Facility Location and Clock Tree Synthesis

Dissertation

zur Erlangung des Doktorgrades

der Mathematisch-Naturwissenschaftlichen Fakultät

der Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Jens Uwe Maßberg

aus Hannover

im Oktober 2009

Erstgutachter:	Prof. Dr. Jens Vygen
Zweitgutachter:	Prof. Dr. Dr. h.c. Bernhard Korte

Tag der Promotion: 18.12.2009

Contents

Introduction

A fundamental problem in chip design is the construction of networks that distribute an electrical signal from a given source to a set of sinks. In most cases these networks are repeater trees. A repeater tree consists of horizontal and vertical wires connecting the source and the sinks. Moreover, it contains repeaters (inverters or buffers) which 'refresh' the signal. Recent chips contain millions of repeater trees.

Clock trees are very similar to repeater trees. Their task is to distribute a clock signal from a source to a large set of sinks. Beside the repeaters, they can contain additional special circuits modifying the clock signal. One main goal of clock tree construction is to minimize the power consumption. Typically, 80% to 90% of the total power consumption of a clock tree occurs in the last stage of the tree.

This motivates us to have a closer look at this stage. It contains circuits (inverters or special circuits) driving the sinks. Each sink is assigned to one of these drivers and each driver is connected to the sinks that are assigned to it by a rectilinear Steiner tree. Each driver has to drive the electrical capacitance of the tree plus the input capacitances of its sinks. The total capacitance it can drive is limited. The power consumption of the last stage is equivalent to the power consumption of the trees (proportional to their length) plus the power consumption of the drivers. Typically, the drivers of the last stage are inverters or special circuits of the same size. In both cases the power consumption of all drivers is the same. The key component of every clock tree construction tool is to build a last stage that minimizes power consumption while satisfying the capacitance limits of the drivers.

Mathematically, this task can be formulated as the following Sink Clustering Problem: Given a finite set D of sinks with positions $p(v) \in \mathbb{R}^2$ in the plane and demands (input capacitances) $d(v) \in \mathbb{R}_{\geq 0}$ for all $v \in D$, a facility opening cost $f \in \mathbb{R}_{>0}$ (power consumption of a driver) and a load limit (capacitance limit) $u \in \mathbb{R}_{>0}$, the task is to find a partition $D = D_1 \dot\cup \cdots \dot\cup D_k$ of D and, for all $1 \leq i \leq k$, a rectilinear Steiner tree S_i for $\{p(v) | \, v \in D_i\}$. Each cluster (D_i, S_i), $1 \leq i \leq k$, has to keep the load limit, that means $\sum_{e\in E(S_i)} c(e) + \sum_{s \in D_i} d(s) \leq u$. The goal is to minimize the weighted sum of the length of all Steiner trees plus the number of clusters, i.e. minimize $\sum_{i=1}^{k} \sum_{e\in E(S_i)} c(e) + kf$.

In Chapter 1 we study the Sink Clustering Problem for general metrics and present the first constant-factor approximation algorithms for it. Moreover, we develop several lower bounds that partly rely on fundamental connections of the Sink Clustering Problem to matroid theory. In Section 1.4 we show experimental results on real-life instances from clock tree design. The cost (power consumption) of the solutions computed by our algorithm is in average only about 10% over the lower bounds.

Clock trees have to satisfy several timing constraints. Typically, the signal has to

arrive at all sinks of a clock tree at the same time. In this case we have the SINK CLUSTERING PROBLEM as defined above. However, there are clock trees where the signal has to arrive at each sink within an individual required arrival time window. In this case sinks can only be driven by the same circuit if the time windows of these sinks have a point of time in common. In Chapter 2 we study this generalization of the SINK CLUSTERING PROBLEM, describe algorithms and lower bounds.
In Chapter 3 we present our algorithm BonnClock for sythesizing clock trees. Its key component is the SINK CLUSTERING ALGORITHM as presented in Chapter 1 and generalized in Chapter 2. The clustering algorithms are used to construct the last stage of the clock tree and also to build upper parts of the tree. It is combined with a sink partitioning approach that is a generalization of the well-known H-trees. BonnClock has become the standard tool used by IBM Microelectronics for constructing clock trees. It has been used for the design of hundreds of most complex chips.
Finally, we study the REPEATER TREE TOPOLOGY PROBLEM in Chapter 4. In contrast to clock trees, the timing constraints of repeater trees are different: The signal has to arrive at a sink not later than a given individual required arrival time and therefore cannot arrive too early. We propose a greedy algorithm that can produce trees that are either almost length optimal or timing optimal. Moreover, we present theoretical results, including a characterization of all online algorithms for the minimax and the almost minimax problem, and improved lower bounds for repeater trees.

I would like to express my gratitude to my supervisors, Professor Dr. Jens Vygen and Professor Dr. Bernhard Korte. Without their support and the excellent working conditions they provide at the Research Institute for Discrete Mathematics this thesis would not have been possible.
I am grateful to all past and present colleagues in the VLSI group, especially Christoph Bartoschek, Michael Gester, Dr. Stephan Held and Professor Dr. Dieter Rautenbach. Special thanks go to Florian Berger, Michael Gester, Hendrik Guhlich and Tobias Gödderz who implemented parts of the tool BonnClock.
I would further like to thank all people at IBM who helped integrating, promoting and supporting BonnClock, namely Andreas Arp, Günther Hutzl, Dr. Jürgen Koehl, Michael Koch, Karsten Muuss and especially Dr. Matthias Ringe who has been a great supporter over the last years. Thanks go also to Paul Lippens from Magma Design Automation for sharing his clustering results.
Further thanks go to Jan, Michael, Sarah and Tobias who proofread parts of this thesis.
Finally, I thank Sarah for her loving support and Valentin and Lucie who, unfortunately, have seen their father too rarely in the previous months.

1 The Sink Clustering Problem

In this chapter we deal with the SINK CLUSTERING PROBLEM motivated in the introduction. In Chapter 1.1 we introduce the problem and present approximation algorithms with constant approximation guarantees. In Chapter 1.2 we present two post optimization algorithms that can further improve an existing clustering. Based on some fundamental characteristics of the problem and relations to matroid theory we establish four lower bounds in Chapter 1.3. Finally, we present experimental results of the algorithms and the lower bounds.
Section 1.1 is based on joint work with Jens Vygen (Maßberg and Vygen [2008]).

1.1 Problem and Algorithm

1.1.1 Problem Definition

First, we introduce the SINK CLUSTERING PROBLEM. Given a metric space (V, c), we consider a finite set D of sinks with positions $p : D \to V$ and demands $d : D \to \mathbb{R}_{\geq 0}$. We want to 'cluster' these sinks. A cluster is a pair (D', S') with $D' \subseteq D$ and S' a Steiner tree on $\{p(v) | \, v \in D'\}$ in (V, c). A clustering of D is a family $\{(D_i, S_i)\}_{i=1...k}$ of clusters so that $D = D_1 \dot{\cup} \cdots \dot{\cup} D_k$ is a partition of D.
We aim for a clustering where each cluster keeps some capacitance constraints and which has minimal cost.

SINK CLUSTERING PROBLEM

Instance: A metric space (V, c), a finite set D of terminals/customers with positions $p(v) \in V$ and demands $d(v) \in \mathbb{R}_{\geq 0}$ for all $v \in D$, a facility opening cost $f \in \mathbb{R}_{>0}$ and a load limit $u \in \mathbb{R}_{>0}$.

Task: Find a $k \in \mathbb{N}$ and a clustering $\{(D_i, S_i)\}_{i=1...k}$ of D, $D = D_1 \dot{\cup} \cdots \dot{\cup} D_k$, so that the load limit is kept, i.e.

$$\sum_{e \in E(S_i)} c(e) + \sum_{s \in D_i} d(s) \leq u \qquad \text{for } i = 1, \ldots, k, \tag{1.1}$$

and the clustering cost

$$\sum_{i=1}^{k} \left(\sum_{e \in E(S_i)} c(e) \right) + kf \tag{1.2}$$

is minimized.

Notation

For simplicity we identify sinks with their positions. So e.g. a Steiner tree on a set $D' \subset D$ is a Steiner tree on $\{p(v) | \, v \in D'\}$. Moreover, we denote the cost of a tree G by $c(G) := \sum_{e \in E(G)} c(e)$ and the demand of a set $D' \subseteq D$ by $d(D') := \sum_{v \in D'} d(v)$.
A clustering is called feasible if inequality (1.1) is kept for all its clusters.
A solution of the Sink Clustering Problem can be written as $\big(k, \{(D_i, S_i)\}_{i \in \{1,\dots,k\}}\big)$ where $k \in \mathbb{N}$, $\dot{\bigcup}_{i=1}^{n} D_i = D$ and (D_i, S_i) are clusters for $i \in \{1, \dots, k\}$. A solution is feasible if its clustering is feasible.

Previous Work

A variant of the Sink Clustering Problem, where the sinks are connected by spanning trees instead of Steiner trees, has been studied by Shelar [2007]. He proposes a Kruskal-like strategy. First for each sink a cluster is created. Then the algorithm tries to merge two clusters by adding a shortest edge between two sinks, one of each cluster. If the load limit is still kept, the two clusters are replaced by the new one. Neither Shelar could proof an approximation guarantee for his algorithm, nor could he establish lower bounds for the problem. Note that the following algorithms and lower bounds can be modified to compute spanning trees instead of Steiner trees in a straightforward way. Beside the work of Shelar, the problem has not been studied yet.

1.1.2 Complexity

As the problem contains the Steiner Minimum Tree Problem and the Bin Packing Problem it is strongly NP-complete and $MAXSNP$-hard.

Lemma 1.1. *There is no polynomial $\left(\frac{3}{2} - \epsilon\right)$-approximation algorithm for any $\epsilon > 0$ for the* Sink Clustering Problem *in any metric unless $P = NP$.*

Proof. The Sink Clustering Problem is a generalization of the Bin-Packing Problem. Let $a_1, \dots, a_n \leq 1$ be an instance of the Bin-Packing Problem. Choose an $s \in V$ and set $D := \{v_1, \dots, v_n\}$, $p(v_i) := s$ and $d(v_i) := a_i$ for $i \in \{1, \dots, n\}$. Finally, set $u := 1$ and $f := 1$. Then a solution of the Sink Clustering Problem directly corresponds to a solution of the Bin Packing Problem. It has been shown that there is no $\left(\frac{3}{2} - \epsilon\right)$-approximation algorithm for the latter problem unless $P = NP$ (see Garey and Johnson [1979]). $\square$
Moreover, we can show an even stronger result:

Lemma 1.2. *There is no polynomial $(2 - \epsilon)$-approximation algorithm for any $\epsilon > 0$ for the* Sink Clustering Problem *in metrics where the Steiner tree problem cannot be solved optimally in polynomial time unless $P = NP$.*

Proof. Assume there is a polynomial $(2-\epsilon)$-approximation algorithm. As the Steiner Minimum Tree Problem is NP-complete, the corresponding Steiner Tree Decision Problem is NP-hard: Given a set of terminals $T = \{t_1, \dots, t_n\} \subset V$ and a number $k \in \mathbb{R}_+$, is there a Steiner tree of length $\leq k$?

We construct an instance for the Sink Clustering Problem by setting $D := \{v_1, \dots, v_n\}$, $p(v_i) := t_i$ and $d(v_i) := 0$ for $i \in \{1, \dots, n\}$. Moreover, set $u := k$ and $f > k\frac{2-\epsilon}{\epsilon}$. If there exists no Steiner tree on T of length $\leq k$ then any feasible clustering consists of at least two clusters and has cost $\geq 2f$.
Otherwise, let S be a Steiner tree of length $\leq k$. Then the cluster (D, S) is feasible, i.e. there exists a feasible clustering of cost $\leq f + k$. In this case the approximation algorithm computes a solution of cost at most $(2-\epsilon)\cdot(f+k) < (2-\epsilon)\cdot(f+\frac{\epsilon}{2-\epsilon}f) = 2f$. Thus the solution consists of exactly one cluster.
Hence the approximation algorithm computes a clustering containing one single cluster if and only if there is a Steiner tree of length at most k, but that means we can decide the Steiner Tree Decision Problem. $\square$
By constructing multiple copies of the instance used in the last proof it can be shown that there is not even an asymptotic $(2 - \epsilon)$ approximation algorithm for the Sink Clustering Problem in metrics where the Steiner Minimum Tree Problem is NP-complete.

1.1.3 The Sink Clustering Algorithm

We now describe the first approximation algorithm for the Sink Clustering Problem. It relies on the following simple idea: First we build a minimum spanning tree on D. Then we remove some of the longest edges, which decomposes the tree into some components. Finally, we split up components that are overloaded.

Definition 1.3. *The Steiner ratio is the supremum of the length of a minimum spanning tree over the length of a minimum Steiner tree:*

$$\alpha := \sup_{D' \subseteq V, D' \text{ finite}, |D'|>1} \frac{c(MST(D'))}{c(SMT(D'))}.$$

Remark 1.4. *In any metric space the Steiner ratio is at most* 2*. In the special case of the rectilinear plane* $(\mathbb{R}^2, l_1)$ *Hwang [1976] has shown that* $\alpha = \frac{3}{2}$.

A First Lower Bound

Now we show a simple way to compute a lower bound for the cost of an optimal solution. It will help us to prove the approximation factor of the below algorithm. In Chapter 1.3 we will develop more sophisticated lower bounds. But first we have to introduce some definitions.

Definition 1.5. *A k-spanning forest and a k-Steiner on D is a forest F with $V(F) = D$ and $D \subseteq V(F) \subseteq V$, respectively, containing exactly k components.*

Every clustering $(k, \{(D_i, S_i)\}_{1\leq i\leq k})$ yields a k-Steiner forest $(D, \bigcup_{1\leq i\leq k} E(S_i))$.
Let T be a minimum spanning tree on D and denote by $e_1, \dots, e_{n-1}$ the edges of T in non-increasing order, i.e. $c(e_1) \geq c(e_2) \geq \dots \geq c(e_{n-1})$. Now define $F_1 := T$ and recursively $F_i := (D, E(F_{i-1}) \setminus \{e_{i-1}\})$ for $i = 2, \dots, n$.

Lemma 1.6. *F_k is a minimum k-spanning tree for all $1 \leq k \leq n$.*

Proof. Obviously, F_k if a k-spanning forest. Assume there is a shorter k-spanning forest F'. As T is a spanning tree you can find $k-1$ edges $E \subset E(T)$ so that F' plus E is a spanning tree again. E contains $k-1$ edges so by the order of the e_i's clearly $\sum_{e \in E} c(e) \leq \sum_{i=1}^{k-1} c(e_i)$. But then $c(F') + c(E) < c(F_k) + \sum_{i=1}^{k-1} c(e_i) = c(T)$. This is a contradiction to the minimality of the spanning tree T. □
This Lemma can also be proved using matroid theory (a short excursion into matroid theory can be found in Section 1.3.2). Further note that Kruskal's minimum spanning tree algorithm iteratively computes $F_n, F_{n-1}, \ldots, F_2, F_1 = T$ (see e.g. Korte and Vygen [2008]).

Lemma 1.7. *$\frac{1}{\alpha}c(F_k)$ is a lower bound for the length of a minimum k-Steiner forest for all $1 \leq k \leq n$.*

Proof. For $k \in \{1, \ldots, n\}$ let S_k be a minimum Steiner forest. Replace each connected component of S_k by a minimum spanning tree on the same set of vertices. We get a k-spanning forest F that costs at most α times the cost of S_k. The cost of a minimum k-spanning forest cannot be higher than the cost of F. □

Lemma 1.8. *Let t_{lb} be the smallest integer that satisfies*

$$\frac{1}{\alpha}c(F_{t_{lb}}) + d(D) \quad \leq \quad t_{lb} \cdot u. \tag{1.3}$$

t_{lb} is a lower bound for the number of clusters of any feasible clustering.

Proof. Let $\big(k, \{(D_i, S_i)\}_{i \in \{1,\ldots,k\}}\big)$ be a feasible clustering. Using Lemma 1.7 we conclude:

$$\begin{aligned} \frac{1}{\alpha}c(F_k) + d(D) &\leq \sum_{i=1}^{k} c(S_i) + d(D) \\ &= \sum_{i=1}^{k} (c(S_i) + d(D_i)) \\ &\leq k \cdot u. \end{aligned}$$

Thus (1.3) is a necessary condition that there is a feasible clustering with t_{lb} clusters. □
Now we can establish our first lower bound:

Lemma 1.9.

$$\min_{t_{lb} \leq t \leq n} \left(\frac{1}{\alpha}c(F_t) + t \cdot f \right)$$

is a lower bound for the cost of any feasible clustering.

Proof. Again let $\big(k, \{(D_i, S_i)\}_{i\in\{1,\ldots,k\}}\big)$ be a feasible clustering. The cost of it is

$$\sum_{i=1}^{k} c(S_i) + k \cdot f \quad \geq \quad \frac{1}{\alpha} c(F_k) + k \cdot f.$$

Moreover, by Lemma 1.8 any feasible solution has at least t_{lb} clusters, which completes the proof. □

We denote by t^* the smallest t for which the minimum in Lemma 1.9 is attained, $L_r := \frac{1}{\alpha} c(F_{t^*})$ and $L_f := t^* \cdot f$. Then $L_r + L_f$ is a lower bound for the cost of an optimum solution. Moreover, the proof of Lemma 1.9 implies

$$L_r + d(D) \quad \leq \quad L_f \frac{u}{f}. \tag{1.4}$$

Approximation Algorithm

The previous observations give us the mathematical basis for the Sink Clustering Algorithm that will be presented now.

For $D' \subset D$ and a tree T' with $D' \subseteq V(T') \subseteq V$ we set $\text{load}(D', T') := c(T') + d(D')$. We call a cluster (D', T') overloaded if $\text{load}(D', T') > u$.

The algorithm first computes a minimum spanning tree T on D. Then it computes t^* and removes the $t^* - 1$ longest edges from T. This yields a t^*-spanning forest F'. Every component T' of F' induces a cluster $(D \cap V(T'), T')$. If none of these clusters is overloaded we have a feasible clustering that costs at most α times the optimum.

Otherwise, let (D', T') be an overloaded component. We will split up subtrees from T' and reduce the load of T' by at least $\frac{u}{2}$ for each additional component.

By duplicating vertices and adding edges of length 0 we transform T' into a rooted binary tree with an arbitrarily chosen element $r \in D' := V(T') \cap D$ as the root and the elements in $D' \setminus \{r\}$ as the leaves. We set $d(v) := 0$ for every newly inserted vertex v. For every vertex $v \in V(T')$ we denote by T_v the subtree rooted at v.

Now let $v \in V(T')$ be a vertex with maximum distance to r and $\text{load}(D \cap V(T_v), T_v) > u$. v is no leaf, thus $v \notin D$ and $d(v) = 0$. As T' is a binary tree, v has a successor w with $\text{load}(D \cap V(T_w), T_w) + c(\{v, w\}) \geq \frac{u}{2}$. We split off T_w and remove the edge (v, w) from T'. Then we continue the splitting until no overloaded components are left.

Lemma 1.10. *The algorithm computes a solution of cost at most*

$$\alpha L_r + 3L_f + 2\frac{f}{u}(\alpha - 1)L_r.$$

Proof. The number of new clusters created by splitting overloaded components is at most $\frac{2}{u}\,\text{load}(D, T)$. Thus the cost of the solution computed by the algorithm is at most

$$\begin{aligned} c(F_{t^*}) + t^* \cdot f + \frac{2}{u}\left(c(F_{t^*}) + d(D)\right) \cdot f \quad &= \quad \alpha L_r + L_f + \frac{2f}{u}(\alpha L_r + d(D)) \\ &\leq \quad \alpha L_r + 3L_f + 2\frac{f}{u}(\alpha - 1)L_r. \end{aligned}$$

In the last inequality we used (1.4). □

Using (1.4) again we get

Theorem 1.11. *The* SINK CLUSTERING ALGORITHM *is a* $(2\alpha + 1)$*-approximation algorithm.* □

If $\frac{f}{u}$ is small we get an even better result:

Corollary 1.12. *For instances with* $\frac{f}{u} \le \phi$*, the* SINK CLUSTERING ALGORITHM *computes a solution of cost at most* $\max\{3, \frac{\alpha+2\alpha\phi+\phi}{1+\phi}\}$ *times the optimum.*

Proof. Set $\delta := \max\{3, \frac{\alpha+2\alpha\phi+\phi}{1+\phi}\}$. Then $\frac{f}{u} \le \phi \le \frac{\delta-\alpha}{2\alpha+1-\delta}$. By Lemma 1.10, the cost of the solution is at most:

$$\begin{aligned}\alpha L_r + 3L_f + 2\frac{f}{u}(\alpha-1)L_r &= \alpha L_r + 3L_f + ((\delta-3) + (2\alpha+1-\delta))\frac{f}{u}L_r \\ &\le \alpha L_r + 3L_f + (\delta-3)L_f + (2\alpha+1-\delta)\phi L_r \\ &\le \delta(L_r + L_f).\end{aligned}$$

We used (1.4) in the first inequality. □

Lemma 1.13. *The running time of the* SINK CLUSTERING ALGORITHM *is dominated by computing a minimum spanning tree on* D*.*

Proof. Obviously, detecting overloaded components and splitting them up can be done in linear time. Now we show how to compute t_{lb} in linear time if the minimum spanning tree T is already computed. To this end, we use the WEIGHTED MEDIAN PROBLEM: Given an integer n, numbers $z_1, \dots, z_n \in \mathbb{R}$, $w_1, \dots, w_n \in \mathbb{R}$ and a number W with $0 < W \le \sum_{i=1}^n w_i$, find the unique number z^* for which

$$\sum_{i:z_i<z^*} w_i < W \le \sum_{i:z_i\le z^*} w_i. \tag{1.5}$$

The WEIGHTED MEDIAN PROBLEM can be solved in linear time (see e.g. Bleich and Overton [1983], Johnson and Mizoguchi [1978], Reiser [1978], Shamos [1976]).
Let $e_1, \dots, e_{n-1}$ be the edges of the minimum spanning tree T. We cannot expect that these edges are sorted in some way. Let $\pi : \{1, \dots, n-1\} \to \{1, \dots, n-1\}$ be a permutation so that $c(e_{\pi(1)}) \ge c(e_{\pi(2)}) \ge \dots \ge c(e_{\pi(n-1)})$. Then we can rewrite inequality (1.3) and conclude that t_{lb} is the unique integer satisfying

$$\sum_{i=1}^{t_{lb}-2}\left(u + \frac{1}{\alpha}c(e_{\pi(i)})\right) < \frac{1}{\alpha}c(T) + d(D) - u \le \sum_{i=1}^{t_{lb}-1}\left(u + \frac{1}{\alpha}c(e_{\pi(i)})\right).$$

We get an instance of the WEIGHTED MEDIAN PROBLEM by setting $W := \frac{1}{\alpha}c(T) + d(D) - u$, $w_i := u + \frac{1}{\alpha}c(e_i)$ and $z_i = -c(e_i)$ for $1 \le i \le n-1$. Applying the WEIGHTED MEDIAN ALGORITHM we get a number z^* satisfying (1.5).
Set $Z := \{e \in E(T) | \, c(e) > -z^*\}$ and $W' := \sum_{e\in Z} \frac{1}{\alpha}c(e) + u$. Then $t_{lb} = \lceil \frac{W-W'}{u-z^*} \rceil + |Z| + 1$. Moreover, $t^* = \max\{t_{lb}, |\{e \in E(T) | \, \frac{1}{\alpha}c(e) \ge f\}|\}$, Z, W' and t_{lb} can be computed in linear time.
We conclude that the running time is dominated by computing the minimum spanning tree. □

1.1.4 Variants of the Sink Clustering Algorithm

In the SINK CLUSTERING ALGORITHM we built a spanning tree, removed some of the longest edges and finally split up overloaded components. Now we will present two variants of the algorithm where we first build an approximate Steiner tree and an approximate minimum tour, respectively, remove some of the longest edges and finally split up overloaded components.

Sink Clustering Algorithm on Steiner Trees

For the SINK CLUSTERING ALGORITHM ON STEINER TREES we set $\lambda := \frac{uf}{u+2f}$ and let c' be the metric defined by $c'(v,w) := \min\{c(v,w), \lambda\}$ for all $v, w \in V$. First we compute a Steiner tree S on D in (V, c') with an approximation algorithm with performance guarantee β. Then we delete all edges of length λ from S. If the resulting forest F contains overloaded components, they are split up as in the first algorithm. We get a feasible clustering.

Corollary 1.14. *The cost of the clustering is*

$$c'(S)\left(1+\frac{2f}{u}\right)+f+\frac{2f}{u}d(D).$$

Proof. Let t be the number of edges in S of length λ. Then the load of the forest F is $c'(S) - t\lambda + d(D)$. For each component generated by splitting, this load is reduced by at least $\frac{u}{2}$.
Thus the number of clusters is at most $1 + t + \frac{2}{u}(c'(S) - t\lambda + d(D))$, and the total cost is at most

$$\begin{aligned} & (c'(S)-t\lambda)+f+tf+\frac{2f}{u}c'(S)-\frac{2t\lambda f}{u}+\frac{2f}{u}d(D) \\ = \; & c'(S)\left(1+\frac{2f}{u}\right)+t\left(f-\lambda-\frac{2\lambda f}{u}\right)+f+\frac{2f}{u}d(D). \end{aligned}$$

Using $f = \lambda + \frac{2\lambda f}{u}$ we get the desired formula. □

Let $\left(k, \{(D_i, S_i)\}_{i\in\{1,\ldots,k\}}\right)$ be an optimum clustering. We set $L_r := \sum_{i=1}^{k} c(S_i)$ and $L_f := k \cdot f$. Clearly,

$$L_r + d(D) \;\leq\; L_f\frac{u}{f}. \tag{1.6}$$

Note that the optimum clustering can be extended to a Steiner tree on D by adding $k - 1 = \frac{L_f}{f} - 1$ edges. Hence there is a Steiner tree on D in (V, c') of length at most $L_r + \left(\frac{L_f}{f} - 1\right)\lambda$. Our Steiner tree S is at most β times longer. We conclude that the total cost of our clustering is bounded by

$$\begin{aligned}
&\beta\left(L_r+\left(\frac{L_f}{f}-1\right)\lambda\right)\left(1+\frac{2f}{u}\right)+f+\frac{2f}{u}d(D)\\
=\ & L_r\left(\beta+\beta\frac{2f}{u}\right)+\beta L_f\lambda\left(\frac{1}{f}+\frac{2}{u}\right)+f-\beta\left(\lambda+\frac{2\lambda f}{u}\right)+\frac{2f}{u}d(D)\\
\le\ & \beta L_r+\frac{2f}{u}\left(\beta L_r+d(D)\right)+\beta L_f\\
\le\ & \beta L_r+3\beta L_f.
\end{aligned}$$

We conclude:

Theorem 1.15. *The* Sink Clustering Algorithm on Steiner Trees *has performance ratio* 3β *for any metric.* □

Using the Robins-Zelikovsky algorithm (Robins and Zelikovsky [2000]) for building the Steiner trees we get a 4.648-approximation in polynomial time.
However, by a more careful analysis of this algorithm we can do even better. Recall that the Robins-Zelikovsky algorithm works with a parameter k and analyzes all k-restricted full components. Its running time is $O(n^{4k})$ and the length of the Steiner tree S it computes is at most

$$c'(E(Y^*))+c'(L^*)\ln\left(1+\frac{c'(T)-c'(E(Y^*))}{c'(L^*)}\right), \tag{1.7}$$

where T is a minimum spanning tree, Y^* is any k-restricted Steiner tree on D in (V,c') and L^* is a *loss* of Y^*, i.e. a minimum cost subset of $E(Y^*)$ connecting each Steiner point of degree at least three in Y^* to a terminal.
Let $\epsilon>0$ be fixed. For $k\ge 2^{\lceil\frac{1}{\epsilon}\rceil}$, an optimum k-restricted Steiner tree is at most $1+\epsilon$ times longer than an optimum Steiner tree (Du et al. [1991]). Thus taking an optimum k-restricted Steiner tree for each component of our optimum solution and adding edges to make the graph connected, we get a k-restricted Steiner tree Y^* with $c'(E(Y^*))\le(1+\epsilon)L_r+\left(\frac{L_f}{f}-1\right)\lambda$, and with loss L^* of length $c'(L^*)\le\frac{1+\epsilon}{2}L_r$.
The derivative of (1.7) with respect to $c'(E(Y^*))$ is $1-\frac{c'(L^*)}{c'(L^*)+c'(T)-c'(E(Y^*))}$. This is positive, as $c'(T)\ge c'(E(Y^*))$. Moreover, $c'(T)\le\alpha L_r+\left(\frac{L_f}{f}-1\right)\lambda$, where α is again the Steiner ratio. We conclude

$$\begin{aligned}
c'(S)\ \le\ & (1+\epsilon)L_r+\left(\frac{L_f}{f}-1\right)\lambda+c'(L^*)\ln\left(1+\frac{c'(T)-(1+\epsilon)L_r-\left(\frac{L_f}{f}-1\right)\lambda}{c'(L^*)}\right)\\
\le\ & (1+\epsilon)L_r+\left(\frac{L_f}{f}-1\right)\lambda+c'(L^*)\ln\left(1+\frac{(\alpha-1-\epsilon)L_r}{c'(L^*)}\right)\\
\le\ & (1+\epsilon)L_r+\left(\frac{L_f}{f}-1\right)\lambda+(\alpha-1)L_r\frac{c'(L^*)}{(\alpha-1)L_r}\ln\left(1+\frac{(\alpha-1)L_r}{c'(L^*)}\right)\\
\le\ & (1+\epsilon)L_r+\left(\frac{L_f}{f}-1\right)\lambda+L_r\frac{1+\epsilon}{2}\ln(1+2(\alpha-1)),
\end{aligned}$$

as $\max\{x\ln(1+\frac{1}{x})|\, 0 < x < \frac{1+\epsilon}{2(\alpha-1)}\} = \frac{1+\epsilon}{2(\alpha-1)} \ln\left(1+\frac{2(\alpha-1)}{1+\epsilon}\right)$.
For $\alpha = 2$ we get

$$c'(S) \;\leq\; (1+\epsilon)\left(1+\frac{\ln 3}{2}\right)L_r + \left(\frac{L_f}{f}-1\right)\lambda. \tag{1.8}$$

For simplicity of notation, we set $\beta' := (1+\epsilon)\left(1+\frac{\ln 3}{2}\right)$ (note that $\beta' < 1.5495$ for $\epsilon \leq 10^4$). Using (1.8) with Corollary 1.14 and applying (1.6) we get that the Sink Clustering Algorithm on Steiner Trees computes a solution of total cost at most

$$\begin{aligned}
&\left(\beta' L_r + \left(\frac{L_f}{f}-1\right)\lambda\right)\left(1+\frac{2f}{u}\right) + f + \frac{2f}{u}\left(\frac{u}{f}L_f - L_r\right)\\
=\;& L_r\left(\beta' + (\beta'-1)\frac{2f}{u}\right) + L_f\left(\frac{\lambda}{f}+\frac{2\lambda}{u}+2\right) + f - \left(1+\frac{2f}{u}\right)\cdot\lambda\\
=\;& L_r\left(\beta' + (\beta'-1)\frac{2f}{u}\right) + 3L_f\\
\leq\;& \beta' L_r + (2\beta'+1)L_f.
\end{aligned}$$

We have shown

Lemma 1.16. *The* Sink Clustering Algorithm on Steiner Trees *with the Robins-Zelikovsky algorithm (with parameter $k = 2^{10000}$) is a 4.099-approximation algorithm.* □

Sink Clustering Algorithm on TSP

For the Sink Clustering Algorithm on TSP we set $\lambda' := \frac{fu}{u+f}$ and define the metric (V, c'') by setting $c''(v,w) := \min\{c(v,w), \lambda'\}$ for all $v, w \in V$.
First the algorithm computes a tour T on D in (V, c'') using an approximation algorithm for the TSP with performance ratio γ. Set $\delta := \max\{c''(e)|\, e \in E(T)\}$.
Then the algorithm deletes all edges of length λ' from T. If there is no such edge it deletes an edge of length δ. Let t be the number of deleted edges and let $P^1, \dots, P^t$ be the resulting paths. For each $j = 1, \dots, t$ we split up the path P^j into paths $P^j_1, \dots, P^j_{k^j}$ such that $\text{load}(D \cap V(P^j_i), P^j_i) \leq u$ and $\text{load}(D \cap V(\tilde{P}^j_i), \tilde{P}^j_i) > u$ for $i = 1, \dots, k^j - 1$, where $\tilde{P}^j_i$ is P^j_i plus the edge connecting P^j_i and P^j_{i+1}. This yields a feasible clustering. To show that the tour T is not too long we need the following Corollary.

Corollary 1.17. *Let (V, c) be a metric space, T a tree with $V(T) \subseteq V, |V(T)| \geq 2$, and $a, b \in V(T), a \neq b$. Then there exists a path P from a to b with $V(P) = V(T)$ and $c(P) \leq 2c(T)$.*

Proof. Let Q be the a-b-path in T and H be the graph obtained from T by doubling all edges except those of Q and adding the edge $\{a, b\}$. H is Eulerian and thus contains a Eulerian walk. Removing $\{a, b\}$ and short-cutting where vertices appear not for the first time results in an Hamiltonian path P from a to b which is not longer than $2c(T)$. □

Let $\big(k, \{(D_i, S_i)\}_{i\in\{1,\dots,k\}}\big)$ be an optimum clustering. We again set $L_r := \sum_{i=1}^{k} c(S_i)$ and $L_f := k \cdot f$. And again we get

$$L_r + d(D) \;\; \leq \;\; L_f \frac{u}{f}. \tag{1.9}$$

If $k = 1$ let e' be an arbitrary edge of T. If $k > 1$ we can reorder $S_1, \dots, S_k$ such that T contains an edge $e' = \{v_k, w_1\}$ with $v_k \in V(S_k)$ and $w_1 \in V(S_1)$. Now we choose additional vertices $v_i \in V(S_i)$ and $w_{i+1} \in V(S_{i+1})$ $(i = 1, \dots, k-1)$ such that $v_i \neq w_i$ for $i = 1, \dots, k$ with $|V(S_i)| > 1$. e' is an edge of T so $c''(e') \leq \delta$. Moreover, $c''(\{v_i, w_{i+1}\}) \leq \lambda'$ for $i = 1, \dots, k-1$. Using Corollary 1.17 we conclude that there is a tour on D of length at most

$$L_r + \sum_{i=1}^{k-1} c''(\{v_i, w_{i+1}\}) + c''(e') \;\; \leq \;\; 2L_r + (k-1)\lambda' + \delta.$$

The computed tour T is at most γ times longer, i.e.

$$c''(T) \;\; \leq \;\; 2\gamma L_r + \gamma(k-1)\lambda' + \gamma\delta.$$

The algorithm deleted t edges of length at most δ. So the resulting paths $P^1, \dots, P^t$ has length at most

$$2\gamma L_r + \gamma(k-1)\lambda' + \gamma\delta - t\delta. \tag{1.10}$$

In the case $t > 1$, we have $\delta = \lambda'$ and (1.10) is equal to

$$\begin{aligned} & 2\gamma L_r + \gamma k \lambda' - t\lambda' \\ = \;\; & 2\gamma L_r + \gamma \frac{\lambda'}{f} L_f - t\lambda'. \end{aligned}$$

In the case $t = 1$, (1.10) is maximized for $\delta = \lambda'$ as $\gamma \geq 1$, and we get again

$$c''(T) - t\delta \;\; \leq \;\; 2\gamma L_r + \gamma \frac{\lambda'}{f} L_f - t\lambda'.$$

Now note that by the construction of the paths, $\sum_{i=1}^{k^j-1} \mathrm{load}(D \cap V(\tilde{P}_i^j), \tilde{P}_i^j) > u$ for $j = 1, \dots, t$ and hence $2d(D) + c''(T) - t\delta > \sum_{j=1}^{t} (k^j - 1)u$. We conclude that the number of clusters is bounded by

$$\sum_{i=1}^{t} k^j \;\; < \;\; \frac{1}{u}(2d(D) + c''(T) - t\delta).$$

Thus the total cost of the solution is at most:

$$\begin{aligned}
& \left(2\gamma L_r + \gamma\frac{\lambda'}{f}L_f - t\lambda'\right)\left(1+\frac{f}{u}\right) + 2\frac{f}{u}d(D) + tf \\
\overset{(1.9)}{\leq} & \left(2\gamma L_r + \gamma\frac{\lambda'}{f}L_f - t\lambda'\right)\left(1+\frac{f}{u}\right) + 2L_f - 2\frac{f}{u}L_r + tf \\
= & L_r\left(2\gamma + 2(\gamma-1)\frac{f}{u}\right) + L_f\left(2+\gamma\lambda'\left(\frac{1}{f}+\frac{1}{u}\right)\right) + tf + t\lambda'\left(1+\frac{f}{u}\right) \\
= & L_r\left(2\gamma + 2(\gamma-1)\frac{f}{u}\right) + L_f(2+\gamma) \\
\leq & 2\gamma L_r + 3\gamma L_f.
\end{aligned}$$

We conclude

Theorem 1.18. *The* SINK CLUSTERING ALGORITHM ON TSP *has performance guarantee 3γ for any metric.* □

Using the best known polynomial approximation algorithm for TSP (Christofides [1976]) we get a 4.5-approximation algorithm.
Table 1.1 summarizes the approximation ratios and running times of the presented algorithms.

algorithm	metric	factor	runtime
SINK CLUSTERING ALGORITHM	$(\mathbb{R}^2, l_1)$	4	$n \log n$
SINK CLUSTERING ALGORITHM	general	5	n^2
SINK CLUSTERING ALGORITHM ON STEINER TREES	general	4.099	$n^{2^{10000}}$
SINK CLUSTERING ALGORITHM ON TSP	general	4.5	n^3

Table 1.1: Overview of the approximation ratios and running times of all presented algorithms for the SINK CLUSTERING PROBLEM.

1.1.5 Analysis

Tight Examples for the Sink Clustering Algorithm

Now we give an example showing that the performance guarantee of the SINK CLUSTERING ALGORITHM is not better than five:
First we define the metric (V, c) with $V := \{0, 1, \ldots, 4m(m+2)\}$, $c(0, i) := 1$ and $c(i, j) := 2$ for all $i, j \in V \setminus \{0\}, i \neq j$. Let $D := V \setminus \{0\}$, $u := 4m$, $f \gg m$ and $d(i) := 0$ for all $i \in D$. Then an optimum solution has cost $(m+2)f + 4m(m+2)$. The SINK CLUSTERING ALGORITHM could build the spanning tree

$$T = \{\{i, i+1\} | \, i \in D \setminus \{i(m+2) | \, i = 1, \ldots, 4m\}\} \cup \{\{1, i(m+2)+1\} | \, i = 1, \ldots, 4m-1\}.$$

In this case, the algorithm computes $t^* = m+2$ and could end up with $F_{t^*} = T \setminus \{\{i(m+2)-1, i(m+2)\} | \, i = 1, \ldots, m+2\}$. $4m-1$ components have to be split up until the

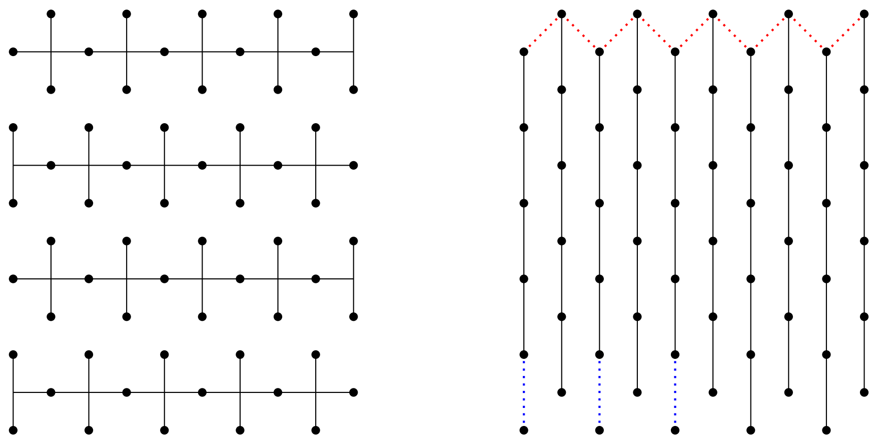

Figure 1.1: Clustering instance in the rectilinear plane for $m = 2$. The left side shows an optimal clustering, the right side a clustering produced by the SINK CLUSTERING ALGORITHM. The blue dotted edges are removed to construct F_{t^*}, the red ones are removed when splitting up components.

clustering is feasible. The total cost of the solution is $(5m + 1)f + 2(4m - 1)(m + 2)$ which is arbitrarily close to 5 times larger than the optimum if f is large enough.
Even in the rectilinear plane the performance guarantee of the SINK CLUSTERING ALGORITHM is not better than 4 (see Figure 1.1).
Let $D := \{v_{i,j} := (2i, 2j), v'_{i,j} := (2i+1, 2j+1) | \, i = 1, \ldots, 3m-1, j = 1, \ldots, 3m\} \subset \mathbb{R}^2$, $u := 12m - 5$, $f \gg m$ and $d(v) := 0$ for all $v \in D$. An optimum solution has cost $2mf + 24m^2 - 10m$.
The algorithm might compute the spanning tree $T = \{\{v_{i,j}, v_{i,j+1}\}, \{v'_{i,j}, v'_{i,j+1}\} | \, i = 2, \ldots, 3m-1, j = 1, \ldots, 3m-1\} \cup \{\{v_{i,3m}, v'_{i,3m}\} | \, i = 1, \ldots, 3m-1\} \cup \{\{v_{i,3m}, v'_{i-1,3m}\} | \, i = 2, \ldots, 3m-1\}$. Each edge of T has length 2. As $\alpha = \frac{3}{2}$ in the rectilinear plane we get $t^* = 2m$. Then the algorithm might compute $F_{t^*} = T \setminus \{\{v_{i,1}, v_{i,2}\} | \, i = 1, \ldots, 2m - 1\}$ and the total cost of the final solution is $36m^2 - 28m + 6 + (8m - 3)f$. For f large enough this is arbitrarily close to 4 times larger than the optimum.
For the SINK CLUSTERING ALGORITHM ON STEINER TREES and the SINK CLUSTERING ALGORITHM ON TSP no tight examples are known. They would depend on the approximation algorithms used to compute an approximate minimum Steiner tree and TSP tour,respectively.

1.2 Post Optimization

In this section we present two heuristical post optimization algorithms that can improve an existing feasible clustering. The first algorithm tries to improve pairs of clusters while the second one tries to improve chains of clusters.
In both algorithms we only consider clusters that are 'near' to each other, so we use a neighborhood graph G on D. Two clusters (D_A, S_A) and (D_B, S_B) are called neighbors if they contain sinks $a \in D_A$ and $b \in D_B$ with $\{a, b\} \in E(G)$. In the rectilinear plane a Delaunay triangulation could be used as neighborhood graph.

1.2.1 TwoClusterOpt

The first post optimization algorithm tries to improve the cost of two clusters that are neighbors by redistributing their sinks (see Figure 1.2):
Let (D_A, S_A) and (D_B, S_B) be two clusters that are neighbors. We compute an approximate Steiner tree $S_{A\cup B}$ on $A \cup B$ by some Steiner heuristic. Then $(D_A \cup D_B, S_{A\cup B})$ forms a (not necessary feasible) cluster. If the load of $(D_A \cup D_B, S_{A\cup B})$ is at most u the cluster is feasible. If, moreover, the cost of the new cluster is smaller than the cost of the initial two clusters we can replace them by the new one and reduce the total cost. In this case we have merged the two clusters.

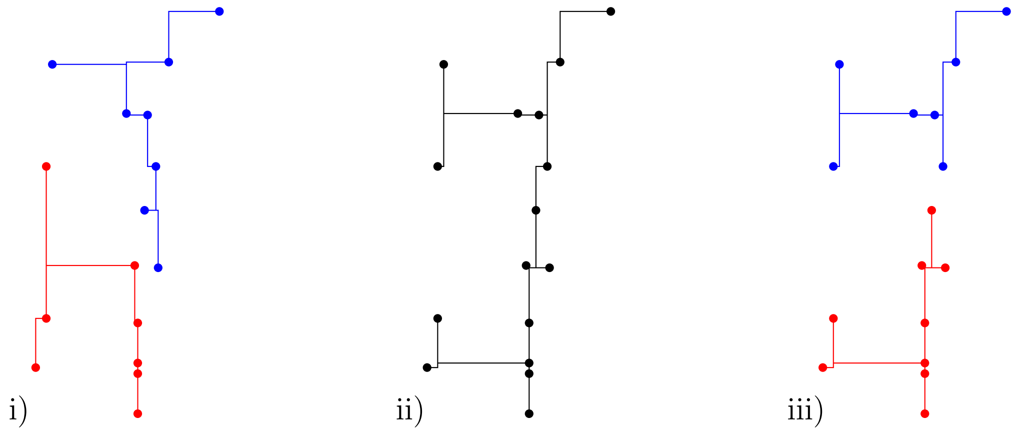

Figure 1.2: Example for *TwoClusterOpt*. i) shows the initial two clusters. In ii) an approximate minimum Steiner tree on all sinks has been computed. iii) shows the resulting two feasible clusters after removing an appropriate edge.

If the two clusters cannot be merged we try to split the tree $S_{A\cup B}$ into two new clusters: Removing an edge e of $S_{A\cup B}$ we get two Steiner trees S_e^1 and S_e^2. Note that one or both endpoints of e might be Steiner points. Let $D_e^1 \subset A \cup B$ be the sinks that are connected by S_e^1 and $D_e^2 \subset A \cup B$ be the sinks that are connected by S_e^2. $C_e^1 = (D_e^1, S_e^1)$ and $C_e^2 = (D_e^2, S_e^2)$ are two clusters covering $D_A \cup D_B$. Let $e \in E(S_{A\cup B})$ be an edge of maximum cost so that C_e^1 and C_e^2 are feasible. If no such edge exists or the cost of the two new clusters is greater than or equal to the cost of the initial two clusters

we cannot improve them. Otherwise, we replace (D_A, S_A) and (D_B, S_B) by the new clusters (D_e^1, S_e^1) and (D_e^2, S_e^2) and reduce the cost of the clustering.
Finding an edge $e \in E(S_{A\cup B})$ of maximum cost so that C_e^1 and C_e^2 are feasible or deciding that no such edge exists can be done in linear time. To this end, we transform the tree into an arborescence by selecting an arbitrary vertex $s \in V(S_{A\cup B})$ of degree 1 as root. By traversing the arborescence bottom-up from the leaves to s we compute for every vertex $v \in V(S_{A\cup B})$ the capacitance c_v of the sub-tree rooted at v. Note that $c_v = d(v) + \sum_{w\in\delta^+(v)}(c(v,w)+c_w)$ with $d(v) = 0$ if $v \notin D$ is a Steiner point. Removing an edge $e = (v,w) \in E(S_{A\cup B})$ creates two clusters, one of load c_w and one of the total load of $S_{A\cup B}$ minus the capacitance of e and c_w.

1.2.2 ChainOpt

The second post optimization algorithm tries to optimize a chain of clusters by moving parts of one cluster to another. Figures 1.3 and 1.4 illustrate the work of *ChainOpt*.
The input of the main subroutine are two clusters that are neighbors. The goal is to move load from the second cluster to the first one. For this, let (D_A, S_A) and (D_B, S_B) be two clusters and $a \in D_A$ and $b \in D_B$ with $c(\{a,b\})$ minimal. $S_{A\cup B} = S_A \cup \{\{a,b\}\} \cup S_B$ is a Steiner tree on $D_A \cup D_B$. If the cluster $(D_A \cup D_B, S_{A\cup B})$ is feasible the routine returns the merged cluster. If not, removing an edge $e \in S_B$ from $S_{A\cup B}$ creates two new clusters - one of them containing S_A. We denote this cluster by C_A^e and the other one by C_B^e. If there is no edge $e \in S_B$ such that both clusters C_A^e and C_B^e are feasible, the subroutine cannot improve the clustering. Otherwise, it chooses an edge $e \in S_B$ so that C_A^e and C_B^e are feasible and the load of C_B^e is minimized.
The algorithm *ChainOpt* applies this subroutine iteratively on a chain of clusters. It starts with two clusters C_1 and C_2 that are neighbors. Let e_1 be an edge of minimum length connecting C_1 and C_2. Now we can use the subroutine on C_1, C_2 and e_1. If the subroutine is not successful we cannot improve this pair of clusters. If the clusters can be merged and cost less than the initial clusters we merge them. Otherwise, let C_1' and C_2' be the two new clusters.
Now the iteration steps on: Assume we have constructed C_{k-1}' and C_k', then we look for a cluster C_{k+1} that is a neighbor of C_k'. Let e_k be an edge of minimum length connecting them and apply the subroutine on C_k', C_{k+1} and e_k. Now again three cases can occur: The subroutine

1. was not successful,
2. merged both clusters into one cluster C_k'' or
3. returns two new clusters C_k'' and C_{k+1}'.

If it was not successful we check if the cost of the clusters $C_1', C_2'', \ldots, C_{k-1}'', C_k'$ is smaller than the cost of the initial clusters $C_1, \ldots, C_k$. In this case we replace the initial clusters by the new ones. Otherwise, we dismiss the chain.
If the clusters were merged into one cluster C_k'' we have got a new chain $C_1', C_2'', \ldots, C_k''$. If its cost is smaller than the cost of the initial clusters $C_1, \ldots, C_{k+1}$ we replace them

and dismiss them otherwise. Note that the old chain contains one cluster more than the new one.
In the third case we continue with the next iteration. In order to limit the running time of the algorithm the lengths of the chains can be limited.

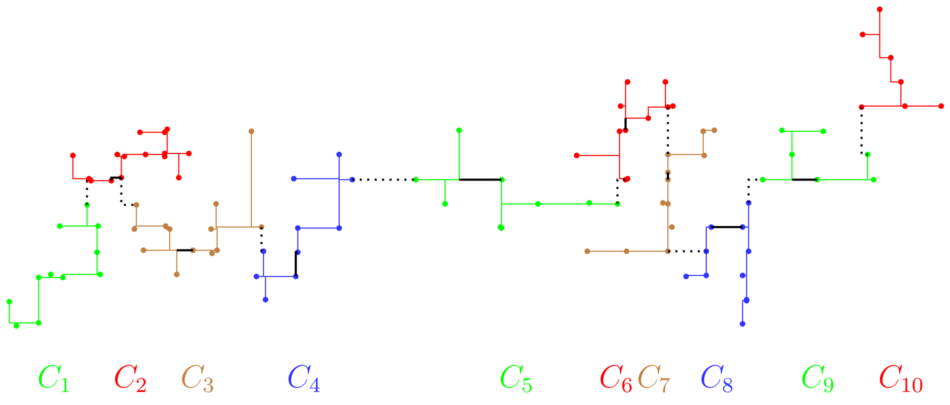

Figure 1.3: Clustering before applying ChainOpt. Only the clusters that are considered in the current chain are plotted. The dotted edges are the ones to be inserted and the black edges the ones to be deleted.

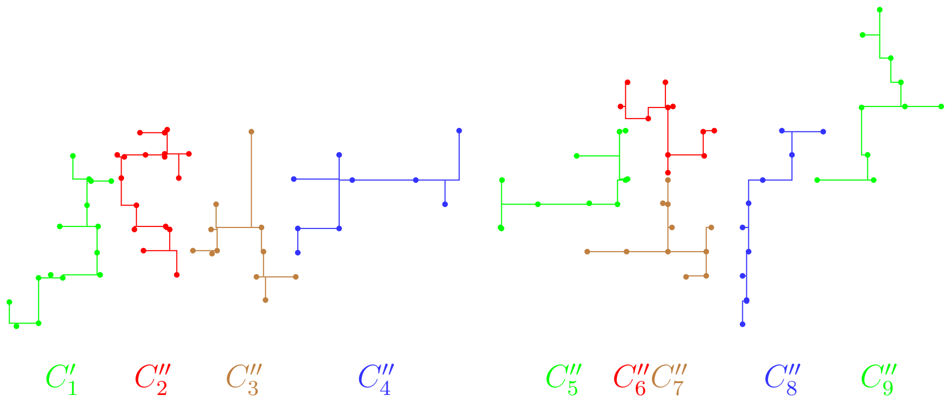

Figure 1.4: Clustering after applying ChainOpt.

1.3 Lower Bounds

In this chapter we establish improved lower bounds for the Sink Clustering Problem. We already presented a simple lower bound in Section 1.1.3 that was based on building a minimum spanning tree and removing the longest edges until a capacitance constraint is satisfied.
In Section 1.3.1 we will use a lower bound for the length of a minimum Steiner tree on D in order to get bounds for our problem. After making a short excursion into matroid theory we will introduce the concept of K-dominating functions in Section 1.3.3. They will help us to analyze clustering instances more locally and yield new lower bounds. Finally, in Section 1.3.4 we establish bounds that combine the ideas of the previous approaches.
For the rest of this chapter let $\left(k^o, \{(D_i^o, S_i^o)\}_{i\in\{1,\dots,k^o\}}\right)$ be an optimum solution, i.e. a feasible solution that minimizes the cost function (1.2).

1.3.1 Lower Bound Based on Minimum Steiner Trees

We have seen that the clusters of a feasible clustering induce a k-Steiner forest on D for some $k \in \mathbb{N}$. But then there exist $k-1$ edges of a minimum spanning tree so that the Steiner forest plus these edges form a Steiner tree on D connecting all sinks. We will use this observation in order to get a lower bound for the cost of our Sink Clustering Problem.
Let l_{SMT} be a lower bound for the length of a minimum Steiner tree on D and let T be a minimum spanning tree on D. Moreover, let $\left(k^{\text{feas}}, \{(D_i^{\text{feas}}, S_i^{\text{feas}})\}_{i\in\{1,\dots,k^{\text{feas}}\}}\right)$ be a feasible clustering with k^{feas} clusters. Then there exists a set of edges $E' \subseteq E(T)$ with $|E'| = k^{\text{feas}} - 1$ so that $G = (D, \bigcup_{i=1}^{k^{\text{feas}}} E(S_i^{\text{feas}}) \cup E')$ is connected. G is a Steiner tree on D and therefore $\sum_{i=1}^{k^{\text{feas}}} c(S_i^{\text{feas}}) + c(E') = c(G) \geq l_{\text{SMT}}$. We get

$$\sum_{i=1}^{k^{\text{feas}}} c\left(S_i^{\text{feas}}\right) \geq l_{\text{SMT}} - \sum_{e\in E'} c(e). \tag{1.11}$$

Let $e_1, \dots, e_{n-1}$ be the edges of T with $c(e_1) \geq \dots \geq c(e_{n-1})$. Then

$$\sum_{e\in E'} c(e) \leq \sum_{i=1}^{k^{\text{feas}}-1} c(e_i). \tag{1.12}$$

As the solution is feasible we get

$$\begin{aligned} k^{\text{feas}} \cdot u &\overset{(1.1)}{\geq} d(D) + \sum_{i=1}^{k^{\text{feas}}} c\left(S_i^{\text{feas}}\right) \\ &\overset{(1.11)}{\geq} d(D) + l_{\text{SMT}} - c(E') \\ &\overset{(1.12)}{\geq} d(D) + l_{\text{SMT}} - \sum_{i=1}^{k^{\text{feas}}-1} c(e_i). \end{aligned}$$

And so we conclude:

Theorem 1.19. *Let t_{SMT} be the smallest integer satisfying*

$$d(D) + l_{\text{SMT}} - \sum_{i=1}^{t_{\text{SMT}}-1} c(e_i) \leq t_{\text{SMT}} \cdot u.$$

Then t_{SMT} is a lower bound for the number of facilities of any feasible solution. Moreover,

$$\min_{t \geq t_{\text{SMT}}} \left(l_{\text{SMT}} - \sum_{i=1}^{t-1} c(e_i) + t \cdot f \right)$$

is a lower bound for the cost of an optimum solution.

Lemma 1.20. *The computation of the lower bound in the last theorem is as fast as computing a lower bound for a minimum Steiner tree and constructing a minimum spanning tree.*

Proof. Observe that t_{SMT} can be computed in linear time by using the WEIGHTED MEDIAN ALGORITHM similar as in the proof of Lemma 1.13. $\square$

1.3.2 Excursion into Matroid Theory

We have already noted that the forests on D form a matroid. As an important property of matroids will be used extensively in the following section we now give a short excursion into matroid theory. We are using the notations of Korte and Vygen [2008].

Definition 1.21. *A set system $(E, \mathcal{F})$ with E finite and $\mathcal{F} \subseteq 2^E$ is a matroid if*

(M1) $\emptyset \in \mathcal{F}$,

(M2) if $X \subseteq Y$ *and* $Y \in \mathcal{F}$ *then* $X \in \mathcal{F}$,

(M3) if $X, Y \in \mathcal{F}$ *and* $|X| > |Y|$, *then there is an* $x \in X \setminus Y$ *with* $Y \cup \{x\} \in \mathcal{F}$.

A set $X \in \mathcal{F}$ is called independent and a set $X \notin \mathcal{F}$ is called dependent. A maximal independent set is called basis and a minimal dependent set is called circuit.

Now we show the well-known result that the forests on D form a matroid:

Lemma 1.22. *Let G be a graph.*
Then $(E(G), \mathcal{F})$ with $\mathcal{F} = \{F \subseteq E(G) | (V(G), F)$ is a forest$\}$ is a matroid.

Proof. (M1) and (M2) are trivial. Let X and $Y \in \mathcal{F}$ with $|X| > |Y|$. We have to show that there is an edge $e \in X$ so that $(V(G), Y \cup \{e\})$ is a forest. As $|X| > |Y|$ the forest $(V(G), Y)$ has at least one connected component more than $(V(G), X)$. But then there must be an edge $e \in X$ that connects two different connected components of $(V(G), Y)$. Thus Y plus e forms a new forest. $\square$
The following lemma is essential for the next section.

Lemma 1.23. *Let $M = (E, \mathcal{F})$ be a matroid and X, Y be two bases. There exists a bijective mapping $\pi : X \to Y$ such that $(X \setminus \{a\}) \cup \{\pi(a)\}$ is a basis for all $a \in X$.*

Proof. The original proof is due to Gabow et al. [1974] (see also von Randow [1975]). It uses some observations that can be concluded directly from the matroid properties. In our proof we will use Hall's condition.
We define the bipartite graph $G = (V, E)$ with $V := X \dot\cup Y$ and $E := \{(a, b) | \, a \in X, b \in Y, (X \cup \{b\}) \setminus \{a\} \in \mathcal{F}\}$. We will show that G satisfies the Hall condition: For all $X' \subseteq X$ the number of neighbors of X' is at least as big as $|X'|$, i.e. $|\Gamma(X')| \geq |X'|$. Then by Hall's Theorem (Hall [1935]) there is a matching covering X and this matching induces the desired bijective mapping.
Let $X' \subseteq X$. M is a matroid and thus there is a set $Y' \subseteq Y$ so that $(X \setminus X') \cup Y'$ is a base. Now we show that for any $b \in Y'$ there is an $a \in X'$ so that $(a, b) \in E$. This implies $Y' \subseteq \Gamma(X')$ and thus $|\Gamma(X')| \geq |Y'| = |X'|$ and we are done.
To this end, assume that there is a $b \in Y'$ so that for each $a \in X'$ we have $(a, b) \notin E$. As X is a basis there is a unique circuit $C \subseteq X \cup \{b\}$ (see e.g. Welsh [1976]). For all $a \in X'$ the set $(X \cup \{b\}) \setminus \{a\}$ is dependent, $X \setminus \{a\}$ is independent and $(X \cup \{b\}) \setminus \{a\} \subseteq X \cup \{b\}$. We conclude $C \subseteq (X \cup \{b\}) \setminus \{a\}$ for all $a \in X'$ and therefore $C \subseteq \overline{X} := (X \cup \{b\}) \setminus X'$, i.e. $\overline{X}$ is dependent. But $\overline{X}$ is a subset of the independent set $(X \cup Y') \setminus X'$ contradicting that M is a matroid. □

1.3.3 An Improved Lower Bound using 'K-dominated' Functions

When computing the previous lower bounds, we always removed the $k - 1$ longest edges from the minimum spanning tree in order to get a lower bound for the length of a k-Steiner forest. This might be too pessimistic as we had only a 'global' look at the instance.
Assume that there is a set of sinks $D' \subseteq D$ so that the distance to any other sink of D is 'long'. Then any cluster containing both sinks in D' and sinks outside of D' also contains a 'long' edge. To make this statement more precise, we introduce the concept of 'K-dominated' functions in this section. They will help us to analyze the instances more locally and yield improved lower bounds for the Sink Clustering Problem.

Preliminaries

Adding an artifical vertex r we define the metric space (V', c') with $V' := V \cup \{r\}$, $c'(v, w) := \min\{\frac{1}{\alpha}c(v, w), f\}$ for $v, w \in V$ and $c'(v, r) := c'(r, v) := f$ for $v \in V$. Here α denotes the Steiner ratio again. Let $\big(k^o, \{(D_i^o, S_i^o)\}_{i \in \{1,\dots,k^o\}}\big)$ be an optimum clustering in (V, c). For $i \in \{1, \dots, k^o\}$ let T_i^o be a minimum spanning tree on $D_i^o \cup \{r\}$ in (V', c'). Then

$$c'(T_i^o) \leq c(S_i^o) + f \tag{1.13}$$

for all i and $\sum_{i=1}^{k^o} c'(T_i^o)$ is a lower bound for the cost of an optimum solution.
$T^o = (D \cup \{r\}, \bigcup_{i=1}^{k^o} E(T_i^o))$ is a spanning tree on $D \cup \{r\}$. We choose a minimum spanning tree T on $D \cup \{r\}$. Each edge has length at most f and all edges incident to

r have length exactly f so we can assume without loss of generality that r has degree 1 in T, i.e. $|\delta_T(r)| = 1$. T^o and T can be interpreted as arborescences rooted at r. Figures 1.5 and 1.6 show an example which will be used throughout the rest of this section to illustrate the next lower bound.

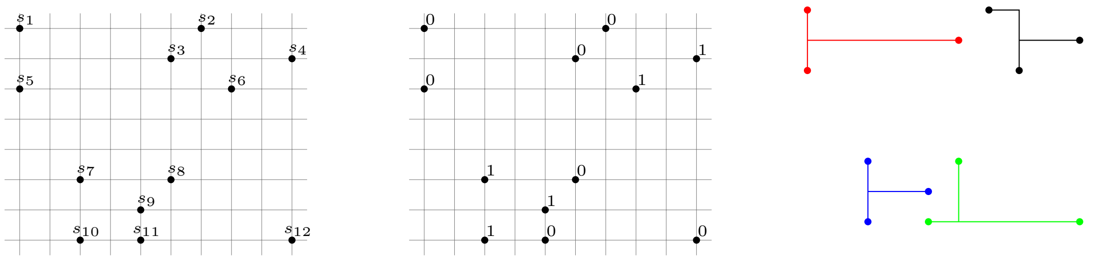

Figure 1.5: An instance in the rectilinear plane with 12 sinks, $D := \{s_1, \dots, s_{12}\}$. The distance between two consecutive grid lines is 1. The demands are printed next to the sinks in the middle picture. Moreover, $u := 7$ and $f := 10$. The right picture shows an optimum clustering of cost 63.

Definition 1.24. *For a sink $v \in D$ let $e_T(v)$ be the unique edge in T ending in v and $E_T(W) := \{e_T(v) | v \in W\}$ for $W \subseteq D$.*

For $i \in \{1, \dots, k^o\}$ we denote by T_i the graph obtained by removing the edges of $E_T(D_i^o)$ from T and adding the edges of $E(T_i^o)$, i.e. $T_i := (D \cup \{r\}, (E(T) \setminus E_T(D_i^o)) \cup E(T_i^o))$.

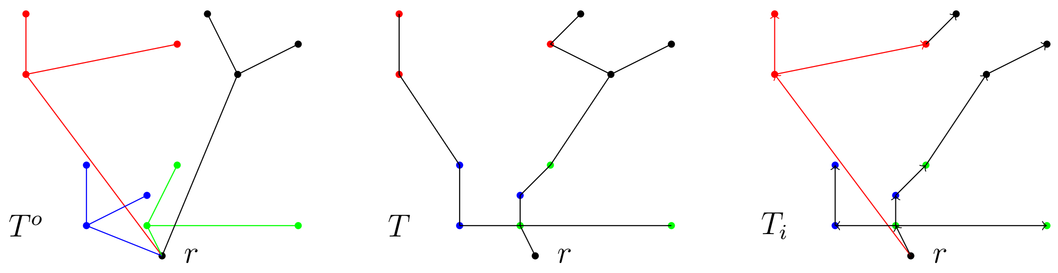

Figure 1.6: The tree T^o, the minimum spanning tree T and the arborescence T_i for the i belonging to the red cluster.

Lemma 1.25. *T_i is a spanning tree on $D \cup \{r\}$ for all $i \in \{1, \dots, k^o\}$.*

Proof. As $|D_i^o| = |E_T(D_i^o)| = |E(T_i^o)|$ it is sufficient to show that every vertex $v \in D$ is reachable from r in T_i. T_i^o is a spanning tree rooted at r so all vertices $v \in D_i^o$ are reachable from r in $T_i^o \subseteq T_i$.
Choose $v \in D \setminus D_i^o$ and let P be the unique path from r to v in T. If $V(P) \cap D_i^o = \emptyset$ then P is completely contained in T_i and we are done. Otherwise, let $w \in V(P) \cup D_i^o$ be the last vertex of D_i^o on P and denote by P_w the path from w to v in P. By the choice of w there is no vertex in $V(P_w) \setminus \{w\}$ that is also contained in D_i^o, so $P_w \subseteq T_i$.

Moreover, we know that $w \in D_i^o$ so w is reachable from r in T_i by a path $P'_w \subseteq T_i$. Connecting P'_w and P_w yields an r-v-path in T_i. □

T and T_i are spanning trees on $D \cup \{r\}$. By Lemma 1.23 there exists a bijective mapping $\pi_i : E_T(D_i^o) \to E(T_i^o)$ so that $(D \cup \{r\}, E(T) \setminus \{e\} \cup \{\pi_i(e)\})$ is a spanning tree for all $e \in E_T(D_i^o)$ and $\pi_i(e) = e$ for all $e \in E(T) \setminus E_T(D_i^o)$.

Remark 1.26. *For all $e \in E_T(D_i^o)$ the graph $T \cup \{\pi_i(e)\}$ contains a unique circuit C with $\{e, \pi_i(e)\} \subseteq E(C)$.*

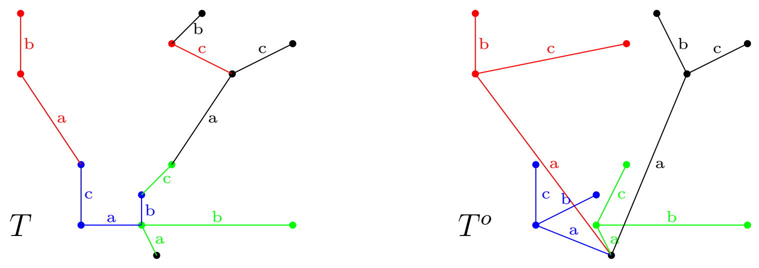

Figure 1.7: Illustration of the bijective function $\pi : E(T) \to E(T^o)$. Edges with the same label in the same color are mapped on each other.

As T is a minimum spanning tree we have $c'(\pi_i(e)) \geq c'(e)$ for $e \in E_T(D_i^o)$.
$E_T(D_1^o)\dot\cup \ldots \dot\cup E_T(D_{k^o}^o)$ is a partition of $E(T)$ so we can extend the mappings π_i, $i \in \{1, \ldots, k^o\}$ to a mapping $\pi : E(T) \to E(T^o)$ with $\pi|_{D_i^o} = \pi_i$ (see Figure 1.7).

Remark 1.27. *Note that $\pi(e) = e$ for all $e \in E(T) \cap E(T^o)$.*

For simplicity we define $c_\pi : E(T) \to \mathbb{R}_+$ by $c_\pi(e) := c'(\pi(e))$ for all $e \in E(T)$.

K-dominated Functions

We have to introduce some more definitions in order to handle the new bounds.
Let $K \subseteq D$. For each vertex $v \in K$ there exists exactly one edge of T ending in v, thus $|K| = |E_T(K)|$. For a function $g : E(T) \to \mathbb{R}_+$ let $e_1^{g,K}, \ldots, e_{|K|}^{g,K}$ be the edges of $E_T(K)$ sorted in non-increasing order with respect to g, i.e. $g(e_1^{g,K}) \geq g(e_2^{g,K}) \geq \ldots \geq g(e_{|K|}^{g,K})$.

Definition 1.28. *Let $g, h : E(T) \to \mathbb{R}_+$ be two cost functions on the edges of T and $K \subseteq D$. g is called K-dominated by h if $g(e_i^{g,K}) \leq h(e_i^{h,K})$ for all $i \in \{1, \ldots, |K|\}$.*

Proposition 1.29. *If $K_1, K_2 \subseteq D$, $K_1 \cap K_2 = \emptyset$, $g, h : E(T) \to \mathbb{R}_+$, g is K_1-dominated by h and g is K_2-dominated by h then g is $K_1 \cup K_2$-dominated by h.*

Proof. It is sufficient to show that for any $j \in \{1, \ldots, |K_1 \cup K_2|\}$ there are at least j edges $e \in E_T(K_1 \cup K_2)$ with $g(e_j^{g,K_1\cup K_2}) \leq h(e)$. To show this, let $p : \{1, \ldots, |K_1 \cup K_2|\} \to \{1, 2\}$ be a function satisfying $e_i^{g,K_1\cup K_2} \in E_T(K_{p(i)})$, $1 \leq i \leq |K_1 \cup K_2|$. As $K_1 \cap K_2 = \emptyset$, each edge $e \in E_T(K_1 \cup K_2)$ is either in $E_T(K_1)$ or in $E_T(K_2)$ and thus

the function p exists and is unique. Moreover, define $\pi : \{1, \dots, |K_1 \cup K_2|\} \to \mathbb{N}$ so that $e_i^{g,K_1\cup K_2} = e_{\pi(i)}^{g,K_{p(i)}}$.
Now choose $j \in \{1, \dots, |K_1 \cup K_2|\}$. Using that g is K_1- and K_2-dominated by h we conclude that for every $i \in \{1, \dots, j\}$ we have

$$g\left(e_j^{g,K_1\cup K_2}\right) \le g\left(e_i^{g,K_1\cup K_2}\right) = g\left(e_{\pi(i)}^{g,K_{p(i)}}\right) \le h\left(e_{\pi(i)}^{h,K_{p(i)}}\right).$$

As $K_1 \cap K_2 = \emptyset$ the set $\left\{e_{\pi(i)}^{h,K_{p(i)}} \,|\, 1 \le i \le j\right\} \subseteq E_T(K_1 \cup K_2)$ contains j elements. □
If g is D-dominated by c_π then $\sum_{e\in E(T)} g(e)$ is a lower bound for the cost of an optimum solution as $\sum_{e\in E(T)} c_\pi(e)$ is a lower bound, too.
As $c'(e) \le c_\pi(e)$ for all $e \in E(T)$, c' is K-dominated by c_π for all $K \subseteq D$.

A Lower Bound for the Number of Components

Let $\left(k^{\text{feas}}, \{(D_i^{\text{feas}}, S_i^{\text{feas}})\}_{i\in\{1,\dots,k^{\text{feas}}\}}\right)$ be a feasible clustering. Again let $K \subseteq D$, $K \ne \emptyset$, be a subset of D.
By $I_K = \{i \,|\, D_i^{\text{feas}} \cap K \ne \emptyset\}$ we denote the set of components that have a sink in common with K. We want to compute a lower bound for $|I_K|$ with the aid of a function $g : E(T) \to \mathbb{R}_+$ that is K-dominated by c_π.
The load of the components of I_K is at least

$$\begin{aligned}
& \sum_{i\in I_K}\left(d\left(D_i^{\text{feas}}\right) + \sum_{e\in E(S_i^{\text{feas}})} c(e)\right) \\
\overset{(1.13)}{\ge}\; & \sum_{i\in I_K}\left(d\left(D_i^{\text{feas}}\right) + \sum_{e\in E(T_i^{\text{feas}})} c'(e) - f\right) \\
\overset{\text{def. of }\pi}{=}\; & \sum_{i\in I_K}\left(d\left(D_i^{\text{feas}}\right) + \sum_{s\in D_i^{\text{feas}}} c_\pi(e_T(s)) - f\right) \\
\ge\; & \sum_{i\in I_K}\left(d\left(D_i^{\text{feas}} \cap K\right) + \sum_{s\in D_i^{\text{feas}}\cap K} c_\pi(e_T(s)) - \max_{s\in D_i^{\text{feas}}\cap K} c_\pi(e_T(s))\right) \\
=\; & d(K) + \sum_{s\in K} c_\pi(e_T(s)) - \sum_{i\in I_K}\left(\max_{s\in D_i^{\text{feas}}\cap K} c_\pi(e_T(s))\right) \\
\ge\; & d(K) + \sum_{s\in K} c_\pi(e_T(s)) - \sum_{i=1}^{|I_K|}\left(c_\pi(e_i^{c_\pi,K})\right) \\
\ge\; & d(K) + \sum_{s\in K} g(e_T(s)) - \sum_{i=1}^{|I_K|}\left(g(e_i^{g,K})\right).
\end{aligned}$$

The first equation comes from the definition of c_π and in the last inequality we used that g is K-dominated by c_π.

As each cluster has load at most u we get:

Lemma 1.30. *The minimum $t \in \mathbb{Z}$ satisfying*

$$d(K) + \sum_{s \in K} g(e_T(s)) - \sum_{i=1}^{t} \left(g(e_i^{g,K}) \right) \quad \leq \quad t \cdot u$$

is a lower bound for the number of clusters that have a sink in common with K for any feasible clustering.

For $K \subseteq D$, $1 \leq t \leq |K|$ and $g : E(T) \to \mathbb{R}_+$ K-dominated by c_π we denote by $L(K, g, t)$ the total length of the t longest edges in $E_T(K)$ with respect to g, i.e.

$$L(K, g, t) \quad := \quad \sum_{i=1}^{t} \left(g \left(e_i^{g,K} \right) \right)$$

and set

$$C(K, g, t) \quad := \quad d(K) + \sum_{s \in K} g(e_T(s)) - L(K, g, t).$$

By Lemma 1.30 the minimum $t \in \mathbb{N}$ satisfying $C(K, g, t) \leq t \cdot u$ is a lower bound for $|I_K|$. We will denote this value by t_K^g.

Improving K-dominated Functions

Now we show how to 'improve' a K-dominated function by analysing parts of the instance more carefully. To this end, we make some preparative observations.

Proposition 1.31. *For all $j \in \{1, \ldots, k^o\}$ with $K \cap D_j^o \neq \emptyset$ there is a sink $v \in K \cap D_j^o$ so that $\pi(e_T(v))$ joins a vertex that is not in K.*

Proof. $E' := \{\pi(e_T(v)) | \, v \in K \cap D_j^o\}$ is a subset of the edges of T_j^o. π is bijective, so $|E'| = |K \cap D_j^o|$. As T_j^o is a spanning tree, E' cannot contain a circuit, so there must be an edge in E' that joins a vertex that is not in $K \cap D_j^o$. Recall that the endpoints of the edges in E' are in $D_j^o \cup \{r\}$, and the proof is complete. □

We denote by $l_K := \min_{e \in \delta_T(K)} c'(e)$ the minimum length of an edge in the cut defined by K in T.

Proposition 1.32. *Let $\hat{v} \in K \cap D_j^o$ be a vertex so that $\pi(e_T(\hat{v}))$ joins a vertex that is not in K. Then $c_\pi(e_T(\hat{v})) \geq l_K$.*

Proof. If $\pi(e_T(\hat{v})) = e_T(\hat{v})$ then $e_T(\hat{v}) \in \delta_T(K)$ and the claim follows directly. Otherwise, by Remark 1.26 there is a unique circuit C in $(V(T), E(T) \cup \{\pi(e(\hat{v}))\})$ with $\{e_T(\hat{v}), \pi(e_T(\hat{v}))\} \subseteq E(C)$. C contains vertices both of K and of $(V \cup \{r\}) \setminus K$, so two edges of $E(C)$ are in $\delta_C(K)$ and at least one of them is an element of $\delta_T(K)$ and therefore in $\delta_T(K) \cap E(C)$. Choose $\hat{e} \in \delta_T(K) \cap E(C)$. As T is a minimum spanning tree $c_\pi(e_T(\hat{v})) = c'(\pi(e_T(\hat{v}))) \geq c'(\hat{e}) \geq l_K$. □

Combining Proposition 1.31 and 1.32 yields:

Corollary 1.33. *There are at least $|I_K|$ edges $e \in E_T(K)$ with $c_\pi(e) \geq l_K$.*

We get the main result of this section:

Lemma 1.34. *Let $K \subseteq D$, $g : E(T) \to \mathbb{R}_+$ be a function that is K-dominated by c_π and $1 \leq t \leq t_K^g$. We define $g' : E(T) \to \mathbb{R}_+$ as*

$$g'(e) := \begin{cases} \max\{l_K, g(e)\} & e = e_i^{g,K} \text{ for an } i \in \{1, \ldots, t\} \\ g(e) & \text{otherwise.} \end{cases}$$

Then g' is K-dominated by c_π.

Proof. Set $L_K := \{e \in E_T(K) | \; c_\pi(e) \geq l_K\}$. As t_K^g is a lower bound for $|I_K|$ and $|L_K| \geq |I_K|$ by Proposition 1.32, we have $|L_K| \geq t_K$ and the claim follows immediately. □

Clearly, Lemma 1.34 is also true when we replace t_K^g by $|I_K|$.

A Lower Bound using Dominated Functions

Now we show how to compute a lower bound by defining a sequence $g_{K_1}, \ldots, g_{K_n}$ of functions on the edges of T that are D-dominated by c_π. To this end, we construct a laminar family $\mathcal{G}$ of subsets of D. Initially $\mathcal{G}$ contains all subsets of D with one element. Let $e_1, \ldots, e_n$ be the edges of the minimum spanning tree T sorted in non-decreasing order, i.e. $c'(e_1) \leq c'(e_2) \leq \ldots \leq c'(e_{n-1}) \leq c'(e_n)$. Recall that T is a minimum spanning tree on $D \cup \{r\}$, $n = |D|$ and that the distance between any sink and r is f. Thus $c'(e_n) = f$. Now assume that there are $k \geq 2$ edges connected to r. We remove $k-1$ of them and get $k-1$ connected components. Adding $k-1$ new edges between sinks of these components we achieve a spanning tree T' where only one edge is connected to r. By the definition of c' the length of any edge is at most f and thus T' is also a minimum spanning tree. So we can assume without loss of generality that only the edge e_n is connected to r.

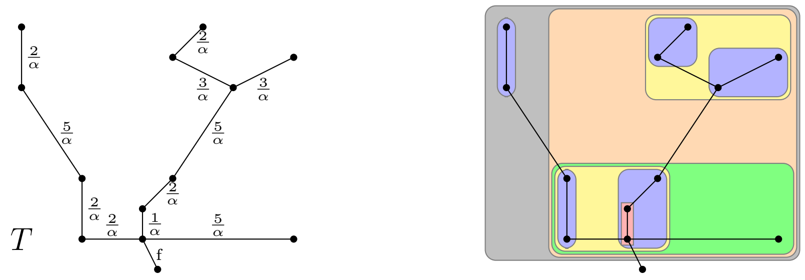

Figure 1.8: The minimum spanning tree T in (V', c') with edge lengths and a corresponding laminar family $\mathcal{G}$. As some lengths appear several times, $\mathcal{G}$ is not unique.

For $i = 1$ to n let A_i and B_i be the two unique maximal sets of $\mathcal{G}$ that are connected by e_i and add the set $K_i = A_i \cup B_i$ to $\mathcal{G}$ (see also Figure 1.8). Note that if there are

edges in T of the same length, $\mathcal{G}$ is not unique. We denote by $T[K]$ the graph induced by a $K \subseteq D$ in T.
By construction we get:

Lemma 1.35. *Let $K_i, A_i, B_i \in \mathcal{G}$ with $K_i = A_i \cup B_i$ defined as above. Then*

- $T[K_i] = (A_i \cup B_i, E(T[A_i]) \cup E(T[B_i]) \cup \{e_i\})$,
- $T[K_i]$ *is connected,*
- e_i *is at least as long as every edge in* $T[A_i]$ *and* $T[B_i]$ *and*
- e_i *is a shortest edge in* $\delta_T(A_i)$ *and* $\delta_T(B_i)$.

Now we define for each set $K \in \mathcal{G}$ a function $g_K : E(T) \to \mathbb{R}_+$ that is K-dominated by c_π. Initially we set $g_K \equiv c'$ for $K \in \mathcal{G}$ with $|K| = 1$.
Now assume we have defined g_{K_j} for $j < i$.
We define $\tilde{g}_{K_i}$:

$$\tilde{g}_{K_i}(e) = \begin{cases} c'(e) & e \in E(T) \setminus E(T[K_i]) \\ g_{A_i}(e) & e \in E(T[A_i]) \\ g_{B_i}(e) & e \in E(T[B_i]) \\ c'(e_i) & e = e_i. \end{cases}$$

$\tilde{g}_{K_i}$ is well defined by Lemma 1.35 and K_i-dominated by c_π by Proposition 1.29. Choose $t_{K_i} := t_{K_i}^{\tilde{g}_{K_i}}$ as above, i.e. let t_{K_i} be the minimum integer satisfying $C(K_i, \tilde{g}_{K_i}, t_{K_i}) \leq t_{K_i} \cdot u$. We have seen that t_{K_i} is a lower bound for the number of components of any feasible solution that have a nonempty intersection with K_i.
Now we can use Lemma 1.34 to construct g_{K_i} from $\tilde{g}_{K_i}$ by setting the length of the $t_{K_i} - 1$ longest edges of $T[K_i]$ with respect to g_{K_i} to the length of the shortest edge in $\delta_T(K_i)$ if they are shorter. By Lemma 1.34, g_{K_i} is still K_i-dominated by c_π.

Lemma 1.36. $\sum_{e \in E_T(D)} g_D(e)$ *is a lower bound for the cost of an optimum solution.*

Proof. $D = K_{n-1}$ and $g_{K_{n-1}}$ is D-dominated by c_π. □

Remark 1.37. *This lower bound can be implemented so that the running time is dominated by constructing the minimum spanning tree.*

In Section 1.3.5 we will give an example computing this lower bound for the instance shown in Figure 1.5.

The Laminar Family $\mathcal{G}$ is Best Possible

The previous lower bound computation can be done with any laminar family $\mathcal{H}$ of sets of edges from T. A question that might arise is if the chosen laminar family $\mathcal{G}$ is the best possible choice or if there is another laminar family that yields better bounds. We will show now that $\mathcal{G}$ is the best one.

Let $\mathcal{G}$, g_K and $t_K^{g_K}$ for all $K \in \mathcal{G}$ be chosen as above. Let $\mathcal{H}$ be another laminar family of subsets of D and $\tilde{h}_K$, h_K and $s_K^{h_K}$ be defined analogously to $\tilde{g}_K$, g_K and $t_K^{g_K}$, but for $K \in \mathcal{H}$ instead of $K \in \mathcal{G}$. We can assume that $D \in \mathcal{H}$ and for all $K \in \mathcal{H}$ either $|K| = 1$ or there exist $A, B \in \mathcal{H}$ with $A \cap B = \emptyset$ and $K = A \dot{\cup} B$. For the sake of simplicity, we abbreviate $t_K := t_K^{g_K}$ and $s_K := s_K^{h_K}$.
In order to show that $\mathcal{G}$ is the best choice for the laminar family it is sufficient to prove that h_D is D-dominated by g_D.
To see this, we first extend the functions $\tilde{g}_K$ and g_K to be defined for all $K \in \mathcal{H}$: For $K \in \mathcal{H}$ let $\hat{K} := \{A \in \mathcal{G} |\ A \subseteq K \text{ and } A \text{ is maximal}\}$. Then for $A, B \in \hat{K}$ with $A \neq B$ we have $A \cap B = \emptyset$ and $\dot{\bigcup}_{A \in \hat{K}} A = K$. So for every $e \in E_T(K)$ there is a unique $A \in \hat{K}$ with $e \in E_T(A)$ and we set $\tilde{g}_K(e) := \tilde{g}_A(e)$ and $g_K(e) := g_A(e)$. $\tilde{g}_K$ and g_K are well defined. For $K \in \mathcal{H}$ set $t_K := \sum_{A \in \hat{K}} t_A$.

Lemma 1.38. *h_K is K-dominated by g_K for all $K \in \mathcal{H}$.*

Proof. By induction on $|K|$: For $K \in \mathcal{H}$ with $|K| = 1$ the functions are equivalent, i.e. $h_K \equiv g_K$ and $s_K = t_K = 1$.
Assume the claim has been proven for all $K \in \mathcal{H}$ with $|K| < m$. Let $K \in \mathcal{H}$, $|K| = m$. Then there exist $A, B \in \mathcal{H}$ with $A \cap B = \emptyset$ and $A \dot{\cup} B = K$. Moreover, $|A| < m$ and $|B| < m$, so by induction h_A is A-dominated by g_A and h_B is B-dominated by g_B. Using the definition of $\tilde{g}_K$ and Proposition 1.29 we see that $\tilde{h}_K$ is K-dominated by $\tilde{g}_K$ and so by g_K.
s_K was defined to be the minimum integer t so that

$$C(K, \tilde{h}_K, t) \leq t \cdot u.$$

We show that $s_K \leq t_K$, i.e. $t = t_K$ satisfies the above inequality:

$$\begin{aligned}
C(K, \tilde{h}_K, t_K) &\leq C(K, \tilde{g}_K, t_K) \\
&= d(K) + \sum_{s \in K} \tilde{g}_K\left(e_T(s)\right) - L(K, \tilde{g}_K, t_K) \\
&= \sum_{A \in \hat{K}} \left(d(A) + \sum_{s \in A} \tilde{g}_A(e_T(s)) \right) - L\left(K, \tilde{g}_A, \sum_{A \in \hat{K}} t_A \right) \\
&\leq \sum_{A \in \hat{K}} \left(d(A) + \sum_{s \in A} \tilde{g}_A(e_T(s)) - L\left(A, \tilde{g}_A, t_A\right) \right) \\
&= \sum_{A \in \hat{K}} C(A, \tilde{g}_A, t_A) \\
&\leq \sum_{A \in \hat{K}} t_A \cdot u \\
&= t_K \cdot u.
\end{aligned}$$

Let again l_K be the length of the shortest edge in the cut $\delta_T(K)$ with respect to c'. Set $L_K := \{e \in e_T(K) |\ g_K(e) \geq l_K\}$. Now we show that $|L_K| \geq t_K$.

Let e be the shortest edge in $\dot{\bigcup}_{A\in\hat{K}}\delta_T(A)$ with respect to c'. Assume $c'(e) < l_K$. Then $e \notin \delta_T(K)$ by the definition of l_K and there exist $A, B \in \hat{K}$ so that e has a vertex in common with each of the two sets. As e is the shortest edge in $\delta_T(A) \cup \delta_T(B)$, the set $A\dot{\cup}B$ must be an element in $\mathcal{G}$. But this is a contradiction to the maximality of the elements of $\hat{K}$. Therefore every edge in $\bigcup_{A\in\hat{K}} \delta_T(A)$ has length at least l_K.
But then by the definition of g_A at least t_A edges in $\{e_T(v)|\ v \in A\}$ have length $\geq l_K$ with respect to g_A for all $A \in \hat{K}$. All these edges are disjoint for the different sets $A \in \hat{K}$ and we get at least $\sum_{A\in\hat{K}} t_A = t_K$ edges in $E_T(K)$ of g_K-length at least l_K.
$\tilde{h}_K$ is K-dominated by g_K. When we compute h_K, the s_K longest edges according to $\tilde{h}_K$ will be increased to l_K if they are shorter. But as at least $t_K \geq s_K$ edges have g_K-length at least l_K, the function h_K has to be K-dominated by g_K, too.

□

Final Remarks

An interesting fact we will use later is the following observation:

Lemma 1.39. *$t_{A_i} + t_{B_i} - 1 \leq t_{K_i} \leq t_{A_i} + t_{B_i}$ for all $i \in \{1, \ldots, n\}$.*

Proof. Recall that e_i is the unique edge connecting A_i and B_i. Thus $e_i \in \delta_T(A_i)$ and $e_i \in \delta_T(B_i)$ and so by induction and Lemma 1.35 $c'(e_i) = g_{A_i}(e_i) \geq g_{A_i}(e)$ for all $e \in E(T[A_i])$ and $c'(e_i) = g_{B_i}(e_i) \geq g_{B_i}(e)$ for all $e \in E(T[B_i])$.
Moreover, e_i is still a shortest edge in the cuts defined by A_i and B_i as $g_{A_i}(e) = c'(e)$ for all $e \in \delta(A_i)$ and $g_{B_i}(e) = c'(e)$ for all $e \in \delta(B_i)$. But that means that the $t_{A_i} - 1$ longest edges in $E(T[A_i])$ with respect to g_{A_i} and the $t_{B_i} - 1$ longest edges of $E(T[B_i])$ with respect to g_{B_i} have cost $c'(e_i)$. According to $\tilde{g}_{K_i}$ there are altogether at least $g_{A_i} + g_{B_i} - 1$ edges of length $c'(e_i)$ in $E(T[K_i])$ and there is no longer edge.
Let e_{K_i} be the shortest edge in the cut defined by K_i. By construction $c'(e_{K_i}) = \tilde{g}_{K_i}(e_{K_i}) \geq \tilde{g}_{K_i}(e)$ for all $e \in E(T[K_i])$. Recall that $E_T(K_i) = E(T[K_i]) \cup \{e_{K_i}\}$.
These observations show that

$$L(K_i, \tilde{g}_{K_i}, t_{A_i} + t_{B_i} - 2) \;=\; L(A_i, g_{A_i}, t_{A_i} - 1) + L(B_i, g_{B_i}, t_{B_i} - 1).$$

For $t = t_{A_i} + t_{B_i} - 2$ we have

$$\begin{aligned} C(K_i, \tilde{g}_{K_i}, t) &= d(K_i) + \sum_{s\in K_i} \tilde{g}_{K_i}(e_T(s)) - L(K_i, \tilde{g}_{K_i}, t) \\ &= C(A_i, \tilde{g}_{K_i}, t_{A_i} - 1) + C(B_i, \tilde{g}_{K_i}, t_{B_i} - 1) \\ &\geq C(A_i, \tilde{g}_{A_i}, t_{A_i} - 1) + C(B_i, \tilde{g}_{B_i}, t_{B_i} - 1) \\ &\geq (t_{A_i} - 1) \cdot u + (t_{B_i} - 1) \cdot u. \end{aligned}$$

The last inequality follows from the choice of t_{A_i} and t_{B_i}. We conclude that $t_{K_i} > t_{A_i} + t_{B_i} - 2$.

On the other hand we get

$$\begin{aligned}(t_{A_i}+t_{B_i})\cdot u &\geq C(A_i,g_{A_i},t_{A_i}-1)+C(B_i,g_{B_i},t_{B_i}-1)\\ &= d(K_i)+\sum_{e\in E_T(K_i)}\tilde{g}_{K_i}(e)-L(K_i,\tilde{g}_{K_i},t_{A_i}+t_{B_i})\\ &= C(K_i,\tilde{g}_{K_i},t_{A_i}+t_{B_i}).\end{aligned}$$

Thus $t_{K_i}\leq t_{A_i}+t_{B_i}$ which completes the proof. □

1.3.4 A Lower Bound Combining Dominated Functions and Steiner Trees

The lower bound using dominated functions has the advantage that it works locally while the lower bound using minimum Steiner trees is more accurate to the Sink Clustering Problem than using spanning trees. In this section we develop a lower bound that combines the advantages of both approaches.
For this we analyze the laminar family $\mathcal{G}$ that has been defined in Section 1.3.3 more closely. For each $K\in\mathcal{G}$ we have found a lower bound t_K for the number of clusters that contain a sink of K for any feasible clustering.
Let $\mathcal{S}:=\{K_i\in\mathcal{G}|\, t_{A_i}+t_{B_i}=t_{K_i}\}$. $\mathcal{S}$ is well defined. Our goal is to find an edge $e_S\in E(T[S])$ for every $S\in\mathcal{S}$ and a set of edges $E\subseteq E(T)$ with $|E|=k^o-|\mathcal{S}|-1$ so that $(D,E\cup\{e_S|\,S\in\mathcal{S}\}\cup\bigcup_{i=1}^{k^o}E(S_i^o))$ is a Steiner tree.
To this end, we first have to look at the structure of $\mathcal{S}$.

Proposition 1.40. *For each set $K\in\mathcal{G}$ there are exactly t_K-1 sets $S\in\mathcal{S}$ with $S\subseteq K$.*

Proof. By induction on $|K|$. If $|K|=1$ then $t_K=1$ and there is no set $S\in\mathcal{S}$ with $S\subseteq K$ so the claim is true.
Denote by s_K the number of sets $S\in\mathcal{S}$ with $S\subseteq K$. Let $K\in\mathcal{G}$ with $|K|=m>1$. Then $K=K_i=A_i\dot{\cup}B_i$ for some $i\in\{1,\dots,n\}$. As $\mathcal{S}\subseteq\mathcal{G}$ is laminar, for each set $S\in\mathcal{S}$ with $S\subseteq K_i$ we have either $S=K_i$ or $S\subseteq A_i$ or $S\subseteq B_i$. If $K_i\in\mathcal{S}$ then $t_{K_i}=t_{A_i}+t_{B_i}$ and therefore $s_{K_i}=s_{A_i}+s_{B_i}+1=(t_{A_i}-1)+(t_{B_i}-1)+1=t_{K_i}-1$. Otherwise, $K_i\notin\mathcal{S}$. In this case by Lemma 1.39 $t_{K_i}=t_{A_i}+t_{B_i}-1$ and we get $s_{K_i}=s_{A_i}+s_{B_i}=(t_{A_i}-1)+(t_{B_i}-1)=t_{K_i}-1$. □
This proposition helps us to estimate the number of clusters that have a nonempty intersection with subsets of $\mathcal{S}$.

Lemma 1.41. *Let $\mathcal{K}\subseteq\mathcal{S}$ and denote by t the number of clusters that have a sink in common with $\bigcup_{K\in\mathcal{K}}K$ for a feasible clustering. Then*

$$t\geq|\mathcal{K}|+1.$$

Proof. Let $\mathcal{K}'\subseteq\mathcal{K}$ be the set of maximal elements of $\mathcal{K}$. Consider an instance of the Sink Clustering Problem with the reduced set of sinks $D':=\cup_{K\in\mathcal{K}}K$. For $K\in\mathcal{K}'$ with $K\neq D'$ we want to estimate the minimum cost of an edge $\{v,w\}$ with $v\in K$

and $w \in D' \setminus K$. Using that T is a minimum spanning tree and the definition of K we conclude:

$$\begin{aligned} \min_{v \in K, w \in D' \setminus K} c'(\{v,w\}) &\geq \min_{v \in K, w \in D \setminus K} c'(\{v,w\}) \\ &= \min_{e \in \delta_T(K)} c'(e) \\ &\geq \max_{e \in E(T[K])} c'(e). \end{aligned}$$

It follows that there exists a minimum spanning tree T' on D' with $E(T[K]) \subseteq E(T')$ for all $K \in \mathcal{K}'$, i.e. T' contains all trees induced by K for all $K \in \mathcal{K}'$. Let $\mathcal{G}'$ be the corresponding laminar family of T' and t'_K for all $K \in \mathcal{K}$ the lower bound for the number of clusters that contain a sink of K. Moreover, let $\mathcal{S}'$ be defined analogously to $\mathcal{S}$ with respect to $\mathcal{G}'$.
Let t' be the minimum number of components a feasible clustering for D' can have. D' is a subset of D so each feasible clustering for D can be reduced to a feasible solution for D' and therefore $t \geq t'$. $\mathcal{K}$ is a subset of $\mathcal{S}'$. Putting all together and applying Proposition 1.40 we see that $t - 1 \geq t' - 1 \geq |\mathcal{S}'| \geq |\mathcal{K}|$. □
Now we come to the main observation of this section.

Lemma 1.42. *Let V be a finite set and $\mathcal{T} = \{T_1, \dots, T_t\}$ with $T_i = (V(T_i), E(T_i))$ a spanning tree on $V(T_i) \subseteq V$ for $i \in \{1, \dots, t\}$ so that for all $\mathcal{T}' \subseteq \mathcal{T}$:*

$$\left| \bigcup_{T \in \mathcal{T}'} V(T) \right| \geq 1 + |\mathcal{T}'|. \tag{1.14}$$

Then there exist edges $e_T \in E(T)$ for all $T \in \mathcal{T}$ so that $F = (V, \{e_T | T \in \mathcal{T}\})$ is a forest.

Proof. First observe that by (1.14) the graph $(\mathcal{T} \dot{\cup} V, \{(T,a) | \, a \in V(T)\})$ satisfies the Hall condition. Thus we can find $v_T \in V(T)$ for all $T \in \mathcal{T}$ so that $v_T \neq v_{T'}$ for $T \neq T' \in \mathcal{T}$. Furthermore, as $|V(T)| \geq 2$ and T is a spanning tree there exist $w_T \in V(T)$ so that $e_T^1 := \{v_T, w_T\}$ is in $E(T)$ for all $T \in \mathcal{T}$.
If $G := (V, \{\{v_T, w_T\} | \, T \in \mathcal{T}\})$ does not contain any circuits set $F := G$ and we are done. Otherwise, we will reduce the number of connected components that contain a circuit one by one until none exists anymore.
For this, let $G_1 = (V', E_1)$ be a connected component of G that contains a circuit. Let $\mathcal{R} := \{T \in \mathcal{T} | \, \{v_T, w_T\} \in E_1\}$ be the set of trees that contribute an edge to G. All v_T, $T \in \mathcal{R}$, are different, thus $|V(G_1)| \geq |E(G_1)|$. But as G_1 is connected it follows that the component contains exactly one circuit C_1.
The idea of the remaining proof is to eliminate this circuit by removing an edge of it and replacing it by another one from the same tree $T \in \mathcal{R}$. The problem is that inserting the new edge can create a new circuit C_2. In this case we are looking for another edge either of the initial circuit or the second circuit that will be replaced by an other edge with one endpoint outside of C_1 and C_2. Again this can create a new circuit C_3. We repeat this procedure until we replace an edge by another without creating a new circuit.

We construct a sequence of graphs $G_2 = (V', E_2), G_3 = (V', E_3), \dots$ on the same vertices as G_1 with $E_i = \{e_T^i | T \in \mathcal{R}\}$ where $e_T^i \in E(T)$ are edges that will be defined later. Moreover, there will be exactly one circuit $C_i \in E_i$ in each G_i. We denote by

$$\mathcal{R}_i := \{T \in \mathcal{R} | \, \exists j \in \{1, \dots, i\} \text{ s.t. } e_T^j \in C_j\}$$

the set of trees that contribute an edge to at least one of the circuits $C_1, \dots, C_i$. Let $V(\mathcal{R}_i) = \bigcup_{j=1}^{i} V(C_j)$ be all vertices that are covered by at least one of the circuits $C_1, \dots, C_i$.

Assume we have created $G_1, \dots, G_i$. By (1.14) $|\mathcal{R}_i| + 1 \leq |\bigcup_{T \in \mathcal{R}_i} V(T)|$. But then there exists a tree $T' \in \mathcal{R}_i$ and a vertex $v \in V(T')$ so that v is not contained in any previous circuit, i.e. $v \notin V(\mathcal{R}_i)$.

As T' is connected and $V(T') \cap V(\mathcal{R}_i) \neq \emptyset$ there exist $v', w' \in V(T')$ with $\{v', w'\} \in E(T')$, $v' \notin V(\mathcal{R}_i)$ and $w' \in V(\mathcal{R}_i)$. Choose $k \in \{1, \dots, i\}$ so that the edge T' contributes to G_k is in the circuit C_k, i.e. $e_{T'}^k \in E(C_k)$. Such a circuit must exist as $T' \in \mathcal{R}_i$.

If $v' \in V'$, i.e. it is in the connected component, set $e_T^{i+1} = e_T^i$ for $T \in \mathcal{R} \setminus \{T'\}$ and $e_{T'}^{i+1} = \{v', w'\}$. Then by the choice of $e_{T'}^{i+1}$, $G_{i+1} = (V', \{e_T^{i+1} | T \in \mathcal{R}\})$ again contains exactly one circuit C_{i+1}. As $v' \in V(\mathcal{R}_{i+1})$, $v' \notin V(\mathcal{R}_i)$ and $V(\mathcal{R}_i) \subseteq V(\mathcal{R}_{i+1})$ we conclude $|V(\mathcal{R}_{i+1})| > |V(\mathcal{R}_i)|$. But as there is only a finite set of vertices covered by the trees in $\mathcal{R}$, after a finite number of iterations we get $v' \notin V(\mathcal{R})$.

Now we turn to the case that v' is not inside the connected component G_i, i.e. $v' \notin V'$. The graph $(V', E_k \setminus \{e_T^k\})$ does not contain a circuit and $\{v', w'\}$ connects two different connected components. Thus setting $e'_T := e_T, T \in \mathcal{T} \setminus \mathcal{R}$, $e'_T := e_T^k$, $T \in \mathcal{R} \setminus \{T'\}$ and $e'_{T'} := \{v', w'\}$ yields a graph $G' = (V, \{e'_T | T \in \mathcal{R}\})$ that contains one circuit less than G. We set $G := G'$ and continue in this fashion with the next connected component that contains a circuit. Finally, after a finite number of iterations, we get the proposed forest F. □

Obviously, the lemma is still true if the T_i are connected graphs instead of spanning trees. Now we will apply this lemma on our laminar family $\mathcal{S}$. To this end, let $\big(k^o, \{(D_i^o, S_i^o)\}_{i \in \{1, \dots, k^o\}}\big)$ be an optimum clustering again. Let $V' := \{D_i^o | \, 1 \leq i \leq k^o\}$ be the set of all partition sets of the clustering. For $K \in \mathcal{S}$ let T_K be the graph obtained by merging the sinks of $T[K]$ that are within the same cluster to one vertex. Then T_K is a connected graph on a subset of V'. Set $\mathcal{T} := \{T_K | \, K \in \mathcal{S}\}$. By Lemma 1.41 the preconditions of Lemma 1.42 are satisfied. Thus we can find edges $e_K \in E(T_K)$ for $K \in \mathcal{S}$ so that $\{e_K | \, K \in \mathcal{S}\}$ is a spanning forest on the clusters of the optimum clustering.

For $K \in \mathcal{S}$ let e'_K be the edge in the initial spanning tree T that became e_K after merging. Then $H := (D, \bigcup_{i=1}^{k^o} E(S_i^o) \cup \{e'_K | \, K \in \mathcal{S}\})$ does not contain a circuit. Thus we can find additional $k^o - |\mathcal{S}| - 1$ edges $E' \subseteq E(T)$ that complete the graph H to a Steiner tree on D.

Unfortunately we cannot compute these edges efficiently, but we can bound their total length. Let $K_1, \dots, K_{|\mathcal{S}|}$ be an order of the elements of $\mathcal{S}$ so that $K_i \subseteq K_j$ if $i \leq j$.

Now set

$$\hat{e}_i := \begin{cases} \text{longest edge in } E(T[K_1]) & i = 1, \\ \text{longest edge in } E(T[K_i]) \setminus \{e_1, \dots, e_{i-1}\} & 2 \leq i \leq |\mathcal{S}|, \\ \text{longest edge in } E(T) \setminus \{e_1, \dots, e_{i-1}\} & |\mathcal{S}| < i < |D|. \end{cases}$$

It is obvious that $\sum_{i=1}^{k^o-1} c(\hat{e}_i) \geq \sum_{K \in \mathcal{S}} c(e_K) + \sum_{e \in E'} c(e)$.
In the end, we get a lower bound similar to the one of Theorem 1.19.

Theorem 1.43. *Let t_{smtdom} be the smallest integer satisfying*

$$d(D) + l_{\text{SMT}} - \sum_{i=1}^{t_{\text{smtdom}}} c(\hat{e}_i) \quad \leq \quad t_{\text{smtdom}} \cdot u.$$

Then t_{smtdom} is a lower bound for the number of facilities of any feasible solution. Moreover,

$$\min_{t \geq t_{\text{smtdom}}} \left(l_{\text{SMT}} - \sum_{i=1}^{t} c(\hat{e}_i) + t \cdot f \right)$$

is a lower bound for the cost of an optimum solution.

Remark 1.44. *As this lower bound is a combination of the former two bounds its computation is as fast as computing a lower bound for a minimum Steiner tree and constructing a minimum spanning tree.*

1.3.5 An Example

In this section we will use the example of Figure 1.5 in order to illustrate the computation of the four lower bounds that have been presented. The instance consists of 12 sinks, some have demand 1, the others have demand 0. The load limit is $u := 7$ and the facility cost is $f := 10$. An optimum solution has cost $23 + 4f = 63$.
A minimum Steiner tree on D has length 29. We will use this value as lower bound l_{SMT}.

First Lower Bound from Section 1.1

First we compute the lower bound of Section 1.1. The total demand of the sinks is $d(D) = 5$. Moreover, the length of a minimum 1-,2-,3-spanning forest on D is $c(T_1) = 33, c(T_2) = 28$ and $c(T_3) = 23$. By Lemma 1.8 we get $t_{lb} = 3$ as $\frac{c(T_3)}{\alpha} + d(D) = 23\frac{2}{3} + 5 = 20 + \frac{1}{3} \leq 21 = 3u$. Using Lemma 1.9 we get as lower bound $23\frac{2}{3} + 3f = 45 + \frac{1}{3}$.

Lower Bound Based on Minimum Steiner Trees from Section 1.3.1

We have $l_{\text{SMT}} = 29$ as lower bound for the length of a minimum Steiner tree on D. The three longest edges of the minimum spanning tree on D all have length 5. Thus using Theorem 1.19 we get $t_{\text{SMT}} = 4$ and $l_{\text{SMT}} - (5 + 5 + 5) + 4f = 54$ as lower bound for the cost of an optimum solution.

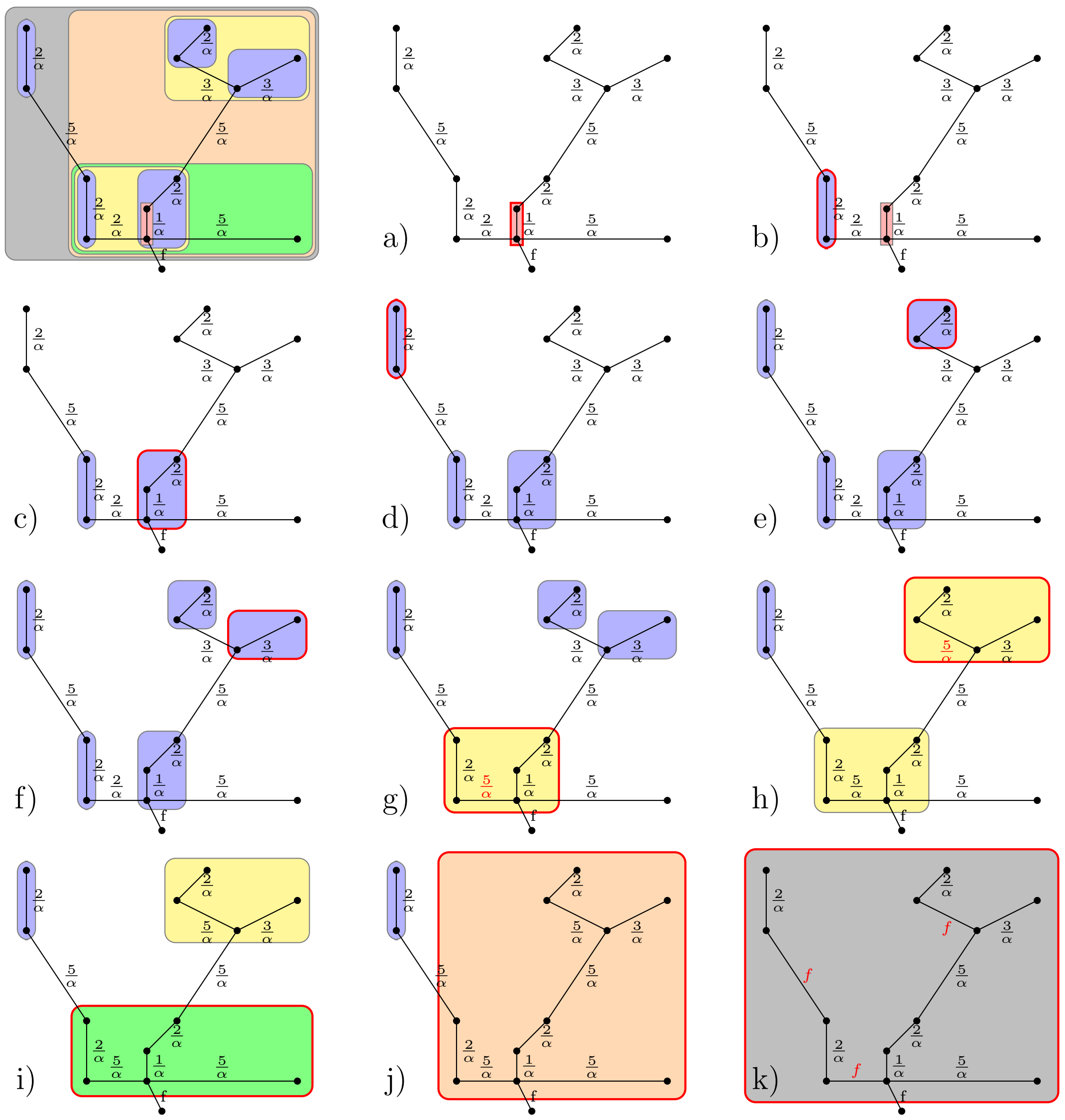

Figure 1.9: The first picture shows the spanning tree with the laminar family $\mathcal{G}$ and the initial cost function c'. Figures a) to k) show the graphs for $g_{K_1}, \ldots, g_{K_{11}}$ with the corresponding sets K_i (marked by a red bounding box). Edge costs that have been updated are written in red.

Lower bound using dominated functions from Section 1.3.3

We compute a minimum spanning tree on $D \cup \{r\}$ and get the initial cost function c' and the laminar family G as shown in Figure 1.9 a).
$t_{K_1} = \ldots = t_{K_6}$ and so $g_{K_1} = \ldots = g_{K_6} = c'$. The first proper computation occurs when considering K_7 (shown in 1.9 b)). There $C(K_7, g_{K_6}, 1) = \frac{5}{\alpha} + 3 = 7 + \frac{1}{3} > u$ but $C(K_7, g_{K_6}, 2) \leq 2u$, thus we get $t_{K_7} = 2$. Moreover, $\lambda_{K_7} = \frac{5}{\alpha}$. Applying Lemma 1.34 two edges of $E_{K_7}(T)$ have length at least λ_{K_7}. Therefore, we have to increase the cost of an internal edge and get a new cost function g_{K_7}.
We continue with K_8, K_9, K_{10} and K_{11} and compute the new cost functions $g_{K_8}, g_{K_9}, g_{K_{10}}$ and $g_{K_{11}}$ in each iteration. Table 1.2 shows all computed values.

i	t_{K_i}	λ_i	$\sum_{e\in E(T)} g_{K_i}(e)$	K_i
1	1	$2\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$\{s_9\}\dot\cup\{s_{11}\}$
2	1	$2\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$\{s_7\}\dot\cup\{s_{10}\}$
3	1	$2\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$K_1\dot\cup\{s_8\}$
4	1	$5\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$\{s_1\}\dot\cup\{s_5\}$
5	1	$3\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$\{s_2\}\dot\cup\{s_3\}$
6	1	$3\frac{1}{\alpha}$	$32\frac{1}{\alpha} + f$	$\{s_4\}\dot\cup\{s_6\}$
7	2	$5\frac{1}{\alpha}$	$35\frac{1}{\alpha} + f$	$K_2\dot\cup K_3$
8	2	$5\frac{1}{\alpha}$	$37\frac{1}{\alpha} + f$	$K_5\dot\cup K_6$
9	2	$5\frac{1}{\alpha}$	$37\frac{1}{\alpha} + f$	$K_7\dot\cup\{s_{12}\}$
10	3	$5\frac{1}{\alpha}$	$37\frac{1}{\alpha} + f$	$K_8\dot\cup K_9$
11	4	f	$22\frac{1}{\alpha} + 4f$	$K_4\dot\cup K_{10}$

Table 1.2: Values computed for the lower bound using K-dominated functions.

Finally, $g_{K_{11}}$ gives us $\frac{22}{\alpha} + 4f = 54 + \frac{2}{3}$ as lower bound for the cost of an optimum solution.

Lower Bound Combining Dominated Functions and Steiner Trees from Section 1.3.4

Finally, we compute the last lower bound that combines the ideas of the previous two ones. Again we have $l_{\text{SMT}} = 29$. Moreover, we can conclude from the previous lower bound that $\hat{e}_1 = 2$, $\hat{e}_2 = 3$, $\hat{e}_3 = 5$ and $\hat{e}_4 = 5$ (we do not need the other values). By Theorem 1.43 we get $t_{\text{smtdom}} = 4$ and $l_{\text{SMT}} - (2 + 3 + 5) + 4f = 59$ as a lower bound for the cost of an optimum solution.
Table 1.3 shows a summary of the four lower bounds.

1.3.6 Analysis

We now analyze the gap between the lower bounds and an optimum solution in the worst case. First we show that there is no lower bound computable in polynomial time and better than $\frac{1}{2}$ times the cost of an optimum clustering for any instance unless $P = NP$.

cost of optimum solution	63
Lower bound Section 1.1	$45 + \frac{1}{3}$
Lower bound Section 1.3.1	54
Lower bound Section 1.3.3	$54 + \frac{2}{3}$
Lower bound Section 1.3.4	59

Table 1.3: Summary of the four lower bounds applied on the example instance.

Proposition 1.45. *Let (V, c) be a metric space and L be a polynomial time algorithm that computes a lower bound $L(I)$ for the cost $O(I)$ of an optimum solution for any instance $I = (V, c, D, d, f, u)$ of the* Sink Clustering Problem. *If there is an $\epsilon > 0$ so that $L(I) > \frac{O(I)}{2} + \epsilon$ for every instance I then the* Steiner Tree Decision Problem *can be solved in polynomial time.*

Proof. Let (D, k) be an instance of the Steiner Tree Decision Problem (Is there a Steiner tree on D of length at most k?). We construct an instance $I = (V, c, D, d, f, u)$ of the Sink Clustering Problem by setting $d(s) := 0$ for all $s \in D$, $u := k$ and $f \gg k$. W.l.o.g. we may assume that $k \le \frac{\epsilon}{2}$ by scaling k and c with the same appropriate value.
Assume there is a Steiner tree on D of length k. Then $L(I) \le O(I) \le f + k$. Otherwise, there have to be at least two clusters and we get $O(I) \ge 2f$. Therefore $L(I) \ge \frac{O(I)}{2} + \epsilon \ge f + \epsilon > f + k$. We conclude that there is a Steiner tree of length $\le k$ if and only if $L(I) \le f + k$. □
In Section 1.1.3 we presented the Sink Clustering Algorithm. It computes a feasible clustering of cost at most $(2\alpha + 1)$ times the lower bound presented in Section 1.1. As the lower bound using K-dominated functions is at least as good as this bound, both have a gap of at most $2\alpha + 1$.
For the rest of this section we give an example for an instance where the lower bound of Section 1.1 and the lower bound using K-dominated functions are about $\frac{1}{4}$ from the cost of an optimum clustering. So the gap is at least 4. Recall that the Sink Clustering Algorithm in Section 1.1.3 always computes a feasible solution of cost at most 5 times the lower bound of Section 1.1. The lower bound using K-dominated functions cannot be worse. Thus the gap is at most 5.
For $m, r \in \mathbb{N}$ we define the graph $G_r^m = (V, E)$ with $V = \{s\} \cup \bigcup_{i=1}^{r} \{v_i^1, \ldots, v_i^m\}$ and $E = \bigcup_{i=1}^{m} \{(s, v_i^1), (v_i^1, v_i^2), (v_i^2, v_i^3), \ldots, (v_i^{m-1}, v_i^m)\}$ (see Figure 1.10). Set $c(e) := 1$ for all $e \in E$ and let $c(\{v_1, v_2\})$ be the length of the unique path between v_1 and v_2 in G. It can be verified easily that (V, c) is a metric.
Consider the set $D' := \{v_1^m, \ldots, v_r^m\}$. $c(\{v_i^m, v_j^m\}) = 2m$ for $i \neq j$ and so the length of a minimum spanning tree on D' is $2m \cdot (r - 1)$. On the other side, G_r^m is a Steiner tree on D' and has length $m \cdot r$. This shows that the Steiner ratio α in (V, c) is at least $\frac{2(r-1)}{r}$. It can be shown that indeed α is equal to $\frac{2(r-1)}{r}$.
Now set $D := V$, $d(v) := 0$ for $v \in V$, $u := 2m - 1$ and $f \gg u$. An optimum solution has at least r clusters as $D' \subseteq V$ and the distance between two elements in D' is greater than u.

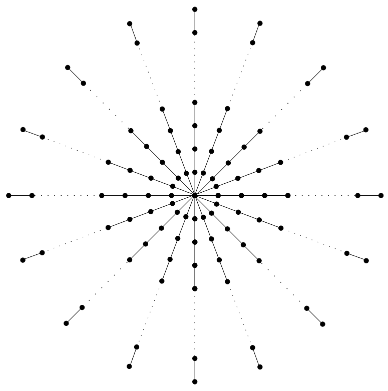

Figure 1.10: Illustration of the example showing that the optimum solution can cost 4 times more than the lower bound of Section 1.1 and the lower bound using K-dominated functions in general metrics.

G_r^m is a minimum spanning tree on D. As every edge in G_r^m has length 1, the lower bound of Section 1.1 and the bound using K-dominated functions have the same value and so we only have to analyze the first one. For any k let T_k be the minimum k-spanning forest we obtain by removing $k-1$ edges of G_r^m. It is easy to see that $c(T_k) = m \cdot r - k + 1$. Now we look for the minimum integer t so that $\frac{1}{\alpha}c(T_t) \le t \cdot u$, i.e.

$$\frac{r}{2(r-1)}(m \cdot r - t + 1) \quad \le \quad t(2m-1).$$

This is equivalent to

$$\frac{mr^2 + r}{4rm - r - 4m + 1} \quad \le \quad t.$$

For m and r large enough we get $t \approx \lceil \frac{r}{4} \rceil + 1$ and the lower bound $L = f \cdot \left(\lceil \frac{r}{4} \rceil + 1 \right) + \frac{1}{\alpha}c(T_{\lceil \frac{r}{4} \rceil + 1})$. An optimum solution has cost $O = f \cdot r + c(T_r)$. Thus for any $\epsilon > 0$ there exist r, m and f so that $\frac{O}{L} \ge 4 - \epsilon$.
In the special case of the rectilinear plane it can be shown by a similar example that the gap is at least $\frac{8}{3}$.

1.4 Experimental Results

We have implemented the Sink Clustering Algorithm and the four presented lower bounds. As metric we use the rectilinear plane.

instance	$\|D\|$	$d(D)$	f	u	length MST	l.b. SMT
Z1	529	1.6	0.088	0.164	0.6	0.53
Z2	2 444	9.5	0.127	0.333	14.7	13.52
Z3	7 708	29.8	0.132	0.400	31.2	28.66
Z4	10 896	20.4	0.102	0.126	26.2	23.50
Z5	24 610	74.5	0.122	0.212	39.7	35.07
Z6	39 473	114.1	0.127	0.479	158.7	n.c.
Z7	74 974	162.5	0.097	0.131	341.3	n.c.
Z8	130 064	476.9	0.141	0.379	505.5	n.c.
Z9	141 503	436.5	0.088	0.164	239.2	n.c.
Z10	260 402	653.9	0.112	0.518	745,6	n.c.
Z11	387 666	915.4	0.097	0.446	911,9	n.c.
Z12	802 804	1 760.2	0.127	0.383	1 306.2	n.c.

Table 1.4: The test instances. n.c. = not computed.

The instances summarized in Table 1.4 come from clock trees of some recent chips. The table shows the number of sinks, the total demand, the facility opening cost, the load limit and the length of a minimum spanning tree on D. For the smaller instances we could compute lower bounds for the length of a Steiner minimum tree, using the software GeoSteiner 3.1 Warme et al. [2003].

	core Sink Clustering alg.			... plus post opt.		
instance	total cost	wire cost	#clusters	total cost	wire cost	#clusters
Z1	1.8	0.5	14	1.8	0.5	14
Z2	19.6	9.3	81	19.2	9.5	76
Z3	48.9	27.5	162	48.0	27.4	156
Z4	61.3	21.3	393	58.9	21.6	366
Z5	104.8	35.3	570	101.1	35.1	541
Z6	214.8	135.1	628	210.2	134.3	597
Z7	663.7	281.6	3 939	642.7	276.4	3 776
Z8	828.9	446.7	2 711	808.5	445.2	2 577
Z9	599.9	214.2	4 383	579.8	211.7	4 183
Z10	999.6	680.6	2 848	984.6	680.0	2 719
Z11	1 278.1	857.0	4 341	1 262.3	857.9	4 169
Z12	2 241.9	1 171.8	8 426	2 187.4	1 174.1	7 979

Table 1.5: Total cost of the solution, wiring cost and number of clusters after the initial Sink Clustering Algorithm and after additional post optimization.

Table 1.5 shows the results we got on the instances by applying the Sink Clustering Algorithm and after additional application of the two post optimization algorithms presented in Section 1.2. The table contains the total cost, the wiring cost and the number of clusters for each computed clustering.
Table 1.6 shows the simple lower bound of Section 1.1.3 and the improved lower bound using K-dominated functions. These lower bounds also give us lower bounds for the number of clusters of each feasible clustering. The table shows the lower bound for the cost of an optimum solution, the lower bound for the number of clusters (#clusters) and the ratio of the total cost of the clustering of Table 1.5 and the lower bound.

	lower bound 1 Section 1.1.3			lower bound 2 Section 1.3.3		
instance	cost	#clusters	ratio	cost	#clusters	ratio
Z1	1.5	13	1.146	1.5	13	1.143
Z2	12.6	50	1.529	14.9	55	1.293
Z3	32.3	117	1.487	35.9	124	1.335
Z4	41.1	271	1.432	45.3	289	1.301
Z5	81.1	466	1.247	83.1	472	1.218
Z6	142.0	426	1.480	156.1	450	1.346
Z7	433.5	2 621	1.483	469.8	2 780	1.368
Z8	599.3	2 070	1.349	621.9	2 114	1.300
Z9	437.2	3 461	1.326	460.9	3 555	1.258
Z10	718.5	2 186	1.370	732.9	2 209	1.344
Z11	905.8	3 359	1.394	921.3	3 388	1.370
Z12	1 696.6	6 777	1.289	1 721.8	6 826	1.270
average			1.369			1.296

Table 1.6: Simple lower bounds and lower bounds using K-dominated functions.

In Table 1.7 the corresponding values for the the lower bound using a lower bound for the length of a minimum Steiner tree and the lower bound combining K-dominated functions and the lower bound for the length of a minimum Steiner tree.

	lower bound 3 Section 1.3.1			lower bound 4 Section 1.3.4		
instance	cost	#clusters	ratio	cost	#clusters	ratio
Z1	1.7	14	1.015	1.7	14	1.012
Z2	14.5	53	1.322	17.4	60	1.103
Z3	39.3	130	1.219	44.0	139	1.091
Z4	47.9	300	1.230	52.0	318	1.133
Z5	92.6	500	1.092	94.9	507	1.066
average			1.176			1.081

Table 1.7: Lower bounds based on a lower bound for the length of a minimum Steiner tree.

Finally, Table 1.8 shows the running time of the core clustering algorithm and the two post optimization heuristics and the number of successful applications of TwoClusterOpt and ChainOpt.

	initial	TwoClusterOpt		ChainOpt	
instance	runtime (s)	runtime (s)	succ.	runtime (s)	succ.
Z1	< 0.1	< 0.1	3	< 0.1	1
Z2	0.1	0.4	47	0.1	7
Z3	0.2	1.7	108	0.3	8
Z4	0.4	2.4	354	0.6	75
Z5	1.0	8.6	554	1.7	56
Z6	1.5	12.3	539	3.0	69
Z7	3.6	12.7	2 703	6.8	556
Z8	8.7	45.7	2 360	12.7	303
Z9	7.3	34.8	3 621	13.1	650
Z10	17.5	100.0	2 494	30.5	267
Z11	27.6	117.0	3 529	43.1	360
Z12	68.3	353.7	8 062	98.0	777

Table 1.8: Running time in seconds and number of successful applications of *TwoClusterOpt* and *ChainOpt*.

The experimental results show the quality of the clustering algorithms. For the instances where we could compute a lower bound for the length of a Steiner minimum tree the costs of the clusterings are only about 8% higher than the lower bounds. The costs of the instances are about 30% over the lower bounds based on K-dominated functions (lower bound 2). The results indicate that the performance ratio of our clustering approach is nearly independent of the size of the instances and a lot better than the theoretical worst case scenario.
Moreover, the clustering appears to be extremely fast. Even the biggest instance has been clustered in about one minute and in 9 minutes with additional applied post optimization.

Comparison to Industrial Clustering tool

In this section we compare the results of our clustering algorithm with the clustering used by the clock tree construction tool used by Magma Design Automation Inc.. Paul Lippens from Magma Design Automation Inc. was friendly enough to compute and give us their clustering results.
Table 1.9 shows the instances we used for comparison. In Table 1.10 you can see the total cost, wiring cost and number of clusters in the clusterings computed by the Magma tool and by our clustering algorithm (core algorithm plus post optimization). Our results are throughout better. In any testcase our clustering algorithm uses less wiring and less clusters. In average the total cost is reduced by 6.6% .

instance	$\lvert D\rvert$	$d(D)$	f	u	length MST	lb SMT
A	4 108	9.0	0.140	0.046	7.7	6.7
B	10 968	15.8	0.140	0.133	70.9	65.7
C	17 140	38.5	0.140	0.111	60.3	59.7
D	35 305	285.6	0.140	0.158	105.4	n.c.
E	119 461	258.8	0.140	0.111	314.5	n.c.

Table 1.9: Instances for comparison with Magma clustering.

	Magma clustering			our clustering		
instance	total cost	wire cost	#clusters	total cost	wire cost	#clusters
A	59.9	6.8	379	57.0	6.3	362
B	150.4	51.7	705	140.6	45.8	677
C	210.0	65.1	1 035	179.9	52.7	908
D	454.8	93.7	2 579	441.0	85.2	2 541
E	1 075.7	305.5	5 501	1 007.1	277.7	5 210

Table 1.10: Comparision between the clustering produced by Magma and our clustering.

Lower bounds for the total cost and the number of clusters are shown in Table 1.11. Here we computed the lower bound using K-dominated functions and, if we have a lower bound on the minimum Steiner length, the combined bound. The clusterings produced by the Magma tool have costs about 40%, while our clusterings have costs about 30% over the lower bound.

	lower bound 1		ratio		lower bound 2		ratio	
instance	cost	#clusters	Magma	our	cost	#clusters	Magma	our
A	45.9	296	1.303	1.240	50.6	321	1.183	1.126
B	103.9	487	1.447	1.353	118.4	528	1.269	1.187
C	129.4	670	1.623	1.390	159.9	792	1.313	1.125
D	367.1	2 196	1.239	1.201	-	-	-	-
E	767.3	4 095	1.402	1.312	-	-	-	-
average			1.403	1.299			1.255	1.146

Table 1.11: Lower bounds for the cost and number of clusters of each feasible clustering.

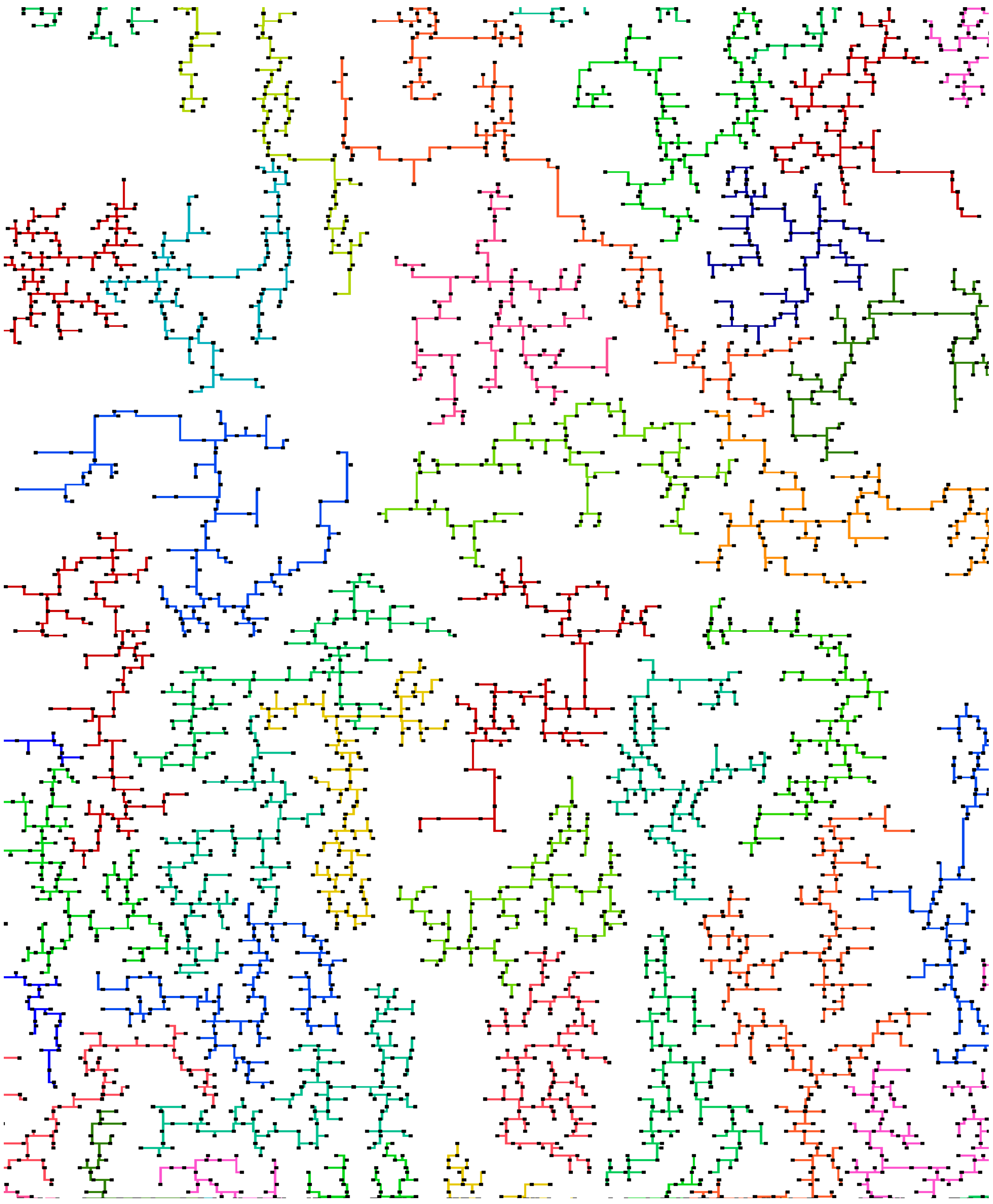

Figure 1.11: Computed clustering on instance Z12. 0.1% of the instance is shown. The coloring is used to distinguish the clusters.

2 The Sink Clustering Problem with Time Windows

As mentioned in the introduction, the Sink Clustering Problem occurs in the clock tree construction when building one stage of a tree. Clock trees have to satisfy some timing constraints. More precisely, the clock signal has to arrive at each sink of the tree within an individual required arrival time window. Often all these time windows have a point of time in common and we can apply on of the Sink Clustering Algorithms presented in the previous chapter. But this is not always the case, for example when skew scheduling has been applied in order to speed up a chip (see Held [2008] for more details). In this case we have to guarantee that the time windows of the sinks of one cluster have at least one point of time in common.
In this chapter we will study this generalization of the Sink Clustering Problem, present an approximation algorithm and lower bounds and, finally, show some experimental results.

2.1 Problem and Algorithm

Sink Clustering Problem with Time Windows

Instance: A metric space (V, c), a finite set D of terminals/customers with positions $p(v) \in V$, demands $d(v) \in \mathbb{R}_+$, time windows $\tau(v) = [\underline{rat}(v), \overline{rat}(v)]$ for all $v \in D$, a facility opening cost $f \in \mathbb{R}_+$ and a load limit $u \in \mathbb{R}_+$.

Task: Find a $k \in \mathbb{N}$ and a clustering $\{(D_i, S_i)\}_{i=1\ldots k}$ of D so that the load limit is kept, i.e.

$$\sum_{e \in E(S_i)} c(e) + \sum_{s \in D_i} d(s) \leq u \qquad \text{for } i = 1, \ldots, k, \tag{2.1}$$

the timing constraints are satisfied, i.e.

$$\max_{v \in D_i} \underline{rat}(v) \leq \min_{v \in D_i} \overline{rat}(v) \qquad \text{for } i = 1, \ldots, k, \tag{2.2}$$

and the clustering cost

$$\sum_{i=1}^{k} \left(\sum_{e \in E(S_i)} c(e) \right) + kf, \tag{2.3}$$

is minimized.

In the special case that all sinks are on the same point, the Sink Clustering Problem with Time Windows reduces to the Bin Packing Problem with Conflicts: Given a set of items $\{1, \dots, n\}$ with sizes $s_1, \dots, s_n \in \mathbb{R}_{\geq 0}$ and a conflict graph $G = (V, E)$, the goal is to find a partition of the items into independent sets of G, where the total size of the items in each independent set is at most 1 so that the number of independent sets is minimized. In our case, the conflict graph is the complement of the interval graph induced by the time windows of the sinks. This graph is perfect. Epstein and Levin [2008] presented a 2.5-factor approximation algorithm for the Bin Packing Problem with Conflicts for perfect conflict graphs. However, the Sink Clustering Problem with Time Windows has not been studied yet.

2.1.1 Initial Observations

From now on let $I = ((V, c), D, d, \tau, u, f)$ be an instance of the Sink Clustering Problem with Time Windows. For simplicity of notation we define by $I_{D'} := ((V, c), D', d|_{D'}, \tau|_{D'}, u, f)$ the reduced instance for the sinks $D' \subseteq D$.
Let $O(I)$ be the cost of an optimum solution for I. Ignoring the time windows of the sinks we get an instance of the Sink Clustering Problem, where $O'(I)$ denotes the cost of an optimum solution. As every solution of the Sink Clustering Problem with Time Windows is also a solution of the Sink Clustering Problem, clearly $O'(I) \leq O(I)$.

Proposition 2.1. *Let $D' \subseteq D$. Then $O(I_{D'}) \leq O(I_D)$.*

Proof. Let $(D_1, S_1), \dots, (D_k, S_k)$ be an optimum solution for instance I. For $1 \leq i \leq k$ define $D'_i = D' \cap D_i$. S_i is still a Steiner tree on D'_i. Thus $(D'_1, S_1), \dots, (D'_k, S_k)$ is a feasible solution for the reduced instance $I_{D'}$ and an optimum solution for $I_{D'}$ cannot be more expensive. □
Note that this lemma is not true if the S_i have to be spanning trees instead of Steiner trees, as a minimum spanning tree on a set of terminals D' can be longer than a minimum spanning tree on a bigger set $D \supset D'$.

Proposition 2.2. *Let $D^a, D^b \subseteq D$ be two subsets of sinks so that*

$$\max_{s \in D^a} \overline{rat}(s) < \min_{s \in D^b} \underline{rat}(s).$$

Then $O(I_{D^a \cup D^b}) = O(I_{D^a}) + O(I_{D^b})$.

Proof. It is obvious that $O(I_{D^a \cup D^b}) \leq O(I_{D^a}) + O(I_{D^b})$. Now let $(D_1, S_1), \dots, (D_k, S_k)$ be an optimum solution for $I_{D^a \cup D^b}$. As the timing constraints have to be satisfied we have either $D_i \cap D^a = \emptyset$ or $D_i \cap D^b = \emptyset$ for $1 \leq i \leq k$. Set $A := \{i | 1 \leq i \leq k, D_i \cap D^b = \emptyset\}$ and $B := \{1, \dots, k\} \setminus A$. Then $D^a = \dot{\bigcup}_{i \in A} D_i$ and $\{D_i, S_i\}_{i \in A}$ is a feasible solution for I_{D^a} that costs at least $O(I_{D^a})$. In the same manner we get that $D^b = \dot{\bigcup}_{i \in B} D_i$ and $\{D_i, S_i\}_{i \in B}$ is a feasible solution for I_{D^b} that costs at least $O(I_{D^b})$. As both solutions together cost $O(I_{D^a \cup D_b})$ we conclude that $O(I_{D^a \cup D^b}) \geq O(I_{D^a}) + O(I_{D^b})$, which completes the proof. □

2.1.2 An Approximation Algorithm

In this section we introduce the first approximation algorithm for the Sink Clustering Problem with Time Windows. It will use an approximation algorithm for the Sink Clustering Problem as a subroutine. But first of all we need some definitions.

Definition 2.3. *Let $T = \{t_1, \dots, t_k\} \subseteq \mathbb{R}$ be a finite set of points and $D' \subseteq D$ be a set of sinks with time windows. We say that T covers D' if for every sink $v \in D'$ there exists a $t \in T$ with $t \in \tau(v)$.*

Definition 2.4. *Let $D' \subseteq D$ be a set of sinks with time windows. We define $\psi(D') := \min_{T \subseteq \mathbb{R}, T \text{ covers } D'} |T|$ to be the minimum size of a set covering D'.*

Proposition 2.5. *A set T covering D' with $\tau(D') = |T|$ can be found in time $O(|D'| \log |D'|)$.*

Proof. Set $D^1 := D'$ and $i := 1$. As long as $D^i \neq \emptyset$ let $v_i \in D^i$ be a sink with $t_i := \overline{rat}(v_i) = \min_{v \in D^i} \overline{rat}(v)$. Set $D^{i+1} = D^i \setminus \{v \in D^i | \, t_i \in \tau(v)\}$ and increment i. Let k be the biggest integer such that $D^k \neq \emptyset$ and $T = \{t_1, \dots, t_k\}$. By construction the time windows $\tau(v_1), \dots, \tau(v_k)$ are pairwise disjoint. Thus $\psi(D') \geq k = |T|$. On the other hand it is obvious that T covers D'. □

Note that $\psi(D')$ could also be defined as the maximum number of pairwise disjoint arrival time windows of sinks in D'.

For the rest of this section let $T_D = \{t_1, \dots, t_k\}$ be a set that covers D with $|T_D| = \psi(D)$. First we compute a partition $D_{t_1} \dot\cup \dots \dot\cup D_{t_k} = D$ of D so that $t_i \in \tau(v)$ for all $v \in D_{t_i}$ and $i \in \{1, \dots, k\}$.

To this end, we will recursively partition a set of sinks into three subsets. The time windows of the sinks in one of these sets will have a non-empty intersection. On this set we can later apply an algorithm for the Sink Clustering Problem ignoring the timing restrictions. The other two sets will be further partitioned if necessary.

In each partition step we have a set of sinks $D' \subseteq D$, a finite set of points $T' = \{t'_1, \dots, t'_h\} \subseteq T_D$ that covers D' and an integer m that will be used later for analysis. We compute a partition $D_{t'_1} \dot\cup \dots \dot\cup D_{t'_h} = D'$ of D' so that $t'_i \in \tau(v)$ for all $v \in D_{t'_i}$ and $i \in \{1, \dots, h\}$. If T' consists of only one element t'_1, we set $depth(t'_1) := m$ and $D_{t'_1} := D'$. Otherwise let j be the median of the indices, i.e. $j := \left\lfloor \frac{h}{2} \right\rfloor$. We set $depth(t'_j) := m$ and

- $D_{\text{left}} := \{v \in D | \, \overline{rat}(v) < t'_j\}$,
- $D_{t'_j} := \{v \in D | \, \underline{rat}(v) \leq t'_j \leq \overline{rat}(v)\}$,
- $D_{\text{right}} := \{v \in D | \, t'_j < \underline{rat}(v)\}$.

By the definition of T' the set $T_{\text{left}} := \{t \in T_D | \, t < t_j\}$ covers D_{left} and $T_{\text{right}} := \{t \in T_D | \, t_j < t\}$ covers D_{right}. Moreover, $D = D_{\text{left}} \dot\cup D_{t'_j} \dot\cup D_{\text{left}}$, $T' = T_{\text{left}} \dot\cup \{t'_j\} \dot\cup T_{\text{left}}$ and $\psi(T') = \psi(T_{\text{left}}) + 1 + \psi(T_{\text{left}})$. If D_{left} and D_{right}, respectively, are not empty, we repeat

the partition step on $(D_{\text{left}}, T_{\text{left}}, m+1)$ and $(D_{\text{right}}, T_{\text{right}}, m+1)$, respectively. The algorithm returns the result of these sub-calls plus $D_{t'_j}$.
Applying this recursive partition on $(D, T_D = \{t_1, \dots, t_k\}, 1)$ we get a partition $D_{t_1}, \dots, D_{t_k}$ of D. For each $i \in \{1, \dots, k\}$ the set $\{t_i\}$ covers D_{t_i}, i.e. the time windows of all sinks in D_{t_i} have a non-empty intersection and we can apply an approximation algorithm for the Sink Clustering Problem on them. The union of the clusters produced by the calls of this algorithm on all D_{t_i}, $i \in \{1, \dots, k\}$, gives a feasible clustering for D.

Lemma 2.6. *Let A_{SCP} be an approximation algorithm for the* Sink Clustering Problem *and δ_{SCP} its approximation guarantee. Then the presented algorithm for the* Sink Clustering Algorithm with Time Windows *using A_{SCP} as sub function has an approximation guarantee of $\delta_{\text{SCP}} \lceil \log_2 (\psi(D)+1) \rceil$.*

Proof. Let T be a minimum set covering D. First observe that $d_{max} := \max_{t \in T} depth(t) = \lceil \log_2 (\psi(D)+1) \rceil$, as we always choose the median of T' when partitioning the sinks D' in each iteration. Set $T(d) := \{t \in T \mid depth(t) = d\}$. For $t, t' \in T(d)$, $t < t'$, we get by construction $\max_{s \in D_t} \overline{rat}(s) < \min_{s \in D_{t'}} \underline{rat}(s)$. Using Proposition 2.1 and 2.2 we see

$$O(I_D) \geq O\left(I_{\bigcup_{t \in T(d)} D_t}\right) = \sum_{t \in T(d)} O(I_{D_t}) = \sum_{t \in T(d)} O'(I_{D_t}).$$

Let $co(D)$ be the cost of the computed solution and $co(D_t)$ be the cost of the partial solution computed by the Sink Clustering algorithm for the set D_t, $t \in T$. We conclude

$$\begin{aligned} co(D) &= \sum_{t \in T} co(D_t) \\ &\leq \delta_{\text{SCP}} \sum_{t \in T} O'(D_t) \\ &= \delta_{\text{SCP}} \sum_{t \in T} O(D_t) \\ &= \delta_{\text{SCP}} \sum_{i=1}^{d_{max}} \sum_{t \in T(d_i)} O(D_t) \\ &\leq \delta_{\text{SCP}} \sum_{i=1}^{d_{max}} O(D) \\ &= \delta_{\text{SCP}} \lceil \log_2 (\psi(D)+1) \rceil O(D). \end{aligned}$$

□

Using the algorithms for the Sink Clustering Problem of Chapter 1 we get:

Lemma 2.7. *The* Sink Clustering Problem with Time Windows *for an instance $((V,c), D, d, \tau, u, f)$ can be solved with approximation ratio*

- $(1+2\alpha)\lceil\log_2(\psi(D)+1)\rceil$,
- $3\beta\lceil\log_2(\psi(D)+1)\rceil$ *and*
- $3\gamma\lceil\log_2(\psi(D)+1)\rceil$

where α is the Steiner ratio, β is the approximation ratio of a minimum Steiner tree algorithm and γ is the approximation ratio of a minimum traveling salesman tour algorithm.

$\square$

Now we present an example showing that the above algorithm could produce a clustering with cost $\log_2(\psi(D))$ times the cost of an optimum solution.
Initially we set $D^1 := \{s\}$ with $p(s) := 1, \underline{rat}(s) := \overline{rat}(s) := 1$ and $d(s) := 0$. For $s \in D^k$ let $\hat{s}^k$ be a sink with position $p(\hat{s}^k) := p(s)+2^k$, required arrival times $\underline{rat}(\hat{s}^k) := \underline{rat}(s)+2^k$, $\overline{rat}(\hat{s}^k) := \overline{rat}(s)+2^k$ and demand $d(\hat{s}^k) := 0$. Set $\hat{D}^k := \{\hat{s}^k | \, s \in D^k\}$.
Moreover, we define for each $1 \le j \le 2^k-1$ the sink s_j^k by setting the position $p(s_j^k) := j$, required arrival times $\underline{rat}(s_j^k) := 1$, $\overline{rat}(s_j^k) := 2^k-1$ and demand $d(s_j^k) := 0$. Set $L^k := \{s_j^k | \, 1 \le j \le 2^k-1\}$. Finally, we need the sink s^k with $p(s^k) := 2^{k-1}$, $\underline{rat}(s^k) := \overline{rat}(s^k) := 2^{k-1}$ and $d(s^k) := 0$.
Now we define recursively $D^k := D^{k-1} \cup \hat{D}^{k-1} \cup L^k \cup \{s^k\}$ (See Figure 2.1). Finally, set the facility cost $f := 1$ and the load limit $u := \frac{1}{2}$. $I^k := ((\mathbb{R}, l_1), D^k, d, \tau, u, f)$, with $\tau(s) = [\underline{rat}(s), \overline{rat}(s)]$ for $s \in D$, forms an instance for the Sink Clustering Problem with Time Windows in the rectilinear plane $(\mathbb{R}, l_1)$.

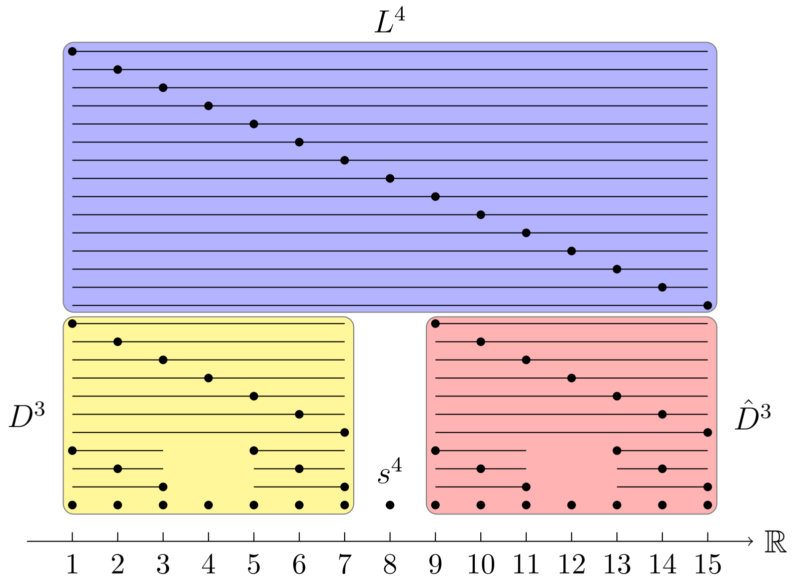

Figure 2.1: Worst case example for the approximation algorithm of Lemma 2.6 for the Sink Clustering Problem with Time Windows. The set D^4 is printed. Each line and dot represent one sink, with the line as time interval and the dot as position.

Set $k \in \mathbb{N}$. Then $\{p(s) | s \in D^k\} = \{1, \ldots, 2^k - 1\}$. As two sinks with different positions have distance at least 1 they cannot be clustered together. Thus we have to open a new cluster for each of the $2^k - 1$ positions. For $1 \leq j \leq 2^k - 1$ set $C_j := \{s \in D^k | p(s) = j\}$. By construction $\underline{rat}(s) \leq p(s) \leq \overline{rat}(s)$ so all sinks in C_j have a point of time in common. That means that C_j and the Steiner tree consisting just of the point $\{j\}$ is a feasible clustering. We conclude that an optimum clustering of I^k has cost $2^k - 1$.
Note that $T^k := \{1, \ldots, 2^k - 1\}$ covers D^k and $\psi(D^k) = |T^k| = 2^k - 1$. The median of T^k is $j = 2^{k-1}$. Therefore the algorithm computes $t'_j = 2^{k-1}$ and the partition $D^k = D_{\text{left}} \dot\cup D_{t'_j} \dot\cup D_{\text{right}}$ with $D_{\text{left}} = D^{k-1}$, $D_{\text{right}} = \hat{D}^{k-1}$ and $D_{t'_j} = L^k \cup \{s^k\}$. Then an algorithm for the Sink Clustering Problem on $D_{t'_j}$ is applied. Note that the sinks in $D_{t'_j}$ have $2^k - 1$ different positions and thus any feasible clustering of these sinks contains at least $2^k - 1$ clusters. The algorithm then continues by partitioning the other two sets.
By induction we conclude that the final solution contains

$$\begin{aligned}
& 2^k - 1 + 2 \cdot (2^{k-1} - 1) + 4 \cdot (2^{k-2} - 1) + \ldots + 2^{k-1} \cdot (2^1 - 1) \\
= \; & k \cdot 2^k - (1 + 2 + 2^2 + \ldots + 2^{k-1}) \\
= \; & (k - 1) \cdot 2^k + 1
\end{aligned}$$

clusters.
Thus the clustering computed by the algorithm has cost $O(k \cdot 2^k) = O(\log_2(\psi(D)) 2^k)$. This is factor $\log_2(\psi(D))$ from the cost of an optimum solution.
In this section we have presented a polynomial $\log_2(\psi(D))$-approximation algorithm for the Sink Clustering Problem with Time Windows. The last observations show that the approximation ratio is tight. It is an open question if there is a polynomial approximation algorithm with constant approximation ratio for the problem.

2.2 Post Optimization

The post optimization algorithm presented in Section 1.2 can also be used, with some small modifications, for the Sink Clustering Problem with Time Windows.
First of all, we modify the neighborhood graph G on D used to find pairs of clusters that are near to each other. We ensure that sinks are connected by an edge only if their required arrival time windows have a point in common. Then, again, two clusters (D_A, S_A) and (D_B, S_B) are neighbors if they contain sinks $a \in D_A$ and $b \in D_B$ with $\{a, b\} \in E(G)$. Note that by this definition there might be clusters that are neighbors but the time windows of all their sinks do no have a point of time in common.
When applying *TwoClusterOpt* we additionally have to check if the two clusters we get by removing an edge of the Steiner tree satisfy the timing constraints, i.e. if all sinks of each cluster have a point of time in common.
Recall that within the main loop of *ChainOpt* we analyze two clusters (D_A, S_A) and (D_B, S_B). We add an edge $\{a, b\}$, $a \in A, b \in B$, in order to get a Steiner tree T on $A \cup B$. Then we remove an edge of $E(T) \setminus E(S_A)$ maximizing the load of the cluster containing S_A. Again we have to ensure that the timing constraints are satisfied. Obviously, this operation can only be successful if the arrival time windows of a and b have a point in common.

2.3 Lower Bounds

In this section we want to develop lower bounds for the cost of optimum solutions of the Sink Clustering Problem with Time Windows. Similar to the approximation algorithm we will describe a framework where lower bounds for the Sink Clustering Problem are used for parts of the initial instance and then are combined to form a lower bound for the complete instance.
The main idea is given by the following observation:

Proposition 2.8. *Let $I = ((V, c), D, d, \tau, u, f)$ be an instance of the* Sink Clustering Problem with Time Windows*, $r \in \mathbb{N}_+$ and $t_1, \ldots, t_{r+1} \in \mathbb{R}$ with $t_1 < t_2 < \ldots < t_{r+1}$. Define $D^i := \{s \in D, t_i < \underline{rat}(s), \overline{rat}(s) \leq t_{i+1}\}$ for $i \in \{1, \ldots, r\}$. Then*

$$O(I) \quad \geq \quad O'(I_{D^1}) + \ldots + O'(I_{D^r}).$$

Proof. First observe that all sets D^i are pairwise disjoint and the time windows of sinks of different sets have an empty intersection. Using Proposition 2.1 and 2.2 we conclude:

$$\begin{aligned} O(I_D) \quad &\geq \quad O(I_{D^1 \cup \ldots \cup D^r}) \\ &= \quad O(I_{D^1}) + \ldots + O(I_{D^r}) \\ &\geq \quad O'(I_{D^1}) + \ldots + O'(I_{D^r}). \end{aligned}$$

□

For $D' \subseteq D$ let $l(I_{D'})$ be a lower bound for the cost of an optimum solution on $I_{D'}$ ignoring the timing constraints, i.e. $l(I_{D'}) \leq O'(I_{D'})$. Now we can find a lower bound for the Sink Clustering Problem with Time Windows using l in the following way:
For $a < b$ we define the set $D[a, b] = \{s \in D, a < \underline{rat}(s), \overline{rat}(s) \leq b\}$.
Choose $r + 1$ points $t_1 < t_2 < \ldots < t_{r+1}$. By the previous proposition

$$\sum_{1 \leq i \leq r} l(I_{D[t_i, t_{i+1}]}) \quad \leq \quad O(I).$$

The problem is to find points t_i that yield a good lower bound. The next lemma will show that we can find the best choice for the t_i's efficiently.

Lemma 2.9. *Let $n = |D|$ and $\theta_{|D'|}$ be the running time to compute $l(I_{D'})$ for $D' \subseteq D$. Then we can compute*

$$\max_{r \in \mathbb{N}, t_1, \ldots, t_{r+1} \in \mathbb{R}, t_1 < \ldots < t_{r+1}} \sum_{1 \leq i \leq r} l(I_{D[t_i, t_{i+1}]}) \tag{2.4}$$

in time $O(n^2 \theta_n)$.

Proof. Let $r' \in \mathbb{N}, t'_1, \dots, t'_{r'+1} \in \mathbb{R}$ and $t'_1 < \dots < t'_{r'+1}$ numbers maximizing (2.4).
Let $t_{-\infty} \in \mathbb{R}$ be a number with $t_{-\infty} < \underline{rat}(s)$ for all $s \in D$. Set $T := \{\underline{rat}(s) | \, s \in D\} \cup \{\overline{rat}(s) | \, s \in D\} \cup \{t_{-\infty}\}$. First we want to show that we can assume without loss of generality that $t'_i \in T$.
For $1 \leq i \leq r+1$ set $A_i := \{\underline{rat}(s) | s \in D, \underline{rat}(s) \leq t'_i\} \cup \{t_{-\infty}\}$ and $B_i := \{\overline{rat}(s) | s \in D, \overline{rat}(s) \leq t'_i\} \cup \{t_{-\infty}\}$. Set $a_i := \max A_i$, $b_i := \max B_i$ and $c_i := \max\{a_i, b_i\}$. Note that $c_i \in T$. By definition $D[t'_i, t'_{i+1}] = D[a_i, t'_{i+1}]$ and $D[t'_i, t'_{i+1}] = D[t'_i, b_{i+1}]$. As $a_i \leq c_i \leq t'_i$ and $b_{i+1} \leq c_{i+1} \leq t'_{i+1}$ we conclude $D[t'_i, t'_{i+1}] = D[c_i, c_{i+1}]$. Therefore we can replace t'_i by c_i for all i and get the same lower bound.
Now we show how to compute the lower bound efficiently. To this end, let $p_0, \dots, p_m$ be the elements of T in increasing order, i.e. $t_{-\infty} = p_0 < p_1 < p_2 < \dots < p_m$ and $\{p_0, \dots p_m\} = T$. For simplicity we set for $J = \{j^1, \dots, j^l\} \subseteq \{0, 1, \dots m\}$, $j^1 < \dots < j^l$,

$$l(J) \; := \; \sum_{1 \leq i \leq l} l\left(I_{D[p_{j^i}, p_{j^{i+1}}]}\right).$$

By the previous observations we have to find a set $J \subseteq \{0, \dots, m\}$ maximizing $l(J)$.
By induction on i we will compute a set $J_i \subseteq \{0, \dots, i\}$ maximizing $l(J)$ subject to $J \subseteq \{0, \dots, i\}$..
Set $J_0 := \{0\}$ and $J_1 := \{0, 1\}$. Assuming we have computed J_i, we will now compute J_{i+1}. To this end, we need the following observations: Let $a_1, \dots, a_k$ be the elements of J_{i+1} in increasing order. If $a_k < i+1$ then $J_{i+1} = J_{a_k}$. But as $l(J_{i+1}) \geq l(J_i) \geq l(J_{a_k})$ we can set w.l.o.g. $J_{i+1} = J_i$ in this case. Otherwise, we conclude

$$\begin{aligned} l(J_{i+1}) &= l(\{a_1, \dots, i+1\}) \\ &= l(\{a_1, \dots, a_{k-1}\}) + l(\{a_{k-1}, i+1\}) \\ &= l\left(J_{a_{k-1}}\right) + l(\{a_{k-1}, i+1\}) \\ &= l\left(J_{a_{k-1}}\right) + l\left(I_{D[p_{a_{k-1}}, p_{i+1}]}\right). \end{aligned}$$

Thus $l(J_{i+1}) = \max_{0 \leq j \leq i} l(J_j) + l(I_{D[p_j, p_{i+1}]})$. If the maximum is obtained for $j = j'$ we get $J_{i+1} = J_{j'} \cup \{i+1\}$. We conclude that we have to compute $i+1$ lower bounds and take the maximum of $i+2$ values in order to compute J_{i+1}.
Finally, J_m is the set we are looking for. Altogether the computations of the $1 + 2 + \dots + m = O(m^2) = O(n^2)$ lower bounds dominates the running time, and the lemma follows. □

2.3.1 Analysis

We have seen that the approximation algorithm for the Sink Clustering Problem with Time windows presented in Section 2.1 can be $\Omega(\log \psi(D))$ away from the cost of an optimum solution. In the analysis of that algorithm we compared the cost of the computed solution with the simple lower bound for the Sink Clustering Problem presented in Section 1.1. It is obvious that the lower bound of Lemma 2.9 is at least as good. Thus it can be at most $O(\log \psi(D))$ away from the cost of an optimum solution.

In this section we will give an example that shows that this is tight, i.e. that the lower bound can be indeed $\Omega(\log\psi(D))$ away from the cost of an optimum solution. To this end, we define an instance for the Sink Clustering Problem with Time Windows within the metric space $(V,c)=(\mathbb{R},l_1)$.
Let $h\in\mathbb{N}$, $k:=2^h$, and $t_i:=i$ for $i\in\{1,\dots,k\}$. For $1\leq i\leq j\leq k$ we define the sinks $s_{i,j}$ with position $p(s_{u,j}):=j-i$, $d(s_{i,j}):=0$, $\underline{rat}(s_{i,j}):=i$ and $\overline{rat}(s_{i,j}):=j$. Let $D=\{s_{i,j}|\,1\leq i\leq j\leq k\}$. Obviously, $\psi(D)=k$. We choose $f:=1$ as facility opening cost and $u:=0.5$ as load limit.
Now we analyze the following sets of sinks (see Figure 2.2):

- $P_1:=\{s_{1,1},s_{2,2},\dots,s_{k,k}\}$,
- $P_2:=\left\{s_{1,2},s_{3,4},\dots,s_{2\left\lfloor\frac{k}{2}\right\rfloor-1,2\left\lfloor\frac{k}{2}\right\rfloor}\right\}$
- $P_3:=\left\{s_{1,3},s_{4,6},\dots,s_{3\left\lfloor\frac{k}{3}\right\rfloor-2,3\left\lfloor\frac{k}{3}\right\rfloor}\right\}$
- $\dots$
- $P_k:=\{s_{1,k}\}$

Any two sinks of the same set P_i have disjoint time windows and cannot be in the same cluster. Moreover, any two sinks of two different sets P_i and P_j cannot be clustered together, as their distance is $|i-j|\geq 1>0.5=u$. Hence any feasible clustering has at least $\sum_{1\leq i\leq k}|P_i|$ clusters. It can be shown that this is exactly the number of clusters of any optimum solution.
The number of elements in P_i is $\left\lfloor\frac{k}{i}\right\rfloor$. In total we get:

$$\begin{aligned}
\sum_{1\leq i\leq k}|P_i| &= \sum_{1\leq i\leq k}\left\lfloor\frac{k}{i}\right\rfloor\\
&\geq \left(\sum_{1\leq i\leq k}\frac{k}{i}\right)-k\\
&= \sum_{2\leq i\leq k}\frac{k}{i}\\
&\geq k\cdot\sum_{2\leq i\leq 2^h}\frac{1}{2^{\lceil\log_2(i)\rceil}}\\
&= k\cdot\left(\sum_{2^0<i\leq 2^1}\frac{1}{2^{\lceil\log_2(i)\rceil}}+\sum_{2^1<i\leq 2^2}\frac{1}{2^{\lceil\log_2(i)\rceil}}+\dots+\sum_{2^{h-1}<i\leq 2^h}\frac{1}{2^{\lceil\log_2(i)\rceil}}\right)\\
&= k\cdot\left(\sum_{2^0<i\leq 2^1}\frac{1}{2^1}+\sum_{2^1<i\leq 2^2}\frac{1}{2^2}+\dots+\sum_{2^{h-1}<i\leq 2^h}\frac{1}{2^h}\right)\\
&= k\cdot\frac{1}{2}\cdot h=\frac{k\log_2 k}{2}.
\end{aligned}$$

(Hoehn and Ridenhour [1989] proved that $\sum_{1\leq i\leq k}\left\lfloor\frac{k}{i}\right\rfloor = k(\log_2 k + 2\gamma - 1) + O(\sqrt{k})$ where γ is is the Euler-Mascheroni number.) We conclude that an optimum solution for this instance costs at least $\Omega(k\log k)$.

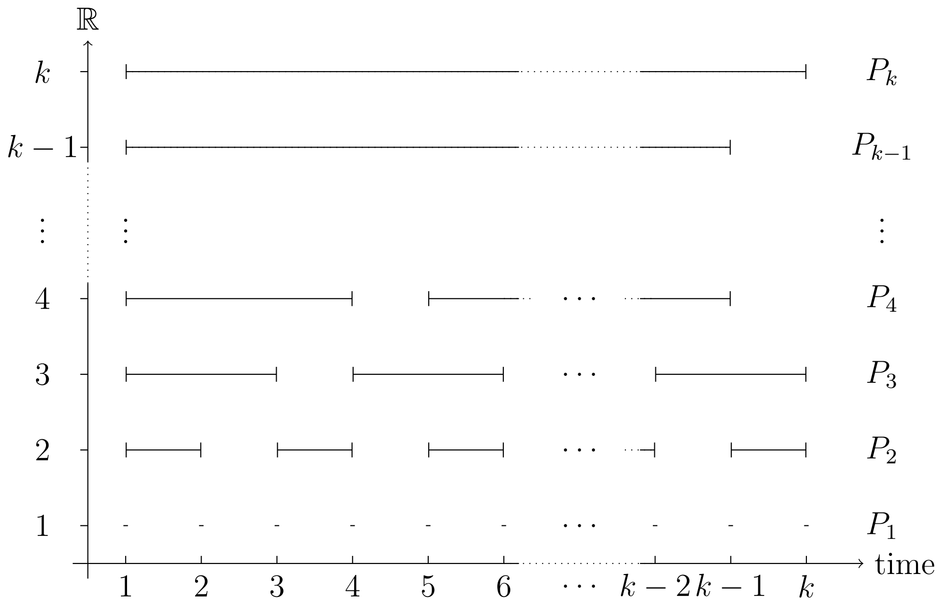

Figure 2.2: Worst case example for lower bound of Lemma 2.9.

In the next step we compute the lower bound given by Lemma 2.9. To this end, we first have a look at $D[a,b]$ for $a,b \in \{0,\ldots,k\}, a<b$. As by definition $D[a,b] = \{s_{i,j} \mid a < i \leq j \leq b\}$ all sinks in $D[a,b]$ have a position in $\{0,\ldots,b-a-1\}$. The lower bound l ignores the time windows and therefore $l(I_{D[a,b]}) = (b-a)\cdot f = b-a$.
Let $J = \{j_0,\ldots,j_r, j_0 < j_1 < \ldots < j_r\} \subseteq \{0,1,2,3,\ldots,k\}$. From what has already been proved, we conclude $l(J) = \sum_{1\leq i\leq r} l(I_{D[j_{i-1},j_i]}) = (j_1-j_0)+(j_2-j_1)+\ldots+(j_r-j_{r-1}) = j_r - j_0 \leq k$. Thus the best lower bound we get by the previous approach for this instance is k, but an optimum solution has cost $O(k\log k) = O(k\log\psi(D))$.

2.4 Experimental Results

The instances summarized in Table 2.1 are again from clock trees of some recent chips. The table shows the number of sinks, the total demand, the facility opening cost, the load limit and the length of a minimum spanning tree on D. Additionally we added the minimum size of a set that covers D (see Section 2.1.2). We applied the Sink Clustering Algorithm with Time Windows using the Sink Clustering Algorithm working on spanning trees as sub-function. By Lemma 2.7 the approximation ratio is $4\lceil\log_2(\psi(D)+1)\rceil$.

instance	$\|D\|$	$d(D)$	f	u	MST	$\|\psi(D)\|$
S1	16 260	58.1	0.127	0.142	76.7	5
S2	33 479	107.9	0.112	0.155	111.9	6
S3	35 305	285.6	0.132	0.167	105.2	329
S4	57 402	118.7	0.093	0.087	93.1	4
S5	71 356	211.8	0.122	0.196	86.0	234
S6	119 461	258.8	0.142	0.117	314.5	8
S7	141 503	436.5	0.127	0.164	239.2	132
S8	638 129	1 337.8	0.093	0.087	1 094.7	5

Table 2.1: The test instances.

Table 2.2 shows the results we got on the instances after applying the Sink Clustering Algorithm with Time Windows and after additional use of the two post optimization algorithms presented in Section 1.2. The table contains the total cost, the wiring cost and the number of clusters of the computed clustering. The post optimization reduced the total cost of the solution by 9.3% in average.

	core Sink Clustering alg.			... plus post opt.		
instance	total cost	wire cost	#clusters	total cost	wire cost	#clusters
S1	286.7	77.8	1 645	249.5	67.1	1 436
S2	310.4	113.9	1 755	277.6	100.6	1 581
S3	1 017.9	249.8	5 819	893.6	194.3	5 298
S4	359.6	97.9	2 814	333.6	90.8	2 611
S5	1 478.1	649.3	6 793	1 370.9	573.0	6 540
S6	1 072.6	297.1	5 461	1 001.7	284.5	5 051
S7	1 604.4	431.3	9 237	1 547.9	405.7	8 994
S8	4 205.4	1 149.1	32 863	3 917.5	1 073.4	30 582

Table 2.2: Total cost of the solution, wiring cost and number of clusters after the initial Sink Clustering Algorithm with Time Windows and after additional post optimization.

Table 2.3 shows the computed lower bounds. The first one is the lower bound based on K-dominated functions without considering time windows. The second is the one presented in the last section using the lower bound for K-dominated functions as subfunction. As expected, the cost of our solution is far away from the lower bound in some cases. The reason can be both a bad lower bounds or a bad solution.

	lower bound 1			lower bound 2		
instance	cost	#clusters	ratio	cost	#clusters	ratio
S1	134.2	731	1.859	147.4	778	1.693
S2	187.1	1 115	1.484	192.8	1 135	1.440
S3	334.8	2 082	2.669	551.0	3 175	1.622
S4	246.7	2 028	1.353	260.2	2 105	1.282
S5	222.2	1 366	6.169	598.4	2 387	2.291
S6	746.5	3 886	1.342	750.2	3 901	1.335
S7	599.6	3 555	2.582	1 131.7	6 411	1.368
S8	2 833.2	23 145	1.383	3 034.1	24 285	1.291
average			2.355			1.540

Table 2.3: Comparison to lower bounds. The table shows the lower bound for the cost (cost) , the lower for the number of clusters (#clusters) and the ration of the cost of our solution and the lower bound (ratio).

Finally, Table 2.4 shows the running time of the core clustering algorithm and the two post optimization heuristics and the number of successful applications of TwoClusterOpt and ChainOpt.

	initial	TwoClusterOpt		ChainOpt	
instance	runtime (s)	runtime (s)	succ.	runtime (s)	succ.
S1	1.1	7.6	1 876	4.6	702
S2	3.9	24,7	2 643	9.7	781
S3	2.4	11.3	3 877	21.0	2 812
S4	2.6	26.5	3 377	10.9	716
S5	3.8	47.0	6 402	92,7	1 150
S6	14.4	42.4	6 177	14.0	1 355
S7	10.4	33.7	3 595	40.2	1 580
S8	37.2	297.7	38 900	188.3	5 856

Table 2.4: Running times in seconds and number of successful applications of *TwoClusterOpt* and *ChainOpt*.

3 BonnClock

In this chapter we will give an introduction to clock tree design and present our algorithm BonnClock developed to build such trees. A clock tree is an electrical network with the task of distributing a clock signal from a source to several sinks placed on a chip. A given initial clock tree has to be replaced by a logically equivalent tree that satisfies some timing and several other constraints. To this end, inverters are inserted and special circuits (circuits that are neither buffers nor inverters) have to be cloned. Moreover, all circuits have to be placed. The clock signal, starting at the root, has to arrive at each sink within an individual required arrival time window.
We will describe the Clock Tree Construction Problem more formally in the next section. For the sake of simplicity, we will restrict our description to trees that do not contain other circuits than inverters. However, the algorithm can be extended in a straightforward way to handle more complex tree structures.

3.1 The Clock Tree Construction Problem

3.1.1 Problem Definition

Clock Tree Construction Problem

Instance: An instance consists of:

- A rectangular chip area $\mathcal{A} \subset \mathbb{R}^2$,
- a source r with timing rules and a location $p_r \in \mathcal{A}$, and input slew sl_r,
- a finite set S of sinks. For each sink s a parity in $\{+,-\}$, a location $p_s \in \mathcal{A}$ and a required arrival time windows $rat(s) = \left[\underline{rat}(s), \overline{rat}(s)\right] \subset \mathbb{R}$,
- a library $\mathcal{L}$ of inverters with timing rules and input capacitances,
- a set of rectilinear axis-parallel blockages $\mathcal{B}$ where circuits cannot be placed,
- a slew limit $maxslew$ and
- additional technology dependent parameters (wire resistance, wire capacitances per unit length, ...)

The task is to compute an arborescence T with root r and leaves S, a mapping $bhc : V(T) \setminus (S \cup \{r\}) \to \mathcal{L}$ from the internal vertices to the inverter library, a position $\pi(v) \in \mathcal{A}$ for each vertex $v \in V(T)$, a rectilinear Steiner tree S_v for each vertex

$v \in V(T) \setminus S$ with terminals $\{\pi(v)\} \cup \{\pi(w) | (v, w) \in E(T)\}$, a starting time at_r of the signal and a number $\delta \in \mathbb{R}_+$ indicating how well the timing requirements are met, so that

- $\pi(s) = p_s$ for $s \in S \cup \{r\}$ and no internal vertex of T is placed on a blockage, i.e. $\pi(v) \notin B$ for all $B \in \mathcal{B}$ and all $v \in V(G) \setminus (S \cup \{r\})$.
- For any sink $s \in S$ the number of internal vertices on the r-s path is even if the parity of s is $+$ and odd if it is $-$.
- The slew limit is kept at all input and output pins.
- The signal, starting at time at_r and slew sl_r at the source, arrives at sink $s \in S$ within the time interval $[\underline{rat}(s) - \delta, \overline{rat}(s) + \delta]$.

The main goal is to build a tree that satisfies the timing requirements as good as possible, i.e. to minimize δ. Further important goals are to minimize the resource consumption. This means on the one hand to minimize the capacitance (proportional to the power consumption) of the tree $\sum_{v \in V(T) \setminus S} c(S_v)$ where $c(S_v)$ is the total capacitance of the Steiner tree S_v. On the other hand we have to minimize the power consumption of the inserted circuits $\sum_{v \in V(T) \setminus (S \cup \{r\})} c(bhc(v))$.

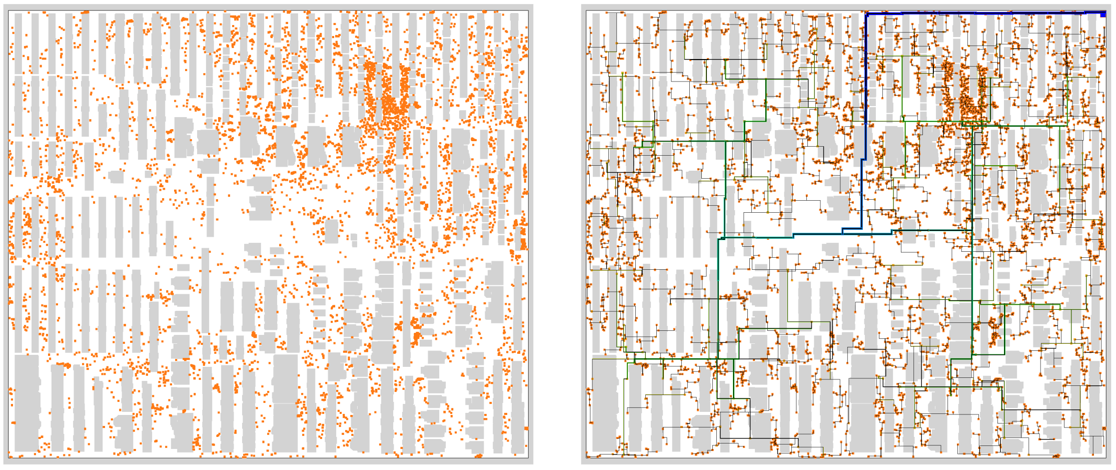

Figure 3.1: A clock tree instance and a complete clock tree. The sinks are in orange, the source is placed in the upper right corner. The color and thickness of the wires in the right picture depend on the arrival time of the clock signal (blue and thick is early, red and thin is late).

One variant of the Clock Tree Construction Problem handles several sources instead of a single one. Then the algorithm has to compute an assignment from the sinks to the sources and build a clock tree for each source on the assigned sinks. Moreover, in practice there appear more complicated clock tree structures like clock trees running into each other. Sometimes there are actually ‘clock trees’ that are not trees but contain circuits that receive the clock signal from several other circuits.

3.1.2 Previous Work

The literature on clock tree design distinguishes between three different parts of the problem. First the creation of the tree topology, second the embedding of a given topology into the plane and third the insertion of inverters. Some papers deal with only one of these aspects, some combine two or all three parts.

One classical tree construction algorithm that guarantees the same length of all source-sink paths are the H-trees. These trees are build top-down. First a vertex is placed in the center of the chip area and connected to the source (see Figure 3.2). Then the area is divided into two halves. In the center of both halves a new vertex is placed and connected to the one in the center of the area. This procedure will be repeated until all sinks are near enough to a leaf of the tree so that they can be connected directly to them. A big disadvantage of this approach is that the sinks have to be distributed symmetrically over the chip area which is not the usual case in practice. Moreover, blockages cannot be considered and only zero-skew trees can be implemented.

The top-down partition algorithm that will be presented in Section 3.6 can be seen as a generalization of the H-tree method. Unlike the classical H-tree algorithm, it can handle arrival time windows and blockages.

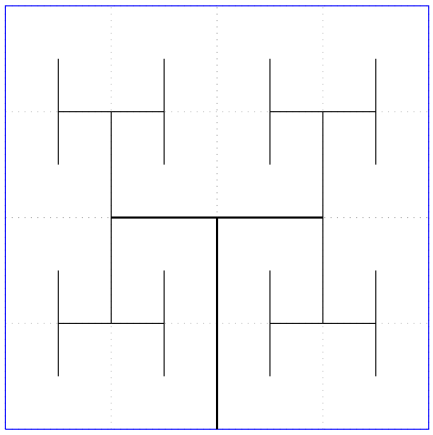

Figure 3.2: An H-tree

One widely used algorithm for embedding a given topology T into the plane is the Deferred-Merge-Embedding (DME) (Chao et al. [1992], similar approaches proposed by Edahiro [1991], Muuss [1994], Tsay [1993]). In the final embedding all source-sink paths have the same length. Without loss of generality let T be a strict binary tree.

For every vertex $v \in V(T)$ of the tree the DME-algorithm stores a merging region $MR(v)$ that consists of a line segment (that could also be a single point) and a value $len(v)$ that is the length of the path from v to any sink of the subtree rooted at v in the final embedding. The merging region of a sink v is its position and the length is $len(v) := 0$. Let r and l be the children of a vertex $v \in V(T)$ where we have already computed the merging regions and lengths. Now let $len(v)$ be the smallest value so that there exists a point x with $len(v) \geq \max\{d(x, MR(r)) + len(r), d(x, MR(l)) + len(l)\}$. Here $d(x, MR(l))$ is the (minimum) distance from x

to the line segment $MR(l)$ which can be computed efficiently. Set $MR(v) := \{x \in \mathbb{R}^2 | \max\{d(x, MR(r)) + len(r), d(x, MR(l)) + len(l)\} \leq len(v)\}$.

Finally, in a top-down process the positions of the vertices are computed. First an arbitrary position $x \in MR(r)$ for the source r will be selected. Then the algorithm chooses for every vertex $v \in V(T)$ a position in the merging region $MR(v)$ of v that is nearest to the position of its predecessor.

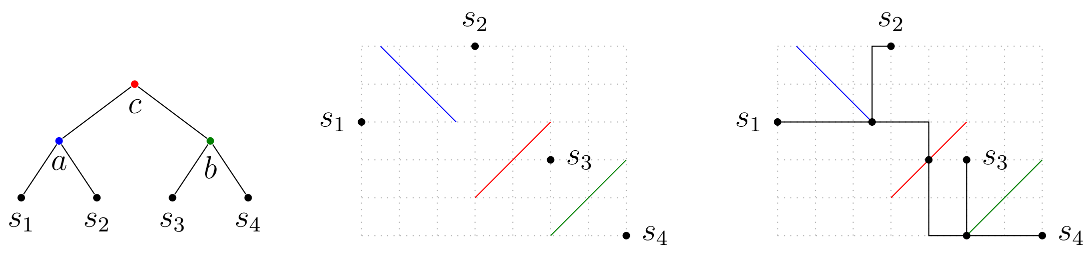

Figure 3.3: Illustration of the Deferred-Merge Embedding algorithm. The left side shows the initial topology, the middle the merging regions and the right side the final vertex positions.

Chao et al. proved that the Deferred-Merge-Embedding algorithm computes a zero skew tree that has minimum length for the given topology.

The length of the final tree depends on the topology of the tree. This topology can be precomputed or it can be chosen within the DME algorithm. In the latter case often a simple greedy strategy is used: in each iteration of the algorithm two vertices with minimum distance between their merging regions are taken and get a new predecessor (see e.g. Edahiro [1992]).

There are many variants of the DME algorithm. One is to allow that the lengths of the source-sink paths need not be the same but might differ by at most some value $b \in \mathbb{R}_{\geq 0}$, or to allow more general skew constraints (Tsao and Koh [2002]). In our definition of the clock tree problem this means that the required arrival time windows are not identical points but intervals of equal or individual length. In both cases the merging regions become octagons instead of line segments.

Actually, we are interested in the delay between the source and the sinks and not the lengths of the source-sink paths. Chong et al. [1998] extended the DME algorithm in order to use the Elmore delay model. In this case, however, the merging regions get even more complicated and can only be approximated by simpler geometrical structures. Moreover, it cannot be guaranteed anymore that the computed embedding is best possible for the given topology.

A problem of all algorithms that are based on the Deferred-Merge Embedding concept is that long wire detours are necessary in order to balance the tree. This leads to an extensive use of wiring resources and a high power consumption.

3.2 Outline of BonnClock

The algorithm BonnClock constructs clock trees as defined in the last section. It builds the tree bottom-up, but it also uses a top-down partitioning algorithm in order to compute roughly the topology of the tree. It does not use balanced nets. Rather all nets will be as short as possible. Delays are balanced by the constructed tree topology and accurate inverter sizing.

For each circuit a placement area is computed where the vertex can be placed. In addition, a set of 'solution candidates' is stored to help us with the selection of good inverter sizes. More details on solution candidates are presented in Section 3.3.

The topology creation and the placement of the internal vertices are done simultaneously. The size of the circuits is not chosen until the complete tree has been computed. While building the tree we maintain a list of 'solution candidates' for each vertex that is computed by a dynamic program.

An outline of the outer loop of the BonnClock Algorithm is shown in Algorithm 1. We always keep a set of active vertices, i.e. vertices that have no predecessor yet. First, these vertices are preliminarily clustered together. We are most interested in 'late' clusters. To determine them let $t := \max_{v \text{ active}} \underline{rat}(v)$ be the latest starting point of the required arrival time window of an active vertex. Denote by $A_{late} := \{v \text{ active} | t \leq \overline{rat}(v)\}$ all active vertices that have a late feasible arrival time and denote by C the set of all clusters only containing vertices of A_{late} (note that C is not empty). Let A be the set of all vertices that are in a cluster of C. The final positions of the vertices in A are chosen and new solution candidates are created. After dismissing the preliminary clusters, the vertices in A will be clustered again, using their new positions. For each cluster we insert a new vertex having the vertices of the cluster as successors. Then the placement areas and preliminary solution candidates for the new vertices are computed. This procedure has to be repeated until the remaining circuits can be connected to the source of the tree. After the main loop, the algorithm chooses a good solution candidate for the source and sets the circuit sizes according to it. In this simplified version of the algorithm we have not considered the parities of the sinks. The algorithm can be extended to respect them too by maintaining two sets of active vertices, one for the vertices that require an even and one for the vertices that require an odd number of vertices on the path towards the source. Only vertices of the same set are clustered together and new vertices are sorted into the correct set.

Within the algorithm we will use two values, *maxdist* and d_{wire}. *maxdist* denotes the maximum distance between two inverters so that the slew limit is kept and the capacitance constraints are satisfied. d_{wire} is the best possible delay per unit of wire that can be achieved without violating the constraints (this value is computed in a preprocessing step). Moreover, we only use a finite set of feasible slew values in order to handle the dynamic program and the computation of solution candidates (see Section 3.3).

In the next sections we will dicuss the different main parts of the algorithm. Section 3.3 will show more precisely how the dynamic program is used to compute solution candidates. How the SINK CLUSTERING ALGORITHMS are used for the clustering is described in Section 3.4. Section 3.5 shows how to compute the placement areas and

Algorithm 1: BonnClock main loop.

```
Active vertices $V_{act} := S$;
while $V_{act}$ cannot be connected to root do
    // Pre-Clustering
    Cluster active vertices;
    Let $A$ be all active vertices with latest arrival time windows;
    // Placement
    Place vertices of $A$ within their placement area;
    Recompute solution candidates for vertices $A$;
    // Clustering
    Recluster $A$ and let $B$ the new vertices driving $A$;
    $V_{act} := (V_{act} \setminus A) \cup B$;
    // Dynamic Program Update
    Compute solution candidates and placement areas for vertices $B$;
end
```

in Section 3.6 the top-down partition is explained. Finally, in Section 3.7 we will give some ideas how the problem of on-chip variation can be handled.

3.3 Solution Candidates

A solution candidate for a vertex v of the clock tree describes a feasible assignment of the vertices in the subtree rooted at v to the inverters. But before we define solution candidates more formally and show how they can be computed we will have a closer look at the delay and slew functions of the circuits.

For the computation of the solution candidates we have to discretize the slew values in some situations. To this end, let Sl be a finite set of feasible slew values. In practice we set $Sl = \left\{\frac{i}{k} maxslew | \, i \in \{0, \dots, k\}\right\}$ for $k = 20$.

For the rest of this section a circuit can be either the source, a sink, an internal inverter or a special gating circuit.

We associate with each vertex of the clock tree a set of solution candidates. A solution candidate sc for a vertex c (or more precisely the input pin of a circuit) consists of a circuit type T, a capacitance cap, a slew $slew$, an arrival time window $rat = [\underline{rat}, \overline{rat}]$ and a solution candidate $succ(w)$ for each successor w of c. We will write the solution candidate as tuple $sc = (T(sc), cap(sc), slew(sc), rat(sc), succ(sc))$.

A solution candidate can be interpreted in the following way. Assume we realize the subtree rooted at a vertex v according to a solution candidate sc of v, i.e. we set the inverter types according to the types given by the candidate and its successors. If the clock signal arrives at the input pin of circuit v with slew $slew(sc)$ and within the time window $rat(sc)$, then the signal arrives at each sink of the subtree within its time window. In the next section we will extend the definition of solution candidates to vertices that do not correspond to input pins of circuits but to output pins of circuits

or points on the Steiner trees connecting the circuits, and thus do not have a circuit type. In these cases we set $T(sc) := NULL$.
Now we show how to compute the set of solution candidates $SolCand(v)$ for a vertex $v \in V(T)$ of the clock tree. Obviously, the solution candidates of the successors $\delta^+(v) = \{v_1, \ldots, v_k\}$ of v have to be computed already (Figure 3.4 shows an example with three successors). We compute an (approximate) minimum Steiner tree on $\{v\} \cup \delta^+(v)$ and transform it into a tree, defined in the following. To this end, we double the vertex v and insert a 'circuit' edge between them (red in Figure 3.4). These two vertices represent the input and output pin of the circuit v. The first one will be the root of the tree. We also replace each Steiner point of degree at least 3 by a vertex for every incident edge. We insert edges between the 'merging' vertex (blue in Figure 3.4) belonging to the edge leading to v and the other ones. Thus we get a directed tree T' where for each edge of the Steiner tree there exists a corresponding 'timing' edge (black in Figure 3.4). Note that the first vertices of these edges all have degree 2. We will call these vertices 'wire' vertices. Altogether, there are four types of vertices: the sinks (white circles in Figure 3.4), the wire vertices (green circles), the merging vertices (blue circles), and the root (black square).
Propagating a signal from the root of the timing tree to the sinks, i.e. computing its arrival times and slews, can be done easily using the delay and slew functions for the circuit at the root and the Elmore delay model for the wiring (recall that the Elmore-delay on a piece of wire of capacitance c and resistance r driving a downstream capacitance $dcap$ located at the end of the wire is $\delta \cdot r(\frac{c}{2} + dcap)$ where δ is some constant.). But our task is to do a backward propagation of the solution candidates and combining them to feasible new ones at the root.

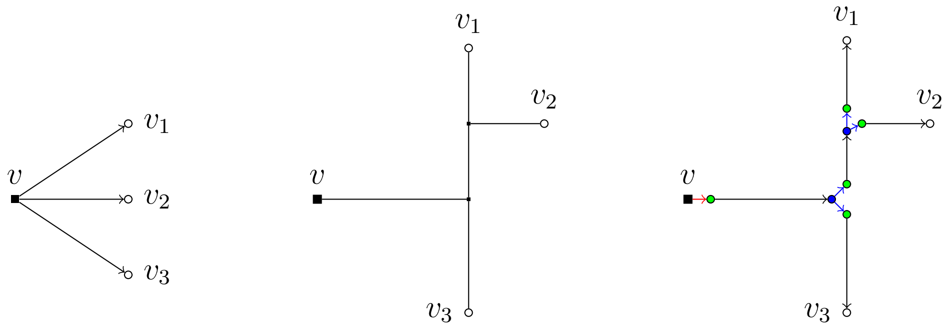

Figure 3.4: Example showing a part of the clock tree with four vertices and the corresponding Steiner tree. The right picture shows the resulting timing tree.

Next we show how to compute a set of solution candidates for each vertex.

Solution Candidates of a Wire Vertex

A wire vertex w has exactly one successor w'. Let $cap(w, w')$ and $res(w, w')$ be the capacitance and resistance of the corresponding wire segment and sc' be a solution candidate of w'. Assume we want to realize sc'. Then the signal has to arrive at w'

within the time window $rat(w')$. It will be delayed by some delay del and the slew increase by some value sd. But because we know the downstream capacitance $dcap(w')$ at w' they can be computed easily using the Elmore model. Thus we get a new solution candidate

$$sc := (NULL, cap(sc') + cap(w, w'), slew(sc') - sd, [\underline{rat}(sc') - del, \overline{rat}(sc') - del], (sc'))$$

for w with $del := res(w, w')\left(\frac{1}{2}cap(w, w') + cap(sc')\right)$. Note that $|SolCand(w)| = |SolCand(w')|$ and $SolCand(w)$ can be computed in linear time.

Solution Candidates of a Merging Vertex

A merging vertex w has at least two successors $w_1, \ldots, w_k$. Let $sc_1 \in SolCand(w_1), \ldots, sc_k \in SolCand(w_k)$ be solution candidates for the successors. In order to combine these candidates to a new one of w we have to choose ones with the same input slew $slew(sc_1) = \ldots = slew(sc_k)$. To this end, we discretize all slew values by rounding them to the nearest value in Sl. We get a new solution candidate

$$sc := \left(NULL, \sum_{i=1}^{k} cap(sc_i), slew(sc_1), rat(sc), (sc_1, \ldots, sc_k)\right)$$

with $\underline{rat}(sc) := \max_{i=1}^{k} \underline{rat}(sc_i)$ and $\overline{rat}(sc) := \min_{i=1}^{k} \overline{rat}(sc_i)$.
Combining all permutations of solution candidates would take too long and could produce many candidates with unusable require arrival time windows. So we will restrict ourselves to 'good' solution candidates that are not 'dominated' by others. We call a solution sc_a dominated by sc_b if $slew(sc_a) = slew(sc_b)$ and $rat(sc_a) \subset rat(sc_b)$. In respect of timing, it will be better to use sc_b instead of sc_a as the former one has a bigger required arrival time window. Note that this is not always true, as these candidates can have different capacitances that influence the delays and slews of the wires and circuits driving these vertices. Moreover, dominated solution candidates might result in trees with a lower power consumption. In order to limit the number of solution candidates and to speed up their computation we restrict ourselves to dominated ones. This limitation seems to have no big influence on the quality of the final trees in practice. This has to be verified for each new technology as this might change. We remove all solution candidates that are dominated by others and store the solution candidates $SolCand(v)$ of each vertex v in ordered lists, one for each slew. That means, we have for any slew $sl \in Sl$ a list $SolCand_{sl}(v) := (sc_1, \ldots, sc_k)$ with $slew(sc_i) = sl$ and $\underline{rat}(sc_j) < \underline{rat}(sc_{j+1})$ for $1 \leq i \leq k$ and $1 \leq j \leq k-1$. As no candidate is dominated by another one we conclude $\overline{rat}(sc_j) < \overline{rat}(sc_{j+1})$ for $1 \leq i \leq k$ and $1 \leq j \leq k-1$.
Now it is easy to compute the solution candidates for the merging vertex w. The left boundary $t := \underline{rat}(sc)$ of the time interval of a new solution candidate sc is equal to the left boundary of the time window of one of the successor candidates of w. We want to maximize $\overline{rat}(sc)$, otherwise we could construct a candidate that dominates sc. To this end, we choose for each successor w_i the unique candidate sc_i with $\underline{rat}(sc_i) \leq t$ and $\overline{rat}(sc_i)$ maximum. This computation can be performed by a simple sweep-line algorithm (see Algorithm 2), that computes for any slew $sl \in Sl$ the list $SolCand_{sl}(w)$.

Algorithm 2: Sweep-line method to compute the solution candidates.

```
Input: Vertex $w$ with successors $w_1, \dots, w_k$, slew $sl \in Sl$ and solution candidates
       $SolCand_{sl}(w_i) = (sc^i_1, \dots, sc^i_{|SolCand_{sl}(w_i)|})$ for $i \in \{1, \dots, k\}$ with
       $slew(sc^i_j) = sl$.
Result: Solution candidates $SolCand_{sl}(w)$ for vertex $w$.
$p_i := 1$ for $i \in \{1, \dots, k\}$;
$t := \max \underline{rat}(sc^i_{p_i})$;
while $t \neq \infty$ do
    for $i \in \{1, \dots, k\}$ do
        while $p_i < |SolCand_{sl}(w_i)|$ and $\underline{rat}(sc^i_{p_i+1}) \leq t$ do
            $p_i := p_i + 1$;
        end
        Create new solution candidate $sc$ with successors $sc^i_{p_i}$ for $i \in \{1, \dots, k\}$;
        $SolCand_{sl}(w) := SolCand_{sl}(w) \cup \{sc\}$;
        $t := \min \left( \left\{ \underline{rat}(sc^i_{p_i+1}) \middle| p_i < |SolCand_{sl}(w_i)|, 1 \leq i \leq k \right\} \cup \{\infty\} \right)$;
    end
end
```

We immediately conclude that $|SolCand(w)| \leq \sum_{i=1}^{k} |SolCand(w_i)| - k + 1$ and that the computation is dominated by sorting the solution sets.

Solution Candidates of the Root Vertex

Finally, we have to compute the solution candidates for the root vertex v. Assume we have the solution candidates for its successor w. v and w represent the input and output pins of a circuit. In order to get a finite set of solution candidates we discretize all slew values by setting them to the nearest value in Sl. For any solution candidate $sc' \in SolCand(w)$ and any inverter type $T \in \mathcal{L}$ we compute the set of input slews $InSlews := \{sl' \in Sl |\, sl_T(cap(sc'), sl') = slew(sc')\}$ that produce an output slew of $slew(sc')$ when driving a capacitance of $cap(sc')$. For any slew $sl \in InSlews$ we get a new solution candidate

$$sc := \big(T, incap(T), sl, rat(sc) = [\underline{rat}(sc), \overline{rat}(sc)], (sc')\big)$$

for v with

$$\underline{rat}(sc) := \underline{rat}(sc') - \mathrm{del}_T(cap(sc'), sl)$$

and

$$\overline{rat}(sc) := \overline{rat}(sc') - \mathrm{del}_T(cap(sc'), sl)$$

and where $\mathrm{del}_T(cap(sc'), sl)$ is the delay of a circuit of type T when driving a capacitance $cap(sc')$ and an input slew sl.

$InSlews \subseteq Sl$ can be empty, contain one or more elements, depending on the slew function of the circuit. We conclude that $|SolCand(v)| \leq |\mathcal{L}||Sl||SolCand(w)|$.

Remarks

Altogether we get at most

$$|SolCand(v)| \leq |\mathcal{L}||Sl| \cdot \sum_{w \in \delta^+(v)} |SolCand(w)|$$

solution candidates for a vertex $v \in V(T)$.
Within the BonnClock algorithm we first compute preliminary solution candidates for the newly created active vertices. At that stage the vertices have no final positions but placement areas where they can be placed. In order to use the previously described method we have to choose for every vertex a preliminary position within its placement area. Later in the algorithm, when the locations of the vertices have been fixed, the final solution candidates are computed.
After computing the solution candidates for a vertex $v \in V(T)$ we dismiss the timing tree rooted at v and the solution candidates of the internal vertices (i.e. the wire and merging vertices) as they are not required anymore.
For the clustering we need a time window for each active vertex. To this end, we take the bounding box of the time windows of the solution candidates of the vertex.

3.4 Clustering

An important part of the clock tree construction, if not the most important, is the clustering. It computes which circuits have to be driven by the same driver circuit. More precisely, the input of the clustering is a set of circuits (vertices) in the plane with input capacitances that have to be driven by new inserted circuits. All circuits that are in the same cluster are connected by a rectilinear Steiner tree and the capacitance of this Steiner tree (proportional to its length) plus the input capacitances has to be at most the load limit of the drivers. Moreover, the required arrival time windows of the circuits in the same cluster must have a point of time in common. The goal is to find a clustering that minimizes the power consumption.
The Sink Clustering Problem and the Sink Clustering Problem with Time Windows introduced in Chapters 1 and 2 are good mathematical models for the clustering within clock trees. The active vertices are the sinks. As position of a vertex we take a point of the placement area. If the vertex is a leaf of the tree this point is unique. Otherwise, we take a point that is nearest to the center of the placement area. The demand of a vertex is the input capacitance of the corresponding input pin. Each cluster will be driven by a special circuit or an inverter. As load limit we use the load limit of the strongest driver type and its power consumption as facility cost. The arrival time windows of the vertices are directly taken as time windows for the sinks. Often all time windows have a point of time in common so that the Sink Clustering Algorithm can be applied.
80% to 90% of the total power consumption of a clock tree arise in the last stage (wires and driver circuits). Moreover, recent designs can contain up to 1 000 000 sinks.

We have shown in the experimental results in Section 1.4 and 2.4 that the proposed clustering algorithms are extremely fast and compute almost power optimal solutions. Our clustering algorithms can be modified easily to consider additional constraints. For example we can limit the maximum distance between two sinks within a cluster or the total number of sinks, both possibilities to reduce the wire delays. Note that we cannot prove approximation guarantees for the problem with such additional constraints but in practice the results are still not far from the lower bounds.
Nevertheless, there are some limitations of the clustering algorithm. The Sink Clustering Problem does not compute the positions of the driver circuits. Instead of this, we compute for each new vertex v a placement area $Pl(v)$ where v can be placed without violating the constraints (see Section 3.5). As the clustering does not consider blockages and the position of the source, the placement area might be empty. In this case we choose a position for v which minimizes any violations.
Moreover, the algorithm requires sink positions. When we perform the pre-clustering of vertices that are not sinks, no final positions have been selected for them. Thus we have to select appropriate positions for these vertices within their placement areas in order to run the clustering.
Finally, the clustering algorithm ignores the wire delays from the driver to the sinks within a clusters. Even if the required arrival time windows of the sinks of a cluster have a point of time in common, different wire delays from the driver to the sinks might lead to an infeasible result. That means there can be sinks where the clock signal does not arrive within the time window.
It is an interesting open problem whether there are approximation algorithms for the variant of the Sink Clustering Problem where driver positions have to be computed and the wire delays from the drivers to the sinks are considered.

3.5 Placement Areas

For every vertex v we store a feasible placement area $Pl(v)$ where v can be placed and a feasible placement area $Pl_{pred}(v)$ where the predecessor of v can be placed without violating any of the constraints. The positions of the sinks and the root are fixed, thus their placement areas consist of a single point. In all other cases the placement area $Pl(v)$ of a vertex v is the intersection of the predecessor placement areas of the successors of v, i.e. $Pl(v) := \bigcap_{w \in \delta^+(v)} Pl_{pred}(w)$. The placement area $Pl_{pred}(v)$, where the predecessor of a vertex v can be placed, has to satisfy the following three conditions:

a) Each point has distance at most $maxdist$ from a feasible position of v,

b) the points are not on a blockage and

c) they are not too far away from the root.

The area that satisfies a) and b) is easy to compute. In order to satisfy c) we must know for each point of the placement area the expected delay of the clock signal from the source to that point in a clock tree that satisfies all constraints. For this, we will introduce a blockage grid that approximates the blockages and a distance graph that will give us the estimated delays from the source to any point.

3.5.1 Blockage Grid and Distance Graph

In order to find a shortest path from the root and to estimate the delay from the root to any point of the chip area while considering the blockages, we use a blockage grid. First we partition the chip area by horizontal and vertical lines that are given by the edges of significant large blockages (similar to the Hanan grid, Hanan [1966]). Additionally, we insert lines so that the distance between two consecutive lines is at most d for some parameter $d \in \mathbb{R}_{>0}$. This partition of the area should be fine enough in order to get a good approximation of the blockages and delays but also coarse enough in order to get small running times. In practice we set the maximum width d to a fraction of $maxdist$. The lines partition the chip area into tiles. A tile is called blocked if it is completely covered by blockages, otherwise it is free.

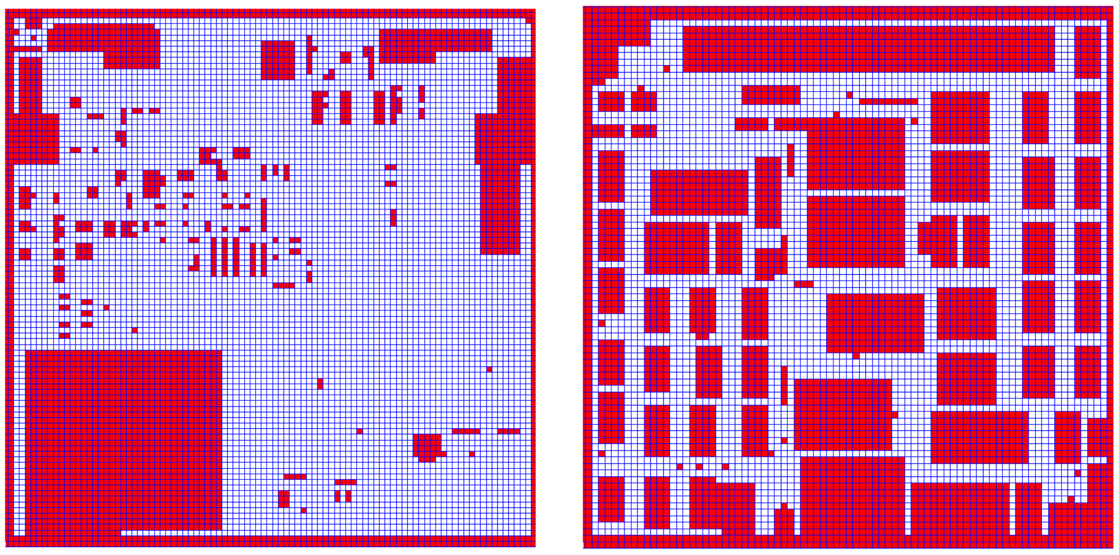
Figure 3.5: Blockage maps of different chips.

For a given source tile s, which in the majority of cases is free but could also be blocked, we compute a shortest distance tree T_s on the tiles that gives us for every tile t an approximate delay from s to t. All blocked tiles, except for the source tile if it is blocked, are leaves of T_s, so on every s-t path in T_s all internal tiles are free.
We define a directed graph G on the tiles. Denote by B the blocked and by U the unblocked tiles. Then set $V(G) := B \cup U$ and $E(G) = \{(v, w) | \, v \in U \cup \{s\} \, w \in B \cup V\}$, i.e. we can reach each tile from an unblocked tile or the source and there is no edge leaving a blocked tile if it is not the source. We will identify the tiles with their midpoints. So the distance between two tiles is the distance between their midpoints. The cost of an edge in $E(G)$ is d_{wire} times its length if the length is at most $maxdist$. Recall that d_{wire} is the best possible delay per unit of distance that can be achieved.

For longer distances we extend this linear cost function to a quadratic function in order to model higher delays if the distance between two inverters is bigger than the optimal distance. Sometimes we cannot place inverters in optimal distances as big blockages might separate sinks from the source. Moreover, it might be better to allow minor distance and slew violations instead of long detours from the source to a sink.
Using this cost function we can compute the shortest path tree T_s in G. This can be done efficiently by a variant of Dijkstra's algorithm (Dijkstra [1959]) without computing $E(G)$ explicitly. The tree gives us for every point $p \in \mathcal{A}$ of the chip area the approximate delay from the source to p.
The grid and the distance graphs will also be used for the top-down partitioning that is described in Section 3.6.

3.5.2 Placement Area Computation

Now we compute the placement area $Pl_{pred}(v)$ where we can place the predecessor of a vertex $v \in V(T)$ without violating any constraints (see Figure 3.6 a)-e)). Initially, we have given $Pl(v)$ (Figure 3.6 a) in blue). We compute the area of all points that are at most *maxdist* from a point in $Pl(v)$ (red dotted area in 3.6 b)). This is also called the l_1-expansion by *maxdist*. Then we remove all blocked areas (gray in 3.6 c)) and the area that lies too far away from the root (green in 3.6 d)). In order to get this area we will estimate a latest starting time of the signal at the root so that all sinks can be reached within their time windows. Using this starting time and the estimated delay from the root to a tile given by the distance graph we can compute the estimated arrival time of the signal at any tile. Then we remove all tiles from the placement area which are reached by the signal later than the required arrival time window. Figure 3.6 e) finally shows $Pl_{pred}(v)$.
The placement areas are represented as unions of overlap-free octagons. The boundaries of the octagons are parallel to the x- or y-axis or the axes rotated by 45 degrees. The rotated boundaries arise by the l_1-expansion of the placement areas and the boundaries parallel to the x- and y-axis by removing the rectilinear blockages and tiles (see Figure 3.6).
The following operations are required in order to compute $Pl_{pred}(v)$ for a vertex v: Computing the intersection of two placement areas and computing the set of points that have a distance of at most *maxdist* from a placement area. Gester [2009] showed how these operations can be implemented efficiently in $O(n \log n)$ time, where n is the number of octagons. Note that removing the blockages and the tiles that are too far away from the source can be implemented as intersection with the complement of these areas.

3.6 Blockage Aware Top-Down Partitioning

Up to now BonnClock builds the clock tree bottom-up and the topology of the tree is determined only by the clustering. In this section we present a top-down partition algorithm that determines the topology of the upper part of the tree. The algorithm

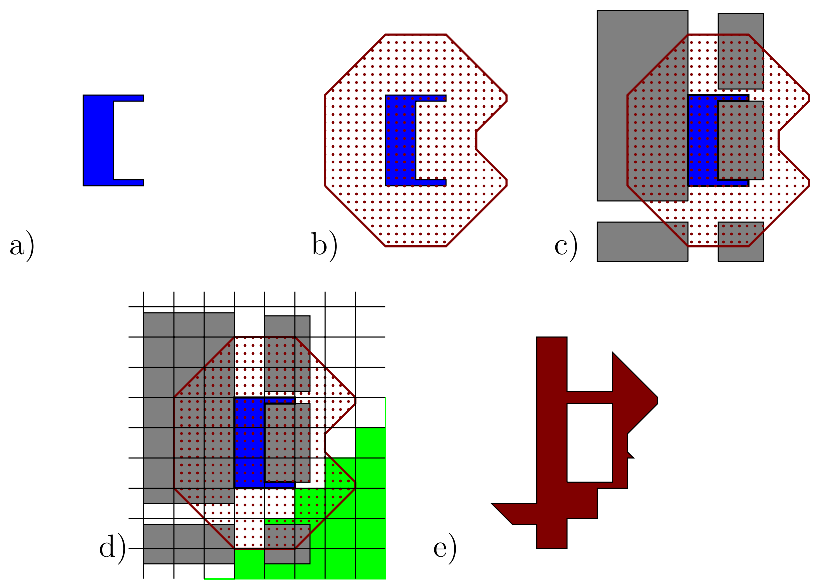

Figure 3.6: Example for computing the predecessor placement area $Pl_{pred}(v)$ for a vertex $v \in V(T)$.

partitions the sinks into two parts. The source of the tree is connected with a merging point where the tree splits into two branches. Each branch then goes to the 'center' of the two partition sets. For each partition set we can repeat this procedure. The connection between the source and the merging point and the connections between the merging point and the centers of the partition sets can be buffered optimally (i.e. the inverters can be placed with optimum distance).

This approach has several advantages. First, the BonnClock algorithm becomes more robust. Without the top-down partition an additional sink or a slight movement of the position of a sink might change the complete tree topology. Moreover, by the optimal buffering the latency (the maximum delay of the signal from the source to a sink) of the tree decreases, which has positive effects on the on-chip variation (see Section 3.7).

We will define the Sink Partition Problem for arbitrary distance functions. In practice we will use the blockage grid and distance graph introduced in the previous section. We assume that any direct connection between two points will be buffered optimally, that means, the delay between the points is proportional to the distance between them. We will give a generalized definition where $k \in \mathbb{N}_{>0}$ sub-sources are computed. The goal is to find a partition and place the sub-sources so that the overall latency does not increase and that the maximum latency of a sub-tree starting at a sub-source in minimized. For $k = 1$ the problem is to place one sub-source so that the maximum distance between sub-source and sinks is minimum while the maximum distance between source and sink does not increase.

Sink Partition Problem

Before we define the SINK PARTITION PROBLEM more formally we introduce some notations and deduct some fundamental properties.
The input is a set of sinks D with required arrival time windows $rat(x) = [\underline{rat}(x), \overline{rat}(x)]$ for all $x \in D$. Moreover, we have a source r and a set of feasible positions F where we can place sub-sources. The function $d : (\{r\} \times (F \cup D)) \cup (F \times D) \to \mathbb{R}_+$ gives us the distance between r and a sink or a feasible position and the distance between a feasible position and a sink.
We use a simplified delay model and assume that the delay to cover a certain distance d is proportional to d. The signal on a path can be further delayed so that we get for $(a, b) \in (\{r\} \times (F \cup D)) \cup (F \times D)$:

$$delay(a, b) \geq d(a, b)$$

if the signal goes from a to b.
The signal starts at the source at a given time $at(r)$ and has to reach a sink $x \in D$ at some time $at(x) \in rat(x)$. By definition $at(r) + delay(r, x) = at(x)$, thus

$$at(r) \leq at(x) - d(r, x).$$

We immediately conclude

$$at(r) \leq \min_{x \in D}(\overline{rat}(x) - d(r, x)).$$

Note that in the following problem formulation the starting time of the signal is part of the input. In most cases the starting time should be as late as possible. In this case we set $at(r) = \min_{x \in D}(\overline{rat}(x) - d(r, x))$.
The problem now is to add k sub-sources $s_1, \ldots, s_k$ at feasible positions and assign each sink to one of the sub-sources by a function $s : D \to \{s_1, \ldots, s_k\}$. The signal starts at the source, goes to the sub-sources and finally reaches the sinks that are assigned to it. We have for $x \in D$

$$delay(r, x) = delay(r, s(x)) + delay(s(x), x).$$

We denote by the non-negative number $\delta(s_i) := delay(r, s_i) - d(r, s_i)$ the additional delay on the path from the source to the i-th sub-source, $i = 1, \ldots, k$. The starting time of the signal should remain the same, thus we have to ensure that we can find arrival times $at(x) \in rat(x)$ satisfying

$$\begin{aligned} at(x) - at(r) &= delay(r, x) \\ &= delay(r, s(x)) + delay(s(x), x) \\ &\geq d(r, s(x)) + \delta(s(x)) + d(s(x), x). \end{aligned}$$

This can only be kept if

$$\overline{rat}(x) - d(r, s(x)) - \delta(s(x)) - at(r) \geq d(s(x), x). \tag{3.1}$$

Our goal is to minimize the maximum latency of a sub-source, that means the maximum delay between a sub-source and a sink that is assigned to it,

$$\min \max_{x \in D} \mathit{delay}(s(x), x). \tag{3.2}$$

For a sink $x \in D$ we have

$$\begin{aligned} \mathit{delay}(s(x), x) &= \mathit{delay}(r, x) - \mathit{delay}(r, s(x)) \\ &= at(x) - at(r) - d(r, s(x)) - \delta(s(x)) \end{aligned}$$

and

$$\mathit{delay}(s(x), x) \geq d(s(x), x).$$

To achieve (3.2) we set

$$at(x) = \max\{at(r) + d(r, s(x)) + \delta(s(x)) + d(s(x), x), \underline{rat}(x)\}$$

and get

$$\mathit{delay}(s(x), x) \;=\; \max\{d(s(x), x), \underline{rat}(x) - at(r) - d(r, s(x)) - \delta(s(x))\}.$$

(3.1) guarantees that $at(x) \in rat(x)$.
After these preparations we can define the problem formally:

Sink Partition Problem

Instance: A source r with starting time $at(r)$, sinks $D = \{x_1, \ldots, x_n\}$ with required arrival time windows $rat(x) = [\underline{rat}(x), \overline{rat}(x)]$ for all $x \in D$ and a $k \in \mathbb{Z}_{>0}$.

In addition, a set of feasible sub-source positions F and a distance function $d : (\{r\} \times (F \cup D)) \cup (F \times D) \to \mathbb{R}_+$.

Task: Find sub-source positions $s_1, \ldots, s_k \in F$, delays $\delta : \{s_1, \ldots, s_k\} \to \mathbb{R}_{\geq 0}$ and an assignment $s : D \to \{s_1, \ldots, s_k\}$ satisfying

$$\overline{rat}(x) - d(s(x), x) - d(r, s(x)) - at(r) \;\geq\; \delta(s(x)) \tag{3.3}$$

for all $x \in D$ so that

$$\max_{x \in D} \max\{d(s(x), x), \underline{rat}(x) - at(r) - d(r, s(x)) - \delta(s(x))\} \tag{3.4}$$

is minimized. (Or decide that no such sub-source positions exist.)

For simplicity we define

$$\begin{aligned} \text{latency}(s_i) &:= \max_{x \in D, s(x) = s_i} \mathit{delay}(s_i, x) \\ &= \max\left\{ \max_{x \in D, s(x) = s_i} d(s_i, x), \max_{x \in D, s(x) = s_i} \underline{rat}(x) - at(r) - d(r, s_i) - \delta(s_i). \right\} \end{aligned}$$

Minimizing (3.4) is equivalent to minimizing $\max_{i=1,\ldots,k} \text{latency}(s_i)$.

Sink Partition for $k = 1$

First we study the case $k = 1$. An easy way to get an optimum sub-source position is to try each $v \in F$ and check if it is feasible, compute latency(v) and take the best one. This can be done in $O(|F||D|)$. We present another approach with the same worst case running time, but that performs a lot better in practice (see Algorithm 3).
To compute an optimum solution we will have for each $v \in F$ an upper bound $ub_\delta(v)$ for $\delta(v)$ and a lower bound $lb_d(v)$ for the $\max d(v,x)$. Set $c(v) := \max_{x \in D} \underline{rat}(x) - at(r) - d(r,v)$ for all $v \in F$. Then $\max\{lb_d(v), c(v) - ub_\delta(v)\}$ is a lower bound for the latency of $v \in F$. If $ub_\delta(v) < 0$ then v is not feasible.

Algorithm 3: Sink Partition Algorithm for $k = 1$.

```
Input: Instance $(r, at(r), D, rat, k = 1, d)$ of the SINK PARTITION PROBLEM.
Result: Optimal sub source position $v \in F$.
if $at(r) > \min_{x \in D}(\underline{rat}(x) - d(r,x))$ then return 'No feasible solution exists.';
$c(v) := \max_{x \in D} \underline{rat}(x) - at(r) - d(r,v)$;
foreach $v \in F$ do
    $lb_d(v) := 0$;
    $ub_\delta(v) := \min_{x \in D} \overline{rat}(x) - d(r,v) - at(r)$;
end
repeat
    $v' = \arg\min_{v \in F, ub_\delta(v) \geq 0} \max\{lb_d(v), c(v) - ub_\delta(v)\}$;
    // Check upper bound $ub_\delta(v')$.
    $x_\delta = \arg\min_{x \in D} \overline{rat}(x) - d(v', x)$;
    if $\overline{rat}(x_\delta) - d(v', x_\delta) - d(r, v') - at(r) < ub_\delta(v')$ then
        foreach $v \in F$ with $ub_\delta(v) \geq 0$ do
            $ub_\delta(v) = \min\{ub_\delta, \overline{rat}(x_\delta) - d(v, x_\delta) - d(r,v) - at(r)\}$;
        end
    else
        // Check lower bound $lb_d(v')$.
        $x_d = \arg\max_{x \in D} d(v', x)$;
        if $lb_d(v') < d(v', x_d)$ then
            foreach $v \in F$ with $ub_\delta(v) \geq 0$ do
                $lb_d(v) := \max\{lb_d(v), d(v, x_d)\}$;
            end
        else
            return $v'$;
        end
    end
until $ub_\delta(v) < 0 \quad \forall v \in F$ ;
return 'No feasible solution exists.'
```

Initially, we set $ub_\delta := \min_{x \in D} \overline{rat}(x) - d(r,v) - at(r)$ and $lb_d(v) := 0$ for all $v \in F$. In each iteration choose $v' \in F$ with $ub_\delta(v') \geq 0$ and minimal $\max\{lb_d(v), c(v) - ub_\delta(v)\}$.

First we have to check if $ub_\delta(v') = \delta(v')$. For this, we compute $x_\delta \in D$ minimizing $\overline{rat}(x) - d(v', x)$. This is a sink minimizing the left side of (3.3). If $ub_\delta(v') > \overline{rat}(x_\delta) - d(v', x_\delta) - d(r, v') - at(r)$ the upper bound is not strict and we update $ub_\delta(v)$ for all $v \in F$ using x_δ and continue with the next iteration.
Otherwise, we check if $lb_d(v')$ is optimum. Let $x_d \in D$ be the vertex maximizing $d(v', x)$. If $lb_d(v') \geq d(v', x)$ then $\max\{lb_d(v), c(v) - ub_\delta(v)\}$ is the latency at v'. In this case v' is optimum by its choice. Otherwise, we update the lower bounds for all $v \in F$ by setting $lb_d(v) := \max\{lb_d(v), d(v, x_d)\}$.
The running time is dominated by the update process. Note that updating ub_δ and lb_d will be performed at most once for each $x \in D$. Therefore the algorithm will terminate. In each iteration at most $|F|$ values will be updated. Thus the running time is $O(|D||F|)$. Nevertheless, the algorithm performs much better in practice. The sink x_δ, chosen in line 9, has the property that it is a sink that is furthest away from a possible sub-source position (plus some constant). In practice, only a few of the sinks have this property and thus the outer loop will be processed only a few times (≤ 10). The running time is in average only $\frac{1}{20}$ times the running time of the trivial computation.

Sink Partition for $k \geq 2$

For $k \geq 2$ the Sink Partition Problem becomes even more complicated. In this case we want to place k sub-sources and assign the sinks to them.
We can assume that the signal starts at the source as late as possible, i.e. $at(r) := \min_{x \in D}(\overline{rat}(x) - d(r, x))$. Otherwise, we apply the algorithm for $k = 1$ and place a single sub-source r' optimally. As (3.4) is minimized, the signal reaches r' as late as possible. Then we apply the problem for $k \geq 2$ using r' as source. Thus we can assume without loss of generality $\delta(s_1) := 0$.
Let $(s_1, \ldots, s_k, \delta(s_2), \ldots, \delta(s_k), s : D \to \{s_1, \ldots, s_k\})$ be an optimum solution. Set $D_i := \{x \in S | \, s(x) = s_i\}$, $i \in \{1, \ldots, k\}$. For $i \in \{2, \ldots k\}$ we can assume that there is an $x \in D_i$ so that (3.3) is satisfied with equality. Otherwise, we could increase $\delta(s_i)$ and get a solution that is still feasible. By increasing $\delta(s_i)$ (3.4) cannot get worse so the solution is also optimum. We conclude that the left hand side of (3.3), and hence $\delta(s_i)$, can have at most $|D_i|$ different values if s_i is fixed.
These observations lead to a simple polynomial algorithm (see Algorithm 4). For each possible tuple $(s_1, \ldots, s_k, \delta_2, \ldots, \delta_k)$ we compute for every sink $x \in D$ the delay between x and s_i for the corresponding δ_i, $1 \leq i \leq k$ (we set the delay to ∞ if x cannot be assigned to that sub-source without violating (3.3)). $latency(s_1, \ldots, s_k, \delta_2, \ldots, \delta_k)$ is the best achievable latency for the tuple. The best tuple and the corresponding partition will be taken. The optimality follows immediately from the previous observations. The running time of the algorithm is $O(|F|^k |D|^k)$.
The running time can be further improved. Assume we have chosen $s_1, \ldots, s_k$ and $\delta_2, \ldots, \delta_{k-1}$. Let $\delta \in \Delta_k$ be fixed and denote by $\hat{D}_i(\delta)$ the set of sinks that can be assigned to s_i, $i \in \{1, \ldots, k\}$, without violating (3.3). Now assign each sink x to a sub-source s_i satisfying $x \in \hat{D}_i(\delta)$ and minimizing (3.4). Note that there might be sinks that cannot be assigned, as all sub-sources do not satisfy (3.3). Let $D_i(\delta)$ be the set of sinks

Algorithm 4: Sink Partition Algorithm for $k \geq 2$.

```
$\delta_1 := 0$;
foreach $(s_1, \dots, s_k) \in F^k$ do
    $\Delta_i := \{\overline{rat}(x) - d(s_i, x) - d(r, s_i) - at(r) | x \in D\}$ for $2 \leq i \leq k$;
    foreach $(\delta_2, \dots, \delta_k) \in \Delta_2 \times \dots \times \Delta_k$, $\delta_2 \leq \dots \delta_k$ do
        foreach $x \in D$ do
            foreach $i \in \{1, \dots, k\}$ do
                // Check if $x$ can be assigned to $s_i$
                if $\overline{rat}(x) - d(s_1, x) - d(r, s_1) - at(r) \geq 0$ then
                    $delay_i(x) := \max(d(s_i, x), \underline{rat}(x) - at(r) - d(r, s_i) - \delta_i\}$;
                else
                    $delay_i(x) := \infty$;
                end
            end
            $delay(x) := \min_{i=1}^{k} delay_i(x)$;
        end
        $latency(s_1, \dots, s_k, \delta_2, \dots, \delta_k) := \max_{x \in D} delay(x)$;
    end
end
```

assigned to s_i and let $latency_i(\delta) := \max_{x \in D_i(\delta)}\{d(s_i, x), \underline{rat}(x) - at(r) - d(r, s_i) - \delta(s_i)\}$, $i \in \{1, \dots, k\}$, be the latency at s_i for the partition given by δ, $\delta(s_1) := 0, \delta(s_k) := \delta$ and $\delta(s_i) = \delta_i$, $2 \leq i \leq k-1$.
Now let $\delta_a \leq \delta_b$. Observe that $\hat{D}_i(\delta_a) = \hat{D}_i(\delta_b)$, $1 \leq i \leq k-1$, as these sets are independent of δ and $\hat{D}_k(\delta_b) \subseteq \hat{D}_k(\delta_a)$ as sinks could get infeasible to be connected to s_k for increasing δ. We conclude $\max_{i=1}^{k-1} latency_i(\delta_a) \leq \max_{i=1}^{k-1} latency_i(\delta_b)$ and $latency_k(\delta_a) \geq latency_k(\delta_b)$. We are looking for $\delta \in \Delta_k$ so that each sink can be assigned feasibly to a sub-source and so that $\max\{latency_1(\delta), \dots, latency_k(\delta)\}$ is minimized. The previous observations showed that this can be done by a binary search over Δ_k.

Theorem 3.1. *The* Sink Partition Problem *for $k \geq 2$ with $\delta_1 = 0$ can be solved in $O(|F|^k |D|^{k-1} \log(|D|))$.* □

There are further options to reduce the running time in practice. For example, it can be shown that there is a sink x' that can be assigned without loss of generality to s_1 and for which the inequality (3.3) is strict. Then s_1 must be placed on a shortest path from the source to x'. Moreover, when looking for s_1, another binary search like heuristic can be used. Nevertheless, the worst case running time does not improve. It is an open question if the Sink Partition Problem can be solved more efficiently.

Usage within BonnClock

Within BonnClock the cases $k = 1$ and $k = 2$ are the interesting ones. The tree starts at the source, goes to the optimum sub-source, then splits into two branches and so on. In order to use the presented algorithms we need a finite set of feasible sub-source positions and an appropriate distance function. For this, we use the blockage grid and distance graphs presented in Section 3.5.1. As sub-source positions we use the centers of the free tiles and compute the distance graphs on-demand. Two examples are shown in Figure 3.7. On both chips first the source is connected to the optimum sub-source position. Then the sinks are partitioned into two parts and then these parts are partitioned once more. The black lines and circles show the resulting upper parts of the clock trees.

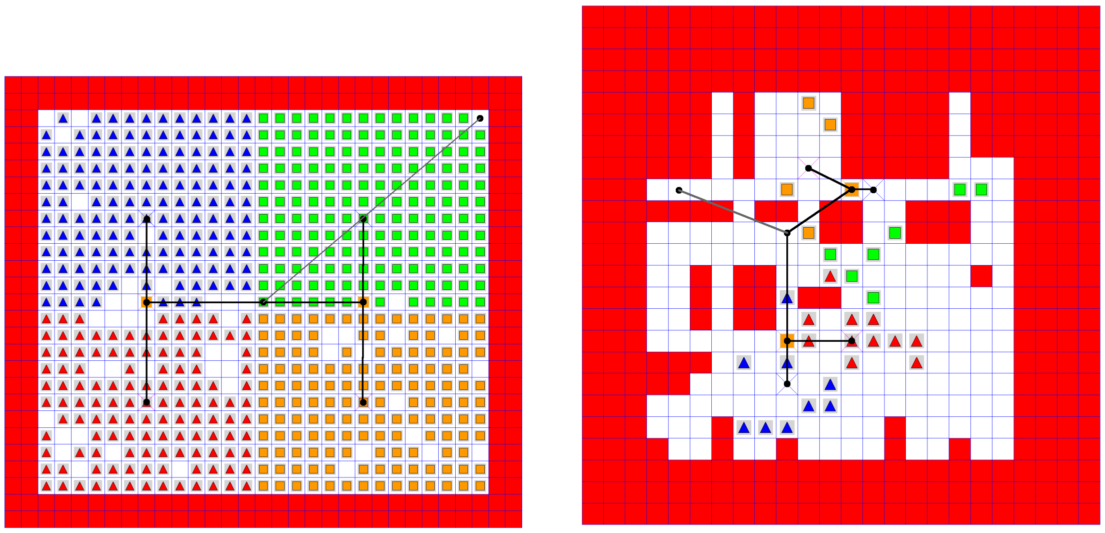
Figure 3.7: Sink partition examples for two zero skew trees.

3.7 On-Chip Variation

One problem in clock tree construction we have not mentioned yet is the on-chip variation. In order to understand the impact of variation we have to take a closer look at the timing of a chip and to define slacks.

Figure 3.8 shows a simplified section of a chip with two registers A and B receiving a clock signal from a clock tree and a logical path P_{AB} starting at A and ending in B. Let del_{AB} be the delay of a data signal on this path. The clock signal arrives at the clock pin of A at time at_A and at the clock pin of B at time at_B. Moreover, the clock tree has a cycle time of C. We define the slacks

$$slack_{late} \quad := \quad at_B + C - (at_A + del_{AB} + \delta_{AB}^{late})$$

and

$$slack_{early} \quad := \quad at_A + del_{AB} - (at_B + \delta_{AB}^{early})$$

where δ_{AB}^{late} and δ_{AB}^{early} are some constants.

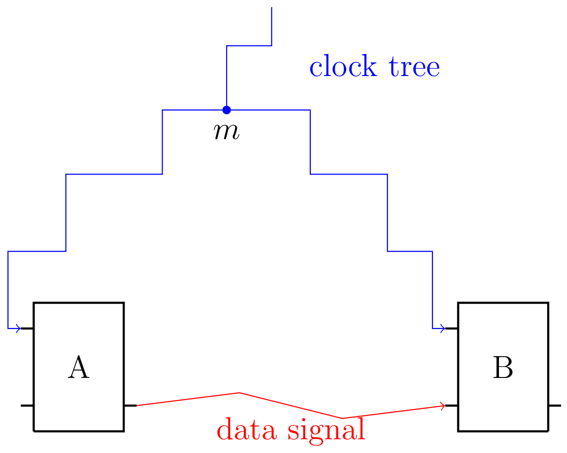

Figure 3.8: Example of on-chip variation.

Both slacks have to be non-negative, otherwise the signal arrives to late ($slack_{late} < 0$) or to early ($slack_{early} < 0$) at register B. If a slack is negative we call it a late mode violation or early mode violation, respectively.
Early mode violations can be fixed easily by inserting inverters into the logical path between A and B and thus delaying the signal. Nevertheless, there should not be too many early mode violations as inserting too many inverters increases the power consumption and might lead to placement problems. Late mode problems are hard to fix, as this can only be done by speeding up the data signal from A to B. But one can assume that the data path is already made as fast as possible by previous design steps. In the worst case, large parts of the chip have to be redesigned or the cycle time of the tree has to be increased by the worst late mode slack and the chip becomes slower.
A produced chip differs from the mathematical model. Wires and circuits can get a little bit bigger or smaller than computed, the voltage might differ and even the temperature might be different on different parts of the chip. Thus some signals might get slower or faster than estimated. Now assume that in our example in Figure 3.8 the path from m to the clock pin of register A within the clock tree becomes slower and the path from m to the clock pin of B gets faster by on-chip variation. Then at_A increases and at_B decreases and $slack_{late}$ becomes worse. If, on the other hand, the path from m to A gets faster and the path from m to B gets slower, $slack_{early}$ gets worse. One possibility to reduce the effect of on-chip variation is to reduce the lengths of the non-common path to the registers within the clock tree.
We propose the following approach to consider on-chip variation within the BonnClock algorithm. First we perform a timing analysis with the assumption that the clock signal arrives at each sink within its required arrival time window (for example by using the tool BonnCycleOpt, Held [2008]). We get a list of pairs of sinks together with late and early slacks. Let L and E be the most critical late and early edges. We want to minimize the effect of on-chip variation on these edges by reducing the lengths of the non-common paths within the clock tree. To this end, we have to adapt the topology of the tree.
The topology of the tree can be influenced directly within the clustering and by the

top-down partition. The goal of the clustering is to minimize the length of all Steiner trees plus the number of clusters. This cost function can be expanded by a reward for each edge of $L \cup E$ where both endpoints are within the same cluster. As early mode problems can be fixed easily their reward is a small constant. Late mode problems are hard to fix, so we want to maximize the minimum late slack. The smaller a late mode slack is the bigger its reward will be. The post optimization algorithms presented in Section 1.2 and 2.2 can be extended to consider these extra rewards.
Considering the late and early edges within the top-down partition is more difficult. If the endpoints of an edge are in two different partition sets the non-common paths get long as the first common predecessor of the vertices within the tree is the merging point. Thus the endpoints of the edges should be in the same partition set. If there are long edges or if the edges form large connected components we have to 'split' some of them. A first simple approach is to perform the sink partitioning as described and achieve two partition sets. Let D_1 be the sinks that have to be assigned to the first sub-source, D_2 be the sinks that have to be assigned to the second sub-source and D' be the remaining sinks that can be assigned to both sets without increasing the cost of the partition. The early and late edges induce a graph on $D_1 \cup D_2 \cup D'$. Now we want to distribute the sinks of D' to the sub sources so that the cost of the edges with endpoints assigned to different sub-sources is minimized (here the cost of an edge is minus the reward it gets within the clustering). Such a distribution can be determined optimally by finding a minimum cut in the induced graph separating D_1 from D_2. Nevertheless, there could be many edges between sinks that can only be assigned to one sub-source.
It is an open problem how the late and early edges can be considered appropriately within the partition algorithm itself. Another problem of on-chip variation is that there is no convincing method to model and estimate its effect.

3.8 An Example

In the following pictures (3.9 to 3.14) we show some 'snap shots' of a run of the algorithm BonnClock on a recent design (technology: 45nm, 3.4mm $\times$ 3.0mm, 5 498 sinks). In order to determine the topology of the upper part of the tree, the sink partition algorithm has been used twice. The result is shown on the left side of Figure 3.7. As the sinks are distributed all over the chip and the blockages are quite small, the upper part of the tree, determined by the sink partition, looks like a classical H-tree.
On the pictures the circuits, placement areas and wires are colored according to the arrival time of the clock signal (blue early, red late).
Figures 3.15, 3.16 and 3.17 show further trees that have been computed by the Bonn-Clock algorithm. The orange blockages in the pictures receive themselves the clock signal.

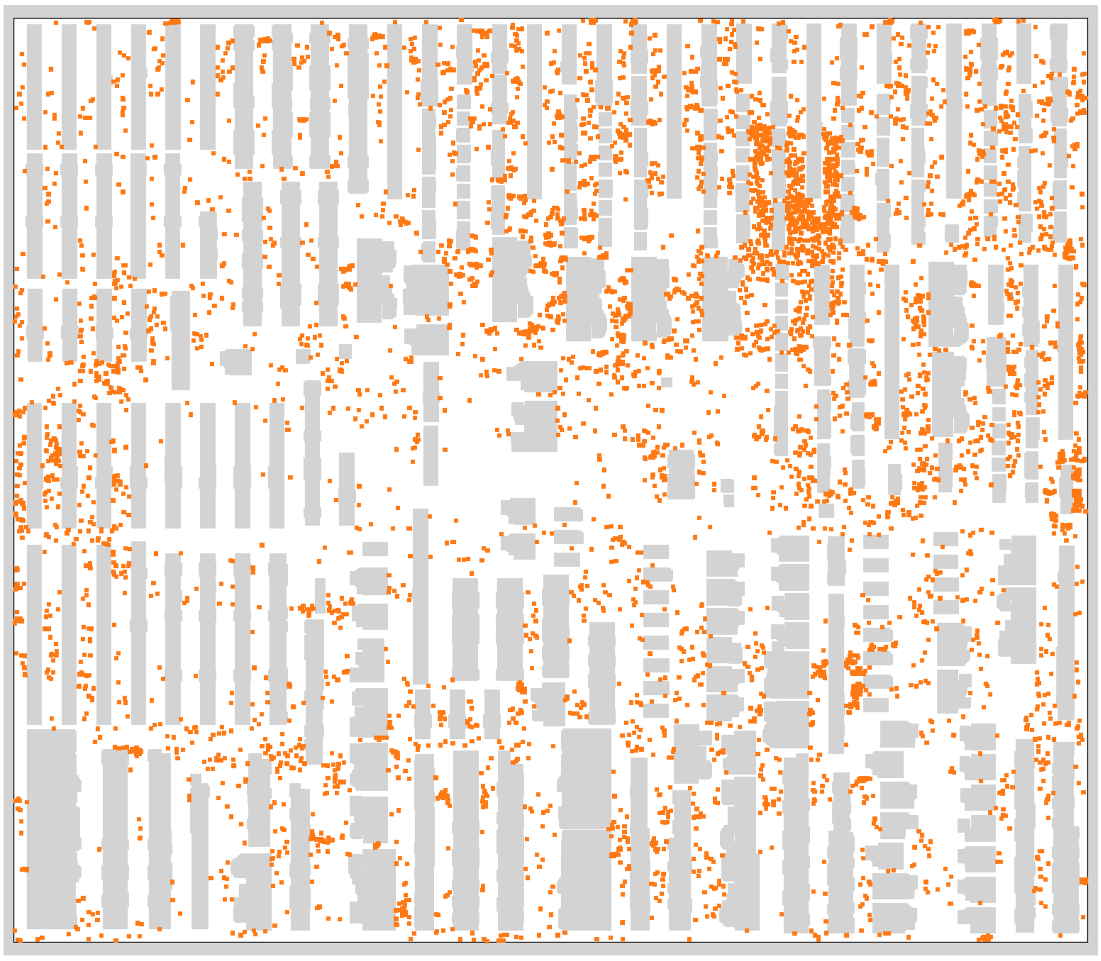

Figure 3.9: Initial instance with sinks in orange and rectangular blockages in gray.

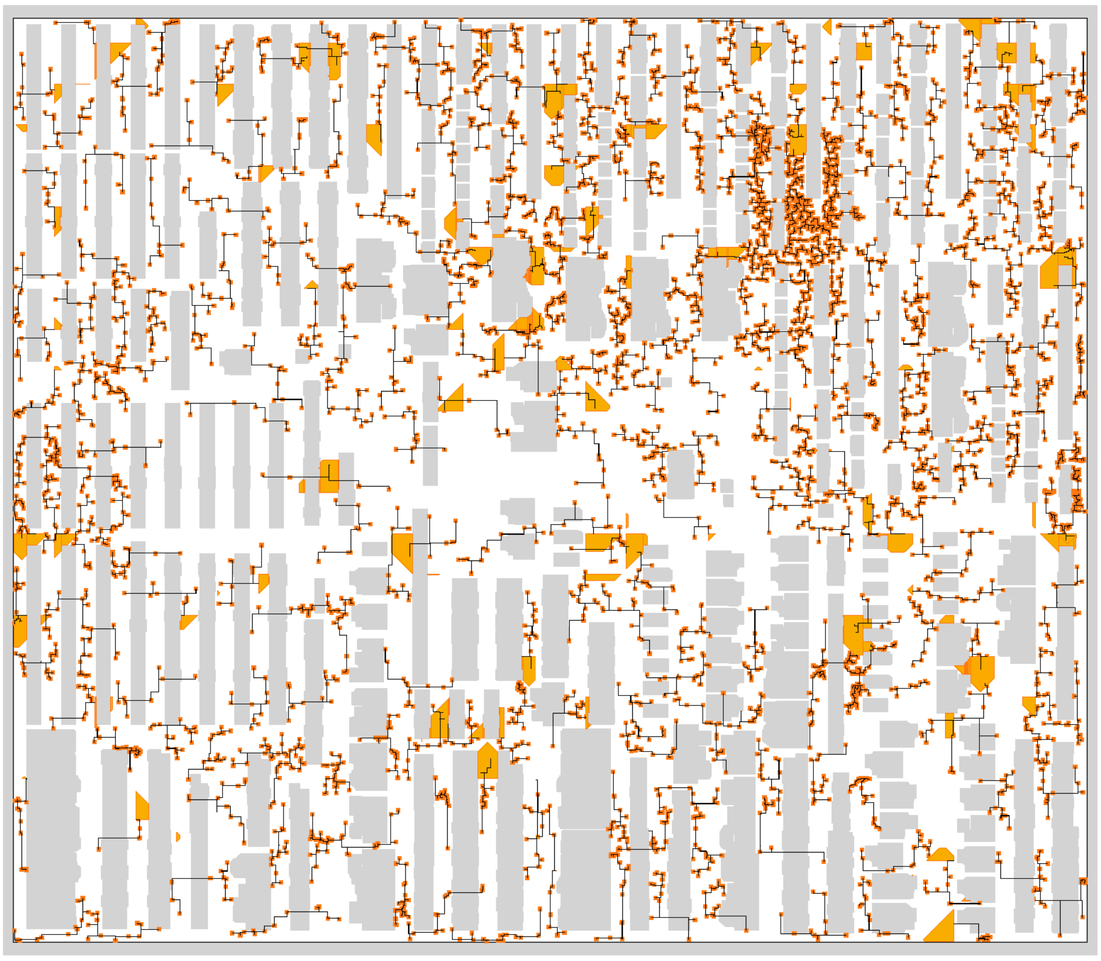

Figure 3.10: The tree after the first clustering. You can see the placement areas of the active vertices. They are in orange as their required arrival time window is still relatively late.

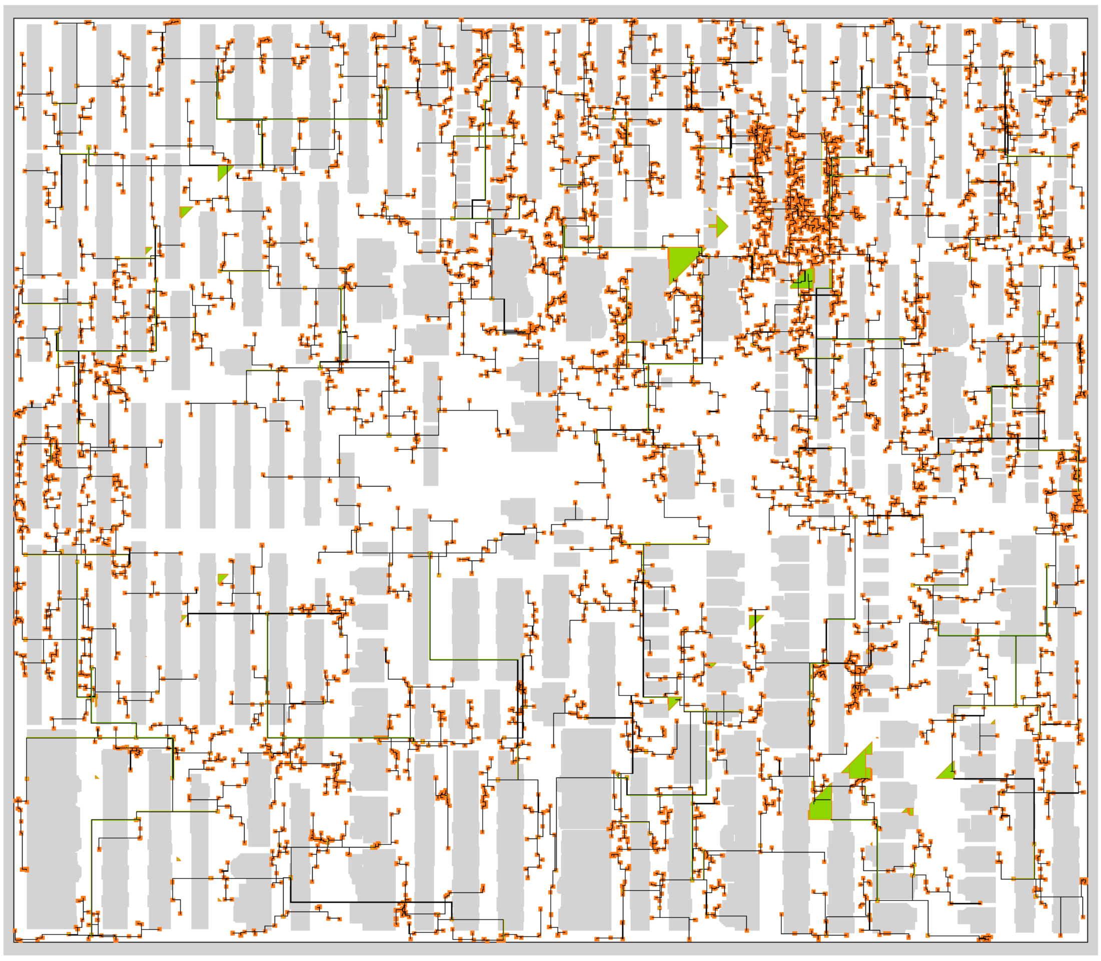

Figure 3.11: The tree some iterations later. Only a few active vertices are remaining. Their placement areas are now in green. The vertices move towards the sub-sources in the four quadrants of the chip.

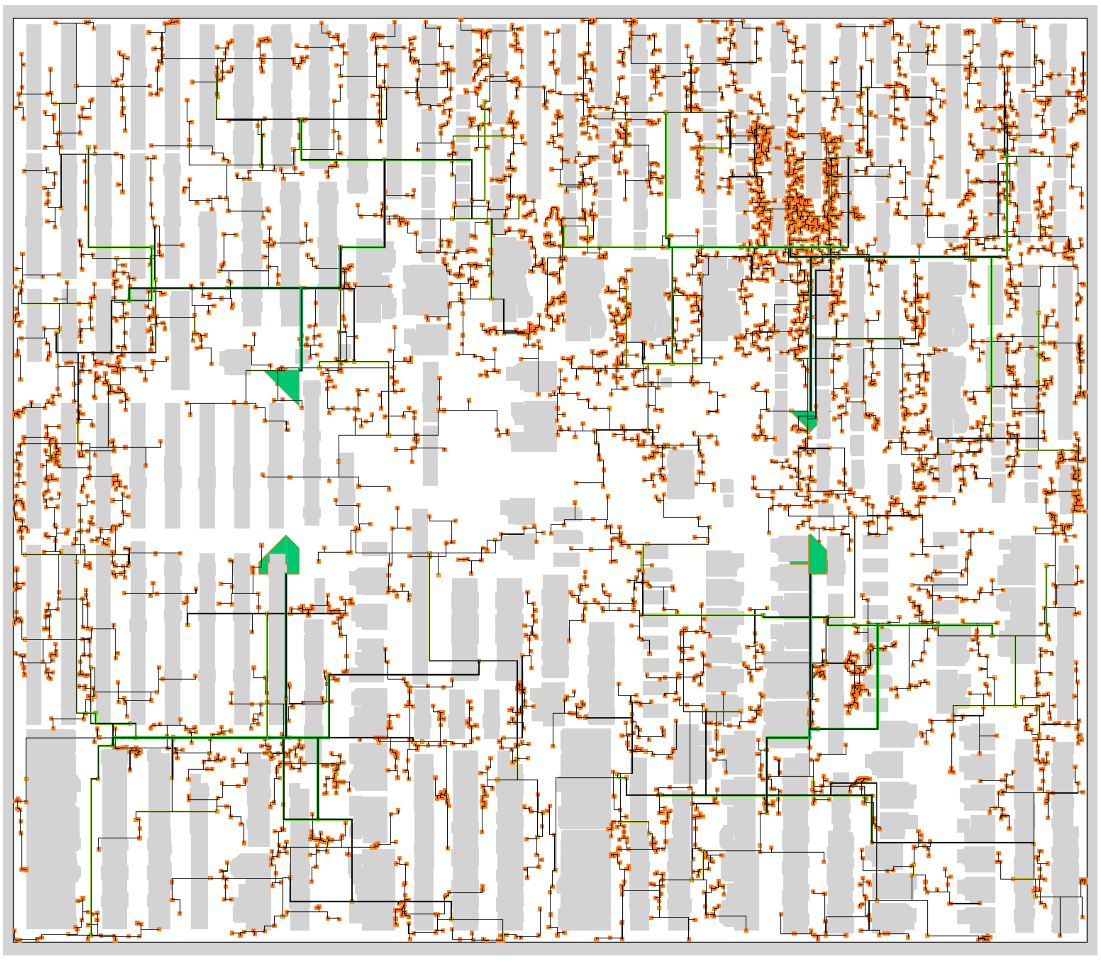

Figure 3.12: The four sub-sources in the four quadrants have been reached. The remaining four active vertices go towards the two sub-sources on the left and the right half.

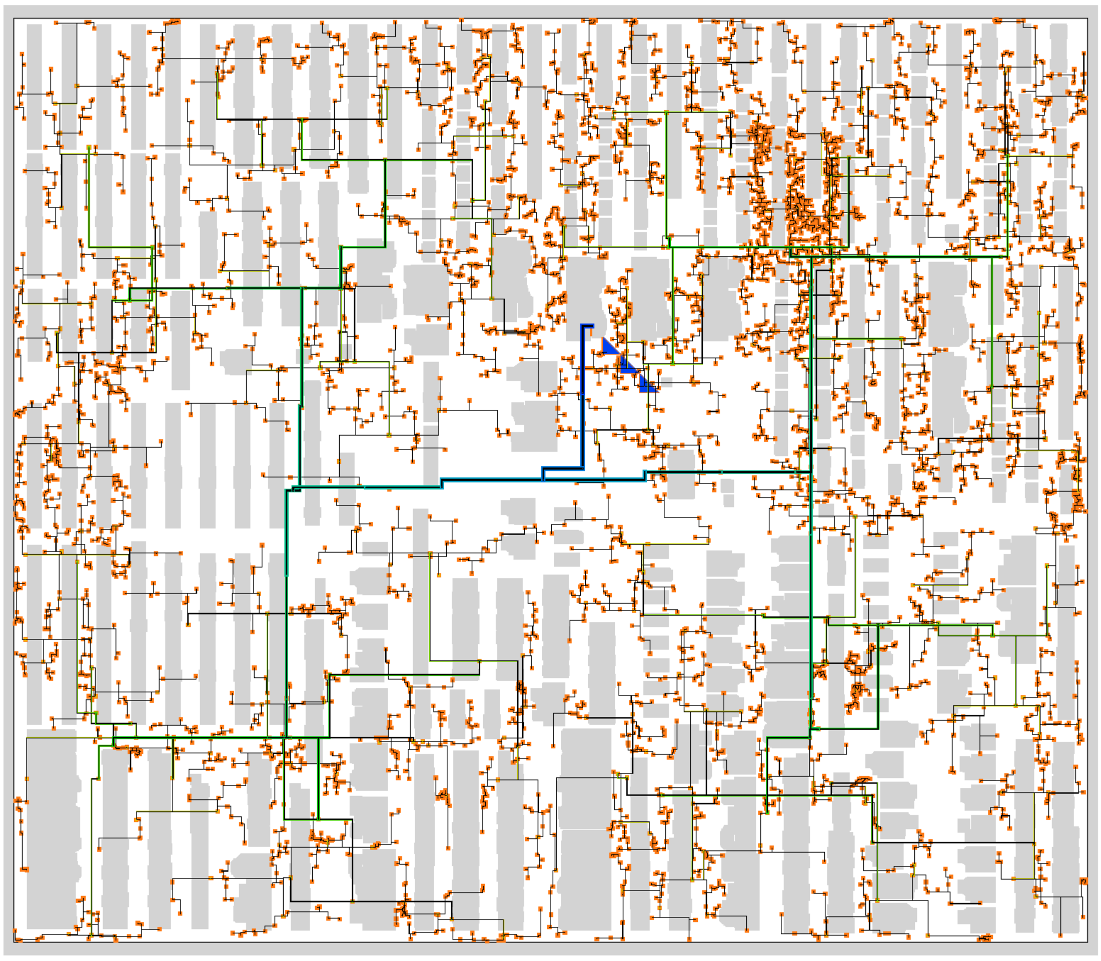

Figure 3.13: Now the final sub-source in the center of the chip has been reached and the last active vertex goes towards the source in the upper right corner. Its placement area is colored in blue. You can see that the area that is too far away from the source has been removed.

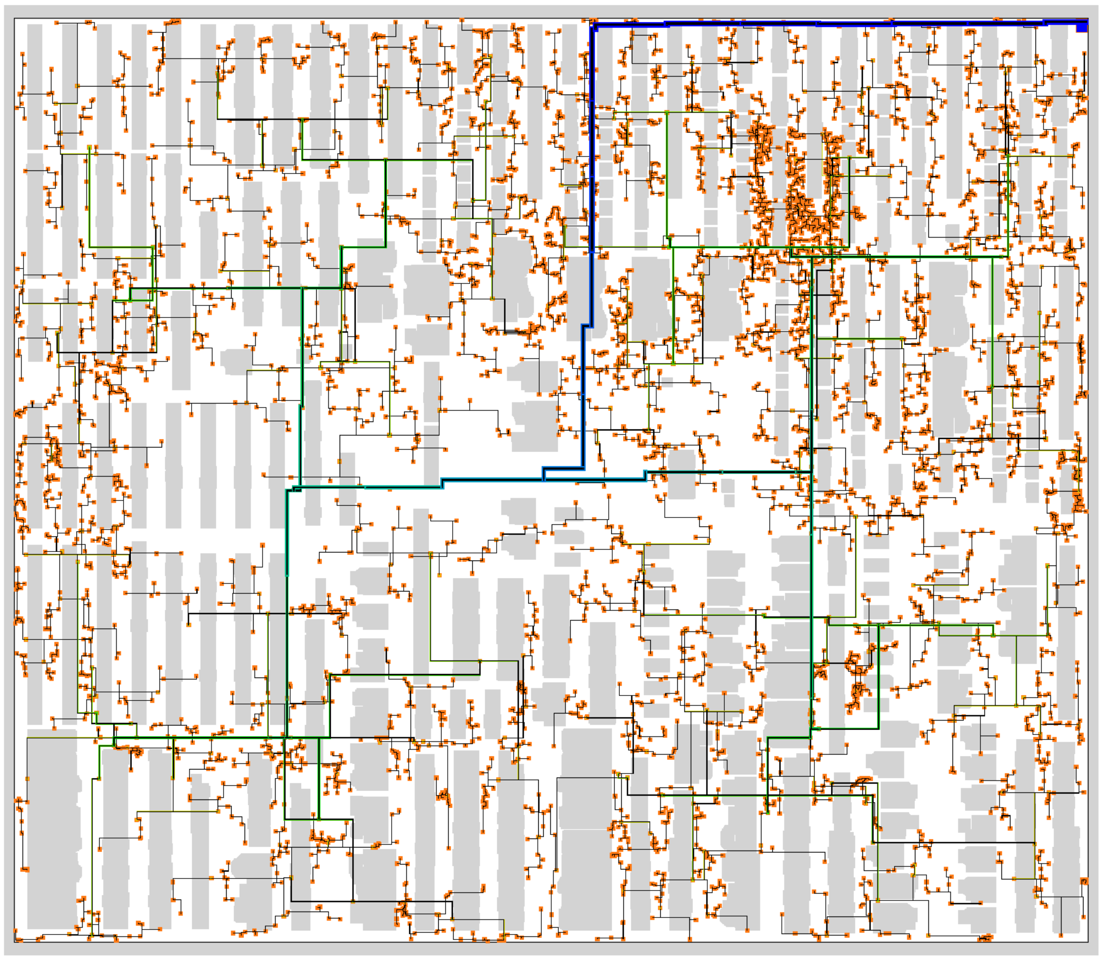

Figure 3.14: The final tree.

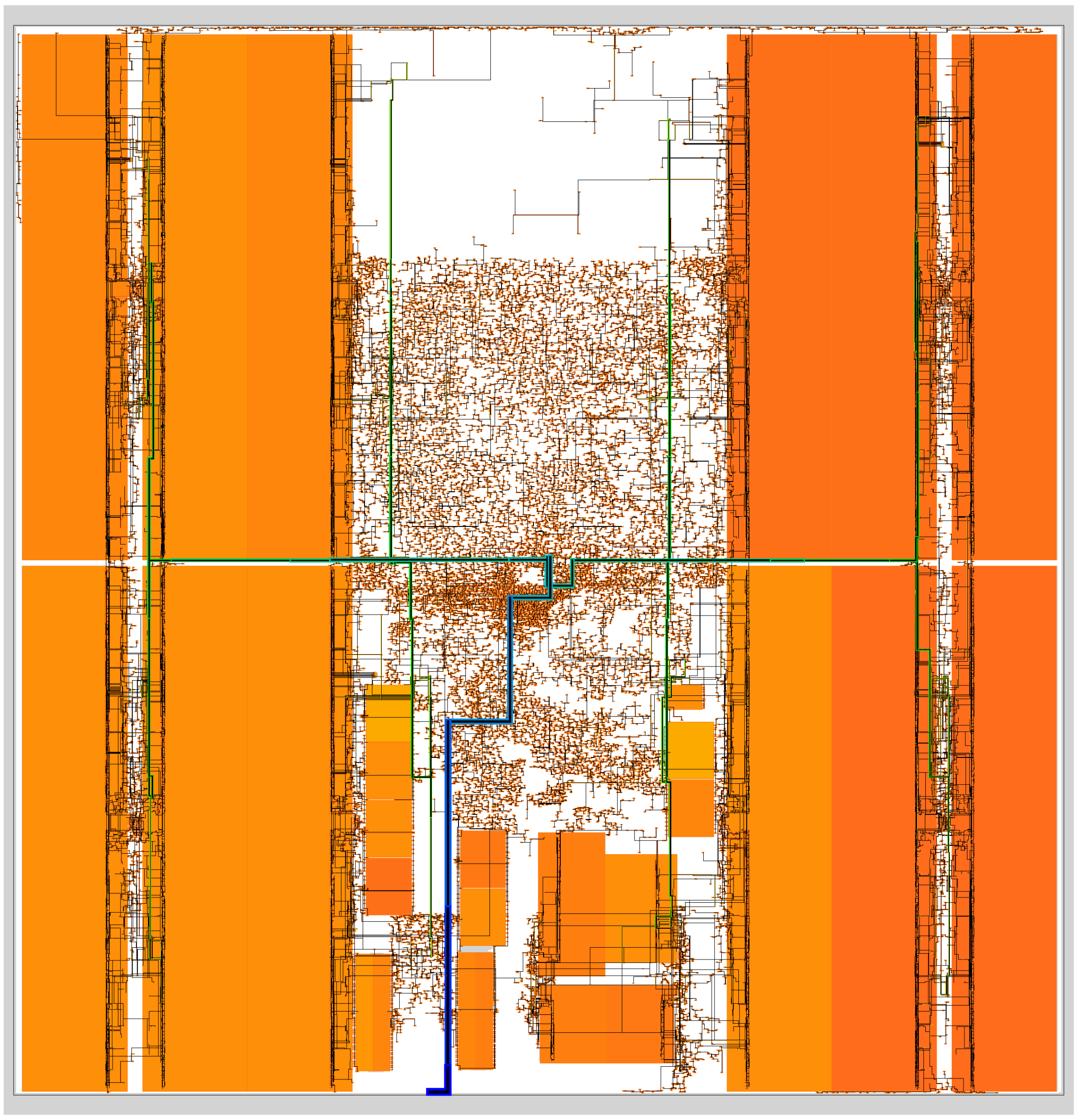

Figure 3.15: Clock tree (65 nm technology, 2.0mm × 2.0mm, 37 107 sinks) with big blockages.

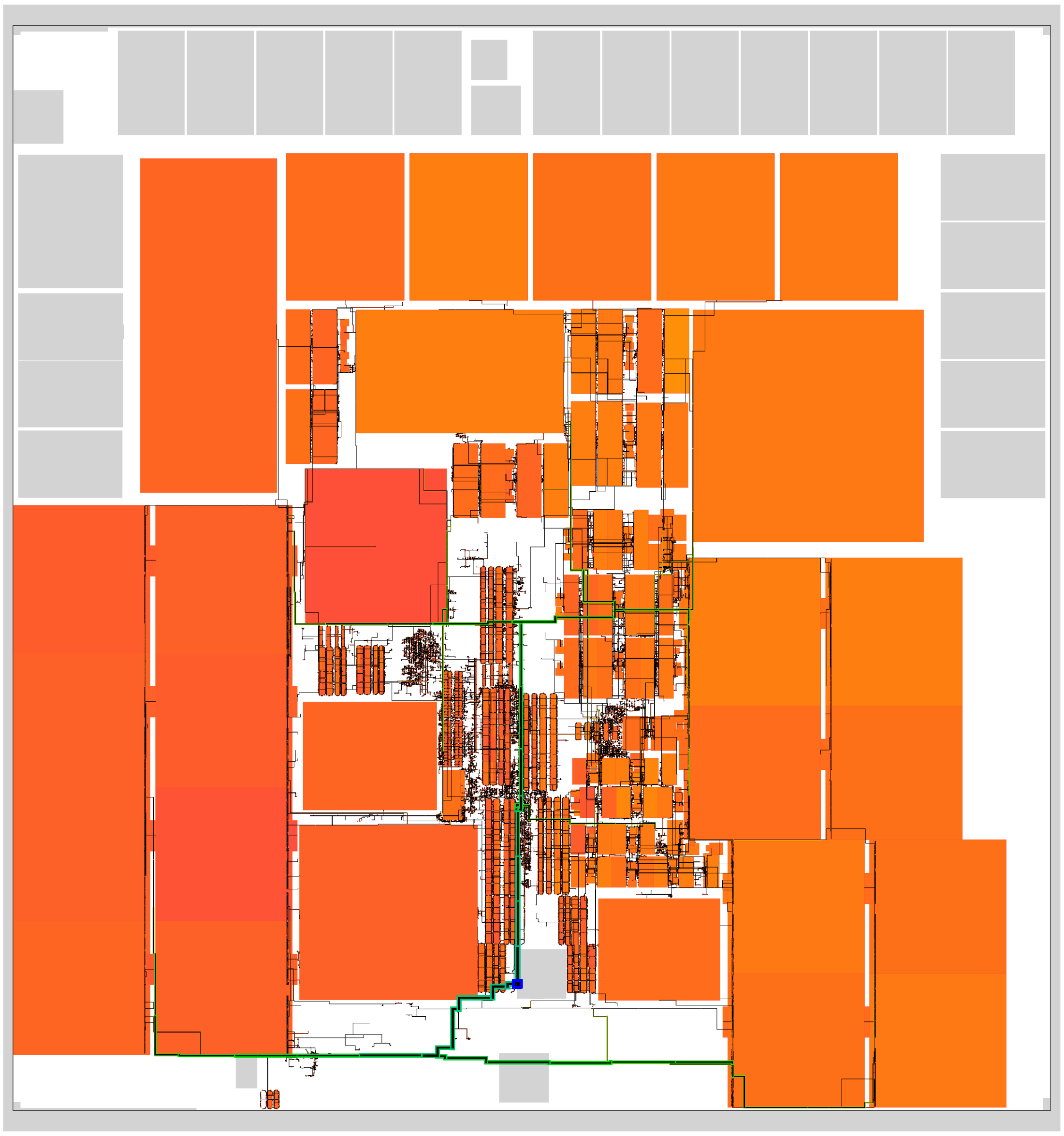

Figure 3.16: Clock tree (65 nm technology, 18.0mm × 18.0mm, 118 319 sinks) with more complex blockages.

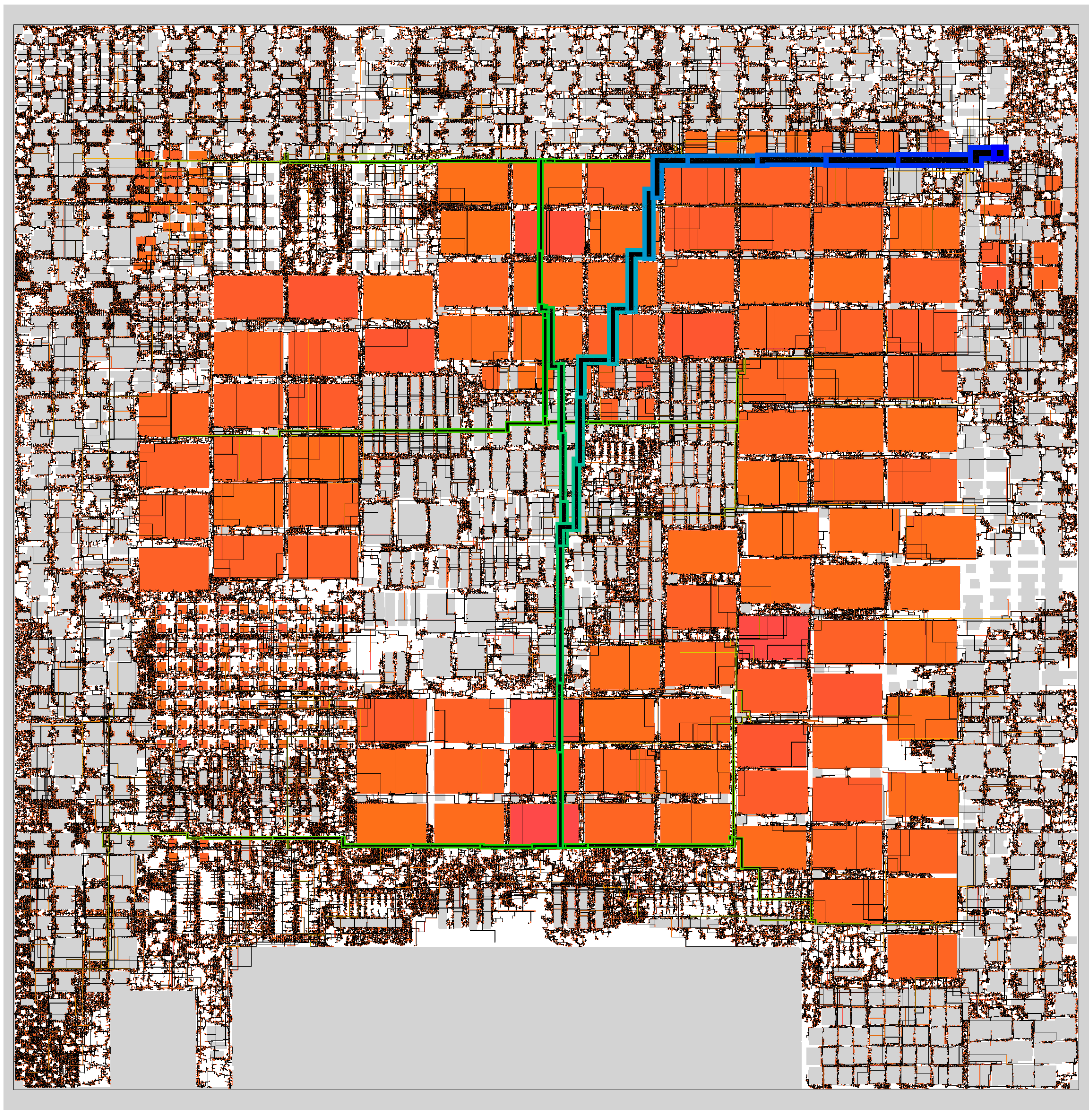

Figure 3.17: Large clock tree (80 nm technology, 15.6mm × 15.6mm, 680 776 sinks).

3.9 Lower Bounds

In this section we develop lower bounds for the power consumption of a clock tree using the lower bounds for the Sink Clustering Problem. We study a special class of clock trees where the sinks are driven by circuits of one special type (that could also be inverters) and where these special circuits are driven by a tree only containing inverters. Furthermore, the nets driven by the special circuits can have different electrical characteristics than the other nets. This type of clock trees often appear in practice: The clock signal is distributed by an inverter tree and then modified in some way by the special circuits before it reaches the sinks. We refer to the special circuits and the nets they are driving as last stage of the tree and the remaining parts as upper stages.

Let D be the set of sinks with positions in the plane, c_{last}, f_{last} and u_{last} the input capacitance, power consumption and load limit of a circuit of the last stage. Assume we have $k \in \mathbb{N}$ different types of inverters we can use in the upper part of the tree. Let c^i_{up}, f^i_{up} and u^i_{up} be the input capacitance, power consumption and load limit of the i'th inverter type. Moreover, denote by κ_{last} and κ_{up} the cost per unit of length in the last and upper stages and by u_{root} the load limit of the root.

Let l_{SMT} be a lower bound for the length of a Steiner tree on D. Now let T be a feasible clock tree driving D and denote by l_{last} be the length of the the wiring in the last stage and by t_{last} the number of circuits in the last stage. Later we will use lower bound for the Sink Clustering Problem in order to estimate l_{last} and t_{last}.

The clock tree induces a Steiner tree on D. Thus the length of the tree in the upper part is at least $\max\{l_{SMT} - l_{last}, 0\}$. We conclude that the load of the upper part of the tree is at least

$$upcap \quad := \quad \kappa_{up} \cdot \max\{l_{SMT} - l_{last}, 0\} + t_{last} \cdot c_{last}.$$

The load $upcap$ has to be driven by the root and by inverters. We can assume that the first u_{root} units of load are driven by the root. Note that for each inserted inverter additional capacitance is added to the tree that itself has to be driven by other inverters. These inverters have to be driven by others and so on. Let t_i be the number of inverters of the i'th type that are used in the upper part of the tree. By the previous observations we get

$$upcap - u_{root} + \sum_{i=1}^{k} t_i c^i_{up} \leq \sum_{i=1}^{k} t_i u^i_{up}.$$

The power consumption of the inverters is $\sum_{i=1}^k t_i f^i_{up}$. In order to find a lower bound for this we relax the problem and allow to insert fractions of inverters. Then the problem is to find $t_i \in \mathbb{R}_{\geq 0}$ satisfying $upcap - u_{root} = \sum_{i=1}^k t_i(u^i_{up} - c^i_{up})$ so that $pow := \sum_{i=1}^k f^i_{up}$ is minimum. For this purpose, we use the argumentation of Bartoschek et al. [2006]. Let j be a type of inverter minimizing $\frac{f^j_{up}}{u^j_{up} - c^j_{up}}$. Obviously, $t_j := \frac{upcap - u_{root}}{u^j_{up} - c^j_{up}}$ and $t_i := 0$, $1 \leq i \leq k$, $i \neq j$ minimizes pow, i.e. we can assume that only type j is used in the upper stages. Set $c_{up} := c^j_{up}, f_{up} := f^j_{up}, u_{up} := u^j_{up}$ and $t := t_j$. Therefore, the total capacitance to drive in the upper stages is at least

$$\left(1+\frac{c_{up}}{u_{up}-c_{up}}\right)\cdot(upcap-u_{root})+u_{root}$$

and the total power consumption of the upper stages is at least

$$\left(1+\frac{f_{up}}{u_{up}-c_{up}}\right)\cdot(upcap-u_{root})+u_{root}.$$

We set $\lambda := \left(1+\frac{f_{up}}{u_{up}-c_{up}}\right)$ and conclude that the total power consumption of the complete tree is at least

$$\lambda\cdot(\kappa_{up}\cdot\max\{l_{SMT}-l_{last},0\}+t_{last}\cdot c_{last})+(1-\lambda)u_{root}+\kappa_{last}\cdot l_{last}+t_{last}\cdot f_{last}. \tag{3.5}$$

Theorem 3.2. *Let T be a clock tree as defined in this section. Assume there is a feasible clustering of the last stage of T and denote by l_{last} the total length of the nets in the clustering and by t_{last} the number of clusters.*
If $\kappa_{last} > \lambda\kappa_{up}$ then the total power consumption is at least the maximum of

$$\lambda\kappa_{up}l_{SMT}+(1-\lambda)u_{root}+(\kappa_{last}-(\lambda\kappa_{up}))l_{last}+(\lambda c_{last}+f_{last})t_{last} \tag{3.6}$$

and

$$(1-\lambda)u_{root}+\kappa_{last}\cdot l_{last}+(\lambda c_{last}+f_{last})t_{last} \tag{3.7}$$

If $\kappa_{last} \le \lambda\kappa_{up}$ then the power consumption of T is at least the maximum of

$$(1-\lambda)u_{root}+\kappa_{last}\cdot l_{SMT}+(\lambda c_{last}+f_{last})t_{last} \tag{3.8}$$

and

$$(1-\lambda)u_{root}+\kappa_{last}\cdot l_{last}+(\lambda c_{last}+f_{last})t_{last} \tag{3.9}$$

Proof. (3.6), (3.7) and (3.9) follow directly from (3.5). In order to see (3.8), observe that (3.5) is at least

$$\begin{aligned}
& \lambda\kappa_{up}\cdot(l_{SMT}-l_{last})+(1-\lambda)u_{root}+\kappa_{last}\cdot l_{last}+t_{last}\cdot f_{last}\\
= \; & (\kappa_{last}-\lambda\kappa_{up})l_{last}+(1-\lambda)u_{root}+\kappa_{last}\cdot l_{SMT}+(\lambda c_{last}+f_{last})t_{last}\\
\ge \; & (1-\lambda)u_{root}+\kappa_{last}\cdot l_{SMT}+(\lambda c_{last}+f_{last})t_{last}
\end{aligned}$$

In the last inequality we used $\kappa_{last} \le \lambda\kappa_{up}$. □
The length of the wiring in the last stage l_{last} and the number of drivers of the last stage t_{last} are not known. But note that in all four cases (3.6) to (3.9) we are looking for a feasible clustering of the sinks with load limit u_{last} minimizing a non-negative weighted sum of the total length of the nets and the number of clusters. But this is exactly the Sink Clustering Problem. Thus we can use the lower bound developed in Section 1.3 in order to get a lower bound for (3.6) to (3.9).

3.10 Experimental Results

In this section we compare the power consumption of trees built by BonnClock to the lower bound for the power consumption presented in Section 3.9. As mentioned in that section the bound can only be applied on trees with a special structure.
The clock tree instances on page 93 to page 96 are from different technologies and vary from small (Kurt with 59 073 sinks) to big instances (Nicolai with 680 776 sinks).
The tables show information about the designs and the main clock tree that has been built for these results. We abbreviate 'power consumption' by 'p.c.' and 'lower bound' by 'l.b.'. 'unplanned skew' is the maximum time a required arrival time window is missed - this is equivalent to the δ in the problem formulation in Section 3.1.1. We also give the parameters that are relevant for the lower bound computation. Note that the clock tree instance 'Nicolai' has a structure different from the one required to compute the lower bound of Section 3.9.
In order to a compute lower bound for the power consumption we need a lower bound l_{SMT} for the length of a minimum Steiner tree on all sinks. The instances were to big to use the tool GeoSteiner for good lower bounds (see Section 1.4), thus we computed the length of a minimum spanning tree. $\frac{2}{3}$ times this length yields a lower bound for the length of a minimum Steiner tree (see Section 1.1.3). The clock trees built by BonnClock have a power consumption that is about 40% more than the lower bound presented in Section 3.9. This is just a little bit worse than the results we have seen for the Sink Clustering Algorithm. As the simple lower bound we used for the length of a minimum Steiner tree is not very good (see results in Section 1.4), we can expect that we are even a lot nearer to the cost of an power-optimum clock tree. Moreover, remark that the lower bounds do not consider any timing or placement restrictions.
We also show a comparison to the clock tree construction tool ClockDesigner that has been developed by IBM. ClockDesigner builds uniform trees, i.e. on all source-sink paths there are the same number of inverters. All inverters are of the same type and in the upper stages the tree is balanced by wiring detours. In order to guarantee a fair comparison, we used the same set of parameter for ClockDesigner and BonnClock. While the unplanned skew of the trees of both tools are nearly the same, the number of inserted circuits and the total power consumption differ a lot. The trees build by ClockDesigner use a lot more resources than our trees.
Finally note, that on all instances about 80 to 90% of the total power consumption is within the last stage of the clock trees. Thus a good sink clustering is the most important part of a clock tree construction tool in order to reduce power consumption.

design	Kurt
technology	45 nm
size (mm× mm)	3.4×3.0
$\lvert D\rvert$	59 073
c_{last} (W)	$7.723 \cdot 10^{-6}$
f_{last} (W)	$3.320 \cdot 10^{-5}$
κ_{last} (W per mm)	$3.416 \cdot 10^{-4}$
c_{up} (W)	$1.772 \cdot 10^{-5}$
f_{up} (W)	$3.543 \cdot 10^{-5}$
κ_{up} (W per mm)	$1.873 \cdot 10^{-4}$
$\kappa_{up} \cdot l_{SMT}$ (W)	0.04679
u_{up} (W)	$1.482 \cdot 10^{-4}$
total p.c. (W)	0.1926
p.c. lower stages (W)	0.1634 (84.8%)
lower bound p.c. (W)	0.13513
total p.c. / l.b. p.c.	1.4253

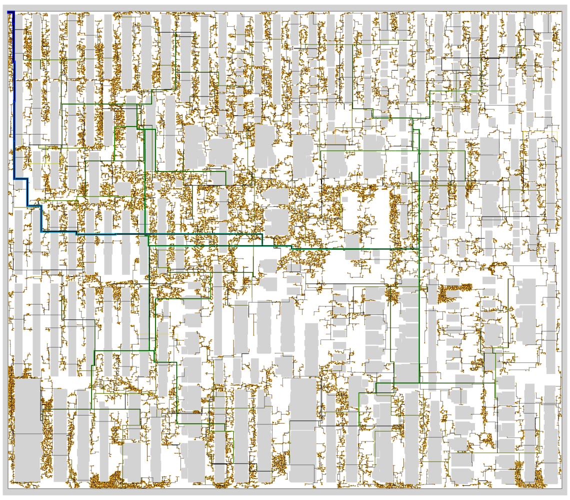

Figure 3.18: Clock tree instance and computed tree on Kurt.

		ClockDesigner (CD)	BonnClock (BC)	CD/BC
	unplanned skew (ps)	27	32	0.084
last stage	number of circuits	1 736	1 348	1.288
	p.c. circuits (W)	0.0576	0.0448	1.287
	p.c. wiring (W)	0.1290	0.1186	1.087
	total p.c. (W)	0.1966	0.1634	1.203
upper stages	number of circuits	153	182	0.841
	p.c. circuits (W)	0.0054	0.0059	0.919
	p.c. wiring (W)	0.0260	0.0233	1.118
	total p.c. (W)	0.0314	0.0292	1.075
all stages	total p.c.(W)	0.2280	0.1926	1.184

Table 3.1: Comparison to ClockDesigner on Kurt.

design	Pascal
technology	65 nm
size (mm× mm)	2.4×2.6
$\lvert D \rvert$	71 356
c_{last} (W)	$3.311 \cdot 10^{-6}$
f_{last} (W)	$4.052 \cdot 10^{-5}$
κ_{last} (W per mm)	$4.660 \cdot 10^{-4}$
c_{up} (W)	$2.815 \cdot 10^{-5}$
f_{up} (W)	$7.016 \cdot 10^{-5}$
κ_{up} (W per mm)	$1.611 \cdot 10^{-4}$
$\kappa_{up} \cdot l_{SMT}$ (W)	0.06245
u_{up} (W)	$2.723 \cdot 10^{-4}$
total p.c. (W)	0.2734
p.c. lower stages (W)	0.2472 (90.4%)
lower bound p.c. (W)	0.20981
total p.c. / l.b. p.c.	1.303

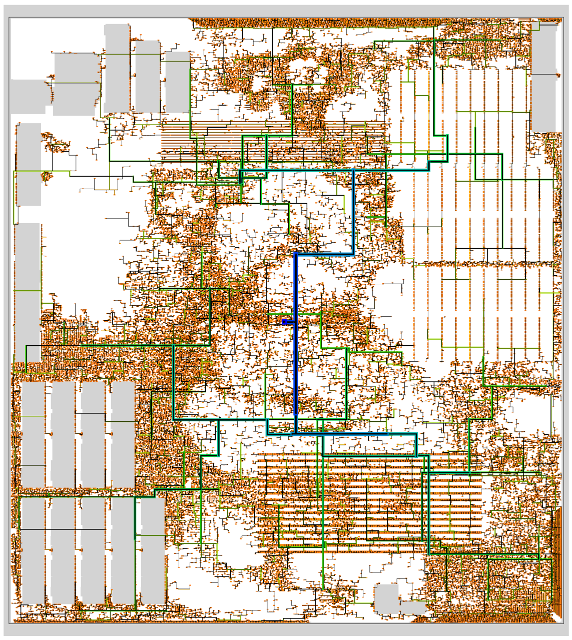

Figure 3.19: Clock tree instance and computed tree on Pascal.

		ClockDesigner (CD)	BonnClock (BC)	CD/BC
	unplanned skew (ps)	39	45	0,867
last stage	number of circuits	2 784	1 560	1.785
	p.c. circuits (W)	0.1128	0.0632	1.785
	p.c. wiring (W)	0.1995	0.1840	1.084
	total p.c. (W)	0.3223	0.2472	1.304
upper stages	number of circuits	165	114	1.447
	p.c. circuits (W)	0.0116	0.0102	1.137
	p.c. wiring (W)	0.0249	0.0160	1.556
	total p.c. (W)	0.0365	0.0262	1.393
all stages	total p.c.(W)	0.3598	0.2734	1.316

Table 3.2: Comparison to ClockDesigner on XppHdm.

design	Katrin
technology	130 nm
size (mm× mm)	10.1×10.1
$\|D\|$	119 461
c_{last} (W)	8.0978×10^{-7}
f_{last} (W)	2.3016×10^{-5}
κ_{last} (W per mm)	6.1407×10^{-5}
c_{up} (W)	1.1944×10^{-5}
f_{up} (W)	2.0308×10^{-5}
κ_{up} (W per mm)	3.5637×10^{-5}
$\kappa_{up} \cdot l_{SMT}$ (W)	0.05107
u_{up} (W)	8.9278×10^{-5}
total p.c. (W)	0.2660
p.c. lower stages (W)	0.2359 (88.7%)
lower bound p.c. (W)	0.1864
total p.c. / l.b. p.c.	1.4273

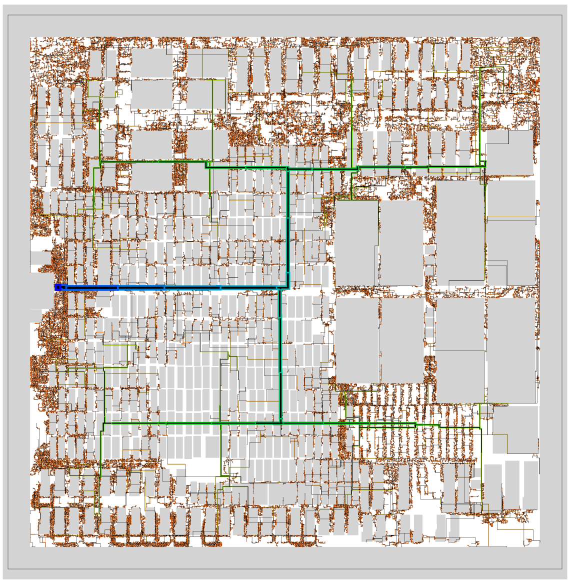

Figure 3.20: Clock tree instance and computed tree on Katrin.

		ClockDesigner (CD)	BonnClock (BC)	CD/BC
	unplanned skew (ps)	53	48	1.104
last stage	number of circuits	8 247	3 616	2.281
	p.c. circuits (W)	0.1897	0.0832	2.280
	p.c. wiring (W)	0.1636	0.1527	1.071
	total p.c. (W)	0.3533	0.2359	1.498
upper stages	number of circuits	585	606	0.965
	p.c. circuits (W)	0.0120	0.0093	1.290
	p.c. wiring (W)	0.0318	0.0208	1.527
	total p.c. (W)	0.0438	0.0301	1.455
all stages	total p.c.(W)	0.3971	0.2660	1.493

Table 3.3: Comparison to ClockDesigner on Katrin.

design	Nicolai
technology	90 nm
size (mm× mm)	15.6 × 15.6
$\lvert D\rvert$	680 776
total p.c. (W)	4.1208
p.c. lower stages (W)	3.4610 (84.0%)

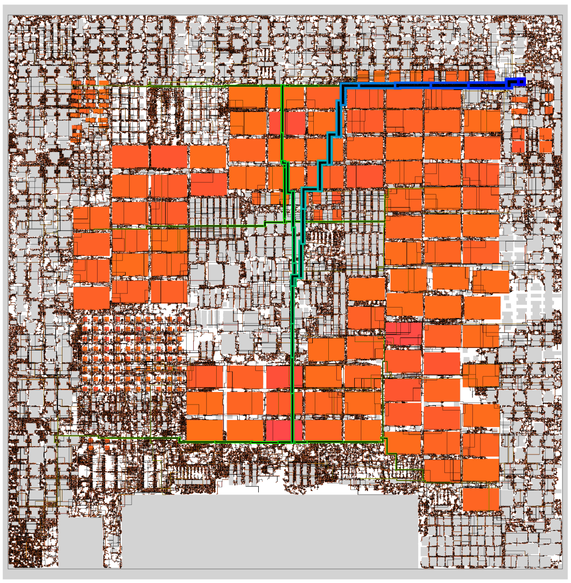

Figure 3.21: Clock tree instance and computed tree on Nicolai.

		ClockDesigner (CD)	BonnClock (BC)	CD/BC
	unplanned skew (ps)	39	43	0.907
last stage	number of circuits	99 472	41 981	2,369
	p.c. circuits (W)	4.1561	1.7540	2.369
	p.c. wiring (W)	1.8189	1.7070	1.066
	total p.c. (W)	5.9750	3.4610	1.726
upper stages	number of circuits	7 675	5 193	1.478
	p.c. circuits (W)	0.3682	0.2008	1.834
	p.c. wiring (W)	0.5123	0.4590	1.116
	total p.c. (W)	0.8805	0.6598	1.334
all stages	total p.c.(W)	6.8555	4.1208	1.664

Table 3.4: Comparison to ClockDesigner on Nicolai.

4 Repeater Tree Topology Problem

Repeater trees are very similar to clock trees. They are also inverter trees distributing a logic signal from a source to a set of sinks. Repeater tree instances are much smaller than clock tree instances, most of them have just a few to some dozen sinks. On the other hand there can be more than a million repeater tree instances on a chip. Another big difference to clock trees are the timing constraints. In clock trees each sink has a required arrival time window, the clock signal has to arrive within it. In repeater trees the signal has to arrive at each sink not later than an individual latest required arrival time, but it is allowed to arrive at any time before.

Most repeater tree algorithms split the task into two parts: first generating a repeater tree topology and then buffering (insertion of inverters) into the tree.

In this thesis we will restrict ourselves to the generation of repeater tree topologies. First, we define the Repeater Tree Topology Problem and present and analyze a simple topology creation algorithm.

Sections 4.1 and 4.2 are based on joint work with Christoph Bartoschek, Stephan Held, Dieter Rautenbach and Jens Vygen.

4.1 Previous Work and Problem Definition

The construction of repeater trees, including buffering, has been studied by various authors. Okamoto and Cong [1996] proposed an algorithm computing a topology by a bottom-up clustering of the sinks and then top-down buffering of the obtained topology. Similarly, Lillis et al. [1996] also integrated buffer insertion and topology integration by considering an algorithm which takes the locality of sinks into account combined with an dynamic programming approach. Hrkic and Lillis [2002, 2003] considered an algorithm, which makes better use of timing information and integrated timing and placement informations. The sinks are partitioned according to their criticality and a given topology is changed by partially separating critical and non-critical sinks. The algorithm of Alpert et al. [2002] is a hybrid combination of the Prim heuristic for minimum spanning trees (Prim [1957], Dijkstra [1959]) and the Dijkstra's shortest path algorithm (Dijkstra [1959]). Further approaches were proposed by Cong and Yuan [2000], Dechu et al. [2004], Hentschke et al. [2007] and Pan et al. [2007].

Before we continue, we define the Repeater Tree Topology Problem formally:

Repeater Tree Topology Problem

Instance: An instance consists of:

- a *source* $r \in \mathbb{R}^2$,
- a finite non-empty set $S \subseteq \mathbb{R}^2$ of *sinks*,
- a *required arrival time* $a_s \in \mathbb{R}$ for every sink $s \in S$, and
- two numbers $c, d \in \mathbb{R}_{>0}$.

Feasible Solution: A feasible solution of such an instance is

- a rooted tree $T = (V(T), E(T))$ with vertex set $\{r\} \cup S \cup I$ where $I \subseteq \mathbb{R}^2$ is a set of $|S| - 1$ points such that r is the root of T and has exactly one child, the elements of I are the internal vertices of T and have exactly two children each, and the elements of S are the leaves of T.

In Bartoschek et al. [2006, 2009] such a feasible solution was called a *repeater tree topology*, because the number, types, and positions of the actual repeaters are not yet determined.
The optimization goals for a repeater tree are related to the wiring, to the number of repeater circuits, and to the timing. We assume that every edge $e = (u, v) \in E(T)$ of T is realized along a path between the two points u and v in the plane which is shortest with respect to some norm $||\cdot||$ on $\mathbb{R}^2$ (in practice the l_1 norm). Furthermore, we assume that repeaters are inserted in a relatively uniform way into all wires in order to linearize the delay within the repeater tree. Hence the wiring and also the number of repeater circuits needed for the physical realization of the edge e are proportional to $||u - v||$. For the entire repeater tree topology, this results in a total cost of

$$l(T) := \sum_{(u,v)\in E(T)} ||u - v||.$$

The delay of the signal starting at the root and traveling through T to the sinks has two components. Let $E[r, s]$ denote the set of edges on the path P in T between the root r and some sink $s \in S$. The linearized delay along the edges of P is modeled by

$$\sum_{(u,v)\in E[r,s]} d||u - v||.$$

Furthermore, every internal vertex on P corresponds to a bifurcation which causes an additive delay of c along P. For the entire path P, these additional delays sum up to

$$c(|E[r, s]| - 1).$$

In practice there is sometimes a certain degree of freedom how to distribute the additional delay caused by a bifurcation to the two branches (see Section 4.4).

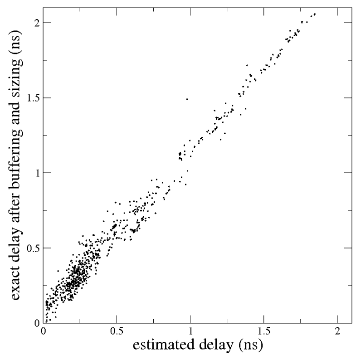

Figure 4.1: Quality of the delay model.

Altogether, we estimate the delay of the signal along P by the sum of these two components.
Assuming that the signal starts at time 0 at the root, the *slack* at some sink $s \in S$ in T is estimated by

$$\sigma(T, s) := a_s - \sum_{(u,v)\in E[r,s]} d||u - v|| - c(|E[r, s]| - 1)$$

and the *worst slack* equals

$$\sigma(T) := \min\{\sigma(T, s) \mid s \in S\}.$$

The restrictions on the number of children of the root and the internal vertices of T imply that the number of sinks contributes logarithmically to the delay, which corresponds to physical experience. The accuracy of our simple delay estimation is shown in Figure 4.1, which compares our estimation with the real physical delay once the repeater tree has been realized and optimized. The parameters c and d are technology-dependent. For the 65nm technology their values are about $c = 20$ps and $d = 220$ps/mm.

In principle, a repeater tree topology is acceptable with respect to timing if $\sigma(T)$ is non-negative, i.e. the signal arrives at every sink $s \in S$ not later than a_s. Nevertheless, in order to account for inaccurate estimations and manufacturing variation, the worst slack $\sigma(T)$ should have at least some reasonable positive value $\sigma_{\min}$ or should even be maximized.
We can formulate three main optimization scenarios: Determine T such that

(O1) $\sigma(T)$ is maximized, or

(O2) $l(T)$ is minimized, or

(O3) for suitable constants $\alpha, \beta, \sigma_{\min} > 0$, the expression

$$\alpha \min\{\sigma(T), \sigma_{\min}\} - \beta l(T)$$

is maximized.

While scenario (O1) is reasonable for instances which are very timing critical, scenario (O2) is reasonable for very timing uncritical instances. Scenario (O3) is probably the practically most relevant one. In the next section, we will show that (O1) can be solved exactly in polynomial time. In contrast to that, (O2) is hard even for restricted choices of the norm such as the l_1-norm, since it is essentially the Steiner tree problem (Garey and Johnson [1977]).

4.2 A Simple Procedure and its Properties

We propose the following very simple procedure for the construction of repeater tree topologies.

Algorithm 5: Repeater tree topology creation

```
Choose a sink $s_1 \in S$;
$V(T_1) \leftarrow \{r, s_1\}$;
$E(T_1) \leftarrow \{(r, s_1)\}$;
$T_1 \leftarrow (V(T_1), E(T_1))$;
$n \leftarrow |S|$;
for $i = 2$ to $n$ do
    Choose a sink $s_i \in S \setminus \{s_1, s_2, \ldots, s_{i-1}\}$, an edge $e_i = (u, v) \in E(T_{i-1})$, and an internal vertex $x_i \in \mathbb{R}^2$;
    $V(T_i) \leftarrow V(T_{i-1}) \dot\cup \{x_i\} \dot\cup \{s_i\}$;
    $E(T_i) \leftarrow (E(T_{i-1}) \setminus \{(u, v)\}) \cup \{(u, x_i), (x_i, v), (x_i, s_i)\}$;
    $T_i \leftarrow (V(T_i), E(T_i))$;
end
```

The procedure inserts the sinks one by one according to some order $s_1, s_2, \ldots, s_n$ starting with a tree containing only the root r and the first sink s_1. The sinks s_i for $i \geq 2$ are inserted by subdividing an edge e_i with a new internal vertex x_i and connecting x_i to s_i. The behavior of the procedure clearly depends on the choice of the order, the choice of the edge e_i, and the choice of the point $x_i \in \mathbb{R}^2$.
In view of the large number of instances which have to be solved in an acceptable time (Bartoschek et al. [2006, 2009]) the simplicity of the above procedure is an important advantage for its practical application. Furthermore, implementing suitable rules for the choice of s_i, e_i, and x_i allows to pursue and balance various practical optimization goals.
We present two variants (P1) and (P2) of the procedure corresponding to the above optimization scenarios (O1) and (O2), respectively.

(P1) The sinks are inserted in an order of non-increasing criticality, where the *criticality* of a sink $s \in S$ is quantified by

$$-(a_s - d||r - s||).$$

(Note that this is the estimated worst slack of a repeater tree topology containing only the one sink s. Since a sink s can be critical because its required arrival time a_s is small and/or because its distance $||r-s||$ to the root is large, this is a reasonable measure for its criticality.)

During the i-th execution of the **for**-loop, the new internal vertex x_i is always chosen at the same position as r — formally this turns $V(T_i)$ into a multiset — and the edge e_i is chosen such that $\sigma(T_i)$ is maximized.

(P2) s_1 is chosen such that $||r-s_1|| = \min\{||r-s|| \mid s \in S\}$ and during the i-th execution of the **for**-loop, s_i, $e_i = (u,v)$, and x_i are chosen such that

$$l(T_i) = l(T_{i-1}) + ||u-x_i|| + ||x_i - v|| + ||x_i - s_i|| - ||u-v||$$

is minimized.

Theorem 4.1. *The largest achievable worst slack σ_{opt} equals*

$$\sigma^*(S) := \max\left\{\sigma \in \mathbb{R} \mid \sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a_s - d||r-s|| - \sigma)\right\rfloor} \leq 1\right\},$$

and (P1) generates a repeater tree topology $T_{(P1)}$ with $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}}$.

Proof: Let $a'_s = a_s - d||r-s||$ for $s \in S$. Let T be an arbitrary repeater tree topology. By the definition of $\sigma(T)$ and the triangle-inequality for $||\cdot||$, we obtain

$$|E[r,s]| - 1 \quad \leq \quad \left\lfloor \frac{1}{c}\left(a_s - \sum_{(u,v)\in E[r,s]} d||u-v|| - \sigma(T)\right)\right\rfloor \leq \left\lfloor \frac{1}{c}\left(a'_s - \sigma(T)\right)\right\rfloor$$

for every $s \in S$. Since the unique child of the root r is itself the root of a binary subtree of T in which each sink $s \in S$ has depth exactly $|E[r,s]| - 1$, Kraft's inequality (Kraft [1949]) implies

$$\sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma(T))\right\rfloor} \quad \leq \quad \sum_{s\in S} 2^{-|E[r,s]|+1} \leq 1.$$

By the definition of $\sigma^*(S)$, this implies $\sigma(T) \leq \sigma^*(S)$. Since T was arbitrary, we obtain $\sigma_{\text{opt}} \leq \sigma^*(S)$.
It remains to prove that $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}} = \sigma^*(S)$, which we will do by induction on $n = |S|$. For $n = 1$, the statement is trivial. Now let $n \geq 2$. Let s_n be the last sink inserted by (P1), i.e. $a'_{s_n} = \max\{a'_s \mid s \in S\}$. Let $S' = S \setminus \{s_n\}$.

Claim

$$\text{frac}\left(\frac{\sigma^*(S)}{c}\right) \quad \in \quad \left\{\text{frac}\left(\frac{a'_s}{c}\right) \mid s \in S'\right\} \qquad (4.1)$$

where $\text{frac}(x) := x - \lfloor x \rfloor$ denotes the fractional part of $x \in \mathbb{R}$.

Proof of the claim: Note that the definition of $\sigma^*(S)$ implies that $\frac{1}{c}\left(a'_s - \sigma^*(S)\right)$ is an integer for at least one $s \in S$. For contradiction of the claim, we assume that $\frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right) \in \mathbb{Z}$ and $\frac{1}{c}\left(a'_s - \sigma^*(S)\right) \notin \mathbb{Z}$ for every $s \in S'$. Since $a'_{s_n} - \sigma^*(S) \geq a'_s - \sigma^*(S)$ for every $s \in S'$, this implies

$$\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor > \max\left\{\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S)\right)\right\rfloor \mid s \in S'\right\}$$

and hence

$$\sum_{s \in S} 2^{-\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S)\right)\right\rfloor} \leq 1 - 2^{-\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor}.$$

Now, for some sufficiently small $\epsilon > 0$, we obtain

$$\sum_{s \in S} 2^{-\left\lfloor \frac{1}{c}\left(a'_s - \left(\sigma^*(S)+\epsilon\right)\right)\right\rfloor} = 2^{-\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor + 1} + \sum_{s \in S'} 2^{-\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S)\right)\right\rfloor} \leq 1$$

which contradicts the definition of $\sigma^*(S)$ and completes the proof of the claim. $\square$

Let $T'_{(P1)}$ denote the tree produced by (P1) just before the insertion of the last sink s_n. By induction, $\sigma\left(T'_{(P1)}\right) = \sigma^*(S')$.
First, we assume that there is some sink $s' \in S'$ such that within $T'_{(P1)}$

$$|E[r, s']| - 1 < \left\lfloor \frac{1}{c}\left(a'_{s'} - \sigma^*(S')\right)\right\rfloor.$$

Choosing e_n as the edge of $T'_{(P1)}$ leading to s', results in a tree T such that

$$\sigma^*(S) \geq \sigma_{\text{opt}} \geq \sigma\left(T_{(P1)}\right) \geq \sigma(T) = \sigma^*(S') \geq \sigma^*(S),$$

which implies $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}} = \sigma^*(S)$.
Next, we assume that within $T'_{(P1)}$

$$|E[r, s]| - 1 = \left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S')\right)\right\rfloor$$

for every $s \in S'$. This implies

$$\sum_{s \in S} 2^{-\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S')\right)\right\rfloor} > \sum_{s \in S'} 2^{-\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S')\right)\right\rfloor} = 1$$

and hence $\sigma^*(S) < \sigma^*(S')$. By (4.1), we obtain

$$\begin{aligned}
\sigma^*(S) &\leq \max\left\{\sigma \mid \sigma < \sigma^*(S'),\ \operatorname{frac}\left(\frac{\sigma}{c}\right) \in \left\{\operatorname{frac}\left(\frac{a'_s}{c}\right) \mid s \in S'\right\}\right\} \\
&= \max\left\{\sigma \mid \sigma < \sigma^*(S'),\ \operatorname{frac}\left(\frac{\sigma - \sigma^*(S')}{c}\right) \in \left\{\operatorname{frac}\left(\frac{a'_s - \sigma^*(S')}{c}\right) \mid s \in S'\right\}\right\} \\
&= c \max\left\{x \mid x < \frac{\sigma^*(S')}{c},\ \operatorname{frac}\left(x - \frac{\sigma^*(S')}{c}\right) \in \left\{\operatorname{frac}\left(\frac{a'_s - \sigma^*(S')}{c}\right) \mid s \in S'\right\}\right\} \\
&= c\left(\frac{\sigma^*(S')}{c} - 1 + \max\left\{\operatorname{frac}\left(\frac{a'_s - \sigma^*(S')}{c}\right) \mid s \in S'\right\}\right) \\
&= \sigma^*(S') - c(1 - \delta)
\end{aligned}$$

for

$$\delta = \max \left\{ \operatorname{frac} \left(\frac{a'_s - \sigma^*(S')}{c} \right) \mid s \in S' \right\}.$$

If $s' \in S'$ is such that

$$\delta = \operatorname{frac} \left(\frac{a'_{s'} - \sigma^*(S')}{c} \right),$$

then choosing e_n as the edge of $T'_{(P1)}$ leading to s', results in a tree T such that

$$\sigma^*(S) \geq \sigma_{\text{opt}} \geq \sigma\left(T_{(P1)}\right) \geq \sigma(T) = \sigma^*(S') - c(1-\delta) \geq \sigma^*(S),$$

which implies $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}} = \sigma^*(S)$ and completes the proof. $\square$

Theorem 4.2. *(P2) generates a repeater tree topology T for which $l(T)$ is at most the total length of a minimum spanning tree on $\{r\} \cup S$ with respect to $||\cdot||$.*

Proof: Let $n = |S|$ and for $i = 0, 1, \ldots, n$, let $\overline{T}_i$ denote the forest which is the union of the tree produced by (P2) after the insertion of the first i sinks and the remaining $n-i$ sinks as isolated vertices. Note that $\overline{T}_0$ has vertex set $\{r\} \cup S$ and no edge, while for $1 \leq i \leq n$, $\overline{T}_i$ has vertex set $\{r\} \cup S \cup \{x_j \mid 2 \leq j \leq i\}$ and $2i-1$ edges.
Let $F_0 = (V(F_0), E(F_0))$ be a spanning tree on $V(F_0) = \{r\} \cup S$ such that

$$l(F_0) = \sum_{uv \in E(F_0)} ||u - v||$$

is minimum. For $i = 1, 2, \ldots, n$, let $F_i = (V(F_i), E(F_i))$ arise from

$$\left(V\left(\overline{T}_i\right), E(F_{i-1}) \cup E\left(\overline{T}_i\right)\right)$$

by deleting an edge $e \in E(F_{i-1}) \cap E(F_0)$ which has exactly one end-vertex in $V(T_{i-1})$ such that F_i is a tree. (Note that this uniquely determines F_i.)
Since (P2) has the freedom to use the edges of F_0, the specification of the insertion order and the locations of the internal vertices in (P2) imply that

$$l(F_0) \geq l(F_1) \geq l(F_2) \geq \ldots \geq l(F_n).$$

Since $F_n = T_n$ the proof is complete. $\square$

For the l_1-norm, the well-known result of Hwang [1976] together with Theorem 4.2 imply that (P2) is an approximation algorithm for the l_1-minimum Steiner tree on the set $\{r\} \cup S$ with approximation guarantee 3/2.
We have seen in Theorems 4.1 and 4.2 that different insertion orders are favorable for different optimization scenarios such as (O1) and (O2).
Alon and Azar [1993] gave an example showing that for the online rectilinear Steiner tree problem the best approximation ratio we can achieve is $\Theta(\log n / \log\log n)$, where n is the number of terminals. Hence inserting the sinks in an order disregarding the locations, like in (P1), can lead to long Steiner trees, no matter how we decide where to insert the sinks.
The next example shows that inserting the sinks in an order different from the one considered in (P1) but still choosing the edge e_i as in (P1) results in a repeater tree topology whose worst slack can be much smaller than the largest achievable worst slack.

Example 4.3. Let $c = 1$, $d = 0$ and $a \in \mathbb{N}$. We consider the following sequences of $-a$'s and 0's

$$\begin{aligned} A(1) &= (-a, 0), \\ A(2) &= (A(1), -a, 0), \\ A(3) &= (A(2), -a, \underbrace{0, \ldots\ldots, 0}_{1+(2^1-1)(a+2)}), \\ A(4) &= (A(3), -a, \underbrace{0, \ldots\ldots\ldots, 0}_{1+(2^2-1)(a+2)}), \ldots, \end{aligned}$$

i.e. for $l \geq 2$, the sequence $A(l)$ is the concatenation of $A(l-1)$, one $-a$, and a sequence of 0's of length $1 + \left(2^{l-2} - 1\right)(a+2)$.
If the entries of $A(l)$ are considered as the requires arrival times of an instance of the repeater tree topology problem, then Theorem 4.1 together with the choice of c and d imply that the largest achievable worst slack for this instance equals

$$\left\lfloor -\log_2 \left(l2^a + \left(1 + \sum_{i=2}^{l} \left(1 + (2^{i-2} - 1)(a+2)\right) \right) 2^0 \right) \right\rfloor .$$

For $l = a + 1$ this is at least $-2 - a - \log_2(a+2)$.
If we insert the sinks in the order as specified by the sequences $A(l)$, and always choose the edge into which we insert the next internal vertex such that the worst slack is maximized, then the following sequence of topologies can arise: $T(1)$ is the topology with two exactly sinks at depth 2. The worst slack of $T(1)$ is $-(a+2)$. For $l \geq 2$, $T(l)$ arises from $T(l-1)$ by (a) subdividing the edge of $T(l-1)$ incident with the root with a new vertex x, (b) appending an edge (x, y) to x, (c) attaching to y a complete binary tree B of depth $l-2$, (d) attaching to one leaf of B two new leaves corresponding to sinks with required arrival times $-a$ and 0, and (e) attaching to each of the remaining $2^{l-2} - 1$ many leaves of B a binary tree Δ which has $a+2$ leaves, all corresponding to sinks of arrival times 0, whose depths in Δ are $1, 2, 3, \ldots, a-1, a, a+1, a+1$. Note that this uniquely determines $T(l)$.
Clearly, the worst slack in $T(l)$ equals $-a-(l+1)$. Hence for $l = a+1$, the worst slack equals $-2a-2$, which differs approximately by a factor of 2 from the largest achievable worst slack as calculated above.

In the next section we show how the sinks can be inserted in order to maximize the slack, even if the sinks are in a different order from the one considered in (P1). However, it is an open question to find a bicriteria approximation algorithm, or an algorithm for (O3).

4.3 The Online Minimax Problem

Assume we want to find a repeater tree topology with optimum slack. In the last section we have shown how to compute such a tree. A disadvantage of the approach is that it does not consider the locations of the sinks and thus can compute trees that are much longer than necessary. In the variant (P1) the sinks are sorted according to their criticality and then inserted into the tree. But can we also achieve trees with optimum or nearly optimum slack if we have to insert the sinks in a given order (not sorted by their criticality)? If the answer of the question is yes we could sort the sinks according to their positions and then build the tree topology, hoping to get shorter trees that are still slack optimal. Moreover, we could allow that the trees are not optimal but to have a slack that is at most some constant c from the optimum. This could further reduce the length of the trees.

In this section we will show that there is an online algorithm that computes a slack optimal tree, i.e. the sinks are inserted into the tree in any given order. Moreover, we give necessary and sufficient conditions for an online algorithm to compute binary trees with a slack that is at least the optimum slack minus c for any constant $c \in \mathbb{N}$. But first we will reformulate the problem so that is in the form of the well-known minimax problem.

4.3.1 Introduction

Let $n \in \mathbb{N}$ and $A := (a_1, \dots, a_n)$ be a tuple with weights $a_i \in \mathbb{N}$, $1 \leq i \leq n$. For a binary tree T with n leaves let t_i be the depths of leaf i, $1 \leq i \leq n$. Then the weight of the tree T is defined as $s^T := \max_{1 \leq i \leq j}(t_i + a_i)$. The definition of weight can be extended to the internal vertices of T. In this case the weight of a vertex is the maximum weight of its two children plus 1. Obviously, s^T is the weight of the root of T. A binary tree that minimizes s^T is called binary minimax tree. Analogously to the proof of Theorem 4.1, by Kraft's inequality (Kraft [1949]), there exists a binary tree T with leaves at depth at most $t_1, \dots, t_n$ if and only if $\sum_{i=1}^{n} 2^{-t_i} \leq 1$. Hence $s^T \geq OPT := \lceil \log_2 \sum_{i=1}^{n} 2^{a_i} \rceil$.

Note that minimizing the depths s^T is equivalent to maximizing the slack in the formulation of the previous section.

We use the following variant of the algorithm from the previous section, adapted to the formulation of the minimax problem: The algorithm starts with the binary tree with leaves 1 and 2. At step $j \in \{3, \dots, n\}$ one of the two following operations are allowed: The first operation is to insert leaf j at an edge e of T by subdividing e and inserting a branch that ends at leaf j. The second operation is to insert the leaf j at the root of T by creating a new root s with two branches, one ending in the original tree T and the other ending in j.

We are interested in the online version of this algorithm: When inserting leaf j we neither know n nor the following weights $a_{j+1}, \dots, a_n$. An open question of the last section is whether there are online algorithms that always compute binary trees of weight $OPT + c$ for an non-negative integral constant c. We will show in this section

that such algorithms exist for any c and we will give necessary and sufficient conditions for them that can be verified efficiently.
The non-online version of the minimax tree problem is well studied. Minimax trees are used for data compression and prefix-codes (Drmota and Szpankowski [2002, 2004]) and some applications in VLSI design to limit circuit fan-ins and fan-outs (Parker [1979], Hoover et al. [1984]). Golumbic [1976] first showed that it can be solved optimally in $O(n \log n)$ using a variant of Huffman's algorithm (Huffman [1952], see also Section 4.2). In the algorithm repeatedly two vertices with minimum weights a_i and a_j are replaced by a new vertex of weight $\max\{a_i, a_j\} + 1$. Recently, Gawrychowski and Gagie [2008] showed that the algorithm can be modified to run in linear time. To our knowledge, the online version of this problem has not be studied yet.

4.3.2 Preliminaries

For a tuple $A = (a_1, \dots, a_j)$ and a binary tree T with sinks $1, \dots, j$ we recursively define a tuple B. Initially set $T_1 := T$ and $B := \emptyset$. For $i = 1, \dots, j$ let b_i the largest integer so that b_i can be inserted in the tree T_i without increasing the weight of it. If no leaf can be inserted we are done. Otherwise let T_{i+1} be the resulting tree and set $B := (b_1, \dots, b_i)$. We call B the filler tuple of (A, T).

Constructing the Filler Tuple B.

Now we describe a method to construct the filler tuple B.
If T only consists of a single vertex we are done as we cannot add another vertex. Otherwise let s be the root, r and l the two children of s and T_r and T_l be the subtrees rooted at r and l, respectively (see Figure 4.2). Then $s^T = 1 + \max\{s^{T_r}, s^{T_l}\}$. W.l.o.g. $s^T = s^{T_l} + 1$.

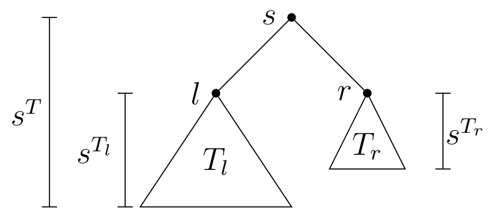

Figure 4.2: A tree T with root r and the sub-trees T_l and T_r.

Obviously, no weight $b \geq s^T - 1$ can be inserted without increasing the weight of T. If $s^{T_r} = s^{T_l}$ we could not even insert $b = s^T - 2$. If $s^{T_r} < s^{T_l}$ we can insert $b = s^T - 2$ at the edge (s, r), i.e. subdividing (s, r) by inserting a new vertex r' and adding a branch that ends at leaf $b = s^T - 2$ (see Figure 4.3) and this is the only possible choice. The root of the new tree T' is driving the sub-trees T_l and T'_r. Observe that $s^{T'_r} = 1 + \max\{s^{T_r}, b\} = 1 + s^T - 2 = s^T - 1 = s^{T_l}$. This implies that the edges (s, l) and (s, r') cannot be subdivided without increasing the weight of the tree. Subdividing an edge of one of the sub-trees does not affect the other tree. Thus the filler tuple of T' is the ordered union of the filler tuples of T_l and

T'_r. (The ordered union of two tuples $C = (c_1, \dots, c_k)$ and $D = (d_1, \dots, d_l)$ is the tuple $(e_{\tau(1)}, \dots, e_{\tau(k+l)})$ with $e_1 = c_1, \dots, e_k = c_k, e_{k+1} = d_{k+1}, \dots, e_{k+l} = d_{k+l}$, where $\tau : \{1, \dots, k+l\} \to \{1, \dots, k+l\}$ is a bijective mapping with $e_{\tau(i)} \geq e_{\tau(i+1)}$ for $1 \leq i \leq k+l-1$.)

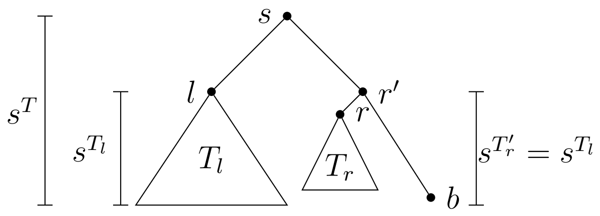

Figure 4.3: The tree after subdividing (s, r) and inserting a branch ending in b.

The construction immediately yields:

Proposition 4.4. *The filler tuple of (A, T) is unique.* □

Moreover, the filler tuple 'fills up' the tree:

Proposition 4.5. $\sum_{a \in A} 2^a + \sum_{b \in B} 2^b = 2^{s^T}$.

Proof. If $\sum_{a \in A} 2^a + \sum_{b \in B} 2^b > 2^{s_{T_i}}$ the weight of the resulting tree would be greater than s^T. That is a contradiction to the definition of B. If $\sum_{a \in A} 2^a + \sum_{b \in B} 2^b < 2^{s^T}$ we can insert at least weight $b = 0$ without increasing the weight of the tree. □

Some Properties of B.

In the following section we always have a tuple $A = (a_1, \dots, a_{|A|})$, a corresponding binary tree T and the filler tuple $B = (b_1, \dots, b_{|B|})$ of (A, T).

Proposition 4.6. *Let $a > \max B$ and T' the tree obtained by inserting weight a at the root of T. Let B' be the filler tuple of $((a_1, \dots, a_{|A|}, a), T')$.*

a) If $a < s^T$ then $B' = (s^T - 1, s^T - 2, \dots, a + 1, a, b_1, \dots, b_{|B|})$.

b) If $a > s^T$ then $B' = (a - 1, a - 2, \dots, s^T + 1, s^T, b_1, \dots, b_{|B|})$.

c) If $a = s^T$ then $B' = B$.

Proof. Simple application of the construction of B. □

Proposition 4.7. *Let $a \in B$ and T' be the tree obtained by inserting weight a at an edge where one of the $b_i \in B$ with $b_i = a$ has been inserted. Then $B' = (b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_{|B|})$ is the filler tuple of $((a_1, \dots, a_{|A|}, a), T')$.*

Proof. We look at the tree T' of the previous construction after inserting b_i at edge (s, l). Denote by l' the inserted vertex, by T_s the subtree rooted at s and let r be the other child of s beside l' (see Figure 4.4). When we replace b_i by a the tree does not change and we get the same filler tuple for the remaining tree. Thus $B' = (b_1, \dots, b_{i-1}, b_{i+1}, \dots, b_{|B|})$. □

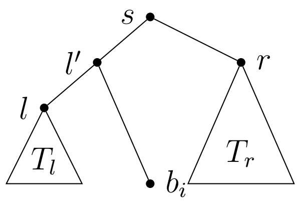

Figure 4.4: The subtree rooted at s after inserting b_i.

Proposition 4.8. *Let a be an integer with $a \notin B$ and $a \leq \max B$. Let $b_i \in B$ be a number so that $b_i > a$ and there is no $b \in B$ with $b_i > b > a$. Let T' be the tree obtained by inserting weight a at the edge where b_i has been inserted. Then $B' = (b_1, \dots, b_{i-1}, b_i - 1, b_i - 2, \dots, a+1, a, b_{i+1}, \dots, b_{|B|})$ is the filler tuple of $((a_1, \dots, a_{|A|}, a), T')$.*

Proof. Again look at the tree T' of the above construction after inserting b_i at edge (s, l). Denote by l' the inserted vertex, by T_s the subtree rooted at s and let r be the other child of s beside l' (see Figure 4.4). The filler tuple of T' is the ordered union of the filler tuples T_l and T_r.

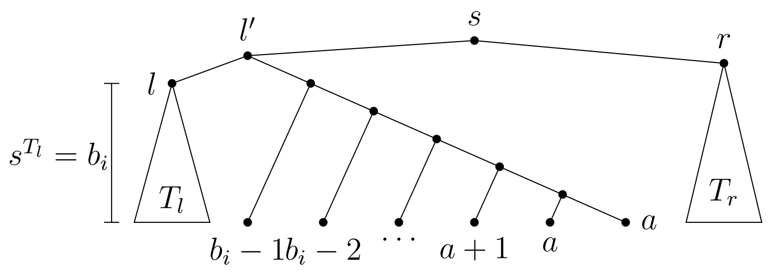

Figure 4.5: The subtree rooted at s after inserting a and the fillers $b_i - 1, b_i - 2, \dots, a$.

As $a \notin B$ and as there is no $b \in B$ with $b_i > b > a$ we conclude $b_i - 1 \notin B$. Thus the edge (l', l) cannot be subdivided and the weight of the subtree T_l rooted at l has to be b_i.

Now we replace b_i by a. The edges (s, l'), (l', l) and (s, r) still cannot be subdivided. But as $s^{T_l} = b_i$ we have to insert $b_i - 1$ at the edge (l', a). Let x' be the vertex we got by subdividing (l', a). x' is driving two subtrees consisting of just one vertex. One has weight $b_i - 1$ the other one weight a. If $b_i - 1 > a$ we can insert $b_i - 2$ at the edge (x', a) and continue by inserting $b_i - 3, \dots, a+1, a$ (see Figure 4.5). The subtrees T_l and T_r are not affected by these insertions. We conclude $B' = (b_1, \dots, b_{i-1}, b_i - 1, b_i - 2, \dots, a+1, a, b_{i+1}, \dots, b_{|B|})$. □

4.3.3 Conditions for an Optimal Online Algorithms

An online algorithm is optimal if and only if for any A and computed tree T it guarantees

$$
\begin{aligned}
\log_2\left(\sum_{a\in A} 2^a + \sum_{b\in B} 2^b\right) &\overset{\text{Prop.4.5}}{=} \log_2\left(2^{s^T}\right) \\
&= s^T \\
&\leq OPT \\
&= \left\lceil \log_2\left(\sum_{a\in A} 2^a\right)\right\rceil \\
&< \log_2\left(\sum_{a\in A} 2^a\right) + 1 \\
&= \log_2\left(2\cdot\sum_{a\in A} 2^a\right).
\end{aligned}
$$

This is equivalent to

$$\sum_{b\in B} 2^b \;<\; \sum_{a\in A} 2^a. \tag{4.2}$$

Let $b_1 \geq b_2 \geq \ldots \geq b_{|B|}$ be the elements of B. Now assume that the next k weights ($1 \leq k \leq |B|$) of the input are $b_1, b_2, \ldots, b_{k-1}, b_k + 1$ (in this order). By the definition of B the first $k-1$ numbers can be inserted into T without increasing its weight. But inserting the last number enforces that the weight of the tree increases by at least one. We have to ensure that also the weight of this tree is optimal:

$$
\begin{aligned}
\log_2\left(\sum_{a\in A} 2^a + \sum_{b\in B} 2^b\right) + 1 &\overset{\text{Prop.4.5}}{=} \log_2\left(2^{s^T+1}\right) \\
&= s^T + 1 \\
&\leq OPT' \\
&= \left\lceil \log_2\left(\sum_{a\in A} 2^a + \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}\right)\right\rceil \\
&< \log_2\left(\sum_{a\in A} 2^a + \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}\right) + 1.
\end{aligned}
$$

And thus

$$\sum_{i=k+1}^{|B|} 2^{b_i} \;<\; 2^{b_k}. \tag{4.3}$$

Inequalities (4.2) and (4.3) are necessary conditions for an online algorithm to produce for each A a tree T with $s^T = OPT$. We claim that these conditions are also sufficient.

First note that (4.2) directly follows from (4.3) and can be neglected. b_1 is the biggest number that can be inserted into T without increasing s^T. Thus $s^T \geq b_1 + 2$. We conclude using Proposition 4.5 and (4.3):

$$\sum_{a\in A} 2^a = 2^{s^T} - \sum_{b\in B} 2^b \geq 2^{b_1+2} - \sum_{b\in B} 2^b > 2 \cdot \sum_{b\in B} 2^b - \sum_{b\in B} 2^b = \sum_{b\in B} 2^b.$$

To show that (4.3) is also sufficient we just have to prove the following:

Lemma 4.9. *For any pair A, T satisfying (4.3) and any weight $a \in \mathbb{N}$ we can insert weight a into T creating a new tree T' so that (4.3) is kept for $(a_1, \ldots, a_{|A|}, a)$ and T'.*

Proof. It is easy to see that inequality (4.3) is true if and only if B does not contain two number with the same value. Now assume we have A, T and the filler tuple B where every number occurs at most one time. Let $a \in \mathbb{N}$.
If $a > \max B$ we insert a at the root of T. Then by Proposition 4.6 the filler tuple B' of $(a_1, \ldots, a_{|A|}, a)$ and the new tree T' does not contain a number more than one time.
If $a \in B$ we insert a at an edge where we have inserted the unique element $b_i \in B$ with $b_i = a$. By Proposition 4.7 the resulting filler tuple B' is a subset of B and does not contain duplicate numbers.
Finally, if $a \leq \max B$ and $a \notin B$ we choose $b_i \in B$ so that $b_i > a$ and there is no $b \in B$ with $b_i > b > a$. Inserting a at the position of b_i we get by Proposition 4.8 the filler tuple $B' = (b_1, \ldots, b_{i-1}, b_i - 1, b_i - 2, \ldots, a+1, a, b_{i+1}, \ldots, b_{|B|})$. By the choice of b_i the tuple B' does not contain a number twice. □
The proof of the Lemma immediately gives an optimal online algorithm.

4.3.4 Conditions for an Online Algorithms with $s^T \leq OPT + c$

Now we turn to the case that we aim for a tree that need not be optimal, but near-optimal. An online algorithm constructs trees of weight at most $OPT + c$ if and only if for any (A, T) and corresponding filler tuple B the following inequality is kept.

$$\begin{aligned}
\log_2\left(\sum_{a\in A} 2^a + \sum_{b\in B} 2^b\right) &\overset{\text{Prop.4.5}}{=} \log_2\left(2^{s^T}\right) \\
&= s^T \\
&\leq OPT + c \\
&= \left\lceil \log_2\left(\sum_{a\in A} 2^a\right)\right\rceil + c \\
&< \log_2\left(\sum_{a\in A} 2^a\right) + c + 1.
\end{aligned}$$

This is equivalent to

$$\sum_{a\in A} 2^a + \sum_{b\in B} 2^b < 2^{c+1} \cdot \sum_{a\in A} 2^a. \tag{4.4}$$

Again let $b_1 \geq b_2 \geq \ldots \geq b_{|B|}$ the elements of B. Assume that $b_1, b_2, \ldots, b_{k-1}, b_k + 1$ (in this order) are the next numbers of the input of the algorithm. By the definition of the filler tuple B the first $k-1$ numbers can be inserted into T without increasing s^T, but the k'th number enforces that the weight of the tree increases by 1. We have to ensure that the weight of this new tree ($= s^T + 1$) is at most $OPT' + c$ where OPT' is the optimum weight for a tree for weights $(a_1, \ldots, a_{|A|}, b_1, \ldots, b_k)$. We conclude that the following inequality has to be satisfied for all $1 \leq k \leq |B|$:

$$\begin{aligned}
\log_2\left(\sum_{a\in A} 2^a + \sum_{b\in B} 2^b\right) + 1 &\overset{\text{Prop.4.5}}{=} s^T + 1 \\
&\leq OPT' + c \\
&= \left\lceil \log_2\left(\sum_{a\in A} 2^a + \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}\right)\right\rceil + c \\
&< \log_2\left(\sum_{a\in A} 2^a + \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}\right) + c + 1.
\end{aligned}$$

In short we have

$$\sum_{a\in A} 2^a + \sum_{b\in B} 2^b < 2^c\left(\sum_{a\in A} 2^a + \min_{k=1}^{|B|}\left\{\sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}\right\}\right). \tag{4.5}$$

Let k_{min} be the minimum index with $b_{k_{min}} = b_{k_{min}+1}$. If no two b_i have the same value set $k_{min} = |B|$.

Lemma 4.10. *The minimum in (4.5) is obtained for $k = k_{min}$.*

Proof. For simplicity of notation set $v(k) = \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1}$. We have to show $v(k_{min}) \leq v(k)$ for all $1 \leq k \leq |B|$. Let $k < k_{min}$. By the definition of k_{min} we know $b_k > b_{k+1}$. We conclude

$$v(k) = \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1} = \sum_{i=1}^{k} 2^{b_i} + 2^{b_k} \geq \sum_{i=1}^{k} 2^{b_i} + 2\cdot 2^{b_{k+1}} = v(k+1).$$

If $k = k_{min} + 1$ we get, using the definition of k_{min},

$$v(k_{min}+1) = \sum_{i=1}^{k_{min}} 2^{b_i} + 2^{b_{k_{min}+1}+1} > \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} = v(k_{min}).$$

Finally, let $k > k_{min} + 1$. Then

$$v(k) = \sum_{i=1}^{k-1} 2^{b_i} + 2^{b_k+1} = \sum_{i=1}^{k_{min}} 2^{b_i} + \sum_{i=k_{min}+1}^{k-1} 2^{b_i} + 2^{b_k+1}$$

$$\geq \sum_{i=1}^{k_{min}} 2^{b_i} + 2^{b_{k_{min}+1}} = \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} = v(k_{min}).$$

□

By Lemma 4.10 inequality (4.5) is satisfied if and only if

$$2^c \left(\sum_{a \in A} 2^a + \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} \right) > \sum_{a \in A} 2^a + \sum_{b \in B} 2^b. \tag{4.6}$$

(4.4) and (4.6) are necessary conditions for the online algorithm to find a tree of weight at most $OPT + c$ for any input.
To show that (4.4) and (4.6) are sufficient we have to prove the following:

Lemma 4.11. *For any A, T satisfying (4.4) and (4.6), any number $x \in \mathbb{N}$ can be inserted into T, creating T', so that $(a_1, \dots, a_{|A|}, x), T'$ also satisfy (4.4) and (4.6).*

Proof. Let A, T a pair satisfying (4.4) and (4.6) and B its filler tuple. Set k_{min} as above. Let $x \in \mathbb{N}$. We will insert x into T and construct the tree T'. By B' we denote the filler tuple of $(a_1, \dots, a_{|A|}, x)$ and T'.
Let b_{min} be the minimum number that appears at least twice in B' or $b_{min} = \min B'$ if all numbers only appear once. Thus to show (4.6) for A', T' we have to prove

$$2^c \left(\sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B', b > b_{min}} 2^b + 2^{b_{min}+1} \right) > \sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b. \tag{4.7}$$

Case 1: $x > \max B$. We insert x at the root of T. By Proposition 4.6 B' is the ordered union of C and B with

- $C = \{s^T - 1, s^T - 2, \dots, x + 1, x\}$ if $x < s^T$,
- $C = \{x - 1, x - 2, \dots, s^T + 1, s^T\}$ if $x > s^T$ or
- $C = \emptyset$ if $x = s^T$.

In all three sub-cases the numbers of C only appear once and are greater than $\max B$. Thus $b_{min} = b_{k_{min}}$ and we conclude

$$\begin{aligned}
& 2^c \left(\sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in C} 2^b + \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} \right) \\
= \; & 2^c \left(2^x + \sum_{b \in C} 2^b \right) + 2^c \left(\sum_{a \in A} 2^a + \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} \right) \\
> \; & 2^x + \sum_{b \in C} 2^b + \sum_{a \in A} 2^a + \sum_{b \in B} 2^b \\
= \; & \sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b.
\end{aligned}$$

Thus (4.7) is satisfied. Inequality (4.4) is equivalent to $OPT + c \geq s^{T'}$. If $x \geq s^T$ then $OPT = \lceil \log_2(\sum_{a \in A} 2^a + 2^x) \rceil = x + 1 = s^{T'}$ and we are done. Otherwise $x < s^T$. Using $x > b_1$ and (4.5) we conclude

$$\begin{aligned}
2^{c+1} \sum_{a \in A \cup \{x\}} 2^a &= 2^{c+1} \left(\sum_{a \in A} 2^a + 2^x \right) \\
&\geq 2^{c+1} \left(\sum_{a \in A} 2^a + 2^{b_1 + 1} \right) \\
&\geq 2 \cdot \left(\sum_{a \in A} 2^a + \sum_{b \in B} 2^b \right) \\
&= \sum_{a \in A} 2^a + \sum_{b \in B} 2^b + 2^{s^T} \\
&= \sum_{a \in A} 2^a + \sum_{b \in B} 2^b + 2^x + \sum_{i=x}^{s^T - 1} 2^i \\
&= \sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b.
\end{aligned}$$

Case 2: $x \in B$: We insert x at an edge where one of the $b_i \in B$ with $b_i = x$ has been inserted. Then by Proposition 4.7 we have $B' = (b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_{|B|})$. (4.4) is clearly satisfied. We removed one element from B. So if $b_i = b_{min}$ then $b_{min} \leq b_{k_{min}}$ otherwise $b_{min} = b_{k_{min}}$ and we conclude

$$\begin{aligned}
& 2^c \left(\sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B', b > b_{min}} 2^b + 2^{b_{min}+1} \right) \\
= {} & 2^c \left(\sum_{a \in A} 2^a + 2^{b_i} + \sum_{b \in B', b > b_{min}} 2^b + 2^{b_{min}+1} \right) \\
\geq {} & 2^c \left(\sum_{a \in A} 2^a + \sum_{i=1}^{k_{min}-1} 2^{b_i} + 2^{b_{k_{min}}+1} \right) \\
> {} & \sum_{a \in A} 2^a + \sum_{b \in B} 2^b \\
= {} & \sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b.
\end{aligned}$$

Case 3) $x \notin B, x \leq \max B$. Let $b_i \in B$ be a number so that $b_i > x$ and there is no $b \in B$ with $b_i > b > x$. Let T' be the tree obtained by inserting x at the edge where b_i has been inserted. By Proposition 4.8 $B' = (b_1, \ldots, b_{i-1}, b_i - 1, b_i - 2, \ldots, x + 1, x, b_{i+1}, \ldots, b_{|B|})$. Set $C := \{b_i - 1, b_i - 2, \ldots, x + 1, x\}$. First observe that

$$\sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b = \sum_{a \in A} 2^a + 2^x - \sum_{b \in B} 2^b - 2^{b_i} + \sum_{i=x}^{b_i - 1} 2^i = \sum_{a \in A} 2^a + \sum_{b \in B} 2^b.$$

Moreover, note that all numbers of C only appear once in B'. If $b_{k_{min}} > b_i$ or $b_{k_{min}} = b_i$ and there are at least three elements in B with value b_i it is easy to verify that (4.7) is satisfied. Otherwise $b_{k_{min}} = b_{min} < x$ and we get

$$\begin{aligned}
& 2^c \left(\sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B', b > b_{min}} 2^b + 2^{b_{min}+1} \right) \\
= \; & 2^c \left(\sum_{a \in A} 2^a + 2^x + \sum_{b \in B, b > b_{k_{min}}} 2^b + 2^{b_{min}+1} - 2^{b_i} + \sum_{i=x}^{b_i - 1} 2^i \right) \\
= \; & 2^c \left(\sum_{a \in A} 2^a + \sum_{b \in B, b > b_{k_{min}}} 2^b + 2^{b_{min}+1} \right) \\
> \; & \sum_{a \in A} 2^a + \sum_{b \in B} 2^b \\
= \; & \sum_{a \in A \cup \{x\}} 2^a + \sum_{b \in B'} 2^b .
\end{aligned}$$

This completes the proof. □

4.3.5 Algorithms

In the last two sections we established necessary and sufficient conditions for online algorithms to be satisfied in order to compute trees of weight at most $OPT + c$. These conditions yield such algorithms at once. A straight forward implementation might result in excessive running times. If for example $a_1 = 0$ and $a_2 = 1\,000\,000$ we get a tree consisting of three vertices and two edges and $B = (999\,999, 999\,998, \ldots, 1, 0)$, i.e. B contains $1\,000\,000$ elements. In this section we will show how to get a strongly polynomial running time by handling the filler tuples efficiently.

To this end, we denote for each edge $e \in E(T)$ by B_e^T the tuple of fillers in decreasing order that are inserted at e. Let $v \in V(T)$ be a vertex with two successors r and l and let T_v, T_r, T_l be the subtrees rooted at v, r and l. We have seen that $s^{T_v} = \max\{s^{T_r}, s^{T_l}\} + 1$. W.l.o.g. $s^{T_v} = s^{T_r} + 1$. We do not insert any filler at the edge (v, r) and thus $B_{(v,r)}^T = \emptyset$. If $s^{T_v} = s^{T_l} + 1$ then also $B_{(v,l)}^T = \emptyset$. Otherwise we will insert the fillers $s^{T_v} - 2, s^{T_v} - 3, \ldots, s^{T_l}$ at (v, l) and get $B_{(v,l)}^T = (s^{T_v} - 2, s^{T_v} - 3, \ldots, s^{T_l})$. For simplicity of notation we denote by $\overline{b_e^T}$ the largest and by $\underline{b_e^T}$ the smallest element in B_e^T. Thus $B_e^T = \left(\overline{b_e^T}, \overline{b_e^T} - 1, \ldots, \underline{b_e^T} + 1, \underline{b_e^T} \right)$.

Lemma 4.12. *For a given tuple of weights A and a corresponding tree T the tuples B_e^T, $e \in E(T)$, can be computed in time $O(n)$.*

Proof. The B_e^T's can be computed by traversing the edges of T bottom-up, computing s^{T_v} for every vertex $v \in V(T)$ and setting B_e^T for each edge as described above. □

Optimal Online Algorithm.

Now we will discuss an online algorithm that computes a tree of optimum weight. Assume we have given A, T and $\{B_e^T\}_{e\in E(T)}$. The edges $\{e_1, \ldots, e_m\}$ of T are ordered so that $\overline{b^T_{e_1}} \geq \overline{b^T_{e_1}} \geq \ldots \geq \overline{b^T_{e_m}}$. Let $x \in \mathbb{N}$ be the next weight in the input. We will insert it according to Proposition 4.6, 4.7 and 4.8, creating the tree T', and show that the filler tuples $B_e^{T'}$ can be computed efficiently.
If $x > \max B$ we add x at the root, i.e. we insert a new root v with two edges ending in r and l, add T at l and assign weight x to r (see Proposition 4.6). Then for all $e \in E(T)$, B_e^T does not change. For the edges (v,l) and (v,r) we get

- if $x < s^T$ then $B^{T'}_{(v,l)} = \emptyset$ and $B^{T'}_{(v,r)} = (s^T, s^T - 1, \ldots, x)$,
- if $x > s^T$ then $B^{T'}_{(v,l)} = (x, x-1, \ldots, s^T)$ and $B^{T'}_{(v,r)} = \emptyset$ and
- if $x = s^T$ then $B^{T'}_{(v,l)} = B^{T'}_{(v,r)} = \emptyset$.

If $x \in B$ then let $b_i \in B$ an element with $x = b_i$ and we insert x at the edge (v,w) where b_i has been inserted (see Proposition 4.7). In this case we intersect (v,w) by a new vertex s and add an edge (s,t). By the choice of b_i we know $b_i \in B^T_{(v,w)} = \left(\overline{b^T_{(v,w)}}, \overline{b^T_{(v,w)}} - 1, \ldots, \underline{b^T_{(v,w)}}\right)$. We get $B^{T'}_{(v,s)} = \left(\overline{b^T_{(v,w)}}, \overline{b^T_{(v,w)}} - 1, \ldots, b_i + 1\right)$, $B^{T'}_{(s,w)} = \left(b_i - 1, b_i - 2, \ldots, \underline{b^T_{(v,w)}}\right)$ and $B^{T'}_{(s,t)} = \emptyset$.
If finally $x \leq \max B$ and $x \notin B$ then we insert x at the unique edge (v,w) with $b_i \in B^T_{(v,w)}$ where b_i is the smallest integer with $b_i \in B$ and $b_i > x$ (see Proposition 4.8). In this case we intersect (v,w) by a vertex s and insert the edge (s,t). By the choice of b_i we have $B^{T'}_{(v,w)} = \left(\overline{b^T_{(v,w)}}, \overline{b^T_{(v,w)}} - 1, \ldots, b_i\right)$ and get

$$B^{T'}_{(v,s)} = \left(\overline{b^T_{(v,w)}}, \overline{b^T_{(v,w)}} - 1, \ldots, b_i + 1\right), B^{T'}_{(s,w)} = \emptyset$$

and

$$B^{T'}_{(s,t)} = \left(\underline{b^T_{(v,w)}} - 1, \underline{b^T_{(v,w)}} - 2, \ldots, x\right).$$

Lemma 4.13. *The online algorithm that inserts a new element x according to Proposition 4.6, 4.7 and 4.8 always computes an optimum solution and can be implemented to run in time $O(n \log n)$ with $n := |A|$.*

Proof. The optimality is given by Lemma 4.9. By the previous observations inserting $x \in \mathbb{N}$ at an appropriate edge e_x and updating the B_e^T's can be done in constant time. But how to find the edge e_x efficiently? As shown in the proof of Lemma 4.9 all numbers b_i^T, $1 \leq i \leq |B^T|$, are different. Thus we can sort the tuples B_e^T with $e \in E(T)$ and $B_e^T \neq \emptyset$ by setting $B_e^T < B_f^T$ if and only if $\overline{b_e^T} < \overline{b_f^T}$. We save the tuples B_e^T with $B_e^T \neq \emptyset$ in an ordered list and can find for any x an appropriate edge e_x by binary search in $O(\log n)$. By each of the three operations the filler tuples B_e^T of at most three edges change. It is easy to see that in all three cases the ordered list can be updated in constant time. $\square$

$OPT + c$ **Online Algorithm.**

In the case that the trees are allowed to have a weight of at most c more than the optimum weight we want to check efficiently if we can insert a new weight $x \in \mathbb{N}$ at an edge e' of an existing tree. To this end, we insert x at e' (creating the tree T'), compute the tuples B_e^T for all $e \in E(T')$ and finally check if the inequalities (4.4) and (4.6) are satisfied. First we handle the latter inequality.
We sort the edges of T so that $B_{e_1}^T \geq B_{e_2}^T \geq \ldots \geq B_{e_{n-1}}^T$. Then we have to compute $b_{k_{min}}$, the biggest number in the filler tuple B that appears at least twice. This number is easy to find. Obviously, we have to look for the smallest number $k \in \{2, \ldots, n-1\}$ so that $\overline{b_{e_k}^T} \in B_{e_{k-1}}^T$. If no such integer exists no two numbers in the filler tuple B are the same. But in this case (4.6) is always satisfied. Otherwise we have

$$(b_1, \ldots, b_{k_{\min}}) = (\overline{b_1}, \ldots, \underline{b_1}, \overline{b_2}, \ldots, \underline{b_2}, \ldots, \overline{b_{k-1}}, \ldots, \overline{b_k}, \overline{b_k}).$$

Moreover, note that $\sum_{b \in B_e^T} 2^e = 2^{\overline{b_e^T}} + 2^{\overline{b_e^T}-1} + \ldots + 2^{\underline{b_e^T}+1} + 2^{\underline{b_e^T}} = 2^{\overline{b_e^T}+1} - 2^{\underline{b_e^T}}$. Thus we can rewrite (4.6) as

$$\sum_{a \in A} 2^a + \sum_{i=1}^{k-2} \left(2^{\overline{b_i}+1} - 2^{\underline{b_i}}\right) + \left(2^{\overline{b_{k-1}}+1} - 2^{\overline{b_k}}\right) + 2^{\overline{b_k}} \quad > \quad 2^{s^T - c}$$

This is equivalent to

$$\sum_{a \in A} 2^a + \sum_{i=1}^{k-1} 2^{\overline{b_i}+1} \quad > \quad 2^{s^T-c} + \sum_{i=1}^{k-2} 2^{\underline{b_i}} \tag{4.8}$$

Moreover, (4.4) is equivalent to

$$\sum_{a \in A} 2^a \quad > \quad 2^{s^T-c-1} \tag{4.9}$$

On both sides of the inequalities (4.8) and (4.9) we have the sum of $O(n)$ integral non-negative powers of two ($n := |A|$). We can verify in $O(n \log n)$ time if these inequalities are satisfied: For both sides of an inequality we create an ordered list of the exponents. Going through the lists in non-decreasing order we replace numbers a that appear twice by one $a+1$. Then we have to check if the list belonging to the left side of the inequality is lexicographically bigger than the right one.

Lemma 4.14. *For a constant $c \in \mathbb{N}$, a pair (A, T) with $s^T \leq OPT + c$, a weight $x \in \mathbb{N}$ and an edge $e \in E(T)$ it can be verified in $O(n \log n)$, $n := |A|$, if x can be inserted at e without violating (4.4) and (4.6).*

4.4 Unbalanced Binary Trees with Choosable Edge Length

This section is based on joint work with Dieter Rautenbach (see Maßberg and Rautenbach [2009]).
In Section 4.1 we have already mentioned that within a repeater tree there is a certain degree of freedom how to distribute the additional delay caused by a bifurcation to the two branches. The reason for this is that one can insert inverters in such a way that they shield off the capacitance load of one of the branches increasing the delay of that branch but reducing the delay of the other branch. This motivates the following definition of $\mathcal{L}$-trees.
We consider rooted strict binary trees T with the property that the two edges leading from every non-leaf to its two children are assigned lengths l_1 and l_2 with $\{l_1, l_2\} \in \mathcal{L}$ for some fixed set $\mathcal{L} \subseteq \binom{\mathbb{R}_{\geq 0}}{2}$ containing two-element sets of non-negative real numbers. We will use the term $\mathcal{L}$-trees for such trees. The depth of a vertex in an $\mathcal{L}$-tree T equals its distance from the root of T with respect to the assigned edge lengths and the depth of an $\mathcal{L}$-tree equals the maximum depth of a leaf. Note that for $\mathcal{L} = \{\{1,1\}\}$ the notion of depth in $\mathcal{L}$-trees coincides with the usual notion of depth in binary trees.
We are mainly interested in sets $\mathcal{L}$ for which $l_1 + l_2$ is constant for all $\{l_1, l_2\} \in \mathcal{L}$ which models that some constant total length can be distributed in certain ways to the pairs of edges leading to the children of non-leaves. Specifically, for an integer $n \geq 3$ we study a continuous version

$$\mathcal{L}(n) = \left\{ \{y, 1-y\} \mid \frac{1}{n} \leq y \leq 1 - \frac{1}{n}, y \in \mathbb{R} \right\}$$

and a discrete version

$$\mathcal{L}'(n) = \{\{i, n-i\} \mid 1 \leq i \leq n-1, i \in \mathbb{N}\}.$$

For some $d \in \mathbb{R}$ let $f_{\mathcal{L}}(d)$ denote the maximum number of leaves in an $\mathcal{L}$-tree of depth at most d. We are interested in the growth of $f_{\mathcal{L}}$. Furthermore, we are interested in an appropriate version of Kraft's inequality: For given values $d_1, d_2, \ldots, d_n \in \mathbb{R}_{\geq 0}$ we want to characterize / algorithmically decide the existence of an $\mathcal{L}$-tree with n leaves having depths at most $d_1, d_2, \ldots, d_n$.
In the Section 4.4.1 we present results on the asymptotic growth of $f_{\mathcal{L}(n)}$. In Section 4.4.2 we study the existence of $\{\{1, m\}\}$-trees for $m \in \mathbb{N}$ and $\mathcal{L}'(4)$-trees. We pose several related conjectures.

Binary trees with unequal edge lengths (Kapoor and Reingold [1989]), the related recursions (Hwang and Tsai [2003]), Kraft's inequality (Kraft [1949], McMillan [1956]), and its variants mainly occur in contexts such as coding theory (Gallager [1968]), combinatorial search (Aigner [1988]), divide-and-conquer strategies, and analysis of running times of algorithm (Sedgewick and Flajolet [1996]). In these contexts the liberty of distributing a total length budget to different edges makes no obvious sense.

The results of Section 4.4.1 allow us to evaluate a lower bound on the achievable delay and to quantify the quality of the final repeater tree with respect to timing, and the results Section 4.4.2 enable the construction of repeater tree topologies with good timing behavior.

4.4.1 Growth of $f_{\mathcal{L}(n)}$

Throughout this section we assume that some integer $n \geq 3$ has been fixed and abbreviate $f_{\mathcal{L}(n)}$ as f. Since every $\mathcal{L}(n)$-tree with at least two leaves has at least one leaf at depth at least $\frac{1}{2}$, the definition of f immediately implies the following recursion

$$f(d) \quad = \quad \begin{cases} 0, & d < 0\,, \\ 1, & 0 \leq d < \frac{1}{2} \end{cases}$$

and $f(d) =$

$$\max_{\frac{1}{n} \leq y \leq 1-\frac{1}{n}} \left(f(d-y) + f(d-(1-y))\right) \tag{4.10}$$

for $d \geq \frac{1}{2}$.
Clearly, f is a monotonously increasing step function, i.e. there is a sequence of 'step points'

$$d_0(n) < d_1(n) < d_2(n) < \ldots$$

with $d_0(n) = 0$ and $d_1(n) = \frac{1}{2}$ such that for $i \in \mathbb{N}_0$, f is constant within the intervals $[d_i(n), d_{i+1}(n))$ and $f(d_i(n)) < f(d_{i+1}(n))$.
Altogether, we will establish some exponential growth for f which implies that the individual 'steps' must become either higher or shorter. In fact, our first result is that every step is of height 1.

Proposition 4.15. *$f(d_{i+1}(n)) - f(d_i(n)) = 1$ for $i \in \mathbb{N}_0$.*

Proof. We abbreviate $d_i(n)$ as d_i and prove the statement by induction on i.
For $i = 0$, we have $f(d_1) - f(d_0) = f\left(\frac{1}{2}\right) - f(0) = 1$.
Now let $i \geq 1$. By (4.10), there is some $\frac{1}{n} \leq y \leq \frac{1}{2} < 1 - \frac{1}{n}$ such that $f(d_{i+1}) = f(d_{i+1} - y) + f(d_{i+1} - (1-y))$. By the induction hypothesis, there is some $\epsilon > 0$ such that $\frac{1}{n} \leq y + \frac{\epsilon}{2} \leq 1 - \frac{1}{n}$, $f(d_i) = f\left(d_{i+1} - \frac{\epsilon}{2}\right)$ and $f(d_{i+1} - y - \epsilon) \geq f(d_{i+1} - y) - 1$. This implies that

$$\begin{aligned} f(d_i) &= f\left(d_{i+1} - \frac{\epsilon}{2}\right) \\ &\geq f\left(\left(d_{i+1} - \frac{\epsilon}{2}\right) - \left(y + \frac{\epsilon}{2}\right)\right) \\ &\quad + f\left(\left(d_{i+1} - \frac{\epsilon}{2}\right) - \left(1 - \left(y + \frac{\epsilon}{2}\right)\right)\right) \\ &= f(d_{i+1} - y - \epsilon) + f(d_{i+1} - (1-y)) \\ &\geq f(d_{i+1} - y) - 1 + f(d_{i+1} - (1-y)) \\ &= f(d_{i+1}) - 1 \\ &\geq f(d_i) \end{aligned}$$

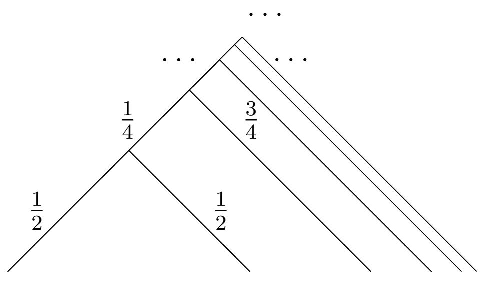

Figure 4.6: Structure of optimal trees for $d \leq 1$.

which completes the proof. □

We first consider the case that n is a power of 2, i.e. $n = 2^i$ for some $i \in \mathbb{N}$ with $i \geq 2$, and solve the recursion (4.10) explicitely for $d \leq 1$.

Proposition 4.16.

$$f(d) = \begin{cases} 0, & d < 0, \\ l, & \sum_{j=1}^{l-1} \frac{1}{2^j} \leq d < \sum_{j=1}^{l} \frac{1}{2^j} \\ & \textit{with } 1 \leq l \leq i, \\ i+1, & 1 - \frac{1}{2^i} \leq d < 1 \textit{ and} \\ i+2, & d = 1. \end{cases}$$

Proof. Every $\mathcal{L}(n)$-tree with at least two leaves in which none of the two children of the root is a leaf, has depth at least 1 and if it has depth 1, then is has exactly 4 leaves. This implies that we can assume that at least one child of every non-leaf in an $\mathcal{L}(n)$-tree with a given number l of leaves which is of minimum depth ≤ 1 is a leaf. Hence there are optimal $\mathcal{L}(n)$-trees that have a very simple structure (cf. Figure 4.6): The non-leaves induce a directed path $v_1 v_2 \dots v_{l-1}$ where v_1 is the root.
Clearly, we may assume that the two edges from v_{l-1} to its two children have equal length $\frac{1}{2}$. In view of this, we may assume that the two edges from v_{l-2} to its two children have lengths $\frac{1}{4}$ and $\frac{3}{4}$, because this balances the delays in a best-possible way. It is obvious how this kind of argument can be applied in turn to the vertices $v_{l-3}, v_{l-4}, \dots, v_1$.
In general, a minimum depth of the form $\sum_{j=1}^{l-1} \frac{1}{2^j}$ would be realized if the two edges from every v_{l-k} to its two children have lengths $\frac{1}{2^k}$ and $1 - \frac{1}{2^k}$. By the definition of $\mathcal{L}(n)$, this is only possible for $k \geq i$. This argument determines $f(d)$ as given in the statement of the Proposition for $d < 1$. For $d = 1$, it is best-possible to assign the lengths $\frac{1}{2^i}$ and $1 - \frac{1}{2^i}$ to the edges from the root to its two children. This leads to an $\mathcal{L}(n)$-tree with $i + 1$ leaves of depth 1 and one leaf of depth $1 - \frac{1}{2^i}$ and the proof is complete. □

As said before, we will establish an exponential growth for f and f is as convex as a step function can be. In order to reduce the recursion (4.10) essentially to a simple

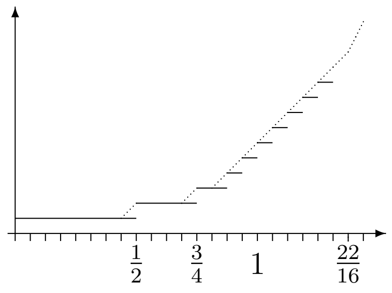

Figure 4.7: f and $\tilde{f}$ for $x = \frac{1}{16}$. The line segments correspond to f and the dotted lines correspond to $\tilde{f}$.

linear recursion that can be solved with textbook methods (see e.g. Sedgewick and Flajolet [1996]) we consider a function

$$\tilde{f} : [0, \infty) \to \mathbb{R}$$

such that

(i) $\tilde{f}\left(\frac{j}{2^i}\right) = f\left(\frac{j}{2^i}\right)$ for $0 \le j \le 2^i$,

(ii) $\tilde{f}$ is linear within the intervals $\left[\frac{j}{2^i}, \frac{j+1}{2^i}\right]$ for $0 \le j \le 2^i - 1$ and

(iii) $\tilde{f}(d) =$

$$\max_{\frac{1}{2^i} \le y \le 1 - \frac{1}{2^i}} \left(\tilde{f}(d-y) + \tilde{f}(d-(1-y))\right) \tag{4.11}$$

for $d > 1$.

Clearly, $\tilde{f}(d) \ge f(d)$ for $d \in [0,1]$ and hence for every $d \ge 0$ (cf. Figure 4.7).

Lemma 4.17. *The function $\tilde{f}$ is convex within $[1, \infty)$ and*

$$\tilde{f}(d) = \tilde{f}\left(d - \frac{1}{2^i}\right) + \tilde{f}\left(d - \left(1 - \frac{1}{2^i}\right)\right)$$

for $d > 1$.

Proof. Clearly, for $0 \le j \le 2^i - 1$, the function $\tilde{f}$ is linear within the interval $\left(\frac{j}{2^i}, \frac{j+1}{2^i}\right)$ with derivative 0 or 2^i. Note that for $2^i - 3 \le j \le 2^i - 1$ the function $\tilde{f}$ has derivative 2^i within the interval $\left(\frac{j}{2^i}, \frac{j+1}{2^i}\right)$. This implies that $\tilde{f}$, wherever it is differentiable, has derivative at least 2^i for $d > 1$. Now, this implies that the derivative of $\tilde{f}$, wherever it exists, is non-decreasing for $d \in [1, 2]$ which easily implies the desired result. □

Proposition 4.18. $f\left(\frac{j}{2^i}\right) = f\left(\frac{j-1}{2^i}\right) + f\left(\frac{j-2^i+1}{2^i}\right)$ *for $j \ge 2^i$.*

Proof. We prove the statement by induction on j. For $j = 2^i$, we have $f(1) = i + 2 = (i+1) + 1 = f\left(\frac{2^i-1}{2^i}\right) + f\left(\frac{1}{2^i}\right)$ and the statement is true.
Furthermore, by (i) of the definition of $\tilde{f}$ and Lemma 4.17, we obtain for $j > 2^i$ that

$$\begin{aligned} f\left(\frac{j}{2^i}\right) &\leq \tilde{f}\left(\frac{j}{2^i}\right) \\ &= \tilde{f}\left(\frac{j-1}{2^i}\right) + \tilde{f}\left(\frac{j-2^i+1}{2^i}\right) \\ &= f\left(\frac{j-1}{2^i}\right) + f\left(\frac{j-2^i+1}{2^i}\right) \\ &\leq f\left(\frac{j}{2^i}\right) \end{aligned}$$

and the proof is complete. □

Whereas the simple linear recursion for some specific values of f given by Proposition 4.18 determines the asymptotic and exponential growth of f, the "local" behavior of f is far from being sufficiently understood. We investigated a lot of explicit numerical data without being able to discover a reasonable pattern.
One notable curiosity is the following. For the essentially convex function $\tilde{f}$ the maximum in (4.11) is always attained only for $y = \frac{1}{2^i}$ for $d > 1$. Therefore, it would seem reasonable to assume that the closely related function f displays a similar behavior at least eventually. Nevertheless, the exponential growth of f on the one hand and Proposition 4.15 on the other hand imply

$$\liminf_{j\to\infty}(d_{j+1}(n) - d_j(n)) = 0.$$

It is easy to see by an inductive argument that for every $j \in \mathbb{N}$ we have $d_{j+1}(n) - d_j(n) = \frac{1}{2^l}$ for an appropriate $l \in \mathbb{N}$. These two facts together immediately imply the existence of a sequence $(d'_j(n))_{j\in\mathbb{N}}$ with $\lim_{j\to\infty} d'_j = \infty$ for which $f(d'_j(n))$ is strictly larger than

$$f\left(d'_j(n) - \frac{1}{2^i}\right) + f\left(d'_j(n) - \left(1 - \frac{1}{2^i}\right)\right)$$

for $j \in \mathbb{N}$. Intuitively, this seems to be in conflict with the asymptotic growth of f as determined by Proposition 4.18.

Now we consider the case that n is not a power of 2. The problem in this case is that the behavior of $f(d)$ for d around 1 is already quite complicated. Nevertheless, we believe that a similar approach as above leads to a result analogous to Proposition 4.18 and hence allows to estimate the asymptotic growth.
To be precise, we conjecture the existence of n step points $d_{i_1}(n), d_{i_2}(n), \ldots, d_{i_n}(n)$ with

(i) $d_{i_{j+1}}(n) - d_{i_j}(n) = \frac{1}{n}$ for $0 \leq j \leq n-1$.

(ii) $f(d_{i_{j+1}}(n)) - f(d_{i_j}(n))$ is monotonously increasing for $0 \leq j \leq n-1$.

(iii) $f(d) \leq \theta f(d_{i_j}(n)) + (1-\theta) f(d_{i_{j+1}}(n))$ for $d = \theta d_{i_j}(n) + (1-\theta) d_{i_{j+1}}(n)$ with $\theta \in (0,1)$.

Defining $\tilde{f}$ as above using the values of f for the $d_{i_j}(n)$ would clearly lead to results similar to Lemma 4.17 and Proposition 4.18. We have verified the existence of appropriate $d_{i_j}(n)$ as above for all $n \leq 32$ and provide some examples in Section 4.4.3.

4.4.2 Existence of $\{\{1,m\}\}$-trees and $\mathcal{L}'(4)$-trees

The existence of $\{\{1,m\}\}$-trees with leaves at specified maximal depths can be characterized by the following variant of Kraft's inequality (Kraft [1949]).

Proposition 4.19. *Let integers $1 \leq d_1 \leq d_2 \leq \ldots \leq d_n$ be given. There exists a $\{\{1,m\}\}$-tree T with n leaves at depths at most $d_1, d_2, \ldots, d_n$ if and only if*

$$\sum_{1 \leq j \leq i} f_{\{\{1,m\}\}}(d_i - d_j) \leq f_{\{\{1,m\}\}}(d_i) \tag{4.12}$$

for $1 \leq i \leq n$.

Proof. We consider the 'infinite' $\{\{1,m\}\}$-tree T_∞ of which every $\{\{1,m\}\}$-tree T is a finite subtree whose leaves correspond to vertices of T_∞ which are not descendants of each other. Because of the edges of length 1 in T_∞, T_∞ has exactly $f_{\{\{1,m\}\}}(d)$ vertices of depth exactly (!) d. Clearly without loss of generality, we can allow non-leaves of the desired $\{\{1,m\}\}$-trees to have just one child and the edge leading to that unique child to have length exactly 1. Hence to decide existence it suffices to consider such modified $\{\{1,m\}\}$-tree T with n leaves at depths exactly $d_1, d_2, \ldots, d_n$.
If such a (modified) $\{\{1,m\}\}$-tree T with n leaves at depths $d_1, d_2, \ldots, d_n$ exists, then counting the descendants at depth l_i within T_∞ of the first i of the leaves of T at depths $d_1, d_2, \ldots, d_i$ yields $\sum\limits_{1 \leq j \leq i} f_{\{\{1,m\}\}}(d_i - d_j)$ vertices within T_∞ at depth l_i which implies (4.12).
Conversely, the validity of (4.12) for all $1 \leq i \leq n$ implies that a desired (modified) $\{\{1,m\}\}$-tree can be found within T_∞ by successively choosing its leaves. After having chosen the first $i-1$ vertices within T_∞ at depths exactly $d_1, d_2, \ldots, d_{i-1}$ there remain

$$\begin{aligned} & f_{\{\{1,m\}\}}(d_i) - \sum_{1 \leq j \leq i-1} f_{\{\{1,m\}\}}(d_i - d_j) \\ \geq \quad & f_{\{\{1,m\}\}}(d_i - d_i) = 1 \end{aligned}$$

vertices within T_∞ at depth d_i that are not descendants of the previously chosen vertices. This clearly implies the existence of the desired $\{\{1,m\}\}$-tree by an inductive argument and completes the proof. □

Now we turn our attention to the simplest case of the discrete version $\mathcal{L}'(n)$ with $|\mathcal{L}'(n)| \geq 2$ which is $\mathcal{L}'(4)$. In the previous section we saw that choosing $y = \frac{1}{4}$ is

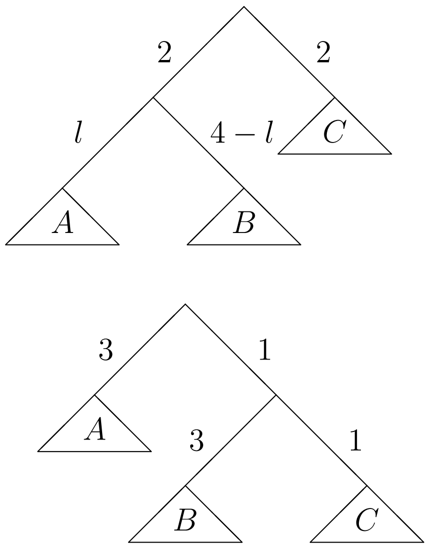

Figure 4.8: $l \in \{1, 2\}$.

eventually the optimal choice within the recursion (4.10) for $\mathcal{L}(4)$. For the discrete version this observation reflects in the property that in an optimal $\mathcal{L}'(4)$-tree T the edge lengths pair $\{2, 2\} \in \mathcal{L}'(4)$ can be assumed to occur only 'close' to the leaves of T, i.e. in the 'inner' part of T only the edge lengths pair $\{1, 3\} \in \mathcal{L}'(4)$ is used. The following proposition makes this statement precise.

Lemma 4.20. *If there is an $\mathcal{L}'(4)$-tree T with leaves at depths $d_1, d_2, \ldots, d_n$, then there is an $\mathcal{L}'(4)$-tree T' with leaves at depths at most $d_1, d_2, \ldots, d_n$ such that every edge of T' of length 2 is incident to a leaf of T'.*

Proof. Every edge of length 2 in T that is not incident to a leaf can be replaced as indicated in Figure 4.8 without increasing the depths of leaves. □

We will describe how to decide the existence of an $\mathcal{L}'(4)$-tree T with leaves at depths at most $d_1, d_2, \ldots, d_n$. Clearly, if all inequalities (4.12) are satisfied, then the desired tree T exists by Proposition 4.19 and does not need to contain any edge of length 2. Hence we can assume that (4.12) is violated for some index. The following result captures the essential observation.

Proposition 4.21. *Let integers $1 \le d_1 \le d_2 \le \ldots \le d_n$ be given and let $1 \le i^* \le n$ be such that inequality (4.12) holds for $1 \le i \le i^* - 1$ and does not hold for $i = i^*$.*
There exists an $\mathcal{L}'(4)$-tree T with leaves at depths at most $d_1, d_2, \ldots, d_n$ if and only if $d_{i^*-1} = d_{i^*}$ and there exists an $\mathcal{L}'(4)$-tree T' with leaves at depths at most

$$d_1, d_2, \ldots, d_{i^*-2}, d_{i^*} - 2, d_{i^*+1} \ldots, d_n.$$

Proof. Let the integers $1 \le d_1 \le d_2 \le \ldots \le d_n$ and the index i^* be as in the statement.

Since the ‘if’-part of the statement is trivial, we only prove the ‘only if’-part. By Proposition 4.19, the existence of T implies that T contains edges of length 2. By Lemma 4.20, we may assume that there are two such edges which are incident and lead to leaves of T. Clearly, the depth requirements for these two leaves must be equal, since otherwise we could use the edge lengths pair $\{1,3\}$ without violating the depth requirements. Hence $d_j = d_{j+1}$ for some $1 \leq j \leq i^* - 1$. By contracting the edge pair, this implies the existence of an $\mathcal{L}'(4)$-tree T'' with leaves at depths at most

$$d_1, d_2, \ldots, d_{j-1}, d_j - 2, d_{j+2}, \ldots, d_n. \tag{4.13}$$

For these new depth requirements, the left hand side of inequality (4.12) for index i^* changes by

$$f_{\{\{1,3\}\}}(d_{i^*} - (d_j - 2)) - 2f_{\{\{1,3\}\}}(d_{i^*} - d_j).$$

Since $f_{\{\{1,3\}\}}(d+2) - 2f_{\{\{1,3\}\}}(d)$

$$= \begin{cases} 1 - 2\cdot 0 = 1 & , d = -2, \\ 1 - 2\cdot 0 = 1 & , d = -1, \\ 1 - 2\cdot 1 = -1 & , d = 0, \\ 2 - 2\cdot 1 = 0 & , d = 1, \\ 3 - 2\cdot 1 = 1 & , d = 2, \\ 4 - 2\cdot 2 = 0 & , d = 3 \end{cases}$$

and, by induction,

$$\begin{array}{rl} & f_{\{\{1,3\}\}}(d+2) - 2f_{\{\{1,3\}\}}(d) \\ = & (f_{\{\{1,3\}\}}(d+1) + f_{\{\{1,3\}\}}(d-1)) \\ & -2(f_{\{\{1,3\}\}}(d-1) + f_{\{\{1,3\}\}}(d-3)) \\ = & (f_{\{\{1,3\}\}}(d+1) - 2f_{\{\{1,3\}\}}(d-1)) \\ & +(f_{\{\{1,3\}\}}(d-1)) - 2f_{\{\{1,3\}\}}(d-3)) \\ \geq & 0 \end{array}$$

for $d \geq 4$, considering (4.13) can only reduce the left hand side of inequality (4.12) for index i^* if $d_j = d_{i^*-1} = d_{i^*}$. This completes the proof. $\square$

Based on Proposition 4.21 we now describe an algorithm that decides the existence (and could also construct it, in case of existence) of an $\mathcal{L}'(4)$-tree with leaves at depths at most some n given positive integers $d_1, d_2, \ldots, d_n$.
If $d_{\max} = \max\{d_1, d_2, \ldots, d_n\}$, then the running time will be $O(n + d_{\max})$. Let

$$n(i) = |\{j \mid 1 \leq j \leq n, d_j = i\}|$$

for $1 \leq i \leq d_{\max}$. The algorithm can be viewed as choosing an appropriate subtree of the infinite tree T_∞ considered in the proof of Proposition 4.19. Whenever it is necessary to use a pair of edges of lengths 2 for a pair of leaves, the algorithm replaces the two leaves by one whose depth requirement is two units less according to Proposition 4.21.

With $b(i)$ we will denote the number of blocked vertices with depth i, i.e. those vertices of T_∞ at depth i that have been chosen as leaves of the desired tree and also those that are descendants of chosen leaves of the desired tree with smaller depths.
In view of Lemma 4.20, it can never be necessary for the existence of the desired tree that a 'new' leaf which replaced two other leaves will itself be replaced together with some further leaf by another 'new' leaf. Hence we keep track of the number $m(i)$ of 'new' leaves with depth i and only consider the $n(i) - m(i)$ 'old' leaves with depth i for possible replacements.

Algorithm 6: Construction of $\mathcal{L}'(4)$-trees.

```
Input: $d_{max} \in \mathbb{N}$, $n(\cdot) : \{1, \dots, d_{max}\} \mapsto \mathbb{N}$
Result: Returns true iff there exists an $\mathcal{L}'(4)$-tree for $n(\cdot)$
1  Compute $f_{\{\{1,3\}\}}(i)$ for $1 \le i \le d_{max}$;
2  Set $b(i) \leftarrow 0$ for $1 \le i \le d_{max}$;
3  $i \leftarrow 1$;
4  while $i \le d_{max}$ do
5      $b(i) \leftarrow b(i-1) + b(i-3) + n(i)$;
6      if $b(i) > f_{\{\{1,3\}\}}(i)$ then
7          if $n(i) - m(i) \ge 2$ then
8              $n(i) \leftarrow n(i) - 2$;
9              $n(i-2) \leftarrow n(i-2) + 1$;
10             $m(i-2) \leftarrow m(i-2) + 1$;
11             $i \leftarrow i - 2$;
12         else
13             return false;
14         end
15     else
16         $i \leftarrow i + 1$;
17     end
18 end
19 return true;
```

We think that the results of this section extend to $\mathcal{L}'(n)$-trees for $n \ge 5$. To be precise, we conjecture that optimal $\mathcal{L}'(n)$-trees can always be assumed to use edge lengths pairs different from $\{1, n-1\}$ only 'close' to leaves, i.e. every such edge is within a distance bounded in terms of n from some leaf, which would generalize Lemma 4.20. Furthermore, we think that it might be possible to extend Proposition 4.21 to $n \ge 5$. In Proposition 4.21 the feasibility of some instance of the existence problem is reduced to the feasibility of exactly one (!) smaller instance. We fear but believe that for $n \ge 5$ this will change which might make it much harder to describe an efficient algorithm for the existence problem.

4.4.3 Examples of the Function $f_{\mathcal{L}(n)}$ for Different Values of n

- $n = 3:$

  $f\left(\frac{1}{2}\right) = 2,\quad f\left(\frac{5}{6}\right) = 3,\quad f\left(\frac{7}{6}\right) = 5$

- $n = 5:$

  $f\left(\frac{279}{160}\right) = 14,\quad f\left(\frac{311}{160}\right) = 19,\quad f\left(\frac{343}{160}\right) = 26,\quad f\left(\frac{75}{32}\right) = 36,\quad f\left(\frac{407}{160}\right) = 50$

- $n = 6:$

  $f\left(\frac{367}{192}\right) = 20,\quad f\left(\frac{133}{64}\right) = 26,\quad f\left(\frac{431}{192}\right) = 34,\quad f\left(\frac{463}{192}\right) = 45,$
  $f\left(\frac{165}{64}\right) = 60,\quad f\left(\frac{527}{192}\right) = 80$

- $n = 7:$

  $f\left(\frac{75}{56}\right) = 8,\quad f\left(\frac{83}{56}\right) = 10,\quad f\left(\frac{13}{8}\right) = 13,\quad f\left(\frac{99}{56}\right) = 17,\quad f\left(\frac{107}{56}\right) = 22,$
  $f\left(\frac{115}{56}\right) = 28,\quad f\left(\frac{123}{56}\right) = 36$

- $n = 9:$

  $f\left(\frac{9\,659}{4\,608}\right) = 37,\quad f\left(\frac{10\,171}{4\,608}\right) = 45,\quad f\left(\frac{1\,187}{512}\right) = 55,\quad f\left(\frac{11\,195}{4\,608}\right) = 67,$
  $f\left(\frac{11\,707}{4\,608}\right) = 82,\quad f\left(\frac{4\,073}{1\,536}\right) = 101,\quad f\left(\frac{12\,731}{4\,608}\right) = 125,\quad f\left(\frac{13\,243}{4\,608}\right) = 155,$
  $f\left(\frac{4\,585}{1\,536}\right) = 192$

- $n = 10:$

  $f\left(\frac{35}{16}\right) = 47,\quad f\left(\frac{183}{80}\right) = 57,\quad f\left(\frac{191}{80}\right) = 69,\quad f\left(\frac{199}{80}\right) = 83,$
  $f\left(\frac{207}{80}\right) = 100,\quad f\left(\frac{43}{16}\right) = 121,\quad f\left(\frac{223}{80}\right) = 147,\quad f\left(\frac{231}{80}\right) = 179,$
  $f\left(\frac{239}{80}\right) = 218,\quad f\left(\frac{247}{80}\right) = 265$

- $n = 20:$

  $f\left(\frac{271\,155}{65\,536}\right) = 8\,162,\quad f\left(\frac{1\,372\,159}{327\,680}\right) = 9\,180,\quad f\left(\frac{1\,388\,543}{327\,680}\right) = 10\,329,$
  $f\left(\frac{1\,404\,927}{327\,680}\right) = 11\,624,\quad f\left(\frac{1\,421\,311}{327\,680}\right) = 13\,082,\quad f\left(\frac{287\,539}{65\,536}\right) = 14\,722,$
  $f\left(\frac{1\,454\,079}{327\,680}\right) = 16\,565,\quad f\left(\frac{1\,470\,463}{327\,680}\right) = 18\,634,\quad f\left(\frac{1\,486\,847}{327\,680}\right) = 20\,955,$
  $f\left(\frac{1\,503\,231}{327\,680}\right) = 23\,557,\quad f\left(\frac{303\,923}{65\,536}\right) = 26\,472,\quad f\left(\frac{1\,535\,999}{327\,680}\right) = 29\,736,$
  $f\left(\frac{1\,552\,383}{327\,680}\right) = 33\,390,\quad f\left(\frac{1\,568\,767}{327\,680}\right) = 37\,481,\quad f\left(\frac{1\,585\,151}{327\,680}\right) = 42\,063,$
  $f\left(\frac{320\,307}{65\,536}\right) = 47\,198,\quad f\left(\frac{1\,617\,919}{327\,680}\right) = 52\,957,\quad f\left(\frac{1\,634\,303}{327\,680}\right) = 59\,421,$
  $f\left(\frac{1\,650\,687}{327\,680}\right) = 66\,682,\quad f\left(\frac{1\,667\,071}{327\,680}\right) = 74\,844$

Bibliography

Aigner, M. [1988]: Combinatorial Search. Wiley & Sons, New York, 1988.

Alon, N. and Azar, Y. [1993]: On-line Steiner trees in the Euclidean plane. *Discrete and Computational Geometry 10*, pp. 113–121, 1993.

Alpert, C.J., Gandham, G., Hrkic, M., Hu, J., Kahng, A.B., Lillis, J., Liu, B., Quay, S.T., Sapatnekar, S.S. and Sullivan, A.J. [2002]: Buffered Steiner trees for difficult instances. In *IEEE Transactions on Computer-Aided Design 21*, pp. 3–14, 2002.

Bartoschek, C., Held, S., Rautenbach, D., and Vygen, J. [2006]: Efficient generation of short and fast repeater tree topologies. In *Proceedings of the International Symposium on Physical Design*, pp. 120–127, 2006.

Bartoschek, C., Held, S., Rautenbach, D., and Vygen, J. [2009]: Fast buffering for optimizing worst slack and resource consumption in repeater trees. In *Proceedings of the International Symposium on Physical Design*, pp. 43–50, 2009.

Bleich, C. and Overton, M.L. [1983]: A linear-time algorithm for the weighted median problem. Technical Report 75, Department of Computer Science, New York University, 1983.

Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., and Tarjan, R.E. [1973]: Time bounds for selection. *Journal of Computer and System Sciences 7*, pp. 448–461, 1973.

Chao, T.-H., Ho, J.-M., and Hsu, Y.-C. [1992]: Zero skew clock net routing. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pp. 518–523, 1992.

Chong, J., Kahng, A.B., Koh, C.-K., and Tsao, C.-W.A. [1998]: Bounded-skew clock and Steiner routing. *ACM Transactions on Design Automation of Electronic Systems 3*, pp. 341–388, 1998.

Cong, J. and Yuan, X. [2000]: Routing tree construction under fixed buffer locations. In *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 379–384, 2000.

Christofides, N. [1976]: Worst-case analysis of a new heuristic for the traveling salesman problem. Technical Report CS-93-13, G.S.I.A., Carnegie Mellon University, Pittsburgh, 1976.

Dechu, S., Shen, Z.C. and Chu, C.C.N [2004]: An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations. *IEEE Transactions on Computer-Aided Design of Intergated Circuits and Systems 24*, pp. 600–608, 2004.

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

Drmota, M. and Szpankowski, W. [2002]: Generalized Shannon code minimizes the maximal redundancy. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, pp. 306–318, 2002.

Drmota, M. and Szpankowski, W. [2004]: Precise minimax redundancy and regret. *IEEE Transactions on Information Theory 50*, pp. 2686–2707, 2004.

Du, D.-Z., Zhang, Y., and Feng, Q. [1991]: On better heuristic for "Euclidean Steiner" minimum trees. In *32nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 431–439, 1991.

Edahiro, M. [1991]: Minimum skew and minimum path length routing in VLSI layout design. *NEC Research and Development 32*, pp. 569–575, 1991.

Edahiro, M. [1992]: Minimum path-length equi-distant routing. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 41–46, 1992.

Epstein L. and Levin A. [2008]: On bin packing with conflicts. *SIAM Journal on Optimization*, pp. 1270–1298, 2008.

Gabow, H., Glover, F., and Klingman, D. [1974]: A note on exchanges in matroid bases. Technical Report C.S.184, Center for Cybernetic Studies, University of Texas, Austin, Texas, USA, 1974.

Gallager, R.G. [1968]: Information theory and reliable communication. Wiley & Sons, New York, 1968.

Garey, M. and Johnson, D. [1979]: Computers and Intractability: A guide to the theory of NP-completeness. W.H. Freeman and Company, New York, 1979.

Garey, M. and Johnson, D. [1977]: The rectilinear Steiner tree problem is NP-complete. *SIAM Journal on Applied Mathematics 32*, pp. 826–834, 1977.

Gawrychowski, P. and Gagie, T. [2008]: Minimax trees in linear time. CoRR abs/0812.2868, 2008.

van Ginneken, L.P.P.P. [1990]: Buffer placement in distributed RC-tree networks for minimal Elmore delay . In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 865–868, 1990.

Gester, M. [2009]: Voronoi-Diagramme von Achtecken in der Maximum-Metrik. Diploma thesis (in german), University of Bonn, 2009.

Golumbic, M. [1976]: Combinatorial merging. *IEEE Transactions on Computers 25*, pp. 1164–1167, 1976.

Hall, P. [1935]: On representatives of subsets. *Journal of the London Mathematical Society 10*, pp. 26–30, 1935.

Hanan, M. [1966]: On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics 14*, pp. 255–265, 1966.

Hentschke, R.F. Narasimham, J., Johann, M.O. and Reis, R.L. [2007]: Maze routing Steiner trees with effective critical sink optimization. IN *ISPD '07: Proceedings of the 2007 International Symposium on Physical Design*, pp. 135–142, 2007.

Held, S. [2008]: *Timing Closure in Chip Design.* PhD thesis, University of Bonn, 2008.

Hoehn, L. and Ridenhour, J. [1989]: Summations involving computer-related functions. *Mathematics Magazine 62*, pp. 191–196, 1989.

Hoover, H.J., Klawe, M.M., and Pippenger, N. [1984]: Bounding fan-out in logical networks. *Journal of the ACM 31*, pp. 13–18, 1984.

Hrkic, M and Lillis J. [2002]: S-Tree: a technique for buffered routing tree synthesis. In *Proceedings of the IEEE/ACM International Conference on Computed-Aided Design*, pp. 578–583, 2002.

Hrkic, M. and Lillis, J. [2003]: Buffer tree synthesis with consideration of temporal locality, sink polarity requirements, solution cost, congestion, and blockages. *IEEE Transactions on Computed-Aided Design of Integrated Circuits and Systems 22*, pp. 481–491, 2003.

Huffman, D.A. [1952]: A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE 40*, pp. 1098–1101, 1952.

Hwang, F. [1976]: On Steiner minimal trees with rectilinear distance. *SIAM Journal of Applied Mathematics 30*, pp. 104–114, 1976.

Hwang, H.-K. and Tsai, T.-H. [2003]: An asymptotic theory for recurrence relations based on minimization and maximization, *Theoretical Computer Science 290*, pp. 1475–1501, 2003.

Johnson, D.B. and Mizoguchi, T. [1978]: Selecting the kth element in $X + Y$ and $X_1 + X_2 + \ldots + X_m$. *SIAM Journal on Computation 7*, pp. 147–153, 1978.

Kapoor, S. and Reingold, E.M. [1989]: Optimum lopsided binary trees. *Journal of the ACM 36*, pp. 573–590, 1989.

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

Korte, B. and Vygen, J. [2008]: *Combinatorial Optimization: Theory and Algorithms.* Springer Berlin. Forth edition, 2008.

Kraft, L.G. [1949]: A device for quantizing grouping and coding amplitude modulated pulses. Master's thesis, EE Dept., MIT, Cambridge, 1949.

Lillis, J., Cheng, C.-K., Lin, T.-T. Y. and Ho, C.-Y. New performance driven routing techniques with explicit area/delay Tradeoff and simultaneous wire sizing. In *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 395–400, 1996.

Maßberg, J. and Rautenbach, D. [2009]: Binary trees with choosable edge lengths. *Information Processing Letters 109*, pp. 1087–1092, 2009.

Maßberg, J. and Vygen, J. [2008]: Approximation algorithms for a facility location problem with service capacities. *ACM Transactions of Algorithms 4*, pp. 1–15, 2008.

B. McMillan, Two inequalities implied by unique decipherability, *IRE Transactions on Information Theory 2*, pp. 115–116, 1956.

Muuss, K. [1994]: Clockskew Optimierung. Diploma thesis (in german), University of Bonn, 1994.

Okamoto, T. and Cong J. [1996]: Buffered Steiner tree construction with wire sizing for interconnect layout optimization. In*Proceedings of the International Conference on Computer Aided Design*, pp. 44-49, 1996.

Pan, M., Chu, C. and Patra, P. [2007]: A novel performance-driven topology design algorithm. In *ASP-DAC '07: Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pp. 244–249, 2007.

Parker, D.S., Jr. [1979]: Combinatorial merging and Huffman's algorithm. *IEEE Transactions on Computers 28*, pp. 365–367, 1979.

Prim, R.C. [1957]: Shortest connection networks and some generalizations. *Bell System Technical Journal 36*, pp. 1389–1401, 1957.

Reiser, A. [1978]: A linear selection algorithm for sets of elements with weights. *Information Processing Letters 7* , pp. 159–162, 1978.

Robins, G. and Zelikovsky, A. [2000]: Improved Steiner tree approximation in graphs. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 770–779, 2000.

Sedgewick, R. and Flajolet, P. [1996]: Analysis of Algorithms. Addison-Wesley, Reading, 1996.

Shamos, M.I. [1976]: Geometry and statistics. In *Algorithms and Complexity: New Directions and Recent Results*, Editor J.F. Traub, Academic Press, pp. 251–280, 1976.

Shelar, R.S. [2007]: An efficent clustering algorithm for low power clock tree synthesis. In *ISPD '07: Proceedings of the 2007 International Symposium on Physical Design*, pp. 181–188, 2007.

Tsao, C.-W.A. and Koh, C.-K. [2002]: UST/DME: a clock tree router for general skew constraints. *ACM Transactions on Design Automation of Electronic Systems 7*, pp. 359–379, 2002.

Tsay, R.-S. [1993]: An exact zero-skew clock routing algorithm. *IEEE Transactions on CAD of Integrated Circuits and Systems 12*, pp. 242–249, 1993.

von Randow, R. [1975]: *Introduction to the Theory of Matroids.* Lecture Notes in Economics and Mathematical Systems. Springer, 1975.

Vygen, J. [2001]: *Theory of VLSI Layout.* Habilitation thesis, University of Bonn, 2001.

Warme, D., Winter, P., and Zachariasen, M. [2003]: Geosteiner 3.1 home page. www.diku.dk/geosteiner, 2003.

Welsh, D.J.A. [1976]: *Matroid Theory.* Academic Press London, 1976.

Summary

The construction of clock trees and repeater trees are major challenges in chip design. Such trees distribute an electrical clock signal from a source to a set of sinks on a chip. On recent designs there can be millions of repeater trees with only a few up to some hundred sinks and several clock trees with up to some hundred thousand of sinks. In repeater trees the signal has to arrive at each sink not later than an individual required arrival time, while in clock trees it has to arrive at each sink within an individual required arrival time window. In this thesis, we present new theory and algorithms for the construction of clock trees and repeater trees and an essential sub-problem, the SINK CLUSTERING PROBLEM. We also describe our clock tree construction tool BonnClock, which has been used by IBM Microelectronics for the design of hundreds of most complex chips.

First, we introduce the SINK CLUSTERING PROBLEM, the main sub-problem of clock tree design. Given a metric space (V, c), a finite set D of terminals with positions $p(v) \in V$ and demands $d(v) \in \mathbb{R}_{\geq 0}$ for all $v \in D$, a facility opening cost $f \in \mathbb{R}_{>0}$ and a load limit $u \in \mathbb{R}_{>0}$, the task is to find a partition $D = D_1 \dot{\cup} \cdots \dot{\cup} D_k$ of D and, for all $1 \leq i \leq k$, a Steiner tree S_i for $\{p(v) | v \in D_i\}$. Each cluster (D_i, S_i), $1 \leq i \leq k$, has to keep the load limit, that means $\sum_{e \in E(S_i)} c(e) + \sum_{s \in D_i} d(s) \leq u$. The goal is to minimize the weighted sum of the length of all Steiner trees plus the number of clusters, i.e. minimize $\sum_{i=1}^{k} (\sum_{e \in E(S_i)} c(e)) + kf$.

We present the first constant-factor approximation algorithm for the SINK CLUSTERING PROBLEM. It is based on decomposing a minimum spanning tree on the sinks and has an approximation guarantee of $1 + 2\alpha$, where α is the Steiner ratio of the underlying metric. Moreover, we introduce two variants of the algorithm that rely on decomposing an approximate minimum Steiner tree and an approximate minimum traveling salesman tour. These algorithms have approximation guarantees of 3β and 3γ, respectively, where β and γ are the approximation guarantees of the Steiner tree and TSP approximation algorithms, respectively. We also propose two post-optimization algorithms that can further improve an existing clustering.

We analyze the structure of the SINK CLUSTERING PROBLEM and exhibit its connections to matroid theory. In particular, we use the property of matroids that for any two bases B_1, B_2 there is a bijection $\pi : B_1 \to B_2$ so that $(B_1 \setminus \{b\}) \cup \{\pi(b)\}$ is again a basis for each $b \in B_1$.

We replace each Steiner tree of an optimum solution by a minimum spanning tree and connect all trees to a new artificial vertex s and get a tree S. In a modified metric the total length of S is a good lower bound for the cost of an optimum solution. Due to the matroid property we can compare a minimum spanning tree T on $D \cup \{s\}$ with S; the length of any edge of T is bounded by the length of an edge of S. We introduce the concept of K-dominated functions that helps us to increase the 'cost' of certain edges

of T while still having the property that the total length of all edges of T ending in a vertex of $K \subseteq D$ is bounded by the total length of all edges of S ending in a vertex of K. Applying this procedure to the sets of a laminar family on D yields an improved lower bound.
The bound can be further improved by combining it with a lower bound for the length of a minimum Steiner tree on D. For this bound we prove the following lemma: For any family of trees $\mathcal{T} = \{T_1, \ldots, T_k\}$ with $V(T_i) \subset D$, $1 \leq i \leq k$, with the property that for any subset $\mathcal{T}' \subseteq \mathcal{T}$ the trees in $\mathcal{T}'$ cover at least $|\mathcal{T}'| + 1$ vertices, there exists an edge $e_i \in E(T_i)$ for $i = 1, \ldots, k$ such that these edges $E = \{e_i | \, 1 \leq i \leq k\}$ form a forest, i.e. the set does not contain an edge twice and it does not contain a circuit.
Our experimental results on real-world instances from clock tree design show that the cost of the solutions computed by our algorithms is in average only 10% over the best lower bound. Moreover, we compare our algorithm to another clustering algorithm used in industry. The results show that the total cost of our solutions is 10% less than the cost of the solutions computed by the competitive tool.
Clock trees have to satisfy several timing constraints. More precisely, the signal has to reach each sink within an individual required arrival time window. Sinks can only be clustered together if their required arrival time windows have a point of time in common. Typically, all required arrival time windows are the same. In this case we have the SINK CLUSTERING PROBLEM defined above. However, there are clock trees where the sinks have different required arrival time windows. This motivates a generalization of the SINK CLUSTERING PROBLEM where each sink additionally has an individual time window. As further constraint the time windows of the sinks of a cluster must have at least one point of time in common. We study the SINK CLUSTERING PROBLEM WITH TIME WINDOWS and present a polynomial $O(\log s)$-approximation algorithm for this problem, where s is the size of a minimum clique partition in the interval graph induced by the time windows. Our algorithm is based on a divide and conquer approach and uses the approximation algorithms for the SINK CLUSTERING PROBLEM on sub-sets of the instance. We show that the approximation guarantee of the algorithm is tight.
For the practical construction of clock trees we present our algorithm BonnClock. BonnClock builds a clock tree combining a bottom-up clustering and a top-down partitioning strategy. In the bottom-up phase BonnClock is using the SINK CLUSTERING ALGORITHM in order to determine the drivers of unconnected sinks or inverters. The 'global' topology of the tree is determined by the top-down partitioning considering big blockages and timing restrictions. BonnClock uses a dynamic program in order to determine the sizes of the inverters that are inserted. All components of the algorithm are discussed in detail.
As part of this thesis, we have also implemented this algorithm. BonnClock has become the standard tool to construct clock trees within IBM. We show experimental results with comparisons to another industrial clock tree construction tool and to lower bounds for the power consumption. It turns out that – mainly due to the SINK CLUSTERING ALGORITHM – our power consumption is much smaller than with the other tool and only one third over the lower bound.
Finally, we consider the repeater tree construction problem. In contrast to clock trees,

each sink has a latest required arrival time instead of a time window. We describe a simple algorithm to build such trees where we insert the sinks one by one into an existing tree. Depending on the optimization goal we show a variant of the algorithm computing trees of almost optimal length or trees with guaranteed best possible performance.

Moreover, we analyze the topology of trees with best or almost best performance more closely. Such trees are equivalent to minimax and almost minimax trees: Let $a_1, \dots, a_n \in \mathbb{N}_{\geq 0}$ be a set of numbers. The weight of a tree with n leaves is the maximum over all leaves i of the depth of leaf i plus a_i. For a non-negative integral constant c the goal is to build a binary tree with weight at most the optimum weight plus c. This problem can be solved optimally by a greedy algorithm. However, we are interested in the online version of this problem where we have to insert the leaf i with weight a_i into the tree without knowing n and the following weights a_j, $j > i$.

We give necessary and sufficient conditions for an online algorithm to compute trees of weight at most the optimum weight plus c. Moreover, we show how these conditions can be verified efficiently. We obtain an online algorithm that computes an optimum tree in $O(n \log n)$ time.

Finally, we study a further mathematical model of repeater trees that considers that additional delay caused by a bifurcation of a tree can be distributed partially to the two branches. For $c \in \mathbb{R}_{>0}$ and a set $\mathcal{L} \subseteq \{(l_1, l_2) \in \mathbb{R}^2_{\geq 0} | \, l_1 + l_2 = c\}$ of two-element sets of non-negative real numbers we consider rooted binary trees with the property that the two edges emanating from every non-leaf are assigned lengths l_1 and l_2 with $\{l_1, l_2\} \in \mathcal{L}$.

We study the asymptotic growth of the maximum number of leaves of bounded depths in such trees and the existence of such trees with leaves at individually specified maximum depths. Our results yield better lower bounds for repeater trees.