead may lead to multiple roots at the same position, if the bounding boxes of multiple clusters intersect the same blockage. So instead, we choose the closest point to our first candidate on the border of the blockage that still lies on a shortest path from the root to all sinks. This process is illustrated in Figure 4.6d.

The need for a clustering in this procedure also motivated our research on the Capacitated Tree Cover Problem. We developed a new $\mathcal{O}(n \log n)$ time 3-approximation algorithm for the Capacitated Tree Cover Problem, which we will present in the next chapter.



(a) The projection of the root onto the bounding box of the sinks is not blocked.

(b) The projection of the root onto the bounding box of the sinks is blocked. We instead project onto the blockage.

(c) The process from 4.6b places both roots on the same spot.

(d) By picking the closest point to our first candidate on the border of the blockage that still lies on a shortest path from the root to all sinks, we find separate root locations.

Figure 4.6: Configurations that are taken into account when we are choosing a location for a cluster root. The chosen root is marked in blue. Blockages are gray.

5 A Fast 3-Approximation for the Capacitated Tree Cover Problem with Edge Loads

In this chapter, we present a new 3-approximation algorithm for the Capacitated Tree Cover Problem with Edge Loads. A shortened version of this chapter has been accepted as a single-authored paper at the SWAT 2024 conference [Roc24]. It is published as: Benjamin Rockel-Wolff. "A Fast 3-Approximation for the Capacitated Tree Cover Problem with Edge Loads". In: 19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024). Ed. by Hans L. Bodlaender. Vol. 294. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 39:1–39:14.

The full version in this thesis contains the full proofs and an additional estimate for a special case that did not fit into the conference paper. It is also available on arXiv.

The capacitated tree cover problem with edge loads is a variant of the tree cover problem, where we are given facility opening costs, metric edge costs and loads, as well as vertex loads. We try to find a tree cover of minimum cost such that the total edge and vertex load of each tree does not exceed a given bound. This variant has, to our best knowledge, not been considered so far.

We start by giving a formal definition of the problem and reviewing the literature on similar problems in Section 5.1.

We then take a closer look at solutions to our problem in Section 5.2. Let $G = (V, E)$ be the input graph. We observe that each induced subgraph $F[A]$ of a solution F (for $A \subseteq V$) can have at most $|A| - 1$ edges and at most $|A|$ minus total load edges. We use this fact to derive a LP-formulation of the problem. Additionally, we derive a second equivalent formulation by scaling the values on the edges.

Although these LP-formulations have an exponential number of inequalities, we are able to present a $\mathcal{O}(m \log n)$ time greedy algorithm that can solve the second formulation optimally in Section 5.3. The main running time is spent in ordering the edges, while the greedy value assignment can be implemented to run in almost linear time (linear times inverse Ackermann function).

We spend the remainder of this section on proving the algorithm's correctness. In a first part, we show that the solution returned by the algorithm is a solution to the LP. In the second part, we show that it is optimum. We prove this by contradiction. Assuming that our solution is not optimum, we pick an optimum solution that maximizes the index i of the first edge, where the two solutions differ. Among these, it should minimize the difference in that edge. It remains to show that we can increase the value of the i-th edge in the optimum solution without changing the objective or violating the constraints, by decreasing the value of an edge with a higher index.

Once we have obtained an optimum solution, we round up all edges that exceed a value α that we determine later and round the remaining edges down in Section 5.4. This may lead to components that violate the load bounds. We fix the solution by applying a well known splitting technique that is also used in [MV08] and [TT22]. While the overall approach of solving the LP, then rounding and splitting is similar to the one used in [TT22], edges with load require a different algorithm for solving the LP. Furthermore, we need to be more careful in the analysis of our rounding step.

We then thoroughly analyse the structure of such a rounded solution to find bounds on the increase in the number of components that we get by rounding and splitting. The components in a rounded solution are divided into three categories: Singletons, good trees that do not exceed the bound, and large trees that have to be split. For each rounded up edge, we closely examine, in which tree it can occur and bound its contribution to the total number of components. For each rounded down edge, we examine the configurations of trees that it may be incident to and bound the edge's contribution to the total number of components. In a final step, we sum over all edges and their estimates to derive our final bound.

We are able to show that with $\alpha = \frac{2}{3}$, we can bound the number by 3 times the fractional number of components in the LP solution. The edge cost itself is no problem, because it can only increase by a factor of 1.5 (for $\alpha = \frac{2}{3}$) with our rounding strategy.

In total, we are able to show our main theorem:

Theorem. *There is a 3-approximation algorithm for the capacitated tree cover problem with edge loads that runs in time* $\mathcal{O}(m \log n)$.

Additionally, we show that we can get a better approximation guarantee if all edge loads are below $\frac{1}{2}$ and we use $\alpha = \frac{1}{2}$. Then the approximation guarantee is bounded by $2 \cdot (1 + u_{max})$, where u_{max} is the highest edge load that appears in the instance.

Finally, in Section 5.5, we give a series of examples that prove that the integrality gap of our LP is at least 3. This shows that the integrality gap of the LP is exactly 3, which means that we can not improve the rounding step in general.

5.1 Introduction

Graph cover problems deal with the following base problem. Given a graph G, the task is to find a set of (connected) subgraphs of G, the cover, such that each vertex of G is contained in at least one of the subgraphs. Usually, the subgraphs are restricted to some class of graphs, like paths, cycles or trees. Different restrictions can be imposed on the subgraphs, like a maximum number of edges, or a total weight of the nodes for some given node weights. Recently, Schwartz [Sch22] published an overview of the literature on different covering and partitioning problems.

We consider the capacitated tree cover problem with edge loads. It is a variation of the tree cover problem that has not been studied so far to the best of our knowledge.

In the capacitated tree cover problem with edge loads, we are given a complete graph $G = (V, E)$, metric edge costs $c : E \to \mathbb{R}_+$, vertex loads $b : V \to [0, 1)$, metric edge loads $u : E \to \mathbb{R}_{\geq 0}$ with $u(e) < u(f) \Rightarrow c(e) \leq c(f)$, and a facility opening cost $\gamma \geq 0$. The task is to find a number of components $k \in \mathbb{N}_{\geq 1}$ and a forest F in G consisting of k trees minimizing

$$\sum_{e \in E(F)} c(e) + \gamma k,$$

such that each tree T_i has total load

$$u(T_i) := \sum_{e \in E(T_i)} u(e) + \sum_{v \in V(T_i)} b(v) \leq 1.$$

The capacitated tree cover problem with edge loads is closely related to the facility location problem with service capacities discussed by Maßberg and Vygen in [MV08]. Their problem uses Steiner trees to connect the nodes, not spanning trees. Furthermore, in their case edge cost and edge load are the same. They make use of this fact to prove a lower bound on the value of an optimum solution. Both problems have important practical applications in chip design. In [Hel+11] they are called the sink clustering problem and used for clock tree construction. In [Bar14] they are used for repeater tree construction. In these applications terminals and edges have an electrical capacitance. A source can drive only a limited capacitance. Edge cost and capacitance usually are proportional to the length of an edge. As the edge length is given by the l_1-distance between its endpoints, this naturally matches our problem.

Our problem is also related to other facility location and clustering problems, like the (capacitated) k-center problem ([HS85], [KS00]) or the k-means problem ([Kan+02], [Llo82]).

Other tree cover problems include the k-min-max tree cover problem and the bounded tree cover problem ([AHL06], [Eve+04], [KS14]). In the k-min-max tree cover problem, we are given edge weights and want to find k trees such that the maximum of the total weights of the trees is minimized. In the bounded tree cover problem, we are given a bound on the maximum weight of a tree in the cover and try to minimize the number of trees that are required. For these problems Khani and Salavatipour [KS14] gave a 3- and 2.5-approximation respectively. They improve over the previously best known results by

Arkin et al. [AHL06], who presented a 4-approximation algorithm for the min-max tree cover problem and a 3-approximation algorithm for the bounded tree cover problem. Even et al. [Eve+04] independently gave a 4-approximation algorithm for the min-max tree cover problem. Furthermore, a rooted version of these problems has been studied. The best known approximation ratio for the capacitated tree case is 7 and was developed by Yu and Liu [YL19].

Many algorithms for cycle cover problems are also based on tree cover algorithms ([Eve+04], [TT22], [XXZ12]). An example is the capacitated cycle covering problem, where the cover consists of cycles (and singletons) and are given an upper bound on the total nodeweight of the cycles. The task is to minimize the total weight of the cycles plus the facility opening costs. Traub and Tröbst [TT22] presented a $2 + \frac{2}{7}$-approximation for this problem. They use an algorithm for the capacitated tree cover problem as a basis for their $2 + \frac{2}{7}$-approximation. In particular, they present a 2-approximation for the capacitated tree cover problem without edge loads.

5.2 The LP-formulation

We may assume that $\gamma \geq c(e)$ for all $e \in E$, as an edge with $c(e) > \gamma$ will never be used in an optimum solution (and could be removed from the solution of the algorithm without increasing the cost).

For simplicity, we will introduce some notation here: For any function $f : A \to B \subseteq \mathbb{R}$ from a finite set A into a set $B \subseteq \mathbb{R}$ and $X \subseteq A$ we write $f(X) := \sum_{x \in X} f(x)$.

Given a solution F to our problem with k components $\{T_1, \ldots, T_k\}$, we know that each tree T_i contains exactly $|V(T_i)| - 1$ edges and hence $k = |V| - |E(F)|$. Each induced subgraph of F is a forest. So we know

$$|E(F[A])| \leq |A| - 1 \text{ for each } A \subseteq V.$$

Let us now consider the load on the subgraph of F, induced by $A \subseteq V$. Each connected component in $F[A]$ can have load at most 1. So there must be at least $b(A) + u(E(F[A]))$ components in $F[A]$. As each of the components is a tree, the inequality

$$|E(F[A])| \leq |A| - (b(A) + u(E(F[A])))$$

must be fulfilled. Using these properties, we can formulate the following LP relaxation of this problem:

$$\min \quad c^t x + \gamma(|V| - x(E)) \tag{5.1}$$

$$\text{s.t.} \quad x(E(G[A])) \leq |A| - 1 \qquad \text{for each } A \subseteq V \tag{5.2}$$

$$\sum_{e \in E(G[A])} (1 + u(e))x(e) \leq |A| - b(A) \qquad \text{for each } A \subseteq V \tag{5.3}$$

$$0 \leq x(e) \leq 1 \qquad \text{for each } e \in E \tag{5.4}$$

Here $x(e)$ denotes the fractional usage of the edge e. We will call an edge e *active* if $x(e) > 0$. The LP can be reformulated by using variables $y(e) := x(e)(1 + u(e))$:

$$\min \quad \sum_{e \in E} \frac{c(e)}{1+u(e)} y(e) + \gamma \left(|V| - \sum_{e \in E} \frac{y(e)}{1+u(e)} \right) \tag{5.5}$$

$$\text{s.t.} \quad \sum_{e \in E(A)} \frac{y(e)}{1+u(e)} \leq |A| - 1 \qquad \text{for each } A \subseteq V \tag{5.6}$$

$$y(E(G[A])) \leq |A| - b(A) \qquad \text{for each } A \subseteq V \tag{5.7}$$

$$0 \leq y(e) \leq 1 + u(e) \qquad \text{for each } e \in E \tag{5.8}$$

For simplicity, we will always consider solutions x, y of both LPs at once. In the following, we will denote by $u_x(e) := x(e) \cdot u(e)$ the fractional load of edge e.

Definition 5.2.1. *For a solution x, y to the LP, we define the* support graph $G_x := (V, \{e \in E \mid x(e) > 0\})$*, i.e. the graph consisting of the vertices V and all active edges.*

We call an edge tight *if $y(e) = 1 + u(e)$, and we call a set $A \subseteq V$ of vertices* tight *if inequality (5.7) is tight.*

Our goal will be to solve the LP exactly and then round to a forest that may violate the capacity constraints. This increases the edge cost by at most a factor of 2. In a final step each tree T in the forest with a load $b(V(T)) + u(E(T)) > 1$ can be split into at most $2 \cdot (b(V(T)) + u(E(T)))$ trees. This may decrease the edge cost, but loses a factor of 3 in the number of components, compared to the LP solution.

5.3 Solving the LP

Although the LP has an exponential number of inequalities, we can solve it using a simple greedy algorithm, shown in Algorithm 7. We will focus on solving the second LP (5) – (8).

As a first step, we sort the edges $\{e_1, \ldots, e_m\} = E(G)$ such that

$$\frac{c(e_1) - \gamma}{1 + u(e_1)} \leq \ldots \leq \frac{c(e_m) - \gamma}{1 + u(e_m)}.$$

In each iteration, we compute a partition $\mathcal{A}_i \subset 2^{V(G)}$ of the vertices of G, based on the previous partition $\mathcal{A}_{i-1}$. We initialize y to 0 and start with $\mathcal{A}_0 := \{\{v\} | v \in V(G)\}$. Then we iterate through the edges from e_1 to e_m. For each edge e_i, we do the following:

If e_i has endpoints in two different sets of the partition $A_i^1, A_i^2 \in \mathcal{A}_{i-1}$, we increase $y(e_i)$ to the maximum possible value. This maximum value is the sum of the slacks of inequalities (5.7) for the sets A_i^1 and A_i^2: $|A_i^1| - b(A_i^1) - y(E(G[A_i^1])) + |A_i^2| - b(A_i^2) - y(E(G[A_i^2]))$. However, we assign at most $1+u(e_i)$, such that we do not violate inequality (5.8).

Finally, if we increased $y(e_i)$ by a positive amount, we create the new partition $\mathcal{A}_i$ that arises from $\mathcal{A}_{i-1}$ by removing A_i^1 and A_i^2 and adding their union.

We set $\mathcal{A} := \bigcup_{i=1,\dots,m} \mathcal{A}_i$. Observe that $\mathcal{A}$ is a laminar family. This guarantees that the support graph is always a forest and inequality (5.6) is automatically fulfilled.

Algorithm 7: Algorithm for solving the LP (5) – (8).

Input : G, c, u.
Output: y optimum solution of the LP (5) – (8).

```
1 Sort edges such that c(e_1)−γ/(1+u(e_1)) ≤ ... ≤ c(e_m)−γ/(1+u(e_m));
2 Set A_0 := {{v}|v ∈ V(G)} and y := 0;
3 for i = 1...m do
4     if there are sets A_i^1, A_i^2 ∈ A_{i−1} with e_i ∩ A_i^1 ≠ ∅, e_i ∩ A_i^2 ≠ ∅ and A_i^1 ≠ A_i^2
       then
5         y(e_i) := min{1+u(e_i), |A_i^1|−b(A_i^1)−y(E(A_i^1))+|A_i^2|−b(A_i^2)−y(E(A_i^2))};
6         if y(e_i) > 0 then
7             A_i := (A_{i−1} \ {A_i^1, A_i^2}) ∪ {A_i^1 ∪ A_i^2};
8         else
9             A_i := A_{i−1}
```

Lemma 5.3.1. *Let x, y be the solution computed by Algorithm 7. If a set $A \in \mathcal{A}$ from the algorithm is not tight, then all the edges in its induced subgraph $G_x[A]$ of the support graph are tight.*

Proof. Assume this were false. Take a minimal counterexample A. As the claim certainly holds for sets consisting only of one vertex ($G_x[A]$ has no edges if $|A| = 1$), we know that $|A| \geq 2$. We can write $A = A_i^1 \cup A_i^2$ with their associated edge e_i (for some i). We know that e_i has to be tight by line 5, as A is not tight. Otherwise, the algorithm could have increased $y(e_i)$ further. At least one of the subsets A_i^1 and A_i^2 of A is not tight, otherwise, A were tight. W.l.o.g we may assume that A_i^1 is not tight. Then all of its edges are tight, by minimality of A. However, then we know that $x(E(G[A_i^1])) = |A_i^1| - 1$. Thus,

$$\begin{aligned} &|A_i^1| - b(A_i^1) - y(E(G[A_i^1])) = |A_i^1| - b(A_i^1) - x(E(G[A_i^1])) - u_x(E(G[A_i^1])) \\ &= |A_i^1| - b(A_i^1) - (|A_i^1| - 1) - u_x(E(G[A_i^1])) = 1 - (u_x(E(G[A_i^1]) + b(A_i^1)) \leq 1 + u(e_i). \end{aligned}$$

This implies that e_i uses up all the slack of A_i^1, when it is made tight. Thus, there must be slack remaining on A_i^2 and it cannot be tight. As A contains an edge that is not tight in its support graph, this edge must be contained in A_i^2. We can conclude that A_i^2 is a smaller counterexample. This contradicts the minimality of A. □

Corollary 5.3.2. *Let $e_i \in E$, A_i^1 and A_i^2 such that they fulfill the requirements in line 4 of the algorithm. If e_i is tight then exactly one of the following is true:*

- *Neither A_i^1, nor A_i^2 are tight.*
- *A_i^j is tight, A_i^{3-j} is not tight and $u(e_i) = u_x(E(G[A_i^{3-j}])) + b(A_i^{3-j}) = 0$.*

Proof. If one of the sets is tight. W.l.o.g. we can assume that this set is A_i^1. If the other set A_i^2 is tight, we know that $y(e_i) = 0$, which contradicts our assumption. Otherwise, by Lemma 5.3.1, all edges in $E(G_x[A_i^2])$ are tight. Then $y(e_i) = |A_i^2| - b(A_i^2) - y(E(G[A_i^2])) = 1 - (u_x(E(G[A_i^2])) + b(A_i^2)) \leq 1 + u(e_i) = y(e_i)$. Hence, $u_x(E(G[A_i^2])) + b(A_i^2) = u(e_i) = 0$. □

Theorem 5.3.3. *Algorithm 7 works correctly and has running time $\mathcal{O}(m \log n)$.*

Proof. The running time is dominated by sorting.
For the correctness, we first check that the algorithm outputs a solution to our LP. The minimum in line 5 guarantees that inequality (5.8) is fulfilled. We have already seen that the support graph of our solution is a forest, which means that inequality (5.6) is also satisfied. It remains to check that inequality (5.7) holds. Each $A \in \mathcal{A}$ fulfills the inequality, when it is introduced by line 5. Since $\mathcal{A}$ is a laminar family, we never change the value of $y(E(G[A]))$ after A has been introduced, so we already know that all $A \in \mathcal{A}$ satisfy inequality (5.7) when the algorithm is finished.

We define the slack of a set $A \subseteq V$ as the slack of inequality 5.7 for that set and denote it by

$$\sigma(A) := |A| - b(A) - y(E(G[A])).$$

Then we can prove that when the algorithm introduces a new set A, it has no more slack than each of the joined subsets.

Claim (1)**.** *Let $A_i^1, A_i^2, A \in \mathcal{A}$ such that $A = A_i^1 \cup A_i^2$. We claim that $\sigma(A) \leq \sigma(A_i^j)$ for $j = 1, 2$.*

First, note that

$$\begin{aligned}\sigma(A) &= |A| - b(A) - y(E(G[A])) \\ &= |A_i^1| + |A_i^2| - (b(A_i^1) + b(A_i^2)) - (y(E(G[A_i^1])) + y(E(G[A_i^2])) + y(e_i)) \\ &= \sigma(A_i^1) + \sigma(A_i^2) - y(e_i). \qquad (\star)\end{aligned}$$

So it is sufficient to show that $y(e_i) \geq \max\{\sigma(A_i^1), \sigma(A_i^2)\}$. By Line 5 of the algorithm, we have

$$y(e_i) = \min\{1 + u(e_i), \sigma(A_i^1) + \sigma(A_i^2)\}.$$

If we are in the second case, then $y(e_i) = \sigma(A_i^1) + \sigma(A_i^2)$ and we are done. Otherwise, $y(e_i) = 1 + u(e_i)$, so e_i is tight. By Corollary 5.3.2, at least one of the sets A_i^1, A_i^2 is not tight. W.l.o.g. let this set be A_i^1. Then all edges in $E(G_x[A_i^1])$ are tight and we have $\sigma(A_i^1) = |A_i^1| - b(A_i^1) - y(E(G[A_i^1])) = 1 - (b(A_i^1) + u_x(E(G[A_i^1]))) \leq 1 + u(e_i) = y(e_i)$.

For the other set, we have two cases. Either A_i^2 is not tight and we can apply the same proof again, or it is tight, then $\sigma(A_i^2) = 0 \leq y(e_i)$. This proves the claim.

As a next step, we extend Claim (1) to all subsets of A.

Claim (2)**.** *Let $A \in \mathcal{A}$. We claim that each subset $B \subseteq A$ has slack $\sigma(B) \geq \sigma(A)$.*

We prove this by induction on the number of iterations. Let i' be the first index such that the algorithm sets $y(e_{i'}) > 0$. The claim holds for all iterations before that, because all sets are singletons until this point. The algorithm creates $A := A_{i'}^1 \cup A_{i'}^2$ with $|A| = 2$. Since all subsets of A are in $\mathcal{A}$, we are in the situation of Claim (1) and there is nothing left to prove.

Now let $i > i'$ and assume, that our claim holds in all previous iterations. The algorithm creates $A := A_i^1 \cup A_i^2$. Let $B \subseteq A$. For $|B| = 1$, we have $B \in \mathcal{A}$, because all singletons are in $\mathcal{A}$. So we can apply the Claim (1). If $|B| > 1$, we split B into two sets $B_1 := B \cap A_i^1$ and $B_2 := B \cap A_i^2$. We observe that $E(G[B]) \setminus (E(G[B_1]) \cup E(G[B_2])) \subseteq \{e_i\}$ and thus

$$\begin{aligned}
\sigma(B) = \quad & |B| - b(B) - y(E(G[B])) \\
= \quad & |B_1| + |B_2| - b(B_1) - b(B_2) - y(E(G[B_1])) \\
& - y(E(G[B_2])) - y(E(G[B]) \setminus (E(G[B_1]) \cup E(G[B_2]))) \\
\geq \quad & \sigma(B_1) + \sigma(B_2) - y(e_i).
\end{aligned}$$

We use our induction hypothesis on B_1 and B_2, if they are nonempty, to see that $\sigma(B_1) \geq \sigma(A_i^1)$ and $\sigma(B_2) \geq \sigma(A_i^2)$ respectively. So we can use equation $(\star)$ to compute

$$\sigma(B) \geq \sum_{j=1,2} \sigma(B_i) - y(e_i) \geq \sum_{j=1,2} \sigma(A_i^j) - y(e_i) = \sigma(A).$$

This proves Claim (2).

Finally, we observe that for $B_1 \subseteq V$ and $B_2 \subseteq V$ from different connected components of G_x, we have $\sigma(B_1 \cup B_2) = \sigma(B_1) + \sigma(B_2)$. This means that we can split any subset of the vertices $B \subseteq V$ into subsets of the toplevel sets of $\mathcal{A}$, which are exactly the connected components of G_x. Then we can use Claim (2) on each of the subsets to see that they have nonnegative slack. The observation implies that B also has nonnegative slack. So inequality 5.7 is always satisfied.

Next we want to prove optimality. Assume that y were not optimum. Let y^* be an optimum solution which maximizes the index of the first edge in the order of the algorithm in which y and y^* differ and among those minimizes the difference in this edge. Let this index be denoted by k. As the algorithm always sets the values to the maximum that is possible without violating an inequality, we know that $y^*(e_k) < y(e_k)$.

By the ordering of the algorithm, we know that

$$\frac{c(e_k) - \gamma}{1 + u(e_k)} \leq \frac{c(e_i) - \gamma}{1 + u(e_i)}$$

for all $i > k$. Our goal will be, to find an edge e_i with $i > k$ such that we can increase $y^*(e_k)$ and avoid violating constraints or increasing the objective by decreasing $y^*(e_i)$ in x^*, y^*.

Let G_k be the connected component of e_k in the subgraph of G containing only e_k and the active edges with index less than k. Define

$$\Gamma := \{e_i \in E(G_{x^*}) \mid i > k \text{ and } e_i \text{ incident to } v \in V(G_k)\}.$$

Note that $\Gamma \neq \emptyset$, because otherwise, we could increase $y^*(e_k)$ to $y(e_k)$ without violating any constraints. Since $c(e) \leq \gamma$, this would not increase the objective value.

We will prove that all tight sets containing the vertices of G_k must have a common edge in Γ.

Claim (3)**.** *Let $\mathcal{T} := \{B \subseteq V \mid V(G_k) \subseteq B$ and B tight$\}$ be the family of tight sets of x^*, y^* containing the vertices of G_k. We claim that*

$$\Gamma \cap \bigcap_{B \in \mathcal{T}} E(G_{x^*}[B]) \neq \emptyset.$$

If there is an edge in Γ between vertices of G_k, then this certainly holds. Otherwise, we know that $V(G_k)$ is not tight, because the algorithm was able to set $y(e_k) > y^*(e_k)$. **Observation:** Unions of tight sets are tight and there are no active edges between tight sets.

Let A, B be tight sets. Let $\Delta := E(G[A \cup B]) \setminus (E(G[A]) \cup E(G[B]))$ be the edges between A and B. We have

$$\begin{aligned} y(E(G[A \cup B])) &\leq |A \cup B| - b(A \cup B) \\ &= |A| - b(A) + |B| - b(B) - (|A \cap B| - b(A \cap B)) \\ &= y(E(G[A])) + y(E(G[B])) - (|A \cap B| - b(A \cap B)) \\ &= y(E(G[A \cup B])) - y(\Delta) + y(E(G[A \cap B])) - (|A \cap B| - b(A \cap B)) \\ &\leq y(E(G[A \cup B])) - y(\Delta), \end{aligned}$$

where we used LP inequality (5.7) in the first and in the last step. As $y \geq 0$, we must have equality everywhere, and $y(E(G[A \cup B]) \setminus (E(G[A]) \cup E(G[B]))) = 0$. This proves our observation.

Let $\mathcal{S} := \{S_1, \ldots, S_p\} \subseteq \mathcal{T}$ be a set of $p \geq 2$ tight sets containing the vertices of G_k and set $Z := \bigcup_{S_i \in \mathcal{S}} S_i$. By our observation, Z is tight as well. We will introduce some notation to write down the proof of the claim. For $a \in \mathbb{N}$, we write $[a] := \{1, \ldots, a\}$. Then for any index-set $\emptyset \subsetneq I \subseteq [p]$, denote by $V_I := \bigcap_{i \in I} S_i \setminus V(G_k)$ the vertices of the intersection of the S_i belonging to the indices in I outside of G_k, by $E_I := E(G_{x^*}[V_I])$ the active edges in V_I and by $\Delta_I := \Gamma \cap E(G_{x^*}[\bigcap_{i \in I} S_i])$ the active edges between V_I and G_k. These sets are illustrated in Figure 5.1. Furthermore, we denote for $A \subset V(G)$ by $\sigma^*(A) := |A| - b(A) - y^*(E(G[A]))$ the slack of the inequality for A in the optimum solution.

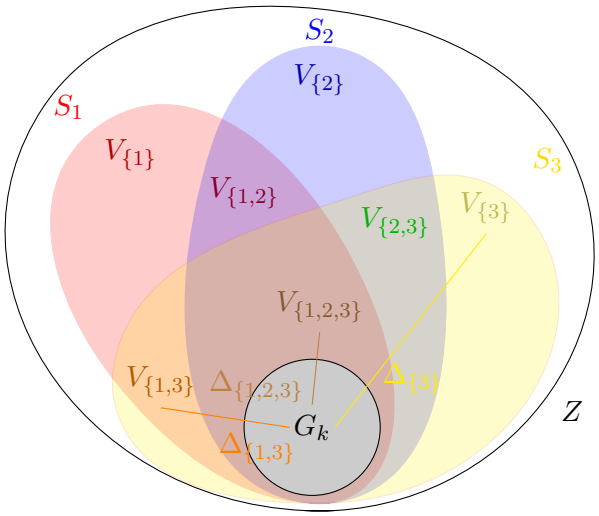


Figure 5.1: An illustration of the sets Z, G_k, S_i, V_I and Δ_I for $p = 3$ and $i \in [3], I \subseteq [3]$. The set $V_{\{1\}}$ for example contains the vertices in the red, the purple, the orange and the brown area, while the set $V_{\{1,2\}}$ contains only the vertices in the purple and the brown area.

We will prove a stronger statement than our claim by induction on p: For each $p \geq 2$ and $I \subseteq [p]$ with $|I| \geq 2$, it holds that

$$y^*(\Delta_I) - \sigma^*(V_I) = \sigma^*(V(G_k)).$$

Since $V(G_k)$ is not tight, the right-hand side must be greater than zero. By our LP-inequalities, $\sigma^*(V_I) \geq 0$. Hence, $y^*(\Delta_I) > 0$ and $\Delta_I \neq \emptyset$.

We start with $p = 2$. Using the tightness of $S_1 \cup S_2$, we get

$$\begin{aligned} y^*(E(G[Z])) &= |Z| - b(Z) \\ &= |S_1| - b(S_1) + |S_2| - b(S_2) - (|V_{\{1,2\}}| - b(V_{\{1,2\}})) - (|V(G_k)| - b(V(G_k))). \end{aligned}$$

Our observation that edges between tight sets can not be active implies that

$$\begin{aligned} &y^*(E(G_k)) + y^*(E_{\{1\}}) + y^*(E_{\{2\}}) - y^*(E_{\{1,2\}}) + y^*(\Delta_{\{1\}}) + y^*(\Delta_{\{2\}}) - y^*(\Delta_{\{1,2\}}) \\ =&y^*(E(G[Z])). \end{aligned}$$

Using the tightness of S_1 and S_2, we compute

$$\begin{aligned} y^*(E(G[Z])) = \quad & y^*(S_1) + y^*(S_2) - (|V_{\{1,2\}}| - b(V_{\{1,2\}})) - (|V(G_k)| - b(V(G_k))) \\ = \quad & 2 \cdot y^*(E(G_k)) + y^*(E_{\{1\}}) + y^*(E_{\{2\}}) + y^*(\Delta_{\{1\}}) + y^*(\Delta_{\{2\}}) \\ & -(|V_{\{1,2\}}| - b(V_{\{1,2\}})) - (|V(G_k)| - b(V(G_k))) \\ = \quad & y^*(E(G[Z])) + y^*(E(G_k)) + y^*(E_{\{1,2\}}) + y^*(\Delta_{\{1,2\}}) \\ & -(|V_{\{1,2\}}| - b(V_{\{1,2\}})) - (|V(G_k)| - b(V(G_k))) \\ = \quad & y^*(E(G[Z])) + y^*(\Delta_{\{1,2\}}) - \sigma^*(V_{\{1,2\}}) - \sigma^*(V(G_k)). \end{aligned}$$

We conclude

$$y^*(\Delta_{\{1,2\}}) - \sigma^*(V_{\{1,2\}}) = \sigma^*(V(G_k)).$$

For the induction step, let $p > 2$ and assume we have for any index-set $I \subseteq [p]$ with $2 \leq |I| < p$ that

$$y^*(\Delta_I) - \sigma^*(V_I) = \sigma^*(V(G_k)).$$

Similarly as in the case $p = 2$, we want to write $|Z| - b(Z)$ as a sum of the $|S_i| - b(S_i)$ minus the values that we counted multiple times. We added $|V(G_k)| - b(G_k)$ with each of the S_i, but only needed it once. So we have to subtract it $(p-1)$ times. Then for the remainder, we use the following observation:

Observation: Let $C_1, \ldots, C_p$ be some sets and $f : \bigcup_{i=1,\ldots,p} C_i \to \mathbb{R}$ a function on the elements of the sets. Then

$$f\left(\bigcup_{i=1,\ldots,p} C_i\right) = \sum_{j=1}^{p} (-1)^{j-1} \sum_{I \in \binom{[p]}{j}} f\left(\bigcap_{i \in I} C_i\right).$$

We compute

$$\begin{aligned} y^*(E(G[Z])) = \quad & |Z| - b(Z) \\ = \quad & \sum_{j=1}^{p} [|S_j| - b(S_j)] - (p-1) \cdot (|V(G_k)| - b(G_k)) \\ & - \sum_{j=2}^{p} (-1)^j \sum_{I \in \binom{[p]}{j}} [|V_I| - b(V_I)]\,. \end{aligned}$$

Then we use tightness of the S_j for $j = 1, \ldots, p$ and the fact that we can split $y^*(E(G[S_j])) = y^*(E_j) + y^*(\Delta_j) + y^*(E(G_k))$ to rewrite this as

$$\begin{aligned} y^*(E(G[Z])) = \quad & \sum_{j=1}^{p} y^*(E(G[S_j])) - (p-1) \cdot (|V(G_k)| - b(G_k)) \\ & - \sum_{j=2}^{p} (-1)^j \sum_{I \in \binom{[p]}{j}} [|V_I| - b(V_I)] \\ = \quad & \sum_{j=1}^{p} [y^*(E_j) + y^*(\Delta_j) + y^*(E(G_k))] - (p-1) \cdot (|V(G_k)| - b(G_k)) \\ & - \sum_{j=2}^{p} (-1)^j \sum_{I \in \binom{[p]}{j}} [|V_I| - b(V_I)]\,. \end{aligned}$$

We use the reverse argument of the first step to assemble $y^*(Z)$ and add the remaining values:

$$\begin{aligned} y^*(E(G[Z])) = \quad & y^*(Z) + \sum_{j=2}^{p}(-1)^j \sum_{I\in\binom{[p]}{j}} [y^*(E_I) + y^*(\Delta_I)] + (p-1)y^*(E(G_k)) \\ & -(p-1)(|V(G_k)| - b(G_k)) - \sum_{j=2}^{p}(-1)^j \sum_{I\in\binom{[p]}{j}} [|V_I| - b(V_I)]\,. \end{aligned}$$

And finally we simplify this to

$$\begin{aligned} y^*(E(G[Z])) = \quad & y^*(Z) - (p-1)\sigma^*(V(G_k)) \\ & + \sum_{j=2}^{p}(-1)^j \sum_{I\in\binom{[p]}{j}} [y^*(\Delta_I) + y^*(E_I) - (|V_I| - b(V_I))] \\ = \quad & y^*(Z) - (p-1)\sigma^*(V(G_k)) + (-1)^p(y^*(\Delta_{[p]}) - \sigma^*(V_{[p]})) \\ & + \sum_{j=2}^{p-1}(-1)^j \sum_{I\in\binom{[p]}{j}} [y^*(\Delta_I) - \sigma^*(V_I)]\,. \end{aligned}$$

Now we can use our induction hypothesis to retrieve

$$\begin{aligned} \sum_{j=2}^{p-1}(-1)^j \sum_{I\in\binom{[p]}{j}} [y^*(\Delta_I) - \sigma^*(V_I)] &= \sum_{j=2}^{p-1}(-1)^j \sum_{I\in\binom{[p]}{j}} \sigma^*(V(G_k)) \\ &= (p-1+(-1)^{p+1})\sigma^*(V(G_k)), \end{aligned}$$

which can be put back into our previous equation:

$$\begin{aligned} y^*(E(G[Z])) = \quad & y^*(E(G[Z])) - (p-1)\sigma^*(V(G_k)) + (-1)^p(y^*(\Delta_{[p]}) - \sigma^*(V_{[p]})) \\ & +(p-1+(-1)^{p+1})\sigma^*(V(G_k)) \\ = \quad & y^*(E(G[Z])) + (-1)^p(y^*(\Delta_{[p]}) - \sigma^*(V_{[p]}) - \sigma^*(V(G_k))) \end{aligned}$$

As before, we conclude

$$y^*(\Delta_{[p]}) - \sigma^*(V_{[p]}) = \sigma^*(V(G_k)).$$

Finally, we can pick an edge $f \in \Gamma$ that is contained in all tight sets that contain the vertices of G_k. If $u(f) \leq u(e_k)$, we know that $\frac{1}{1+u(f)} \geq \frac{1}{1+u(e_k)}$. So we can decrease $y^*(f)$ and increase $y^*(e_k)$ by the same amount without violating any constraints. By the ordering of our algorithm, this can not increase the objective value. This would contradict our choice of y^*. Hence, $u(f) > u(e_k)$. But then $c(f) \geq c(e_k)$ and we could decrease $x^*(f)$ and increase $x^*(e_k)$ without increasing the objective value. Furthermore, we also do not create a violation this way, because $\epsilon \cdot (1 + u(f)) > \epsilon \cdot (1 + u(e_k))$ for $\epsilon > 0$. This contradicts our choice of y^* and concludes the proof. □

The support graph of the LP solution computed by Algorithm 7 is always a forest. Thus, Theorem 5.3.3 implies the following:

Corollary 5.3.4. *There is always a solution x, y to both LPs, such that the support graph G_x is a forest.*

5.4 The Rounding Strategy

Now we want to round the LP solution, computed by Algorithm 7, to get an integral solution. We do so by rounding up edges e with $x(e) \geq \alpha$, for some $0 \leq \alpha \leq 1$ to be determined later. All other edges are rounded down. The forest arising from this rounding step may contain components T with $b(V(T)) + u(E(T)) > 1$. These large components will be split into at most $2 \cdot (b(V(T)) + u(E(T)))$ legal components. We achieve this by using a splitting technique that is often used for these cases, for example in [MV08] and also in [TT22].

We will start by explaining how the splitting technique works in Section 5.4.1. However, for the analysis, we only require the result that it is possible to split the trees into $2 \cdot (b(V(T)) + u(E(T)))$ legal components. In Section 5.4.2, we will study the LP solution, that we get from Algorithm 7. We will exploit the structure of this solution in our analysis. Then we will bound the number of components that we get after rounding and splitting in Section 5.4. We do this by providing an upper bound on the value of each edge after rounding and splitting. Finally, we determine two different α and the implied bounds on the value of our solution compared to the LP solution. The first bound in Section 5.4.3 will depend on the edge loads that are occurring in the instance and is better, when only light edges (with load $< \frac{1}{2}$) occur. The second one in Section 5.4.3 will give a factor of 3 independent of the edge loads.

5.4.1 Splitting large trees

Given a rounded component T with total load $b(V(T)) + u(E(T)) > 1$, we want to split this tree into a forest consisting only of trees with total load less or equal to 1.

Lemma 5.4.1 (Maßberg and Vygen 2008 [MV08])**.** *There is a linear time algorithm that splits a tree with total load $b(V(T)) + u(E(T)) > 1$ into at most $2 \cdot (b(V(T)) + u(E(T)))$ legal trees.*

Proof. We choose an arbitrary vertex $r \in V(T)$ as a root and direct the tree away from r. This way, we get an arborescence. During the algorithm, we will maintain a set of trees $\mathcal{T}$, as well as an upper bound on the total load of each tree $l : \mathcal{T} \to [0, 1]$ and an assignment of vertices to trees $t : V(G) \to \mathcal{T}$. We initialize $\mathcal{T} := \{\{v\} | v \in V(T)\}$, $l(\{v\}) := b(v)$ and $t(v) := \{v\}$. Furthermore, we denote for all $v \in V \setminus \{r\}$ by e_v the unique incoming edge of the vertex v.

We traverse the arborescence in reverse topological order. At the leaves, we do nothing. At each other vertex v, we construct an instance of the bin packing problem. The

maximum size of a bin is 1. Let C be the children of v. Then we want to pack $\{v\} \cup C$, where we assign to v the weight $w(v) := l(t(v))$ and to the children $c \in C$ the weight $w(c) := l(t(c)) + u(e_c)$.

Then we optimize this bin packing instance. For our purposes, the first fit approach will be enough. For our splitting, we only require, that each bin except for one is packed with at least weight $\frac{1}{2}$ and that the last bin could not have been added to any of the other bins.

Let $B_1, \ldots, B_k$ be the resulting bins ordered in decreasing order of their weight. For $i = 1, \ldots, k$, we do nothing, if $|B_i| = 1$. Otherwise, we join the trees $t(p)$ for $p \in B_i$ to a new tree U by shortcutting the paths in T that are connecting them. This can only decrease the load (and cost), because u was metric (and c was metric). Then we remove $t(p)$ from $\mathcal{T}$ for each $p \in B_i$ and we add U to $\mathcal{T}$. We set $l(U) := w(B_i)$. This is an upper bound on the total load of U because we included the weight of the connecting edges in the element weights of the bin packing instance. For the last bin with $i = k$, we also set $t(v) := U$.

Note that due to the reverse topological order, we do not access $t(c)$ or $l(t(c))$ for $c \in C$ again after this point. Also, $\mathcal{T}$ will always contain a partition of T and each edge load will have been counted exactly once.

In the end, each tree in $\mathcal{T}$ will correspond to a bin that we produced during our algorithm. Since we always assign the smallest bin to the current vertex, we make sure that it gets passed on to the next iteration. This way, we can guarantee that at most one of the trees has total load less than $\frac{1}{2}$. Furthermore, there was at least one bin, where this tree did not fit. Consequentially the loads of these two trees sum up to more than 1. This means that on average they have total load at least $\frac{1}{2}$. Since every other tree has total load at least $\frac{1}{2}$, we have created at most $2 \cdot (b(V(T)) + u(E(T)))$ trees this way. □

5.4.2 The general structure of the LP solution

Let x, y be a solution found by the algorithm. Recall that for edges $e \in E(G)$, $u_x(e) := x(e) \cdot u(e)$ was the fractional load of the edge e in our solution. Note that then it holds for each set $A \subseteq V(G)$ and edge $e \in E(G)$ that

$$y(E(G[A])) = x(E(G[A])) + u_x(E(G[A])) \text{ and } y(e) = x(e) + x(e)u(e).$$

Without loss of generality, we can assume that $0 < x(e)$ for all edges. We simply remove all edges with $x(e) = 0$. Then we contract all inclusionwise maximal sets $A \in \mathcal{A}$ such that all edges in their respective induced support graph are tight and set the load of the new vertex to $b(A) + u_x(E(G[A]))$. This only makes the approximation guarantee worse, because these components will have the same value in the rounded solution as in the LP-solution. Corollary 5.3.2 implies that all remaining edges $e \in E$ with $x(e) = 1$, are edges with load $u(e) = 0$. In the remaining graph the following assertions hold:

1. $|\{v\}| - b(\{v\}) - y(E(G[\{v\}])) = 1 - b(v) \leq 1$ for all $v \in V$.

2. All $A \in \mathcal{A}$ containing more than 1 vertex are tight, by Lemma 5.3.1.

Now, we will take a closer look at the sets A_i^j for $i = 1, \ldots, m$ and $j = 1, 2$. In the following analysis, we will assume without loss of generality that $|A_i^1| \leq |A_i^2|$ and $\sigma(A_i^1) \geq \sigma(A_i^2)$. By the above assertions, we have for an edge e_i and the two associated sets A_i^1, A_i^2, either

(i) both A_i^1 and A_i^2 contain only one vertex and one of them (A_i^1) is not tight, or

(ii) A_i^1 contains only one vertex and is not tight and A_i^2 contains more vertices and is tight

To make the following easier to read, we add the following definitions

Definition 5.4.2. *Edges that fulfill condition (i) are called* seed edges *and edges that fulfill condition (ii) are called* extension edges. *For each edge e_i we denote by v_{e_i} the unique vertex in set A_i^1.*

Note that every edge e_i is either a seed edge or an extension edge, but this only holds after contracting the sets of tight edges as described above.

Thus, whenever e_i is a seed edge, the algorithm sets

$$y(e_i) := |A_i^1| - b(A_i^1) - y(E(G[A_i^1])) + |A_i^2| - b(A_i^2) - y(E(G[A_i^2])) = 1 - b(A_i^1) + 1 - b(A_i^2),$$

where we use the fact that $E(G[A_i^j]) = \emptyset$ for $j = 1, 2$. Since both A_i^1 and A_i^2 were singletons, we can conclude

$$x(e_i) + u(e_i)x(e_i) = y(e_i) = 2 - b(A_i^1 \cup A_i^2).$$

Similarly, for extension edges, we get

$$x(e_i) + u(e_i)x(e_i) = 1 - b(A_i^1).$$

In the analysis of the rounding step, we need some further observations:

Observation. *Let T be a connected component in G_x. Then*

- *T is a tree.*
- *If $|V(T)| > 1$, then T contains exactly one seed edge and all other edges are extension edges.*
- *If T contains a seed edge e_i, then $i = \min_{e_j \in E(T)} j$ or in other words, e_i was the first edge of T considered in the algorithm.*

Proof. As G_x is a forest, each connected component must be a tree.
Suppose there were two seed edges $e, f \in E(T)$ $(e \neq f)$. Let A be the set that the algorithm creates, when increasing e, and B the same for f. By the definition of seed edges, $|A| = |B| = 2$. Let $C = A_i^1 \dot{\cup} A_i^2$ be the smallest set in the laminar family $\mathcal{A}$ with $A, B \subset C$. Then $A \subseteq A_i^k$ and $B \subseteq A_i^{3-k}$ for some $k \in \{1, 2\}$. Hence, $|A_i^j| \geq 2$ for $j = 1, 2$ and both must be tight. Thus, $y(e_i) = 0$, contradicting the fact that Algorithm 7 has joined A_i^1 and A_i^2. This implies that T can only contain one seed edge.
Let $|V(T)| > 1$ and e_i be the first edge in T that was considered during the algorithm. Then $|A_i^j| = 1$ for $j = 1, 2$. So by definition, it is a seed edge. □

5.4.3 Analyzing the rounding step

First note that by our rounding procedure, the sum of the edge-weights can increase by at most $\frac{1}{\alpha}$. So for the edge-weights it is sufficient to make sure that $\alpha \geq \frac{1}{2}$ and the main difficulty is to bound the number of components.

Before we choose α, let us estimate how many components we get after rounding and splitting. To do this, we take a look at the connected components after rounding. Let T be such a component. We denote by $\text{comp}(T)$ the number of connected components we need to split T into.

Let C^* be the set of components before splitting and C be the set of components after splitting. Our goal here is to estimate $|C|$ by a contribution $\text{est}(e)$ of each edge $e \in E(G)$, such that the number of components after splitting is

$$|C| = \sum_{T \in C^*} \text{comp}(T) \leq \sum_{T \in C^*} |V(T)| - \sum_{e \in E(G)} \text{est}(e) = |V(G)| - \sum_{e \in E(G)} \text{est}(e)$$

There are three cases:

1. *singletons*: T consists of only one vertex.
2. *good trees*: T consists of more than one vertex and $u(E(T)) + b(V(T)) \leq 1$
3. *large trees*: T consists of more than one vertex and $u(E(T)) + b(V(T)) > 1$

Case 1: T is a singleton. Its number of components is

$$\text{comp}(T) := 1 = |V(T)| - 0.$$

Case 2: T is a good tree. So we can keep this component for a solution to the problem. The number of components is

$$\text{comp}(T) := 1 = |V(T)| - (|V(T)| - 1) \leq |V(T)| - \sum_{e \in E(T)} \left[1 - 2b(v_e) - 2u(e)\right].$$

For all $e \in E(T)$, we set $\text{est}(e) := 1 - 2b(v_e) - 2u(e)$.

Case 3: T is a large tree. So we have to split this component to get a feasible solution. Denote by e' the edge in T with the lowest index according to the sorting of the algorithm. Let $\bar{v} \neq v_{e'}$ be incident to e'. Note that this does not have to be a seed edge, as the components after rounding do not necessarily contain a seed edge. We rewrite the number of components:

$$\begin{aligned}\text{comp}(T) &\leq 2 \cdot (u(E(T)) + b(V(T))) \\ &= |V(T)| - [2 - 2b(v_{e'}) - 2u(e') - 2b(\bar{v})] - \sum_{e' \neq e \in E(T)} \left[1 - 2b(v_e) - 2u(e)\right].\end{aligned}$$

If T contains a seed edge, then this edge is e'. This means that the number of components can be estimated by edges in T. We set $\text{est}(e') := 2 - 2b(v_{e'}) - 2u(e') - 2b(\bar{v})$ and $\text{est}(e) := 1 - 2b(v_e) - 2u(e)$ for all $e \in E(T) \setminus \{e'\}$.

Otherwise, T only consists of extension edges. In this case, we write

$$[2 - 2b(v_{e'}) - 2u(e') - 2b(\bar{v})] + \sum_{e' \neq e \in E(T)} [1 - 2b(v_e) - 2u(e)] \\ = [1 - 2b(\bar{v})] + \sum_{e \in E(T)} [1 - 2b(v_e) - 2u(e)] .$$

Then, we set $\text{est}(e) := 1 - 2b(v_e) - 2u(e)$ for all $e \in E(T)$. However, in this case we need to account for the additional $1 - 2b(\bar{v})$. To do so, we call the edge incident to $\bar{v}$ that is not contained in T a *filler edge*. For all filler edges $\{v, w\} = e \in E(G)$, we w.l.o.g. assume that e is a filler edge for the component that contains v and set

$$\text{est}(e) := \begin{cases} 2 - (b(v) + b(w)), & \text{if } e \text{ is the filler edge of two components} \\ 1 - b(v), & \text{otherwise.} \end{cases}$$

For all edges not considered before, we set $\text{est}(e) := 0$.

Now we have that

$$|C| \leq |V(G)| - \sum_{e \in E(G)} \text{est}(e)$$

Our next goal is to find a lower bound on $\sum_{e \in E(G)} \text{est}(e)$.

Lower bounds for the extension edges

We start with the simpler case of extension edges. An overview over the cases in which they can appear is shown in Figure 5.2. Let e be an extension edge. If it appears inside a good tree or a large tree. Then

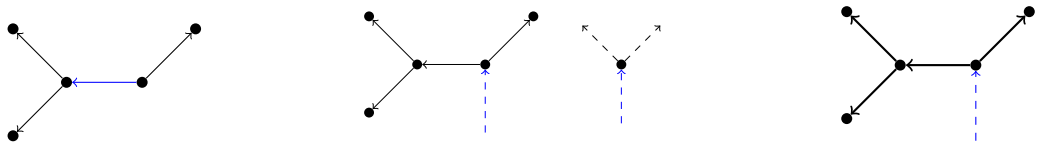
$$\begin{aligned} \text{est}(e) &= 1 - 2b(v_e) - 2u(e) \\ &= 1 - 2(1 - x(e) - x(e)u(e) + u(e)) \\ &= 1 - 2 + 2x(e) + 2x(e)u(e) - 2u(e) \\ &= 2x(e) - 1 - 2u(e)(1 - x(e)). \end{aligned}$$

If it is incident to a singleton or a good tree, we can estimate

$$\text{est}(e) = 0 \geq 2x(e) - 1 - (2x(e) - 1).$$

If it is a filler edge, we can estimate

$$\text{est}(e) = 1 - 2b(v_e) = 1 - 2(1 - x(e) - x(e)u(e)) = 2x(e) - 1 + x(e)u(e) \geq 2x(e) - 1.$$



(a) Inside a (good or large) tree.

(b) Leading towards a good tree or a singleton.

(c) As a filler edge leading to a large tree.

Figure 5.2: The cases in which extension edges can occur. Dashed edges have been rounded down, while solid ones have been rounded up. Thick edges belong to a large tree. For each edge e the arrowhead points towards v_e.

Lower bounds for the seed edges

Next we consider seed edges. An overview over the cases in which they can appear is shown in Figure 5.3. Let e be a seed edge. $e = \{v_e, \bar{v}\}$ can not be contained in a singleton. It can only be contained in a good tree, if e is tight, as otherwise we have

$$1 + u(e) > y(e) = 1 - b(v_e) + 1 - b(\bar{v}) \Leftrightarrow b(v_e) + b(\bar{v}) + u(e) > 1.$$

If it is tight, we are in the second case of Corollary 5.3.2 and $u(e) = 0$. Recall that then one of the sets A_i^1, A_i^2 was tight for $e = e_i$. By our labelling this was $A_i^2 = \bar{v}$. So, we know that $b(\bar{v}) = 1$. We estimate

$$\begin{aligned}
\text{est}(e) &= 1 - 2b(v_e) - 2u(e) \\
&= 1 - 2(2 - x(e) - x(e)u(e) - b(\bar{v}) + u(e)) \\
&= 1 - 4 + 2x(e) + 2b(\bar{v}) \\
&= 2x(e) - 3 + 2b(\bar{v}) \\
&\geq 2x(e) - 2.
\end{aligned}$$

If it is contained a large tree, it was the first edge considered in this component. We estimate

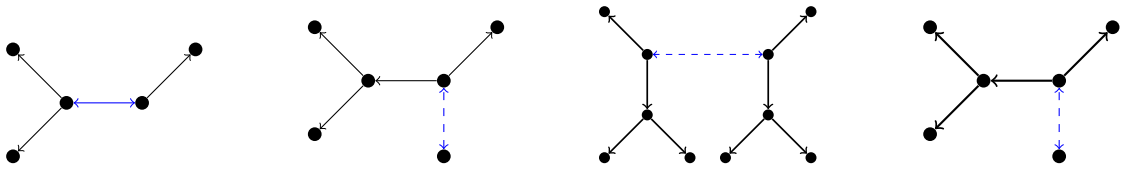
$$\begin{aligned}
\text{est}(e) &= 2 - 2b(v_e) - 2u(e) - 2b(\bar{v}) \\
&= 2 - 2(2 - x(e) - x(e)u(e) + u(e)) \\
&= 2 - 4 + 2x(e) + 2x(e)u(e) - 2u(e) \\
&= 2x(e) - 2 - 2u(e)(1 - x(e)).
\end{aligned}$$

Otherwise both endpoints are incident to different components. This means that it was rounded down. If these components are singletons or good trees, we can estimate

$$\text{est}(e) = 0 \geq 2x(e) - 2.$$

If both are large trees, then e is a filler edge for both and we have

$$\text{est}(e) = 2 - 2(b(v_e) + b(\bar{v})) = 2 - 2(2 - x(e) - x(e)u(e)) = 2x(e) - 2 + 2u(e)x(e) \geq 2x(e) - 2.$$



(a) Inside a good or large tree.

(b) Incident only to good trees or singletons.

(c) As a filler edge for two large trees.

(d) As a filler edge for one large tree and incident to a good tree or a singleton.

Figure 5.3: The cases in which seed edges can occur. Dashed edges have been rounded down, while solid ones have been rounded up. Thick edges belong to a large tree. For each extension edge e the arrowhead points towards v_e. Seed edges have arrowheads on both ends.

The last case is that e is incident to one large tree and a good tree or a singleton. This means it is a filler edge for only one endpoint. W.l.o.g. let this endpoint be v_e. We set $y_1 := (1+u(e))$, $x_1 := 1-b(v_e)$ and $y_2 := (1+u(e))$, $x_2 := 1-b(\bar{v})$. For a later estimate note that then $x_2 \leq \alpha$ as $x(e) \leq \alpha$. We can estimate

$$\begin{aligned}
\text{est}(e) &= 1 - 2b(v_e) \\
&= 1 - 2(1 - x_1 - u(e)x_1) \\
&= 1 - 2 + 2x_1 + 2u(e)x_1 \\
&= 2x_1 - 1 + 2u(e)x_1 \\
&= 2x_1 + 2x_2 - 2 + 2u(e)x_1 + 1 - 2x_2 \\
&\geq 2x(e) - 2 - (2x_2 - 1).
\end{aligned}$$

Now almost all estimates are of the same form.

Summary of the estimates

Before we choose α and derive the approximation guarantee, let us summarize the derived estimates.

$\text{est}(e) \geq$

	seed edges
$2x(e) - 2 - 0$	incident to good trees or singletons, in a good tree, filler for both ends
$2x(e) - 2 - (2x_2 - 1)$	filler for one end
$2x(e) - 2 - 2u(e)(1 - x(e))$	in a large tree
	extension edges
$2x(e) - 1 - 0$	filler edge
$2x(e) - 1 - (2x(e) - 1)$	incident to good tree or singleton
$2x(e) - 1 - 2u(e)(1 - x(e))$	inside a component

The base part, which is left in black, now sums up to at most $2x(E(G)) - |V(G)|$, because there is exactly one seed edge for every component in G that is not a singleton. So it remains to estimate the parts marked in blue. Our goal will be to estimate this part in terms of $|V(G)| - x(E(G))$. That is, find a β, such that we have "sum of blue parts" $\leq \beta(|V(G)| - x(E(G))$. We will achieve this by first estimating for each $\{v_e, w\} \in E(G_x)$ that

$$\text{"blue part"} \leq \begin{cases} \beta(b(v_e) + b(w) + x(e)u(e)) & \text{for seed edges} \\ \beta(b(v_e) + x(e)u(e)) & \text{for extension edges.} \end{cases}$$

Then, we can use that to sum up the estimates of the differences

$$\text{"sum of blue parts"} \leq \beta(b(V(G)) + u(x(E(G)))) \leq \beta(|V(G)| - x(E(G))),$$

where the last inequality follows directly from the LP-inequalities.

In total, we are left with

$$\begin{aligned} |C| &\leq |V(G)| - \sum_{e \in E(G)} \text{est}(e) \\ &\leq |V(G)| - (2x(E(G)) - |V(G)| - \beta(|V(G)| - x(E(G)))) \\ &= (2 + \beta)(|V(G)| - x(E(G))) \end{aligned}$$

A first approximation guarantee

Let $u_{max} := \max_{e \in E(G)} u(e)$

Lemma 5.4.3. *If we set $\alpha := \frac{1}{2}$, the number of components after splitting is at most*

$$(2 + 2\, u_{max})(|V(G)| - x(E(G))).$$

Proof. We will show the following claim:
If $\alpha = \frac{1}{2}$, then

$$\text{est}(e) \geq 2x(e) - 2 - 2\, u_{max}(b(v) + b(w) + x(e)u(e))$$

for all seed edges $e = \{v, w\}$ and

$$\text{est}(e) \geq 2x(e) - 1 - 2\, u_{max}(b(v) + u(e)x(e))$$

for all extension edges $e = \{v, w\}$.

These estimates are exactly of the form, we required in Section 5.4.3. Thus, they directly imply the statement of the lemma.

First we observe that $-u(e)(1-x(e)) \geq -\, u_{max}(1-x(e))$. Second, that for seed edges $e = \{v, w\}$, we have

$$1 - x(e) = b(v) + b(w) + u(e)x(e) - 1 \leq b(v) + b(w) + u(e)x(e)$$

and for extension edges $e = \{v_e, w\}$,

$$1 - x(e) = b(v_e) + u(e)x(e).$$

Furthermore of course $0 \geq -2\, u_{max}(b(v) + b(w) + u(e)x(e))$ and $0 \geq -2\, u_{max}(b(v_e) + u(e)x(e))$. With these observations the claim already holds for most edges. It remains to show that it holds for extension edges e that are incident to good trees or singletons. These edges are rounded down, as they are in no component. This means $x(e) \leq \frac{1}{2}$. We have

$$\text{est}(e) \geq 2x(e) - 1 - (2x(e) - 1) \geq 2x(e) - 1 \geq 2x(e) - 1 - 2\, u_{max}(b(v_e) + u(e)x(e)).$$

And we need to show the inequality for seed edges that are filler edges for only one large tree. These are also rounded down. There we estimated $\text{est}(e) \geq 2x(e) - 2 - (2x_2 - 1)$. With $x_2 \leq \frac{1}{2}$, we get

$$\text{est}(e) \geq 2x(e) - 2 \geq 2x(e) - 2 - 2\, u_{max}(b(v) + b(w) + u(e)x(e)).$$

This proves the claim. □

A general approximation guarantee

However, these estimates were not optimal, as we may choose α in a better way. Specifically, we will choose $\alpha := \frac{2}{3}$ to achieve a 3-approximation.

We want to determine $\beta \leq 1$ optimal, such that we get a $2 + \beta$-approximation. The only step, that required $\alpha \leq \frac{1}{2}$ in the previous proof was to estimate $-(2x(e) - 1) \geq 0$ for edges that were not rounded up. However, if we want a $2+\beta$-approximation, we may choose α, such that $-(2x(e) - 1) \geq -\beta(1 - x(e))$ (for edges that are rounded down).

$$\begin{aligned} & -(2x(e) - 1) \geq -\beta(1 - x(e)) \\ \Leftrightarrow \quad & 1 - 2x(e) \geq -\beta + \beta x(e) \\ \Leftrightarrow \quad & 1 + \beta \geq (2 + \beta)x(e) \\ \Leftrightarrow \quad & \frac{1+\beta}{2+\beta} \geq x(e) \end{aligned}$$

This means for all edges e that we round down, we need

$$x(e) \leq \frac{1+\beta}{2+\beta}.$$

So we set $\alpha := \frac{1+\beta}{2+\beta}$. Now we know for all edges e that we round up, $x(e) \geq \frac{1+\beta}{2+\beta}$. In this case we have $1 - x(e) \leq \frac{1}{2+\beta}$. So we can estimate

$$x(e) \geq \frac{1+\beta}{2+\beta} \geq (1+\beta)(1-x(e)).$$

Then we can use this to estimate

$$-2u(e)(1-x(e)) \geq -\frac{2}{1+\beta}u(e)x(e),$$

as the only edges, where the term $-2u(e)(1-x(e))$ appeared, were edges that we rounded up. Of course then for seed edges this implies

$$-\frac{2}{1+\beta}u(e)x(e) \geq -\frac{2}{1+\beta}(u(e)x(e) + b(v) + b(w))$$

and for extension edges

$$-\frac{2}{1+\beta}u(e)x(e) \geq -\frac{2}{1+\beta}(u(e)x(e) + b(v_e)).$$

For a $2+\beta$-approximation, we need

$$\frac{2}{1+\beta} = \beta \Leftrightarrow \beta \in \{-2, 1\}.$$

Since $\beta \geq 0$, we choose $\beta := 1$ and hence $\alpha := \frac{2}{3}$ for a 3-approximation.

5.5 The integrality gap of the LP

We will now prove that the integrality gap of the LP is 3. This means that using the approach discussed here, we can not achieve a better approximation guarantee.

Theorem 5.5.1. *The integrality gap of the LP-relaxation given in Section 5.2 is at least* 3.

Proof. For an instance I denote by $\mathrm{OPT}(I)$ the value of an optimum (integral) solution and by $\mathrm{OPT}_{\mathrm{LP}}(I)$ the value of an optimum LP-solution. We will provide a sequence I_k of instances, such that $\lim_{k\to\infty} \frac{\mathrm{OPT}(I_k)}{\mathrm{OPT}_{\mathrm{LP}}(I_k)} = 3$.

Let $0 < \epsilon < \frac{1}{2}$. For some $k \geq 3$, let G be a k-star. That is a graph with $k+1$ vertices $\{C\} \cup \{v_1, \ldots, v_k\}$ and edges $\{\{C, v_i\} \mid i = 1, \ldots, k\}$. We set $c \equiv 0$ and $\gamma := 1$. For all edges $e \in E(G)$, we set $u(e) := \frac{1}{2}$. Finally, we set $b(C) := 1 - \epsilon$ and $b(v_i) := \epsilon$ for

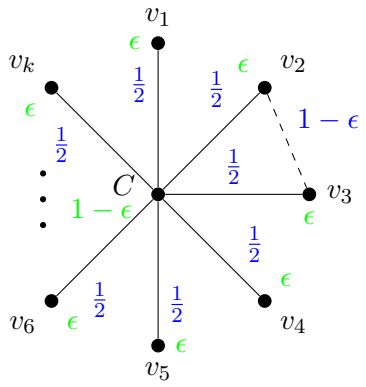


Figure 5.4: A picture showing the instance described in the proof of Theorem 5.5. The solid edges belong to the k-star. Edge loads are marked in blue and node loads are marked in green. The dashed edge is an example for the edges added to complete the graph.

$i = 1, \ldots, k$. In order to get to a complete graph, we extend G, by adding edges between all pairs v_i, v_j for $i < j$ and set $c(\{v_i, v_j\}) = 0$ and $u(\{v_i, v_j\}) := 1 - \epsilon$. Clearly, the resulting u and c are metric. We will denote this instance by $I_{k,\epsilon}$. A depiction of $I_{k,\epsilon}$ is shown in Figure 5.4.

In an optimum integral solution to this instance, no edge can be used. This means that $\mathrm{OPT}(I_{k,\epsilon}) = k + 1$. Now we solve the *LP* using the algorithm from section 2. It will first consider the edges $\{C, v_i\}$ and only afterwards the others. W.l.o.g. we may assume that the edges are considered in the order $e_1 := \{C, v_1\}, \ldots, e_k := \{C, v_k\}$. The first edge will get the value $y(e_1) := 2 - (1 - \epsilon + \epsilon) = 1$ and hence

$$x(e_1) = \frac{y(e_1)}{1 + \frac{1}{2}} = \frac{2}{3}.$$

The edge e_i will get the value $y(e_i) := i + 1 - (i + \epsilon) = 1 - \epsilon$ and hence

$$x(e_i) = \frac{2}{3} - \frac{2\epsilon}{3}.$$

After edge e_k has been considered, the support graph is a tree. This means, that the algorithm will not consider the remaining edges. This shows that

$$\mathrm{OPT}_{\mathrm{LP}}(I_{k,\epsilon}) = |V| - \sum_{i=1,\ldots,k} x(e_i) = k + 1 - \frac{2}{3} - (k-1)\left(\frac{2}{3} - \frac{\epsilon}{3}\right) = \frac{k}{3} + \frac{(k-1)\epsilon}{3} + 1.$$

Setting $I_k := I_{k,\frac{1}{k^2}}$, we get

$$\lim_{k\to\infty} \frac{\mathrm{OPT}(I_k)}{\mathrm{OPT}_{\mathrm{LP}}(I_k)} = \lim_{k\to\infty} \frac{k+1}{\frac{k}{3} + \frac{(k-1)}{3k^2} + 1} = 3$$

□

Remark. *Adding inequalities that forbid edges that can not be taken in an integral solution will not help to lower the integrality gap. We can replace the central vertex C in the proof of Theorem 9 by a K_l for some $l \in \mathbb{N}$ with small edge and vertex loads that add up to* 1*. This will only change the "+1" by a constant depending on l and hence not change the asymptotic ratio of* OPT *to* $\mathrm{OPT}_{\mathrm{LP}}$*.*

Together with the upper bound of 3 given by the analysis of the algorithm, we can conclude:

Corollary 5.5.2. *The integrality gap of the LP is* 3*.*

6 Experimental Results

In the previous chapters, we have developed new algorithms and a framework for solving the buffering problem. Now we test the framework and speedups in practice.

We conducted experiments on real-world instances taken from the industrial flow at IBM. Unfortunately, no public benchmarks are available. Our goal is to evaluate our base algorithm and different versions using our speedup techniques in terms of running time and quality.

We start by explaining our experimental setup and the different combinations of objectives and speedups that we will test our algorithm with in Section 6.1. Furthermore, we try to give some insight into the testbed we are using for our experiments.

In our first set of experiments in Section 6.2, we test the base version of our algorithm that can give us an approximation guarantee and various variants that use different sets of speedup techniques. Since the base version is very slow, we keep the objective simple and the instances small. We compare the variants with heuristic speedups to the base version. For the results, we use the objective value of the solutions as a metric. We show that we can gain significant speedups without a significant loss in result quality in these small instances. Additionally, we show that we can gain even higher speedups if we sacrifice some quality of results.

Finally, we test the fastest variants of our algorithm in a more practical setting in Section 6.3. In this setting, we do not restrict the instance sizes. Furthermore, we evaluate the solutions using higher order delay models and also measure the power usage. Then we compare our solutions to the input solutions. These are taken from the end of the main timing and power optimization step of an industrial flow. Note that the designs we are using are not yet timing clean. Moreover, we compare our solutions to solutions of two other algorithms that are used in practice: The fast buffering by Bartoschek et al. [Bar+09] and a dynamic program similar to the one by Shi and Li [SL05], but with additional net-based layer assignment.

6.1 Setup and Testbed

All experiments were run on an Oracle Linux 8.9 server with two AMD EPYC 7742 64-core processors and 2TB memory.

We ran at most 15 jobs at once, each using up to 8 threads. Each buffering instance is optimized by only one thread if it is solved by a single dynamic program. If it is split into smaller instances by the iterative clustering heuristic, each sub instance may be solved by a different thread. Nonetheless, all running times will be total sequential running times, if not explicitly stated otherwise.

The buffering instances occur as a part of the IBM industrial design flow on 5nm designs. An overview of the designs is displayed in Table 6.2 in the end of this section. The names are anonymized to protect the intellectual property of our industrial partner. The designs labeled C1 – C7 are taken from a testbed of smaller macros, while the designs labelled I1 – I6 are taken from real integration level runs that designers were interested in at the time of extraction.[1] No design could meet all constraints after finishing the main timing and power optimization section of the flow. For each name, the table lists the width and height of the chip image, the maximum distance between vertices that we used in our Hanan graph, the number of routing layers available, as well as the number of width/spacing + layer assignments that were available. Furthermore, we show the minimum and maximum slew that was allowed, as well as the δ for our acc_{const} slew-accuracy function (see Chapter 3.3). Finally, we show the capacitance limit, and some information about the available repeater library, consisting of the minimum and maximum repeater sink capacitance and the number of buffers and inverters available. This should give some context for the running times.

In order to evaluate the different heuristic speedup techniques for our dynamic program, we tested our implementation in various settings that will be explained in the following. Table 6.1 shows a summary of the different setups.

Since we are optimizing real world instances, we also include the required arrival times augmented by a linear approximation of their dependencies in the sink slews. In order to achieve this, we use the negated required arrival time as an initial delay. This means that value in the entry for each sink $t \in T$ becomes the negated slack $-\,rat(t) + delay(t)$, which we then minimize. Furthermore, we have to shift the negated required arrival times by a constant to make them nonnegative, so that we can apply α-dominance in the algorithm. This will only change the slack at the root by a constant.

For each instance (created from a cluster or solved directly), we try to solve it three times. First with a merge arborescence (see Chapter 4.4) that fixes the topology and additional guide penalties. The merge arborescence is based on a delay bounded Steiner arborescence (see Chapter 1.3.2) that we compute using the bicriteria algorithm by Held and Rotter [HR13]. The guide penalties are realized by an additional delay penalty in the lower bound function. They depend on the rectilinear distance of the current vertex to the parent vertex in the merge arborescence, and the number of merges that still have to be performed before all sinks are connected.

[1] Time of extraction is November 2023.

Configuration	Transition	Slew comparison by Bucket	Repeater Map	Pruning by objective cost	no α-dominance	Heuristic future cost increase
singsimpld	rise	×	×	×	×	×
singsimpld_heurfc	rise	×	×	×	×	✓
singsimpld_objcostp	rise	×	×	✓	×	×
singsimpld_noappr	rise	×	×	✓	✓	×
singsimpld_noappr_objcostp_heurfc	rise	×	×	×	✓	✓
singbuckld	rise	×	×	×	×	×
singbuckld_heurfc	rise	✓	×	×	×	✓
singbuckld_objcostp	rise	✓	×	✓	×	×
singbuckld_noappr	rise	✓	×	×	✓	×
singbuckld_noappr_objcostp_heurfc	rise	✓	×	✓	✓	✓
simpld_noappr_objcostp_heurfc	rise/fall	×	×	✓	✓	✓
buckld_noappr_objcostp_heurfc	rise/fall	✓	×	✓	✓	✓
repbuckld_noappr_objcostp_heurfc	rise/fall	✓	✓	✓	✓	✓

Table 6.1: The different configurations that we will be testing in this chapter. The first column shows the shorthand that we will use in tables. The second column shows, which transitions we are optimizing at the source. The remaining columns show which speedup heuristics for our dynamic program are active in the respective setting.

The second pass takes the upper bound that we computed in the first pass. It still uses the merge arborescence, but no guide penalties. Finally, we use the best upper bound that we have found so far to start our third pass that does not fix the topology.

In order to increase our chances of finding a solution, we also return the best solution label so far if the algorithm stops for other reasons than finishing (if it runs into its iteration limit: 20000000 for the first two passes each and 10000000 for the second pass; or if it becomes too slow $\leq$50 iterations per second).

For our slew accuracy function, we use the constant shift (acc_{const}, see Chapter 3.3) by the amount denoted as δ in the Table 6.2. When we generate multiple slew guesses with acc_{const}, two consecutive guesses differ exactly by the constant δ.

The first objective is denoted by *simpld* in our tables. It optimizes the maximum delay over all sinks and both transitions (rise and fall) plus a penalty for slew or load violations (1000 times the difference to the limit). It also includes both rise and fall slew. A solution is dominated, when both rise and fall slew fulfill the dominance requirements. We add labels for all pairs of slew guesses for rise and fall.

The objective *buckld* additionally to simpld defines a bucket for each label. The bucket is identical the bucket from Chapter 4.1. Here, we use the buckets for the slew part of the dominance checks.

The objectives *singsimpld* and *singbuckld* work as buckld and simpld, but only optimize the rise delay at the source. For the other transition, the slew limit is still propagated and used to detect slew violations.

Finally, there is the objective *repbuckld*, which works just like buckld, but uses the repeater map from Chapter 4.5.1. The repeater map is supposed to increase the cor-

relation between a solution of our algorithm and a legalized solution by dividing the placement area into cells and only allowing a limited number of repeaters to be placed in each cell. It is used as a heuristic component. This means that it is not used for dominance checking, but when too many repeaters are placed in the same cell the evaluation operator will evaluate it to ∞. This also means that we might not find a solution if more than the maximum number of repeaters have to be placed in the cell. However, we did not observe this.

The variant singsimpld without any appended modifiers denotes the implementation that yields an approximation guarantee.

Additionally, we also test multiple heuristic speedup techniques.

The shorthand *objcostp* means objective cost pruning. Instead of comparing each tree-monotone component separately for dominance checking, we compare only the objective cost ($\chi(\omega(l)_o)$ for a label l).

Next, we have *noappr*, where we are not using the α-dominance from Chapter 3.3, but normal dominance, losing the approximation guarantee α.

Finally, we have *heurfc*, which heuristically modifies the lower bounds. When running with required arrival times, it can happen that one of the sinks is uncritical compared to the others. The algorithm would first only select labels that connect only this sink, because they have the lowest value. It will do this until the cost reaches the lower bounds for the total tree cost. To balance this difference, we increase the lower bound a label if its connected sinks are much cheaper to connect than the remaining sinks. In detail:

For $t \in T$, denote by $\mathit{pred}(t)$ the minimum lower bound cost to connect to that sink. Informal it is $-\mathit{rat}(t) + \mathfrak{Lb}(t)$. If l connects the sink set $I \subsetneq T$, we increase the lower bound of l for each $t \in I$ by

$$0.95 \cdot \max\left\{0, \max_{t' \in T \setminus I} \mathit{pred}(t') - \max_{t' \in I} \mathit{pred}(t')\right\}.$$

Additionally, we test two other modifications. Their use is marked separately. The first one is to only allow merges at Hanan vertices, that is, not at the subdivisions. The second one is the repeater penalty from Chapter 4.5.1.

Name	Width in nm	Height in nm	Max Vertex Distance in nm	Num Routing Layers	Num Assignments	Slew values in ps			Cap Limit in fF	Repeater Sink Cap in fF		Num Buffers	Num Inverters
						Lower limit	Upper limit	δ		Min.	Max.		
C1	144000	134784	15711	8	3	2.80	70.00	5.60	208.64	0.41	18.90	0	19
C2	218880	513216	15622	10	6	2.80	70.00	5.60	208.64	0.41	18.90	0	19
C3	215040	114048	15622	17	11	2.80	70.00	5.60	208.64	0.41	18.90	0	19
C4	51840	37584	15622	8	3	2.80	70.00	5.60	208.64	0.41	18.90	0	19
C5	90240	73872	15622	8	3	2.80	70.00	5.60	208.64	0.41	18.90	0	19
C6	387840	181440	15548	12	6	2.52	63.00	5.04	210.43	0.41	19.31	0	31
C7	215040	114048	15622	17	11	2.80	70.00	5.60	208.64	0.41	18.90	0	19
I1	209280	207360	15517	10	6	2.52	63.00	5.04	337.41	0.66	20.15	0	29
I2	414720	282528	15618	10	6	2.80	70.00	5.60	217.86	0.43	19.91	0	19
I3	280320	199584	15618	12	6	2.52	63.00	5.04	217.86	0.43	19.91	0	19
I4	1505280	5412096	15618	16	12	2.52	63.00	5.04	217.86	0.43	19.91	0	19
I5	211200	178848	15517	12	6	2.52	63.00	5.04	337.41	0.66	20.15	0	29
I6	691200	736128	15539	16	11	2.52	63.00	5.04	321.54	0.63	61.83	14	44

Table 6.2: Our testbed in anonymized form. The numbers are supposed to give an idea of the complexity of the buffering problem on the respective chip. "Width" and "Height" refer to the dimensions of the chip image. "Max Vertex Distance" means the maximum distance between two vertices in the Hanan graph after subdividing longer edges. "Num Routing Layers" shows the number of available routing layers. "Num Assignments" lists the number of width/spacing + minimum layer assignments that are allowed at that point in the flow. The "Slew values" show the lower and upper slew limits and the δ that we used for our acc_{const} slew accuracy function. We additionally list the capacitance limit as "Cap Limit". Finally, we show the minimum and maximum sink pin capacitances of the available repeaters, as well as the number of available buffers and inverters in the library.

	Running time in s						Add Fraction		
Config	Min.	Max.	Total	Avg.	Successes	Fails	Min.	Max.	Avg.
singsimpld	0.0132	8361.01	12394.42	263.71	47	44	0.013	0.938	0.357
singbuckld	0.0126	34903.03	99847.80	2627.57	38	56	0.013	0.981	0.356
singsimpld_objcostp	0.0134	293263.95	307407.96	6027.61	51	57	0.013	0.862	0.328
singbuckld_objcostp	0.0075	35972.14	103413.26	2522.27	41	70	0.013	0.981	0.343
singsimpld_noappr	0.0051	2066.92	5979.77	52.45	114	7	0.003	0.652	0.100
singbuckld_noappr	0.0044	40.13	266.98	3.14	85	36	0.004	0.301	0.084
singsimpld_noappr_objcostp_heurfc	0.0042	1298.83	4456.52	39.09	114	7	0.003	0.650	0.100
singbuckld_noappr_objcostp_heurfc	0.0041	46.39	288.22	3.39	85	36	0.004	0.299	0.084

Table 6.3: Runtimes of our algorithm in its base version (singsimpld) that can give us an approximation guarantee and with the different speedup heuristics for instances with 1 sink. Running times are only measured on the successful instances. Successful means that the algorithm returned a solution for which we could prove that its cost is within the approximation guarantee. "Add Fraction" shows the number of labels that are not immediately dominated divided by the total number of labels that are created.

6.2 Evaluation of the Speedup Techniques

In our first set of experiments, we want to evaluate the different speedup heuristics for our dynamic program against the correct base version (singsimpld, without any heuristic speedups) that will give us an approximation guarantee. Since the base version is very slow, we only optimize a single transition and instances with only 1–3 sinks. From each chip of our testbed and for each number instance size (1-3 sinks), we selected the 5 instances with the highest worst slack and up to 5 instances with load or slew violations. Some designs did not have 5 instances with load or slew violations in each instance size. Note that for instances with at most 2 sinks, there is only one topology. For instances with only one sink, we directly test our algorithm without guide penalties or merge arborescence. For instances with two sinks, we only use the merge arborescence with guide penalties and do not run the test with fixed topology without guide penalties.

We compare the configurations regarding their running time and result. Here result means the objective cost given by the algorithm, not the worst slack after timing with a higher order delay model. We consider an instance as successful if we are able to prove that our result is within the approximation guarantee. This means that if a run with fixed topology or guide penalties returned a solution, but not the unconstrained run, then we count the instance as unsolved or fail. This can happen if the algorithm does not find a solution at all or if is stopped due to our iteration limits (see Section 6.1.

For the running time on instances with one sink, we present two tables. In Table 6.3, we show the overall result for the whole testbed. The first column shows the name of the configuration. The next few columns show the running time of the successful instances. The number of successful instances and failed instances is shown in the last two columns. Table 6.4 contains only the running times on the instances where all different versions of the algorithm finished successfully.

	Running time in s			
Config	Min.	Max.	Total	Avg.
singsimpld	0.0273	2684.48	3241.13	111.76
singbuckld	0.0126	31569.88	32555.61	1122.61
singsimpld_objcostp	0.0265	2665.08	3223.40	111.15
singbuckld_objcostp	0.0075	22307.28	23153.41	798.39
singsimpld_noappr	0.0051	0.56	4.98	0.17
singbuckld_noappr	0.0044	0.06	0.68	0.02
singsimpld_noappr_objcostp_heurfc	0.0042	0.47	4.58	0.16
singbuckld_noappr_objcostp_heurfc	0.0041	0.07	0.69	0.02

Table 6.4: Runtime comparison of the algorithm with different heuristics for instances with 1 sink, restricted to the set of instances that finished for all configurations.

We observe that the greatest speedup is achieved by adding the noappr heuristic to our dynamic program, which uses dominance instead of α-dominance to prune the labels. Only pruning by the objective cost (objcostp) leads to small improvements in running time, when using the non-bucket implementation as a base. With buckets, it only decreases the running time when not paired with the noappr heuristic.

Interestingly, using buckets increases the running time in the base implementation on the instances where both the bucket and non-bucket version finish. In the runs that use the noappr heuristic on the other hand, it decreases the running time significantly.

Additionally, we can take a look at the successes and fails in Table 6.3. We observe that using buckets decreases the success rate, when paired with the noappr heuristic. So it is faster on the instances it solves, but does not solve all instances.

Let us try to give some explanations for these effects. The effect that fewer instances are solved by the bucket version could be explained by over-pruning. This could also play a role in the increased running time of the bucket version without additional heuristic speedups. When the base version without buckets already solves the instance, it must have found "good" labels fast. If too many of those are pruned in the bucket version, but the overall number of labels is not sufficiently reduced, then it takes longer to find a solution. The decrease in running time if the buckets are paired with the noappr heuristic could then be explained by the radical label reduction.

To support these explanations, let us take a look at Figures 6.1, 6.2 and 6.3.

In Figure 6.1, we can see the number of labels that have been added over the course of the algorithm without immediately being pruned, over the total number of iterations that the algorithm performed before finding the solution. If we take a closer look, we see that the configurations without the noappr heuristic add almost an order of magnitude more labels for the same number of iterations.

The plot in Figure 6.2 confirms that the running time increases with the number of labels added, even if the algorithm returns after the same number of iterations. Nonetheless, we can see in Figure 6.3 that the dependence of the running time on the number of iterations still is polynomial.

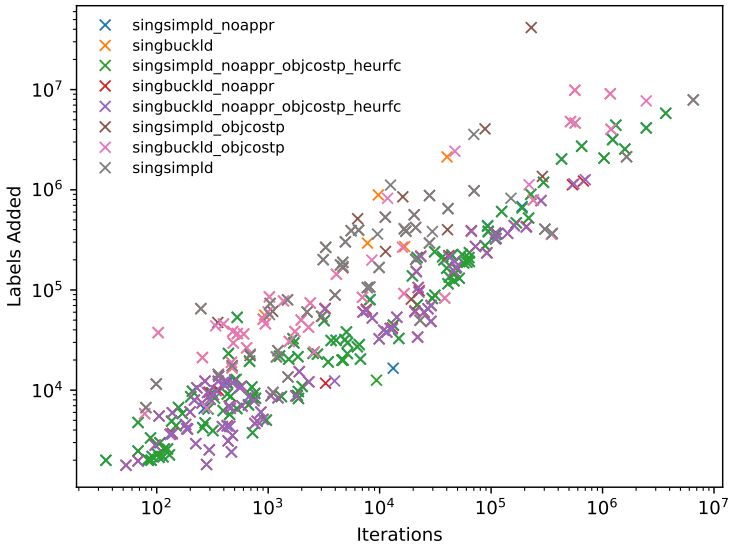


Figure 6.1: Total number of labels added per total number of iterations of individual runs. Labels added means that they were not immediately pruned by dominance or checking against the upper bound.

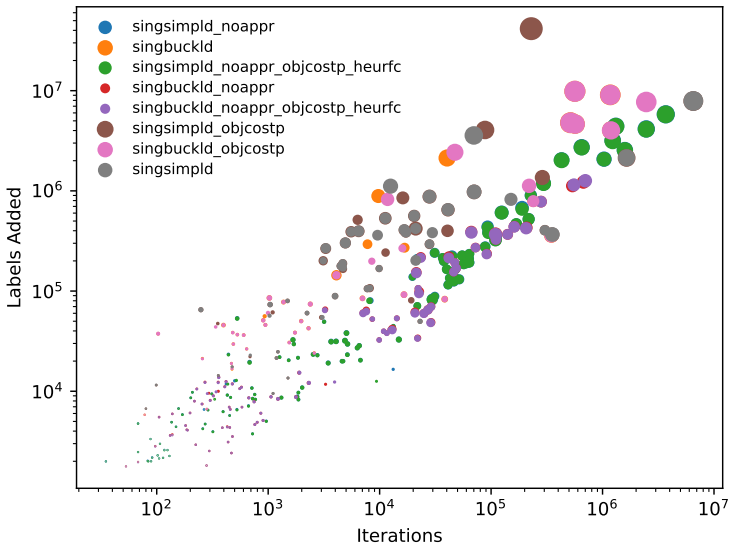


Figure 6.2: Total number of labels added per total number of iterations as a scatter plot, with marker size depending (logarithmically) on running time.

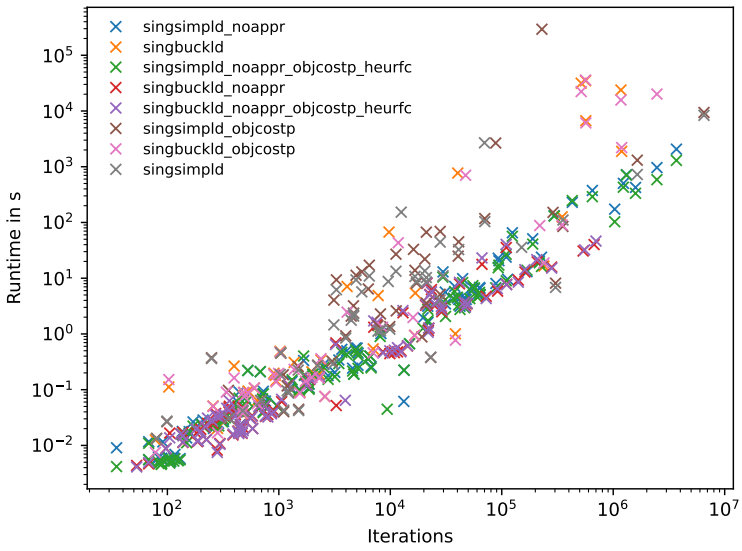


Figure 6.3: Total runtime plotted over total number of iterations used per run.

	Running time in s					
Config	Min.	Max.	Total	Avg.	Successes	Fails
singsimpld	39080.7108	93604.60	310246.52	62049.30	5	114
singsimpld_heurfc	22820.5229	71316.02	295020.24	49170.04	6	113
singsimpld_noappr	67.3058	81257.95	2232914.99	25665.69	87	32
singbuckld_noappr	10.7877	51371.01	582282.17	14930.31	39	80
singsimpld_noappr_objcostp_heurfc	1.8127	10292.97	110601.77	1189.27	93	26
singbuckld_noappr_objcostp_heurfc	4.3770	4503.82	8341.49	208.54	40	79

(a) Running times on instances with 2 sinks. Only the guided and unconstrained version run.

	Running time in s					
Config	Min.	Max.	Total	Avg.	Successes	Fails
singsimpld	inf	-inf	0.00	0.00	0	109
singsimpld_heurfc	inf	-inf	0.00	0.00	0	109
singsimpld_noappr	211.1966	49578.58	271741.58	13587.08	20	89
singbuckld_noappr	6499.3575	42479.43	110455.77	15779.40	7	102
singsimpld_noappr_objcostp_heurfc	3.7465	13038.88	96787.24	3337.49	29	80
singbuckld_noappr_objcostp_heurfc	150.0600	10272.97	26593.70	3799.10	7	102

(b) Running times on instances with 3 sinks. Merge arborescence with guides, merge arborescence without guides and the unconstrained version are tried. The singsimpld and singsimpld_heurfc were not able to solve any instances within the iteration limit.

Table 6.5: Running times of our algorithm in its base version (singsimpld) that can give us an approximation guarantee and with speedup heuristics for instances with 2 and 3 sinks. Running times are only measured on the successful instances.

	Absolute difference in ps				Relative difference		
Config	Min.	Max.	Total	Avg.	Min.	Max.	Avg.
singsimpld	-0.97	0.07	-2.03	-0.04	0.722	1.018	0.991
singbuckld	0.00	11.36	98.82	2.60	1.000	2.702	1.382
singsimpld_objcostp	-0.97	0.21	-1.81	-0.04	0.722	1.024	0.992
singbuckld_objcostp	0.00	11.36	106.46	2.60	1.000	2.730	1.396
singsimpld_noappr	0.00	0.00	0.00	0.00	1.000	1.000	1.000
singbuckld_noappr	-0.64	9.57	277.74	3.35	0.975	3.414	1.416
singsimpld_noappr_objcostp_heurfc	0.00	0.00	0.00	0.00	1.000	1.000	1.000
singbuckld_noappr_objcostp_heurfc	-0.64	9.57	277.74	3.35	0.975	3.414	1.416

Table 6.6: The per instance difference in the result values for instances with 1 sink. singsimpld_noappr serves as the basis, because it has the most successful runs (see Table 6.3). A negative value (value <1 for relative differences) means that the configuration was better than the reference singsimpld_noappr.

	Absolute difference in ps				Relative difference		
Config	Min.	Max.	Total	Avg.	Min.	Max.	Avg.
singsimpld	-0.28	5.11	8.32	1.66	0.589	1.338	1.024
singsimpld_heurfc	-0.31	5.11	10.72	1.79	0.548	1.338	1.028
singsimpld_noappr	-8.92	0.41	-49.36	-0.46	0.411	1.021	0.984
singbuckld_noappr	-1.39	20.56	209.40	5.37	0.412	2.045	1.151
singsimpld_noappr_objcostp_heurfc	0.00	0.00	0.00	0.00	1.000	1.000	1.000
singbuckld_noappr_objcostp_heurfc	-1.12	20.56	214.92	5.37	0.970	2.045	1.165

Table 6.7: The per instance difference in the result values for instances with 2–3 sinks. singsimpld_noappr_objcostp_heurfc serves as the basis, because it has the most successful runs (see Table 6.5). A negative value (value <1 for relative differences) means that the configuration was better than the reference singsimpld_noappr_objcostp_heurfc.

For instances with 2 and 3 sinks the running time results are shown in Table 6.5. The observed trends continue. The largest runtime reduction is achieved by adding the noappr heuristic. For the larger instances, we see the effect of the heuristic future cost increase for the first time. It leads to a notable reduction as well. We were not able to solve instances with 3 sinks in the base version.

Finally, we compare the results of the different configurations for only one sink in Table 6.6. For each instance, we can only compare runs that finished. Since singsimpld_noappr had the most successful runs, we are using it as a baseline.

We can see that we only lose very little by adding the noappr heuristic and even less by using the objcostp heuristic. Using buckets comes with a significant loss in quality.

Similarly, for the instances with 2–3 sinks in Table 6.7, we see that the loss due to the noappr heuristic is within the error of the algorithm. We also see that the heurfc extension to the lower bounds only leads to mild loss in quality.

	Absolute difference in ps				Relative difference		
Config	Min.	Max.	Total	Avg.	Min.	Max.	Avg.
Guided 2 Sinks	-1.13	22.74	260.85	3.14	0.899	1.731	1.102
Guided 3 Sinks	-1.14	8.79	46.11	2.43	0.901	1.640	1.113
Free 3 Sinks	-0.00	1.43	2.69	0.13	1.000	1.080	1.008

Table 6.8: The per instance difference in the result values from the pre-solve runs to the solution with unconstrained topology. We are using runs with singsim-pld_noappr here, since it has the most solutions while being very accurate. A negative value (value <1 for relative differences) means that the configuration was better than the reference (unconstrained topology).

In total, we conclude that using both the noappr and objcostp heuristics on the base version without buckets has the best tradeoff in terms of running time and quality. For instances with more sinks, we will nonetheless have to resort to the bucket variants, because the running time would get too high.

Another interesting question is: How much worse are solutions if we constrain the topology. This is answered for at most 3 sinks in Table 6.8. In practice, we lose almost nothing by constraining the topology for 3 sinks. By adding the guides, we only lose 2-3ps on average, but can be significantly worse. We see degradations up to 22ps or a factor of 1.7.

6.3 Evaluation in a Practical Setting

We do not only want to use our algorithm for benchmarking, but also in practice. We have seen in the previous section, that the high accuracy versions of our dynamic program are too slow for this. For the faster variants however, the running times are good enough. In this second set of experiments, we want to find out how the faster versions of our dynamic program perform in practice.

For this purpose, we will first take a look at the practical running times of the fast variants. Then we compare our solutions to the input solutions (recall that they are taken from the end of the main timing optimization step) and to solutions computed with a dynamic program similar to the one by Shi and Li [SL05], but with additional net based layer and wire-width/spacing assignment. We evaluate the timing by AWE with C_{eff}-computations for the gate delay.

Let us first review the detailed parameters and settings that we use. For this practical mode, we directly solve instances with up to 5 sinks. Instances with more sinks will be solved with the iterative clustering approach from Chapter 4.6. It assigns a criticality to each sink. Then, it iteratively chooses a cutoff criticality and applies a clustering algorithm to the instances that are more critical than the cutoff. The clustering guarantees that the clusters do not become too large. Then a root is chosen for each cluster to create a smaller instance of the TMCMAP. The new instances are solved with multiple solution candidates and finally the roots are used as sinks for the next iteration.

	Runtimes 1-5 Sinks in min				Runtimes 6-10 Sinks in min					Runtimes >10 Sinks in min						
Algorithm	Min.	Max.	Total	Avg.	Min.	Max.	Total	Avg.	TW	Min.	Max.	Total	Avg.	TW	Successes	Fails
simpld_noappr_objcostp_heurfc	<1	672	1167	31	722	722	722	722	727	inf	-inf	<1	<1	<1	39	25
simpld_noappr_objcostp_heurfc + OH	<1	672	1054	30	766	766	766	766	782	inf	-inf	<1	<1	<1	37	37
simpld_noappr_objcostp_heurfc + OH + RP	<1	1358	3238	76	209	209	209	209	220	566	566	566	566	1791	45	32
buckld_noappr_objcostp_heurfc	<1	1083	2738	54	667	667	667	667	672	4196	4196	4196	4196	2263	53	17
buckld_noappr_objcostp_heurfc + OH	<1	1022	2869	63	566	566	566	566	576	5101	5103	10204	5102	7214	49	28
buckld_noappr_objcostp_heurfc + OH + RP	<1	91	395	8	4	10	14	7	21	239	2629	3969	1323	2714	56	14
repbuckld_noappr_objcostp_heurfc	<1	877	2704	62	115	115	115	115	147	inf	-inf	<1	<1	<1	45	17
repbuckld_noappr_objcostp_heurfc + OH	<1	930	2744	60	96	96	96	96	137	5861	5861	5861	5861	5426	48	27
repbuckld_noappr_objcostp_heurfc + OH + RP	<1	117	496	10	6	388	406	136	294	18	910	2937	588	2502	58	22

Table 6.9: Running times of the different algorithm variants. They are grouped by sink sizes. "Min.", "Max.", "Total", "Avg." show sequential running times (also running times of different threads are summed up). "TW" shows the summed Wall (start to finish) running times. Instances with 1–5 sinks were directly solved, while instances with more sinks use the iterative clustering heuristic from Chapter 4.6. The last two columns show the number of runs that returned a solution (successes) and the number of runs that did not return a solution.

We test the configurations (see Table 6.1)

- simpld_noappr_objcostp_heurfc,
- buckld_noappr_objcostp_heurfc and
- repbuckld_noappr_objcostp_heurfc,

which in particular all optimize the maximum of rise and fall delay. There are two additional settings that we will use with them. The first one will be denoted as an appended *+ OH*, which stands for "only merge at Hanan vertices". So if this is added, we do not allow merges at subdivisions of the edges in the Hanan grid. The second one is denoted as an appended *+ RP* and refers to the repeater penalty from Chapter 4.5.1. The repeater penalty adds a penalty for each inserted repeater. It is based on the capacitance change that we get by inserting the repeater. A decrease in capacitance reduces the penalty, while an increase increases the penalty.

As test instances, we take the 5 instances with the worst slack on each of our chips (see Table 6.2) and up to 5 instances with capacitance or slew violations.

Now let us take a look at the results. We start with the running time results in Table 6.9. It shows the sequential running times grouped by number of sinks. The first column that shows the running time on instances with 1–5 sinks coincides with the directly solved instances. The remaining running time columns all show results obtained using the iterative clustering. Since we are using multiple threads in the clustering approach, we also show the summed up wall times (TW). Here, wall time means the time from start to finish, regardless of the number of threads that were used. In the last two columns, we see the number of successes and fails. In contrast to Section 6.2, we count runs as successful that returned any solution (even if the algorithm was stopped).

We see that the versions with both additions OH and RP perform best, both in the number of successes and average running time. They are also able to solve much larger instances. Among the three configurations, repbuckld_noappr_objcostp_heurfc is

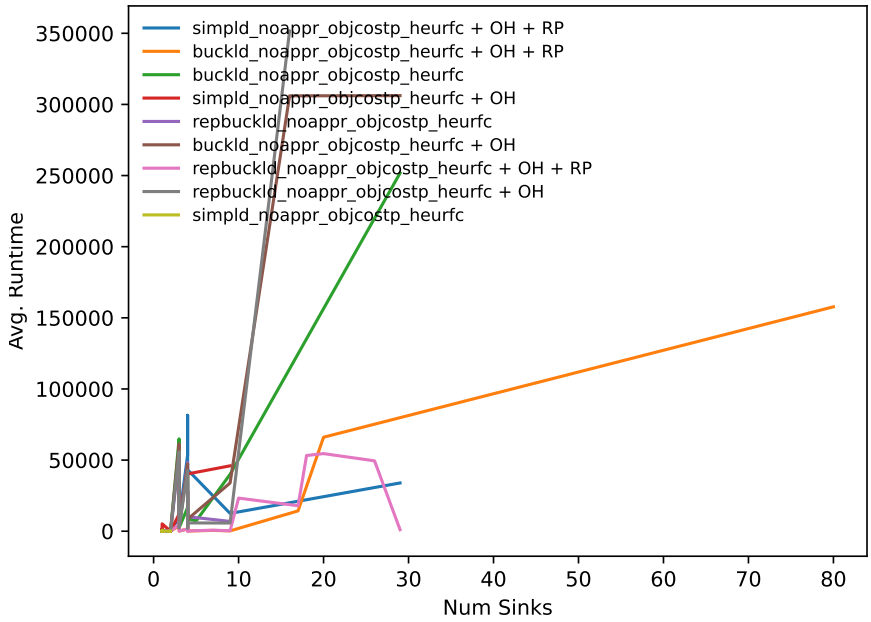


Figure 6.4: A plot of the average sequential running times (instances with the same number of sinks are averaged) of the different algorithm versions over the number of sinks.

the most successful one. We can also see that the running times for up to 10 sinks are reasonable enough to use in some scenarios. Instead of solving these instances by hand, a designer could let our algorithm run on them during the night. This could also be viable for larger running times, as a designer can only solve one instance at a time, but a computer can solve multiple instances in parallel. Finally, we see that we get a factor of roughly 1.5 in the wall time compared to the sequential running time. Though, for solving single instances, we would be able to gain a higher speedup, because here some of the threads may be occupied with solving their own instance.

For a better view of the development of the running times with growing instance sizes, we also show the average sequential running per instance size in Figure 6.4. This further undermines our conclusion that adding OH and RP leads to a significant speedup. Furthermore, we see that the running time grows roughly linear in the instance size for the clustering approach. This is what we would expect.

Finally, we take a look at the results. In Table 6.10, we see differences of our solutions to the input solutions. It shows the three metrics *worst slack* (Wsl), *total negative slack* (Tns) and *power usage*. Total negative slack refers to the sum of all negative slacks at the sinks of our instance. The timing values have been computed by the timing engine EinsTimer, using AWE+C_{eff}. We always show the improvement, so that positive numbers mean that we are better than the input, while negative numbers mean that we are worse. Furthermore, we show one additional algorithm: Fast Buffering. It is

	Wsl Improvement in ps				Tns Improvement in ps				Power Improvement in nW				
Algorithm	Min.	Max.	Total	Avg.	Min.	Max.	Total	Avg.	Min.	Max.	Total	Avg.	Rel. [%]
Fast Buffering, $\xi = 0.8$	-82.62	100.19	169.74	18.86	-437.62	852.92	781.96	86.88	-4.47	1.99	-2.28	-0.25	-5.8
simpld_noappr_objcostp_heurfc	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	0.0
simpld_noappr_objcostp_heurfc + OH	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-8.00	-8.00	-8.00	-8.00	-26.7
simpld_noappr_objcostp_heurfc + OH + RP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-2.79	-1.18	-3.97	-1.99	-13.2
buckld_noappr_objcostp_heurfc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-1.68	-1.68	-1.68	-1.68	-1680.0
buckld_noappr_objcostp_heurfc + OH	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-8.25	-1.68	-9.93	-4.96	-33.1
buckld_noappr_objcostp_heurfc + OH + RP	0.00	96.06	202.75	40.55	-151.08	871.35	638.38	127.68	-8.03	1.06	-12.92	-2.58	-34.6
repbuckld_noappr_objcostp_heurfc	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	0.0
repbuckld_noappr_objcostp_heurfc + OH	0.00	130.36	130.36	65.18	0.00	1028.07	1028.07	514.04	-7.40	-1.08	-8.48	-4.24	-27.9
repbuckld_noappr_objcostp_heurfc + OH + RP	-0.94	96.66	192.53	32.09	0.00	871.35	1016.66	169.44	-8.03	0.27	-11.27	-1.88	-33.0

Table 6.10: The improvement in worst slack (Wsl), Total negative slack (Tns) and Power of our (and fast buffering) solutions over the input solutions. Positive numbers mean that we were better. "Fast Buffering" refers to the fast buffering routine by Bartoschek et al. [Bar+09].

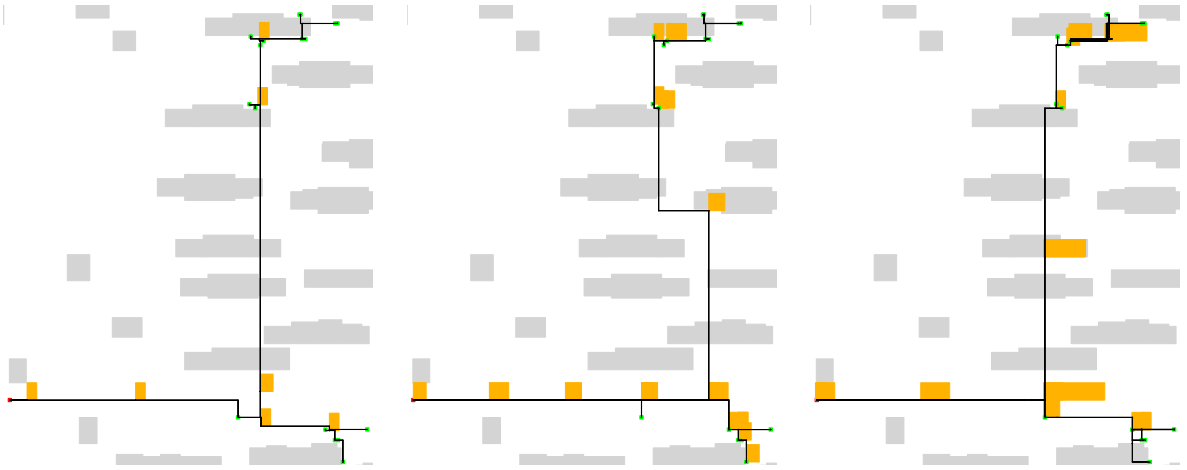
	Wsl Improvement in ps				Tns Improvement in ps				Power Improvement in nW				
Algorithm	Min.	Max.	Total	Avg.	Min.	Max.	Total	Avg.	Min.	Max.	Total	Avg.	Rel. [%]
Fast Buffering, $\xi = 0.8$	-82.19	8.97	-166.12	-18.46	-1175.37	155.95	-1731.08	-192.34	-0.75	5.36	3.70	0.41	8.2
simpld_noappr_objcostp_heurfc	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	0.0
simpld_noappr_objcostp_heurfc + OH	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.83	1.83	1.83	1.83	4.6
simpld_noappr_objcostp_heurfc + OH + RP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.92	7.04	6.12	3.06	15.3
buckld_noappr_objcostp_heurfc	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-1.42	-1.42	-1.42	-1.42	-394.4
buckld_noappr_objcostp_heurfc + OH	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-1.42	1.58	0.16	0.08	0.4
buckld_noappr_objcostp_heurfc + OH + RP	-28.20	52.04	31.69	6.34	-981.76	452.14	-764.92	-152.98	-6.20	1.80	-7.12	-1.42	-16.5
repbuckld_noappr_objcostp_heurfc	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	inf	-inf	0.00	0.00	0.0
repbuckld_noappr_objcostp_heurfc + OH	-3.35	0.00	-3.35	-1.67	0.00	0.00	0.00	0.00	-1.06	2.43	1.37	0.69	3.4
repbuckld_noappr_objcostp_heurfc + OH + RP	-1.39	52.63	59.08	9.85	-3.04	452.14	457.45	76.24	-1.44	1.80	-2.14	-0.36	-4.9

Table 6.11: The improvement of our (and fast buffering solutions) over solutions by a dynamic program similar to the one by Shi and Li [SL05]. For more information see Table 6.10.

the dynamic program by Bartoschek et al. [Bar+09] that is also allowed to change the topology slightly. We run it with $\xi = 0.8$, which means the costs in the algorithm are computed as $0.8 \cdot \mathit{slackcost} + 0.2 \cdot \mathit{powercost}$.

We see that repbuckld_noappr_objcostp_heurfc + OH + RP gives us significant improvements in timing. It outperforms the fast buffering both on average and in total, even though the fast buffering was able to find solutions on all instances, while our algorithm did not. Another observation is that the runs without the repeater penalty improve even more in the timing. In terms of power, it is exactly the reverse. We use more power than the input solutions, and more power than fast buffering. Among our tests, the version with repeater penalty uses less power than the versions without it. This is part of the desired effect. Furthermore, the versions with repeater map use less power for higher delay reductions. This again, was part of the desired effect.

In Table 6.11, we show the same columns as in Table 6.10, but this time we compare to a dynamic program similar to the one by Shi and Li [SL05], but with net based layer and wire-width/spacing assignment. In this comparison, our improvements are no longer as great as before, but still, we are able to improve by almost 10ps on average in the repbuckld_noappr_objcostp_heurfc + OH + RP setting. On the other hand the increase in power usage is much lower.



(a) Input solution.
Wsl= −86ps. The solution uses too few and too small repeaters and chains sinks.

(b) Solution by dynamic program similar to Shi and Li [SL05].
Wsl= −42ps. The solution uses too small repeaters and chains sinks.

(c) Our solution.
Wsl= 2ps. The solution connects multiple sinks in parallel to medium-sized repeaters.

Figure 6.5: Three different solutions of the same instance. Blockages are gray, sinks are green and the root is red (bottom left). Repeaters are the orange rectangles. They have been inflated to make them visible on this picture.

Let us conclude by comparing the solutions in Figure 6.5. The input solution in Figure 6.5a has a worst slack of −86ps (1.84nW power usage). It uses too few and small repeaters. The solution by the dynamic program similar to the one by Shi and Li [SL05] in Figure 6.5b has a worst slack of −42ps (1.17nW power usage). It uses more, but still too few repeaters. Furthermore, the sinks are connected into a long path one by one similar to the input solution, so that the delay to the furthest sinks is large. Finally, our solution in Figure 6.5c has a worst slack of 2ps (2.03nW power usage). It connects the furthest sinks in parallel to a medium-sized repeater. And only after that uses a single path to connect them, which leads to the improvements.

Conclusion

In this thesis, we took a closer look at the buffering problem. We first reviewed the literature and examined how different algorithms try to solve it. We saw that the general trend is to include more aspects into the modeling, e.g. power consumption, placement space usage, routing congestion. Then, we gave an overview over the different aspects that are relevant for buffering, explained how they can be modelled and what decisions we made in modelling. Based on these considerations, we presented a new problem formulation that captures more aspects and allows more degrees of freedom than all the previous algorithms and formulations.

We presented an (exponential time) algorithm that is able to solve this problem either approximately, or even optimally, with higher running time. Both variants depend on a slew accuracy function. We proved that the algorithm finds an optimum solution if start with a certain set of initial slews. Then, we showed how to guess slews that allow us to find an optimum solution. In this way, we solved the inherent problem of including slew in buffering algorithms. Then, we demonstrated that the algorithm can be extended or modified to include higher order delay models or increase the degrees of freedom at the cost of ignoring the slew.

We also developed multiple speed up techniques. We propose a data structure that efficiently maintains the labels in our algorithm. Then, we showed how to construct a sparse graph representing the routing space and buffering positions that retains certain properties that are helpful for finding good solutions. We presented a way to compute an (almost) feasible lower bound function in practice. From there, we turned towards heuristics for which we cannot prove performance guarantees. Among those, we developed a fast way to restrict the topology of solutions. As a final step, we presented a new iterative clustering heuristic that is timing aware. It uses the Capacitated Tree Cover Problem with Edge Loads as a black box.

We present a new 3-approximation algorithm for this problem that runs in $\mathcal{O}(m \log n)$ time. An underlying LP-relaxation can be solved optimally using a greedy algorithm, despite its exponential number of inequalities. We round this LP and applied a well-known splitting technique to get to a feasible solution. In a final step, we proved that the integrality gap of our LP-relaxation is 3, which means that our analysis was optimal.

Finally, we tested our algorithm both in a benchmark setting and a practical setting. We showed that we can gain significant speedups by augmenting our algorithm with

heuristic speedup techniques. For some of these speedup techniques, we saw that in practice they do not lead to a notable loss in result qualities. Then, we examined how the fastest variants of our algorithm perform in a practical setting and saw that it can give us significant improvements at a high runtime cost. Unfortunately, it is still too slow to use it as a default tool in a design flow, but it can be used to optimize a few hard instances that otherwise would have to be optimized manually by a designer.

Our algorithm allows us for the first time to benchmark existing buffering algorithms and heuristics. By comparing solutions of other algorithms to our solutions, we can find out if other solutions are good or if they can be improved. By examining the structural differences, we can improve other algorithms in practice. In fact, BonnRouteBuffer has been improved this way multiple times during the time I was working on this thesis.

Nonetheless, there are still open questions:

We are proving that there is a slew-approximate optimum solution in the search space that our algorithm explores. This is much stronger than just finding a solution with an approximation guarantee. Is it possible to relax the α-dominance and keep the approximation guarantee?

In Section 4.3, we saw that we can still obtain an optimum solution, when our lower bound does not exactly satisfy the feasibility constraints. Can we obtain additional benefits from a correct feasible lower bound?

Can we extend the interval-based buffering to also use slews? A possible idea for that would be to guess slews similar to the main algorithm. Then we store a separate resistance for ranges of input slews of repeaters and shrink the legal intervals to also keep the slew in the respective range.

Can we leave the selection of sinks in the iterative clustering heuristic to the algorithm for the Capacitated Tree Cover Problem with Edge Loads, by including criticality differences in the model?

Can we get rid of the assumption that $u(e) < u(f) \Rightarrow c(e) \leq c(f)$ for the Capacitated Tree Cover Problem with Edge Loads, or is the problem harder if we drop it? I was not able to show that the problem becomes harder. On the other hand, I suspected that if we drop this, there might be instances where all optimum solutions to the LP contain cycles in the support graph. But I was unable to construct a (metric) example, where this is the case, nor was I able to show that this can not happen.

Finally, there is still a lot of practical work that could be done to improve the correlation between solutions from my implementation of the TMCMAP algorithm and the inserted solutions.

Bibliography

[AD97] Charles J. Alpert and Anirudh Devgan. "Wire segmenting for improved buffer insertion". In: *Proceedings of the 34th Annual Design Automation Conference*. ACM Press, 1997, pp. 588–593.

[ADQ99] Charles J. Alpert, Anirudh Devgan, and Stephen T. Quay. "Buffer insertion with accurate gate and interconnect delay computation". In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. ACM Press, 1999, pp. 479–484.

[AHL06] Esther M. Arkin, Refael Hassin, and Asaf Levin. "Approximations for minimum and min-max vehicle routing problems". In: *Journal of Algorithms* 59.1 (2006), pp. 1–18.

[Aut+17] C. Auth et al. "A 10nm high performance and low-power CMOS technology featuring 3rd generation FinFET transistors, self-aligned quad patterning, contact over active gate and cobalt local interconnects". In: *2017 IEEE International Electron Devices Meeting (IEDM)*. 2017, pp. 29.1.1–29.1.4.

[Bak+96] George A Baker, George A Baker Jr, Peter Graves-Morris, and Susan S Baker. *Pade Approximants: Encyclopedia of Mathematics and It's Applications, Vol. 59 George A. Baker, Jr., Peter Graves-Morris*. Vol. 59. Cambridge University Press, 1996.

[Bar+09] Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. "Fast buffering for optimizing worst slack and resource consumption in repeater trees". In: *Proceedings of the 2009 International Symposium on Physical design*. ACM, 2009, pp. 43–50.

[Bar+10] Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen. "The repeater tree construction problem". In: *Information Processing Letters* 110.24 (2010), pp. 1079–1083.

[Bar14] Christoph Bartoschek. "Fast repeater tree construction". Techn. Report No. 141085. Forschungsinstitut für Diskrete Mathematik. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2014.

[BCD89] C. Leonard Berman, J. Lawrence Carter, and Ken F. Day. "The fanout problem: From theory to practice". In: *Proceedings of the Decennial Caltech Conference on VLSI on Advanced Research in VLSI*. MIT Press, 1989, pp. 69–99.

[Bla22] Daniel Blankenburg. "Resource sharing revisited: Local weak duality and optimal convergence". In: *30th Annual European Symposium on Algorithms (ESA 2022)*. Vol. 244. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 20:1–20:14.

[Boe+94] Kenneth D. Boese, Andrew B. Kahng, Bernard A. McCoy, and Gabriel Robins. "Rectilinear Steiner trees with minimum Elmore delay". In: *Proceedings of the 31st Annual Design Automation Conference*. ACM Press, 1994, pp. 381–386.

[Cre19] Pascal Cremer. "Algorithms for cell layout". Techn. Report No. 191179. Forschungsinstitut für Diskrete Mathematik. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2019.

[CW07] Chris Chu and Yiu-Chung Wong. "FLUTE: Fast lookup table based rectilinear Steiner minimal tree algorithm for VLSI design". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.1 (2007), pp. 70–83.

[CY00] Jason Cong and Xin Yuan. "Routing tree construction under fixed buffer locations". In: *Proceedings of the 37th Annual Design Automation Conference*. ACM Press, 2000, pp. 379–384.

[Dab+23] Siad Daboul, Stephan Held, Bento Natura, and Daniel Rotter. "Global interconnect optimization". In: *ACM Transactions on Design Automation of Electronic Systems* 28.5 (2023).

[Dab21] Siad Daboul. "Global timing optimization in chip design". Techn. Report No. 211221. Forschungsinstitut für Diskrete Mathematik. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2021.

[Dij59] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[Elm48] William C Elmore. "The transient response of damped linear networks with particular regard to wideband amplifiers". In: *Journal of Applied Physics* 19.1 (1948), pp. 55–63.

[Eve+04] Guy Even, Naveen Garg, Jochen Könemann, Ramamoorthi Ravi, and Amitabh Sinha. "Min–max tree covers of graphs". In: *Operations Research Letters* 32.4 (2004), pp. 309–315.

[FT87] Michael L. Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[GJ77] Michael R. Garey and David S. Johnson. "The rectilinear Steiner tree problem is NP-complete". In: *SIAM Journal on Applied Mathematics* 32.4 (1977), pp. 826–834.

[Glu23] Adrian Glubrecht. "RC-aware trees in global routing". Techn. Report No. 231293. Forschungsinstitut für Diskrete Mathematik. BS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2023.

[GW01] Youxin Gao and D.F. Wong. "A graph based algorithm for optimal buffer insertion under accurate delay models". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. IEEE Computer Society, 2001, pp. 535–539.

[Han66] Maurice Hanan. "On Steiner's problem with rectilinear distance". In: *SIAM Journal on Applied Mathematics* 14.2 (1966), pp. 255–265.

[Hel+11] Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. "Combinatorial optimization in VLSI design". In: *Combinatorial Optimization - Methods and Applications*. NATO Science for Peace and Security Series - D: Information and Communication Security 31 (2011), pp. 33–96.

[Hel+17] Stephan Held, Dirk Müller, Daniel Rotter, Rudolf Scheifele, Vera Traub, and Jens Vygen. "Global routing with timing constraints". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2 (2017), pp. 406–419.

[HL02] Miloš Hrkić and John Lillis. "S-tree: A technique for buffered routing tree synthesis". In: *Proceedings of the 39th Annual Design Automation Conference*. ACM Press, 2002, pp. 578–583.

[HLA09] Shiyan Hu, Zhuo Li, and Charles J. Alpert. "A fully polynomial time approximation scheme for timing driven minimum cost buffer insertion". In: *Proceedings of the 46th Annual Design Automation Conference*. ACM Press, 2009, pp. 424–429.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[Hou+19] S. Hougardy, B. Korte, P. Rocca, I. Prinz, and J. Vygen. *Mathematik und Ästhetik des Chipdesigns*. Springer, 2019.

[HR13] Stephan Held and Daniel Rotter. "Shallow-light Steiner arborescences with vertex delays". In: *Integer Programming and Combinatorial Optimization: 16th International Conference, IPCO 2013*. Springer. 2013, pp. 229–241.

[HR18] Stephan Held and Benjamin Rockel. "Exact algorithms for delay-bounded Steiner arborescences". In: *Proceedings of the 55th Annual Design Automation Conference*. ACM Press, 2018, pp. 1–6.

[HS85] Dorit S. Hochbaum and David B. Shmoys. “A best possible heuristic for the k-center problem”. In: *Mathematics of operations research* 10.2 (1985), pp. 180–184.

[HSV17] Stefan Hougardy, Jannik Silvanus, and Jens Vygen. “Dijkstra meets Steiner: A fast exact goal-oriented Steiner tree algorithm”. In: *Mathematical Programming Computation* 9.2 (2017), pp. 135–202.

[Hu+03] Jiang Hu, Charles J. Alpert, Stephen T. Quay, and Gopal Gandham. “Buffer insertion with adaptive blockage avoidance”. In: vol. 22. 4. 2003, pp. 492–498.

[Hu+18] Jiang Hu, Ying Zhou, Yaoguang Wei, Stephen T. Quay, Lakshmi Reddy, Gustavo Tellez, and Gi-Joon Nam. “Interconnect optimization considering multiple critical paths”. In: *Proceedings of the 2018 International Symposium on Physical Design*. ACM, 2018, pp. 132–138.

[Huf52] David A. Huffman. “A method for the construction of minimum-redundancy codes”. In: *Proceedings of the IRE* 40.9 (1952), pp. 1098–1101.

[HV22] Stephan Held and Jens Vygen. *Chip Design. Theory and Practice of VLSI Layout*. Lecture notes. Preliminary version. 2022.

[Hwa76] Frank K. Hwang. “On Steiner minimal trees with rectilinear distance”. In: *SIAM Journal on Applied Mathematics* 30.1 (1976), pp. 104–114.

[Ihm23] Benjamin Ihme. “Timing models and buffering”. MS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2023.

[Juh+18] Daniel Juhl, David M. Warme, Pawel Winter, and Martin Zachariasen. “The GeoSteiner software package for computing Steiner trees in the plane: An updated computational study”. In: *Mathematical Programming Computation* 10.4 (2018), pp. 487–532.

[Kam22] Kamal Y Kamal. “The silicon age: Trends in semiconductor devices industry”. In: *Journal of Engineering Science & Technology Review* 15.1 (2022), pp. 110–115.

[Kan+02] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. “A local search approximation algorithm for k-means clustering”. In: *Proceedings of the 18th Annual Symposium on Computational Geometry*. ACM, 2002, pp. 10–18.

[Kar72] Richard M. Karp. “Reducibility among combinatorial problems”. In: *Proceedings of a Symposium on the Complexity of Computer Computations*. The IBM Research Symposia Series. Plenum Press, 1972, pp. 85–103.

[KMB81] Lawrence Kou, George Markowsky, and Leonard Berman. “A fast algorithm for Steiner trees”. In: *Acta Informatica* 15 (1981), pp. 141–145.

[KS00] Samir Khuller and Yoram J. Sussmann. “The capacitated k-center problem”. In: *SIAM Journal on Discrete Mathematics* 13.3 (2000), pp. 403–418.

[KS14] M. Reza Khani and Mohammad R. Salavatipour. "Improved approximation algorithms for the min-max tree cover and bounded tree cover problems". In: *Algorithmica* 69.2 (2014), pp. 443–460.

[KV18] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 6th ed. Springer, 2018.

[KV84] M.R. Kramer and J. Van Leeuwen. "The complexity of wire-routing and finding the minimum area layouts for arbitrary VLSI circuits". In: *Advances in Computing Research* 2 (1984), pp. 129–146.

[LCL11] Yen-Hung Lin, Shu-Hsin Chang, and Yih-Lang Li. "Critical-trunk-based obstacle-avoiding rectilinear Steiner tree routings and buffer insertion for delay and slack optimization". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.9 (2011), pp. 1335–1348.

[LCL96a] John Lillis, Chung-Kuan Cheng, and Ting-Ting Y. Lin. "Optimal wire sizing and buffer insertion for low power and a generalized delay model". In: *IEEE Journal of Solid-State Circuits* 31.3 (1996), pp. 437–447.

[LCL96b] John Lillis, Chung-Kuan Cheng, and Ting-Ting Y. Lin. "Simultaneous routing and buffer insertion for high performance interconnect". In: *6th Great Lakes Symposium on VLSI GLS-VLSI '96*. IEEE Computer Society, 1996, pp. 148–153.

[Lil+96] John Lillis, Chung-Kuan Cheng, Ting-Ting Y Lin, and Ching-Yen Ho. "New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing". In: *Proceedings of the 33rd Annual Design Automation Conference*. ACM Press, 1996, pp. 395–400.

[Llo82] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2 (1982), pp. 129–137.

[LM91] Shen Lin and Malgorzata Marek-Sadowska. "A fast and efficient algorithm for determining fanout trees in large networks". In: *Proceedings of the European Conference on Design Automation.* EEE Computer Society, 1991, pp. 539–544.

[LZS12] Zhuo Li, Ying Zhou, and Weiping Shi. "$O(mn)$ time algorithm for optimal buffer insertion of nets with m sinks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.3 (2012), pp. 437–441.

[MRV11] Dirk Müller, Klaus Radke, and Jens Vygen. "Faster min–max resource sharing in theory and practice". In: *Mathematical Programming Computation* 3.1 (2011), pp. 1–35.

[MV08] Jens Maßberg and Jens Vygen. "Approximation algorithms for a facility location problem with service capacities". In: *ACM Transactions of Algorithms* 4.4 (2008), 50:1–50:15.

[Nag75] Laurence W. Nagel. "SPICE2: A computer program to simulate semiconductor circuits". In: *Technical Report ERL M520, Electronics Research Laboratory, University of California, Berkeley* (1975).

[Nat17] Bento Natura. "Algorithms for routing and buffer insertion". Techn. Report No. 171155. Forschungsinstitut für Diskrete Mathematik. MS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[OC96] Takumi Okamoto and Jason Cong. "Interconnect layout optimization by simultaneous steiner tree construction and buffer insertion". In: *Proceedings of the IEEE International Conference on Computer-Aided Design*. Citeseer. 1996, pp. 44–49.

[Per16] Rodeon Permin. "A near-optimum algorithm for cost-based buffering". MS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2016.

[PR90] Lawrence T. Pillage and Ronald A. Rohrer. "Asymptotic waveform evaluation for timing analysis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 9.4 (1990), pp. 352–366.

[Roc18] Benjamin Marc Rockel. "Exact algorithms for interconnect optimization". Techn. Report No. 191178. Forschungsinstitut für Diskrete Mathematik. MS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2018.

[Roc24] Benjamin Rockel-Wolff. "A fast 3-approximation for the capacitated tree cover problem with edge loads". In: *19th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2024)*. Ed. by Hans L. Bodlaender. Vol. 294. Leibniz International Proceedings in Informatics (LIPIcs). Full version: arXiv: 2404.10638 [cs.DS]. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 39:1–39:14.

[Rom15] Daniel Romen. "Cost-based buffering for multiple resources". Techn. Report No. 151098. Forschungsinstitut für Diskrete Mathematik. MS thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2015.

[Rot17] Daniel Rotter. "Timing-constrained global routing with buffered Steiner trees". Techn. Report No. 171138. Forschungsinstitut für Diskrete Mathematik. PhD thesis. Rheinische Friedrich-Wilhelms-Universität Bonn, 2017.

[Sap04] Sachin Sapatnekar. *Timing*. Springer Science & Business Media, 2004.

[Sch17] Rudolf Scheifele. "Steiner trees with bounded RC-delay". In: *Algorithmica* 78 (2017), pp. 86–109.

[Sch22] Stephan Schwartz. "An overview of graph covering and partitioning". In: *Discrete Mathematics* 345.8 (2022), pp. 112884–112900.

[SL05] Weiping Shi and Zhuo Li. "A fast algorithm for optimal buffer insertion". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24.6 (2005), pp. 879–891.

[SLA04] Weiping Shi, Zhuo Li, and Charles J. Alpert. "Complexity analysis and speedup techniques for optimal buffer insertion with minimum cost". In: *ASP-DAC 2004: Asia and South Pacific Design Automation Conference 2004 (IEEE Cat. No. 04EX753)*. IEEE Computer Society, 2004, pp. 609–614.

[SLP98] Amir H. Salek, Jinan Lou, and Massoud Pedram. "A simultaneous routing tree construction and fanout optimization algorithm". In: *Proceedings of the 1998 IEEE/ACM International Conference on Computer-Aided Design*. ACM / IEEE Computer Society, 1998, pp. 625–630.

[SLP99] Amir H Salek, Jinan Lou, and Massoud Pedram. "MERLIN: Semi-order-independent hierarchical buffered routing tree generation using local neighborhood search". In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. ACM Press, 1999, pp. 472–478.

[Tou90] Hervé Jacques Touati. "Performance-oriented technology mapping". PhD thesis. University of California, 1990.

[TT22] Vera Traub and Thorben Tröbst. "A fast $(2 + \frac{2}{7})$-approximation algorithm for capacitated cycle covering". In: *Mathematical Programming* 192.1 (2022), pp. 497–518.

[Van90] Lukas PPP Van Ginneken. "Buffer placement in distributed RC-tree networks for minimal Elmore delay". In: *1990 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1990, pp. 865–868.

[Vyg06] Jens Vygen. "Slack in static timing analysis". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.9 (2006), pp. 1876–1885.

[XXZ12] Zhou Xu, Dongsheng Xu, and Wenbin Zhu. "Approximation results for a min–max location-routing problem". In: *Discrete Applied Mathematics* 160.3 (2012), pp. 306–320.

[YL19] Wei Yu and Zhaohui Liu. "Better approximability results for min–max tree/cycle/path cover problems". In: *Journal of Combinatorial Optimization* 37.2 (2019), pp. 563–578.

[Zho+99] Hai Zhou, DF Wong, I-Min Liu, and Adnan Aziz. "Simultaneous routing and buffer insertion with restrictions on buffer locations". In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. ACM Press, 1999, pp. 96–99.