Karlsruhe Institute of Technology

# Minimum Linear Arrangement revisited

Master's thesis of

Michael Zündorf

At the Department of Informatics
Institute of Theoretical Informatics

| | |
|---|---|
| Reviewer: | Jun.-Prof. Dr. Thomas Bläsius |
| Second reviewer: | PD Dr. Torsten Ueckerdt |
| Advisor: | Marcus Wilhelm, M.Sc. |

September 30, 2021 – March 30, 2022

KIT – The Research University in the Helmholtz Association www.kit.edu

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I hereby declare that this document has been composed by myself and describes my own work, unless stated otherwise in the text. I also declare that I have read and obeyed the *Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).*

**Karlsruhe, 30.03.2022**

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Michael Zündorf)

## Abstract

The *Minimum Linear Arrangement (MinLA)* problem is a classical combinatorial optimization problem that has been studied for many years. In this work, we revisit the lower bound introduced by Adolphson and Hu which is based on Gomory-Hu trees. Additionally, we introduce a new heuristic upper bound to solve the MinLA problem which is also based on the Gomory-Hu trees. On the way, we introduce a restricted MinLA problem for weighted trees and solve this problem optimally. This also yields a new heuristic for the standard MinLA problem on weighted trees.

In addition to this, we take a closer look on the state-of-the-art algorithms to generate lower bounds. The currently best algorithms in terms of quality of the computed bounds are based on *Linear Programs (LP)*. In this work, we introduce new constraints which can be combined with the existing algorithms. Additionally, we pair one LP approach with a community detection algorithm to get bounds of good quality in much shorter time. This enables us to compute lower bounds for instances of the MinLA problem which were unfeasible before. Finally, we evaluate our proposed approach on a large set of graphs and compare it with the original LP.

## Zusammenfassung

Das *Minimum Linear Arrangement (MinLA)* Problem ist ein klassisches kombinatorisches Optimierungsproblem, das schon seit vielen Jahren untersucht wird. In dieser Arbeit schauen wir uns eine untere Schranke an, die von Adolphson und Hu vorgestellt wurde und auf Gomory-Hu Bäumen basiert. In diesem Zusammenhang stellen wir eine neue Heuristik vor, um das MinLA Problem zu lösen, die ebenfalls auf Gomory-Hu Bäumen basiert. Dafür führen wir zuerst ein eingeschränktes MinLA Problem für gewichtete Bäume ein, das wir optimal lösen. Diese Lösung kann ebenfalls als Heuristik verwendet werden, um das Standard MinLA Problem für gewichtete Bäume zu lösen.

Zusätzlich schauen wir uns aktuelle Algorithmen zur Bestimmung von unteren Schranken an. Die derzeit besten Algorithmen in Bezug auf die Qualität der berechneten Schranke basieren auf *Linearen Programmen (LP)*. In dieser Arbeit führen wir neue Bedingungen ein, die mit den bestehenden Algorithmen kombiniert werden können. Darüber hinaus kombinieren wir ein LP mit einem Algorithmus zur *community detection*, um gute Schranken in kürzerer Zeit zu berechnen. Dieser Ansatz ermöglicht es uns, untere Schranken für Instanzen zu berechnen, für die dies vorher nicht möglich war. Ebenfalls werten wir diesen Ansatz auf einem großen Datensatz auf echten Graphen aus und vergleichen es mit dem ursprünglichen LP.

# Contents

# 1 Introduction

In this work, we want to revisit the Minimum Linear Arrangement problem. We take a look at the existing literature and the state-of-the-art algorithms. We try to find new approaches and improve the existing ones with a focus on optimization for real world graph inputs.

## 1.1 Motivation

Graph layout problems have been studied over the past 50 years, since they naturally arise in different contexts. In theoretical works, the Linear Arrangement problem is one fo the problems to be considered most often. In this problem, a graph is arranged along a linear line such that certain properties of the drawing are optimized. One of the most natural optimization criteria is the total edge length which has first been considered by Harper in 1964 [Har64]. The problem of minimizing this total edge length is called the *Minimum Linear Arrangement* (MinLA) problem.

This is not only a theoretically interesting problem but solving it also helps with designing error correcting codes [Har64]. Some more obvious applications include the creation of efficient peer-to-peer overlay networks or the design of circuits where the track length is minimized [RH08 | BS87]. Additionally, linear arrangements help to analyze and visualize various models arising in studies of gene structures or nervous activity in the cortex.

Unfortunately, solving the MinLA problem appears to be hard in practice. Harper has already proved in 1964 that the problem is $\mathcal{NP}$-hard in general. Later on, it has been shown that the problem remains $\mathcal{NP}$-complete even when restricted to *unweighted bipartite* graphs or *unweighted interval* graphs, for which many other problems become easier [Eve75 | Coh+06].

To find instances that can be solved in polynomial time, we have to look at even more restricted graph classes like *trees*, *rectangular grids*, *proper interval graphs* or *outer-planar graphs*. And even for these classes, only the unweighted problem is solved and the algorithms are fairly complex.

For many interesting graph classes, either no polynomial time algorithm exists or at least none is known. Therefore, in most cases, the only way to solve this problem exactly is either with a *Dynamic Program* (DP) [KH02] or with an *Integer Linear Program* (ILP) [Cou16]. However, this is highly infeasible in practice, since both algorithms are too slow to solve instances with only hundreds of vertices in feasible time. Therefore, in practice, we have to rely on *heuristic* algorithms. These may not find the optimal solution, but at least they find a solution in a feasible time.

## 1.2 Outline

In Chapter 2, we introduce the notations used for this work and elaborate some basic graph theory which we use later.

In Chapter 3, we take a closer look on the relation between Gomory-Hu trees and the MinLA problem. This yields a new heuristic for the MinLA problem but more importantly we find an algorithm that solves some easier MinLA related problems on weighted trees optimally on that way. In particular, we can solve the MinLA problem on weighted trees under the restriction that sub-trees have to be embedded consecutively.

The main results of this work, however, are presented in Chapter 4. There, we revisit the state-of-the-art lower bound algorithms and try to improve them. Since these algorithms are based on Linear Programs (LP), we introduce new constraints and present a new approach to find violated constraints of some types. Furthermore, we introduce the first approach that uses *community detection* in context of the MinLA problem. Our approach enables us to compute lower bounds in much shorter time which may again can be used to improve pruning in heuristic upper bounds. Additionally, good lower bounds enable us to evaluate the quality of the heuristic upper bounds.

In Chapter 5, we evaluate the aforementioned approaches on a set of well-known graphs in the context of the MinLA problem and on a much larger set of real world graphs.

## 1.3 Related Work

As already mentioned, there are only a few graph classes that can be solved optimally and the required algorithms are not simple. among the first graph classes on which the MinLA problem was solved were trees. Goldberg et al. proposed an algorithm which could solve unweighted trees in $\mathcal{O}(n^3)$ [GK76]. Later, this result was improved by Shiloach who found an $\mathcal{O}(n^{2.2})$ algorithm [Shi79]. However, quite recently, Esteban and Ferrer-i-Cancho found and resolved a mistake in Shiloach's algorithm [EF17]. Chung observed that Shiloach's algorithm could be improved to $\mathcal{O}(n^2)$ but also proposed an even faster algorithm with a runtime in $\mathcal{O}(n^{1.585})$ [Chu84]. To the best of our knowledge, no faster or simpler algorithm has been proposed. At the same time, Adolphson and Hu developed an $\mathcal{O}(n \log(n))$ algorithm for *rooted trees* and proposed a non-trivial lower bound based on *Gomory-Hu* trees [AH73].

Other linear arrangement problems have also been solved on unweighted trees. For example, the *Min-cut Linear Arrangement* (McLA) problem wants to minimize the largest cut instead of the sum over all cuts and can be solved in polynomial time due to an algorithm by Yannakakis [Yan85]. However, the same problem was shown to be $\mathcal{NP}$-hard on *weighted trees* by Monien and Sudborough [MS88]. Additionally, the *Bandwidth Linear Arrangement* (BwLA) problem, which wants to minimize the length of the longest edge instead of the sum over all edges, has been shown to remain $\mathcal{NP}$-hard even on *unweighted trees* by Garey et al. [GGJK78].

Additionally, in the past 50 years, many heuristic algorithms have been proposed for the MinLA problem since exact algorithms are infeasible. These heuristics yield valid solutions which, however, may not be optimal. The best results among them are obtained by an algorithm from Safro et al. which is based on multi-level weighted edge contraction [SRB06]. Another good algorithm is introduced by Rodriguez et al. and uses simulated annealing [RHT08].

Besides the heuristic upper bounds, also some algorithms for lower bounds have been proposed. These lower bounds, do not yield valid solutions but help to get insights of the problem instance and can be used to improve heuristics. The first lower bounds were based on basic graph properties like the number of edges and the degree sequence of a graph. Albeit their simplicity, those are the only bounds applicable on large graphs. The best lower bounds so far have been achieved by Linear Programs. However, they have a rather high complexity and could therefore not be used on graphs with more than one thousand vertices.

All these algorithms have been evaluated on a small test suite of 22 graphs composed of uniform random graphs, geometric random graphs, graphs with known optima and graphs from real world applications like VLSI design and graph drawing competitions [Pet03b]. Additionally, the algorithms have been tested on a few larger graphs which arise in finite element discretization and were first used by Koren and Harel [KH02]. However, to the best of our knowledge, none of the algorithms have been tested on larger real world networks or larger *Geometric Inhomogeneous Random Graphs* (GIRG).

# 2 Preliminaries

In the following sections, we define basic concepts of graph theory and computational complexity theory. Additionally, we define notations that are used in this work. In particular, we define *linear arrangements* and *Gomory-Hu trees*.

## 2.1 Graph Theory

Graphs are combinatorial objects that are extensively studied in discrete mathematics and theoretical computer science. An *undirected simple graph* $G = (V, E)$ is a tuple of a set of vertices $V = \{1, 2, \dots, n\}$ and a set of edges $E \subseteq \big\{\{u, v\} \bigm| u, v \in V \text{ and } u \neq v\big\}$. In this work, we are only interested in this type of graphs. Therefore, we omit the words *undirected* and *simple* from now on.

We call two vertices $u$ and $v$ *independent* if the edge $e = \{u, v\}$ is not in $E$. Contrarily, we call $u$ and $v$ *adjacent* if $e$ is in $E$. Based on this, we define the *neighborhood* of the vertex $v$ as the set of vertices that are adjacent to $v$ and denote this with $N(v)$. Further, we define the *degree* $\deg(v)$ of $v$ as the size of its neighborhood.

Further, we denote the complement of a graph as $\overline{G}$. Two vertices in $\overline{G}$ are adjacent if they are independent in $G$ and are independent in $\overline{G}$ if they are adjacent in $G$.

In some occasions, we augment a graph with a *weight function* $w(e)\colon E \longrightarrow \mathbb{R}^+$ which assigns a positive real value to each edge in $E$. Such a graph is referred to as a *weighted graph*.

A *path* from vertex $v_1$ to $v_n$ is an $n$ tuple of vertices $P = (v_1, \dots, v_n)$. We call a path *valid* if any two adjacent vertices in the tuple are also adjacent in the graph. If not stated otherwise we will only talk about valid paths and omit the word valid from now on. Further, we call a path *simple* if each vertex occurs at most once in the tuple.

Based on this, we say that two vertices are *connected* if and only if there exists a path between them. Note that a path with $n = 1$ is always valid and thus the connection relation is reflexive. Further, since our graphs are undirected, the relation is also symmetric and transitive. Thus, the connected relation is an equivalence relation and we say that two vertices belong to the same *connected component* if they are in the same equivalence class i.e. they are connected by a path.

In case of a weighted graph, we also want to define the *length* of a path as the sum over the weights of all $n - 1$ edges along the path. Additionally, we define the *shortest path* as the path whose length is minimal among all possible paths connecting two vertices. Note that our weight functions assigns only positive weights. Thus, there exists at least one shortest path between any two connected vertices.

We define a *cut* as a partition of the vertex set of $G$ into two disjoint subsets. An edge $\{u, v\}$ *crosses* the cut if $u$ and $v$ do not belong to the same subset. The *cut-set* corresponding to a cut is the set of all edges that cross the given cut and the size of a cut is the size of the cut-set. In case of a weighted graph, the size of the cut is defined as the sum of weights over all edges in the cut-set.

### 2.1.1 Graph Classes

In addition to the above definitions we also want to name some specific graph classes. We call a graph an *independent set* if all pairs of vertices are pairwise independent. Contrarily, we call the graph where all vertices are pairwise adjacent a *clique*.

A *cycle* is a connected graph in which all vertices have degree 2. Further, we say that a graph is a *tree* if it is connected and contains no cycle. This implies, that for any pair of vertices there exists exactly one simple path connecting them. In this case, we call all vertices $v$ with $deg(v) \leq 1$ *leaves*. If all but one vertex in the tree are leaves, we may also call it a *star* and say that the remaining vertex is its *center*. In some occasions, we talk about *rooted trees*. In this case some vertex is called the root and all edges are implicitly directed away from the root. This implies that each vertex $v$ that is not the root has exactly one incoming edge. The vertex at the origin of the incoming edge is called the parent $p_v$ of $v$. This also enables us to define a sub-tree for each vertex $v$ where the sub-tree consists of all vertices that can be reached from $v$ without $p_v$. Further, we denote the number of vertices in such a sub-tree with $size(v)$.

We call a graph *bipartite* if its vertices can be partitioned into two independent sets and we say that a graph is *split graph* if its vertices can be partitioned into an independent set and a clique.

## 2.2 Linear Arrangements

*Linear Arrangement* problems are a class of combinatorial optimization problems where the vertices of a graph $G$ are to be ordered along a line, such that each vertex has an integer position and some cost function $c$ is optimized. In this work, we are interested in the *Minimum Linear Arrangement (MinLA)* problem, which is sometimes also referred to as *Optimal Linear Arrangement Problem* or *Optimal Linear Ordering Problem.* In this optimization problem, we want to minimize the total edge length of an arrangement of $G$. This version of the problem was first formulated by Harper [Har64] and was proven to be $\mathcal{NP}$-complete by Garey Johnson and Stockmeyer [GJS74].

Formally, let $\pi$ be a permutation of the vertices i.e. a bijective mapping from the vertex set to the set of integer positions $\{1, \ldots, |V|\}$. We refer to this permutation as *arrangement* or *embedding*. The *length* of an edge $\{u, v\} \in E$ for a fixed arrangement $\pi$ is the distance between $u$ and $v$ in $\pi$, i.e. $|\pi(u) - \pi(v)|$. The cost $c(\pi, G)$ of a *Linear Arrangement* is simply the sum of the edge lengths over all edges.

Let $cut(\pi, G, i)$ (for $1 \leq i < |V|$) denote the cut that separates the vertices with positions less than or equal to $i$ in $\pi$ from those with positions greater than $i$. Then, the cost $c(\pi, G)$ is equal to the sum over $cut(\pi, G, i)$ for all $i$, since an edge with length $x$ belongs to exactly $x$ cut-sets, which gives us another view on the *Optimal Linear Arrangement Problem.* A visualization of both of these views on the cost function is shown in Figure 2.1. The cost function $c$ can be formalized in the following two ways:

$$\begin{aligned} cut(\pi, G, i) &= \Big\{\{u, v\} \;\Big|\; u, v \in V \text{ and } \pi(u) \leq i \text{ and } \pi(v) > i\Big\} \\ c(\pi, G) &= \sum_{\{u,v\} \in E} \big|\pi(u) - \pi(v)\big| = \sum_{i=1}^{|V|-1} \big|cut(\pi, G, i)\big|. \end{aligned}$$

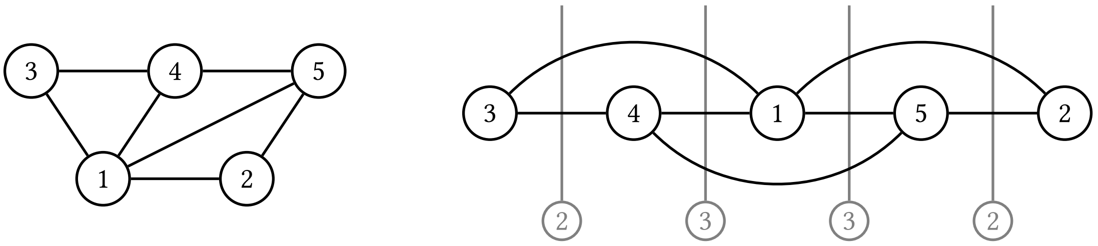


**Figure 2.1:** On the left, a graph $G$ with 5 vertices and 7 edges is drawn. On the right, the same graph is embedded on a line with order $\pi = (3, 5, 1, 2, 4)$. The embedding contains 3 edges of length 2 and 4 edges with length 1, thus, the total cost of this embedding is 10. Additionally, the 4 cuts and their sizes are shown in gray. The total sum of the cut-sizes is also 10.

Since we also want to handle *weighted* graphs, we need to adjust the definition of $c$ to include the weight of an edge. If we look at the sum of the edge lengths, then the natural generalization in this context is to multiply the length of each edge with its weight before summing these values up. Thus, changing the length of an edge $e$ by one changes the cost of the arrangement by $w(e)$. On the other side, we can naturally generalize the size of a cut as the sum of weights of edges in the cut-set instead of the amount of edges in the cut-set. Both generalizations lead to the same cost for a fixed arrangement which we will denote with $c(\pi, G, w)$.

If the weight of an edge $e$ is an integer, then it can also be seen as the number of multi-edges which connect two vertices. Therefore, the cost of an unweighted graph is equivalent to the cost of a graph where all weights are equal to 1. The generalized cost function of a linear arrangement for a weighted graph can be stated as:

$$c(\pi, G, w) = \sum_{\{u,v\}\in E} w\big(\{u,v\}\big) \cdot \big|\pi(u) - \pi(v)\big| = \sum_{i=1}^{|V|-1} \sum_{e\in cut(\pi,G,i)} w(e)\,.$$

## 2.3 Gomory-Hu Tree

We call a weighted tree a *Gomory-Hu tree* $T_G$ of a graph $G$ if both $T_G$ and $G$ have the same vertex set and further, each edge $e = \{s, t\}$ in $T_G$ corresponds to a minimal $s$-$t$-cut of size $w(e)$ in $G$. Formally, the two components of $T_G \setminus e$ must correspond to the two partitions of a minimal cut in $G$. Therefore, for any two vertices in $T_G$, each edge on the path between them corresponds to a cut that separates them.

The Gomory-Hu tree was introduced by Gomory and Hu, who also proposed an efficient algorithm to generate $T_G$ from $G$ [GH61]. A simpler algorithm was later presented by Gusfield [Gus90].

Note that even though a Gomory-Hu tree exists for every graph, the requirements do not necessarily define a unique Gomory-Hu tree for every graph. Therefore, the same graph $G$ can have multiple non-isomorphic Gomory-Hu trees as shown in Figure 2.2.

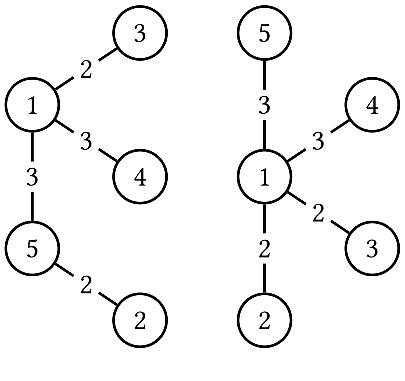


**Figure 2.2:** Two Gomory-Hu trees with their edge weights. Both trees belong to the graph in Figure 2.1.

# 3 Gomory-Hu Tree Bounds

The Gomory-Hu tree has various applications. Among them is a lower bound for the minimum linear arrangement problem that was proposed by Adolphson and Hu [AH73]. In the following sections, we investigate the relation between Gomory-Hu trees and the MinLA problem. Then we take a look at the MinLA problem on weighted trees and finally, we combine these to introduce a new heuristic upper bound based on an arrangement of a Gomory-Hu tree.

## 3.1 Lower Bound

**Theorem 3.1** (Adolphson and Hu [AH73])**:** *The sum of all edge weights in $T_G$ is a lower bound for the value of the minimum linear arrangement problem for $G$.*

Even though this gives us a lower bound of theoretical value, this approach does not perform well in practice. Petit tested several lower bounds for the MinLA problem [Pet03b]. However, he observed that this bound has the same quality as bounds that are computed only from the number of edges or the degree sequence, despite that the Gomory-Hu tree approach is computationally more complex and takes the actual structure of the graph into account.

## 3.2 Upper Bound

**Theorem 3.2:** *For a Gomory-Hu tree $T_G$ from $G$ and a fixed permutation $\pi$, the cost $c(\pi, G)$ is always less than or equal to the cost $c(\pi, T_G)$.*

*Proof.* Let us denote the set of edges in $G$ with $E$ and the set of edges in $T_G$ with $E_T$. Further, let $\sigma := \pi^{-1}$. Given an edge $e = \{u, v\}$ in $E$ with $\sigma(u) < \sigma(v)$, we know that this edge contributes 1 to each cut in the interval $\big[\sigma(u) \dots \sigma(v)\big)$. In $E_T$, the edge $e$ may not be present, but $u$ and $v$ are still connected by a unique simple path $P$ in $T_G$. By definition of the Gomory-Hu tree, we also know that each edge in $P$ represents a cut between $u$ and $v$ in $G$. Obviously, $e$ belongs to any cut which separates $u$ and $v$ in $G$, since it connects them. Therefore, $e$ contributes 1 to the weight of each edge in $P$.

Since $P$ connects $u$ and $v$, it still has to span across all cuts in $\big[\sigma(u) \dots \sigma(v)\big)$. In conclusion this implies that if $e$ contributes to a cut in $G$, then it also contributes the same amount to the same cut in $T_G$. Since this is true for all edges in $E$, we know that $cut(\sigma, G, i) \leq cut(\sigma, T_G, i)$ for each cut $i$. Which, in turn, implies that $c(\pi, G) \leq c(\pi, T_G)$, since $c(\pi, G)$ and $c(\pi, T_G)$ are just the sum of the $|V| - 1$ cuts. ∎

Since Theorem 3.2 yields an upper bound for the cost of a fixed arrangement $\pi$, it also yields an upper bound on the minimal cost for an arrangement of $G$. However, it is worth noting that the upper bound is not tight, i.e., the cost for an optimal arrangement of $T_G$ can be greater than the minimal cost required to arrange $G$. Therefore, it is preferable to find a good arrangement $\pi$ for $T_G$ and then evaluate the costs $c(\pi, G)$ to get a better upper bound. Unfortunately, even this approach is not tight, since the arrangement $\pi$ that minimizes $c(\pi, T_G)$ and the arrangement $\pi'$ that minimizes $c(\pi', T_G)$ are normally not the same as seen in Figure 3.1.

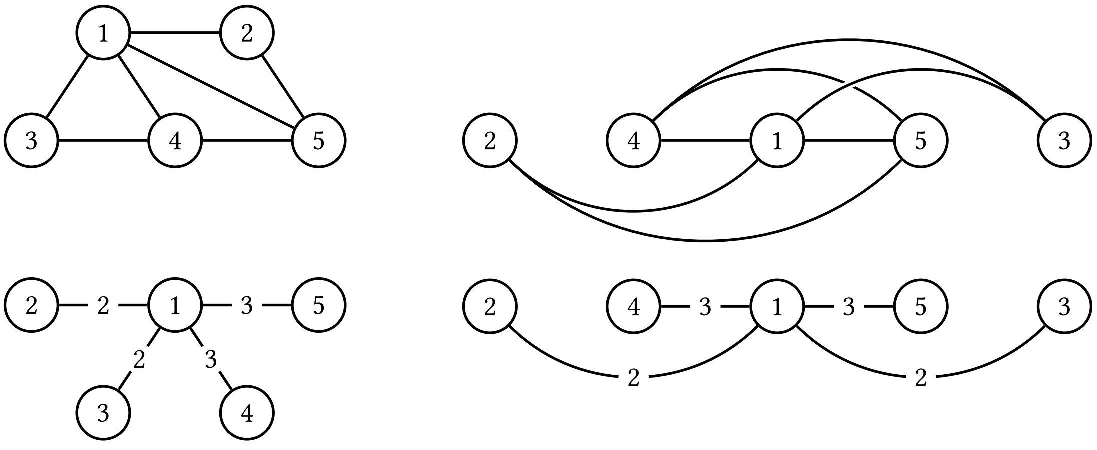


**Figure 3.1:** On the left, a graph $G$ and a Gomory-Hu tree $T_G$ are given. On the right, $G$ and $T_G$ are arranged with $\pi = (2, 4, 1, 5, 3)$. Even though the arrangement is optimal for $T_G$, the cost for $G$ is 4 higher then necessary. An optimal solution can be seen in Figure 2.1

## 3.3 Weighted Trees

Many polynomial algorithms have been proposed to optimally arrange unweighted trees in respect to the total edge length [GK76 | Shi79 | Chu84]. Goldberg and Klipker observed that in the minimal linear arrangement $\pi$ of a tree, the simple path between the first and the last vertex in the arrangement is monotone, i.e., either all vertices on the path have increasing values of $\pi$ or decreasing values. This property is the basis for all of the aforementioned algorithms. However, this property is not true for *weighted* trees, as can be seen in Figure 3.2. This counter example shows that the path between the first and last vertex may need to take multiple turns even though the depicted arrangement is optimal. Thus, unfortunately, none of those approaches can be generalized to solve the problem on *weighted* trees. Further, it is not known whether the MinLA problem for weighted trees can be solved in polynomial time or if it is $\mathcal{NP}$-complete.

Adolphson and Hu proposed an algorithm which handles *rooted* weighted trees [AH73]. However, their algorithm only arranges a tree optimally under the restriction that each vertex has to be placed to the right of its parent. Obviously, this requirement increases the cost of the arrangement. However, we were not able to find a tree for which the costs of such an arrangement increases by more than a factor two which leads us to the following conjecture.

**Conjecture 3.3:** *Arranging a rooted tree such that each vertex is placed to the right of its parent costs at most twice as much as arranging the tree optimally in terms of total edge length.*

If Conjecture 3.3 is true, then it would imply that the algorithm proposed by Adolphson and Hu gives a 2-approximation for the MinLA problem on weighted trees. Thus, it yields both an upper and a lower bound. Note that for general graphs, it was proven that no constant approximation for the MinLA problem exists [DKSV06 | AMS07]. Further, we assume that MinLA problem for a weighted tree itself is hard as stated in Conjecture 3.4. We believe this, since even graphs with simple structures as shown in Figure 3.2 have a complicated layout.

**Conjecture 3.4:** *Given a tree $T$ with edge weights polynomialy bounded in the number of vertices, it is $\mathcal{NP}$-complete to decide if there exists a linear arrangement with total edge weight less than $c$. The corresponding MinLA problem for $T$ is $\mathcal{NP}$-hard.*

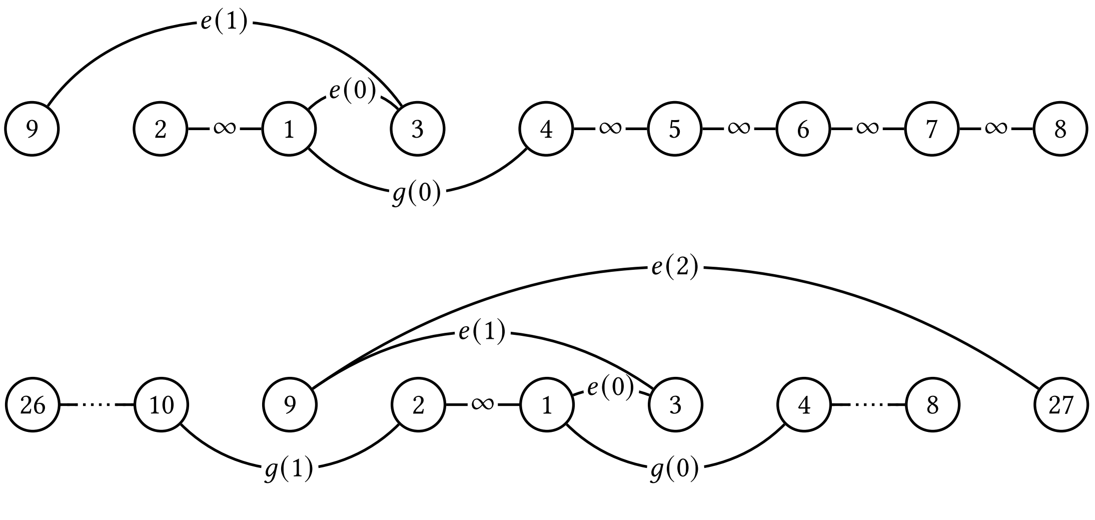


**Figure 3.2:** In the picture we can see a counter example for the monotone property of optimal embedding of a weighted tree. Let $e(i) := 9^{-i}$ and $g(i) := 3^{-i}$, then the given arrangements are optimal. However, the paths between the first and the last vertex in the arrangements are not monotone. Note that $\infty$ only needs to be sufficiently large and can e.g. be replaced by $|V|^2 \cdot e(0)$. Additionally, note that all weights can be multiplied by a large enough power of 3 to make them integers. Thus, these counter examples also work with only integer weights which are polynomial in $|V|$. Further, this example can be generalized to find a larger counter examples.

## 3.4 Gomory-Hu Tree Heuristic

We propose a new heuristic algorithm that shares some similarities with Shiloach's algorithm for unweighted trees. Our algorithm can arrange a weighted *rooted* tree $T$ optimally under the restriction that all vertices in a sub-tree have to be arranged consecutively. In combination with Theorem 3.2 this yields an upper bound since this allows us to arrange the Gomory-Hu tree af a given graph. However, to do this, we first solve a simpler problem with the following lemma.

**Lemma 3.5:** *Let $r$ be the root vertex, and $c_i \in N(r)$ be a child of $r$. Further, let $size(c_i)$ denote the size of the sub-tree rooted at $c_i$. If sub-trees have to be arranged consecutively and each vertex has to be placed to the right of its parent, then the sub-trees of $r$ have to be sorted by the ratio of $size(c_i)$ to $w(\{r, c_i\})$ in non-decreasing order in the optimal arrangement.*

*Proof.* We prove this by contradiction. Assume we have an optimal arrangement $\pi$ and a pair of sub-trees $c_i$ and $c_j$ where $c_j$ appears *directly* before $c_i$ in $\pi$, but they are not sorted correctly. Then the difference of weighted sizes $c_i$ and $c_j$ is negative:

$$\begin{aligned} & \frac{size(c_i)}{w(\{r, c_i\})} < \frac{size(c_j)}{w(\{r, c_j\})} \\ \Longrightarrow\ & size(c_i) \cdot w(\{r, c_j\}) < size(c_j) \cdot w(\{r, c_i\}) \\ \Longrightarrow\ & size(c_i) \cdot w(\{r, c_j\}) - size(c_j) \cdot w(\{r, c_i\}) < 0\,. \end{aligned}$$

However, by moving the sub-tree $c_j$ behind $c_i$ the edge $\{r, c_j\}$ gets exactly $size(c_i)$ longer, whereas the edge $\{r, c_i\}$ gets $size(c_j)$ shorter. Thus, the cost changes to:

$$size(c_i) \cdot w\big(\{r, c_j\}\big) - size(c_j) \cdot w\big(\{r, c_i\}\big)\,.$$

Since this amount is negative, the initial assumption has to be wrong. Considering that the required order is *total*, this also implies that non adjacent sub-trees have to be sorted correctly, which proves the lemma. ∎

Since sub-trees need to be arranged consecutively, a recursive algorithm can solve the problem. The algorithm first appends the root to the result, then sorts its sub-trees according to Lemma 3.5 and, finally, calls itself for each sub-tree.

### 3.4.1 Algorithm

Now, we want to solve the problem without the restriction that each vertex has to be placed to the right of its parent. This can also be done with a recursive algorithm, since sub-trees still need to be arranged consecutively. However, this time we need to decide which sub-tree is placed at which side of the root, This can be decided optimally with a *Dynamic Program* (DP). Note that this is sufficient since the order of the sub-trees on both sides is still implied by Lemma 3.5.

Let $r$ be the root vertex, and $c_i \in N(r)$ for $i \in \big\{1, \dots, deg(r)\big\}$ be the root of the $i$-th sub-tree if all sub-trees are sorted according to Lemma 3.5. Further, let $dp_{i,j}$ (for $0 \leq j < size(r)$) denote the minimal cost of arranging the first $i$ sub-trees such that $j$ vertices are placed to the right of $r$. Note that $dp_{i,j}$ ignores the cost to arrange the sub-trees themselves, since this cost is independent of the placement of the sub-tree. To calculate $dp_{i,j}$, two cases need to be considered. If the $i$-th sub-tree is placed on the right side, then we need to arrange the previous $i-1$ sub-trees optimally, under the restriction that exactly $j - size(c_i)$ vertices have to be placed on the right side. If the $i$-th sub-tree is placed on the left instead, we require that $j$ vertices from the previous $i-1$ trees have to be placed on the right. In both cases, we additionally need to consider the cost of the edge $\{r, c_i\}$ which is stretched depending on the number of vertices which were already placed on the same side. The final $dp$ relation looks as follows:

$$\begin{aligned}
dp_{0,0} &= 0 \\
dp_{0,j} &= \infty \text{ if } j \neq 0 \\
dp_{i,j} &= \infty \text{ if } j < 0 \\
dp_{i,j} &= \min\left\{\begin{matrix} dp_{i-1,j-size(c_i)} + \big(j - size(c_i)\big) \cdot w\big(\{r, c_i\}\big), \\ dp_{i-1,j} + \left(\sum\limits_{k=1}^{i-1} size(c_k)\right) \cdot w\big(\{r, c_i\}\big) \end{matrix}\right\}.
\end{aligned}$$

When arranging the sub-trees of a vertex without a parent, i.e., the root of the complete tree, we just want to find the minimal $dp_{deg(r),j}$ over all $j$ and place the sub-trees according to the DP. For any other sub-tree, let $p$ denote the parent of its root. In this case, we additionally need to consider the cost of the edge $\{p, r\}$ which is stretched depending on the number of vertices that are placed between $r$ and $p$. Thus, we need to minimize $dp_{deg(r),j} + j \cdot w\big(\{p, r\}\big)$ in this case. Note that the cost of the edge $\{p, r\}$ is shared between the sub-tree and its parent.

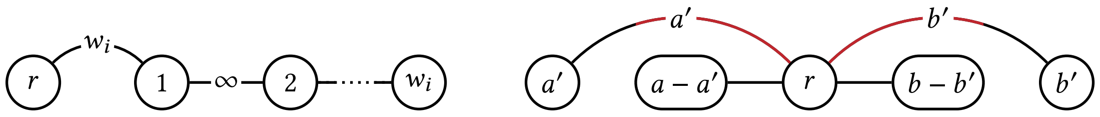


**Figure 3.3:** On the left, we see the sub-tree constructed for each weight of a partition instance. On the right, we see how swapping the outermost sub-trees affects the cost of the linear arrangement. Only the edges marked in red change their length.

### 3.4.2 Runtime

**Lemma 3.6:** *The DP can be evaluated for each vertex in a total running time of $\mathcal{O}(|V|^2)$.*

*Proof.* For a sub-tree rooted in vertex $c$, it takes $\deg(c) \cdot size(c)$ time to evaluate the DP, since any state can be evaluated in constant time and no vertex needs to be placed further than $size(c)$ away from $c$ in an optimal arrangement. Considering that $c$ contributes 1 to $size(p)$ for any direct or indirect parent $p$ of $c$, we can see that $c$ contributes at most $2 \cdot |E|$ to the total runtime. Since our graph is a tree, we know that there are $|V| - 1$ edges and $|V|$ vertices. Thus, the total running time is bounded by $2 \cdot |V|^2$ which proves the lemma. ∎

Lemma 3.6 directly implies that the complete tree can be arranged optimally in $\mathcal{O}(|V|^2)$. Further, the DP only has at most $\deg(v) \cdot |V|$ states for a root vertex $v$. Thus, the calculation for the optimal arrangement requires at most $\mathcal{O}(\Delta(T) \cdot |V|)$ memory, where $\Delta(T)$ denotes the maximum degree for any vertex in $T$.

Let $dp_i$ denote the DP values $dp_{i,j}$ for all $j$. We also refer to $dp_i$ as the $i$-th *row* of the DP. Note that the DP state $(i, j)$ only depends on the states $(i - 1, k)$ for some $k \leq j$. Thus, the $i$-th row of the dp only depends on the previous row. This allow us to evaluate the minimal cost of the DP with only $\mathcal{O}(|V|)$ memory. However, we also want to reconstruct the optimal solution. To do this, we need to find all states that contributed to the minimal cost. This can be done with *backtracking* without increasing the running time asymptotically. Unfortunately, we have to traverse the DP states in reverse order during the backtracking. Since we cannot reconstruct the previous row of the DP from a given one we have to store all rows to allow an efficient reconstruction. The observation that all following states can be calculated from a given row can, however, still be used to reduce the memory consumption. On the other hand, this would increase the total running time.

### 3.4.3 Complexity Results

We now want to reason a elaborate further about the running time of our algorithm. To do this, we first need to introduce the *partition* problem. In this problem $n$ integer weights $w_i$ are given with a total weight of $W$. The problem asks if it is possible to partition the weights into two disjoint sets such that both sets have a total weight of $W/2$. Deciding this problem is known to be weakly $\mathcal{NP}$-hard. To the best of our knowledge, the fastest algorithm to solve the partition problem is a pseudo-polynomial DP which has a running time of $\mathcal{O}(n \cdot W)$ [GJ79].

In Theorem 3.7, we show that any algorithm that can decide which sub-tree has to be arranged on which side of the root in $o(deg(r) \cdot |V|)$ would yield an algorithm that is faster than the aforementioned DP. Therefore, we assume that even the restricted problem where sub-trees have to be arranged consecutively, cannot be solved more efficiently than with our algorithm. Further, Theorem 3.7 shows that any greedy algorithm to solve this problem will likely fail, since this would imply a greedy algorithm for the partition problem.

**Theorem 3.7:** *Let $P = \{w_1, \dots, w_n\}$ be a partition instance. Further, let $W$ be the total weight of all $w_i$. Then, $P$ can be encoded in a rooted weighted tree $T$ with $W + 1$ vertices, such that any optimal linear arrangement corresponds to a solution to the partition problem, should it exist.*

*Proof.* Let $r$ be the root of the tree we want to construct. For each $w_i$, we add a path of length $w_i$. The path is connected to $r$ with an edge with of $w_i$. The other weights along the path are assumed to be infinite, as shown in Figure 3.3. Note that the infinite weight can be modeled as $W^3$, since stretching all other edges to their maximum can cost no more than $W^3$ in total. Further, note that this high weight enforces that all sub-trees of $r$ have to be arranged consecutively.

We prove the theorem by contradiction. Suppose that the partition instance has a solution but there exists an optimal arrangement where $a$ vertices are placed to the left of $r$ and $b = W-a$ to the right of $r$ with $a < b$. Note that the ordering of the sub-trees on each side has no influence on the cost of the arrangement, since all sub-trees have the same size to weight ratio. Therefore, we can assume that the given solution and a partition solution only differ in the outermost sub-trees. This implies that the arrangement can be transformed into a partition by swapping some of the leftmost and rightmost sub-trees. Let $a'$ be the number of vertices which move from left to right and $b'$ be the number of vertices which move from right to left. Since the resulting arrangement should be a partition, we know that $a - a' + b' = b - b' + a'$. If we swap the sub-trees of size $a'$ and $b'$, then the cost of the linear arrangement changes by:

$$-a' \cdot (a - a') + a' \cdot (b - b') - b' \cdot (b - b') + b' \cdot (a - a')\,.$$

Since we move $a'$ vertices from left to right, we change the length of some edges whose total weight is also $a'$. Previously, those edges had to span over $a - a'$ vertices to reach the sub-trees. After the change, those edges need to span over $b - b'$ vertices. Note that we ignore some of the cost to connect the sub-trees to $r$ since it does not change. Similarly, the cost for the sub-trees which move from right to left change. The edges which change during the swap are also highlighted in Figure 3.3.
We now simplify the formula for the change of the cost:

$$\begin{aligned}
& - a' \cdot (a - a') + a' \cdot (b - b') - b' \cdot (b - b') + b' \cdot (a - a') \\
&= (b' - a') \cdot (a - a') + (a' - b') \cdot (b - b') \\
&= (b' - a') \cdot (a - a' + b' - b) \\
&= (b' - a') \cdot (b - b' + a' - b) && \text{Since } a - a' + b' = b - b' + a' \\
&= (b' - a') \cdot (a' - b') \\
&= -(b' - a')^2 \\
&< 0\,.
\end{aligned}$$

Since the change of cost is always negative, the initial arrangement could not be optimal. This proves that only a partition solution is optimal. ∎

# 4 Linear Program

It seems like Linear Programs are the most promising approach to finding good lower bounds for the MinLA problem [Pet13]. In the current literature, two approaches proposed by Caprara et al. can be found which use a similar set of constraints [CLS11 | Cap+11]. In Section 4.1 and Section 4.2, we revisit both proposed approaches. After that, we briefly discuss a third approach in Section 4.3 and introduce some new constraints which can be combined with all three approaches. In the last section, we try to improve the first approach with the help of *community detection.*

## 4.1 Decorous Lower Bounds

The first linear program proposed by Caprara et al. uses $\mathcal{O}(m)$ real-valued variables. Each variable corresponds to the length of one edge. This idea is very natural but requires many constraints to yield good results. In the following, we take at look at the constrains proposed by Caprara et al.

**Rank Constraints** Given a subgraph $H$ in $G$, the sum over all variables which correspond to edges in $H$ has to be at least as large as the cost for the minimum linear arrangement of $H$. These types of constraints are called *rank constraints* and each subgraph $H$ corresponds to exactly one rank constraint. Adding all possible rank constraints would trivially give a perfect lower bound since $G$ itself is also a subgraph of $G$. However, this is obviously not feasible since there are too many possible subgraphs and the constraints themselves would require us to know the cost of the minimum linear arrangement.

**Projected Rank Constraints** It is hard to find good rank constraints since $H$ has to be a subgraph of $G$ and one needs to be able to solve the MinLA problem for $H$. The projected rank constraints try to overcome the first drawback by introducing a *projection* operation which allows us to add valid constraints for any graph $H$ as long as we can solve the MinLA problem for $H$. First, one calculates the shortest path between any pair of vertices in terms of total edge length. Then, the *projected rank* constraint consists of the sum over all shortest $a$-$b$-paths for each edge $\{a, b\}$ in $H$. This means that the constraint ensures that the sum over all shortest paths has to be at least as large as the cost of the minimum linear arrangement of $H$. Caprara et al. showed that these constraints are valid even though some edges appear multiple times in the constraint since an edge can be part of multiple shortest paths [CLS11]. Note that this also implies that the projected rank constraints are a super-set of the set of rank constraints.

**Path Constraints** To further improve projected rank constraints, one should observe that a shortest $a$-$b$-path in $G$ is not allowed to be shorter than the corresponding edge $\{a, b\}$ in $G$ if it exists. For each edge where this requirement is violated, a *path constraint* can be added.

### 4.1.1 Algorithm

Caprara et al. propose an iterative *branch-and-cut* algorithm which consists of two steps that are repeated. First, they find a valid assignment of edge lengths that does not violate any of the current constraints. In the second step, they search new projected rank constraints that are violated by the solution and add them. Since the projected rank constraints still require the MinLA cost for the corresponding subgraph $H$, Caprara et al. decided to only search for graphs from graph classes where the optimal solution is either known or an optimal algorithm was proposed. Namely, they use projected rank constraints based on the following graph classes:

I. Stars
II. Cliques
III. Circuits
IV. Complete Bipartite Graphs
V. Trees

Note that for each of these graph classes at least $2^{|V|}$ constraints can be found. This is true since for each possible subset of vertices at least one graph per graph classes exists. For all classes besides cliques there actually exist multiple graphs per subset. Therefore, we cannot simply try all possible subgraphs belonging to one of the aforementioned graph classes to find violated constraints. Thus, the second step needs some work to be efficient. For stars, Caprara et al. proposed a polynomial algorithm that finds some violated star constraints should there exist one. For the four other graph classes, they proved that finding violated constraints is $\mathcal{NP}$-hard and proposed some simple heuristics to find violated constraints.

Note that Caprara et al. used rank constraints based on trees and unfortunately decided to use Shiloach's algorithm to calculate the MinLA cost for the trees. This is a problem since Esteban and Ferrer-i-Cancho found an error in Shiloach's paper [EF17]. Since we have no access to the original implementation by Caprara et al., we do not know how exactly they evaluated the MinLA cost for trees. If they used Shiloach's algorithm to generate a permutation, they likely used invalid constraints since permutation is not optimal and thus, not a lower bound for the tree constraint. However, since they found a tight lower bound for the complete binary tree of ten levels and Shiloach's algorithm can directly calculate the cost, it is more likely that they used these costs. In this case, the added constraints seem to be valid but not as tight as possible.

## 4.2 Betweenness Variables

The second approach was proposed by Caprara et al. and can be seen as a refinement of the previous one [Cap+11]. This approach needs $|V| - 2$ binary variables for each edge $e = \{u, v\}$. Each such variable states whether or not a vertex $b$ lies between $u$ and $v$, hence the name betweenness variable. The length of an edge can then be formulated as one plus the sum over all $|V| - 2$ variables corresponding to that edge. Since the formula for the edge length is linear any constraint from the previous approach can be restated as a constraint for this approach. This simply requires us to replace the variable of an edge with the mentioned formula. However, the extended set of variables allows to express the following additional constraints.

**Triangle constraints** For any three vertices which are pairwise connected by edges, the sum over the three betweenness variables has to be 1, since exactly one vertex is in the middle of the other two. Note that for these constraints, a projected version can also be introduced. In this case, the sum over the three shortest paths has to be at least 1.

**Feasibility constraints** The biggest advantage of this approach is that it can be used to find a tight lower bound and even allows the reconstruction of an optimal permutation of the vertices. Let $M$ be the $|E|\times|V|$ matrix with $M_{e,v}$ being 1 if and only if there exists a betweenness variable for $e$ and $v$ that is assigned the value 1. Each row of this matrix corresponds to one edge and each column corresponds to one vertex. We can observe that $M$ corresponds to a permutation if and only if the matrix has the *consecutive ones property for rows*.

This property states that the columns can be rearranged such that the ones in each row are consecutive. If we rearrange the matrix with the permutation $\pi$ such that the ones in each row are consecutive, then the matrix corresponds to a valid arrangement of $G$, since each edge can be embedded exactly as required by the matrix with the exact same cost as for that row plus one. Thus, the LP yields a lower bound that is equal to the upper bound of the cost of the arrangement $\pi$.

Fortunately, such matrices with the consecutive ones property for rows can be fully characterized by a set of forbidden sub-matrices. This implies that we can introduce constraints that prohibit these sub-matrices and ensure that the matrix is valid.

### 4.2.1 Algorithm

These constraints directly lead to a branch-and-cut algorithm to solve the MinLA problem. If the solution to the LP is not integral, then a cutting plane is added to remove the non-integral solution. Otherwise, the consecutive ones property of $M$ is checked with an algorithm proposed by Booth and Lueker [BL76]. If the property is violated, then a new feasibility constraint is introduced. After this, a new solution of the LP is calculated. However, if at some point no new constraint can be added, then the solution is optimal and a permutation can be reconstructed.

This basic algorithm can be further sped up with additional constraints. Namely, by adding the *projected triangle constraints* and any projected constraint from the previous model. The algorithm from Caprara et al. reused *path* and *star* constraints since these constraints can be efficiently checked if they are violated [Cap+11]. The resulting algorithm yields better results than the first Linear Program. Note that this new algorithm is also able to optimally solve the problem with enough computation time. However, it converges much slower than the first Linear Program and thus, is not competitive on larger graphs.

## 4.3 Maximum Linear Arrangement

We now want to propose a completely new approach that is based on the *Maximum Linear Arrangement* (MaxLA) problem. In the MaxLA problem, we want to arrange the vertices such that the sum over all edges is maximized. These two problems are closely related. The optimal permutation for the MinLA of $G$ is equal to the optimal permutation for the MaxLA of $\overline{G}$ where $\overline{G}$ denotes the complement graph of $G$. Further, the sum over the optimal cost for the MinLA of $G$ and the MaxLA of $\overline{G}$ is equal to $\binom{|V|+1}{3}$. This relation has already been used in $\mathcal{NP}$-completeness proofs by Garey et al. and by Even and Shiloach [GJS74 | Eve75].

We tried to create a Linear Program for the MaxLA approach with a variable for each pair of vertices. If a pair of vertices is *not* connected by an edge in $G$, then the corresponding variable contributes to the objective function, which we want to *maximize*. This results in a drawback, since a huge amount of variables is required to formulate the LP. However, this could also be an advantage since the complement graph is much denser. Thus, rank constraints are tighter since fewer edges are considered that do not directly contribute to the objective function. Unfortunately, not many results for the MaxLA problem are known and thus, we could only test a limited amount of constraints which we will describe in the following.

**Edge Length Constraint** There can be at most one edge with length $|V| - 1$ and at most two edges with length $|V| - 2$ or, in general, at most $k$ edges with length $|V| - k$. Thus, we have an upper bound we can apply on any set of edges. To easily find a set of edges that violate this requirement, we sort all edges belonging to $\overline{G}$ in non-ascending order and check for each prefix by increasing length if it violates this constraint. In this case, we add the new constraint. This can be done in $\mathcal{O}\big(n^2 \log(n)\big)$ and ensures that we reach the trivial bound of zero for the MinLA problem.

**Star Constraint** For any star, we can add a *rank* constraint. The optimal MaxLA arrangement of a star has its center at the leftmost position and all children at the rightmost positions. Thus, a star with $k$ children can contribute at most $\sum_{i=1}^{k} \big(|V| - i\big) = k \cdot |V| - \binom{k+1}{2}$ to the total cost of an arrangement. To find a violated star constraint with center $v$, we can sort all edges $\{u, v\}$ in $\overline{G}$ in non-ascending order and again check all prefixes. If we do this for all possible root vertices, then we again have a running time of $\mathcal{O}\big(n^2 \log(n)\big)$.

Further, we can also add the MinLA star constraints for the edges not in $\overline{G}$. This ensures that those edges cannot become too short which in itself has no effect on the result but in combination with other constraints results in better bounds.

**Complete Bipartite Graph Constraints** Further, we can add a *rank* constraint for any complete bipartite subgraph. To calculate the maximal cost for a complete bipartite graph with $\ell$ vertices in one partition and $r$ vertices in the other partition, we first solve the simpler case where $\ell + r = |V|$. In this case, we can simply look at the minimal cost to embed the complement and subtract this from the cost for a clique on $\ell + r$ vertices. The complement of the graph would be one clique on $l$ and another on $r$ vertices. To solve the case with $\ell + r \leq |V|$, we need to observe how we the total cost changes. Since the additional vertices are independent, they can only contribute to the cost by enlarging other edges. Thus, each additional vertex can only increase the cost by the size of the maximum cut. For a bipartite graph, any edge belongs to the maximum cut and thus, the total cost grows by $\ell \cdot r$ for any vertex not belonging to the bipartite graph. In total, we get that the maximal cost for a bipartite graph is:

$$\begin{aligned}
&\binom{\ell + r + 1}{3} - \binom{\ell + 1}{3} - \binom{r + 1}{3} + \ell \cdot r \cdot \big(|V| - \ell - r\big) \\
= {} & |V| \cdot \ell \cdot r - \frac{r \cdot \ell^2 + \ell \cdot r^2}{2} .
\end{aligned}$$

We can use the same heuristic to find bipartite graphs with large cost that Caprara et al. used to find bipartite graphs with small cost [CLS11]. Note that finding a violated bipartite graph constraint is $\mathcal{NP}$-hard and thus a heuristic is needed[CLS11].

**Complete Split Graph Constraint** A *complete split graph* $G_{n,k}$ is a clique of size $n$ and $k$ independent vertices connected to each vertex in the clique. If we only consider split graphs with $n + k = |V|$, then the graph is the complement of a clique. Thus, the maximal cost of an arrangement is:

$$\binom{|V|+1}{3} - \binom{k+1}{3}.$$

Finding a violated split graph constraint is $\mathcal{NP}$-hard, since we are basically searching an independent set of size $k$ with minimal cost which in turn is $\mathcal{NP}$-hard for edge weights in $\{0, 1\}$ [GJ78]. Therefore, we use a heuristic to find violated constraints. To be exact, we use the heuristic proposed by Caprara et al. to find cliques [CLS11].

Note that the constraint $n + k = |V|$ is not necessary, i.e., we could also search for other violated split graph constraints. However, this would be more time consuming and the cost formula would be more complex. Therefore, we decided to only use the simplified version.

**Clique Constraint** The cost for a clique of size $k$ is bounded by the cost to embed the clique on integers from 1 to $k$ plus the cost for $|V| - k$ times the maximum cut. Since the maximum cut splits the graph into two parts of almost equal size, the total cost is:

$$\binom{k+1}{3} + \left\lfloor \frac{k}{2} \right\rfloor \cdot \left\lceil \frac{k}{2} \right\rceil \cdot (|V| - k).$$

Finding a violated clique constraint is again $\mathcal{NP}$-hard [CLS11]. However, we can use the heuristic proposed by Caprara et al. as we did before [CLS11].

Note that this also implies that we can introduce a *complete split graph* constraint to the MinLA LP, since it is the complement of a clique. Thus, for a complete split graph $n, k$ the MinLA cost would be:

$$\binom{n+k+1}{3} - \binom{k+1}{3} - \left\lfloor \frac{k}{2} \right\rfloor \cdot \left\lceil \frac{k}{2} \right\rceil \cdot n.$$

However we restrict ourselves to only use complete split graphs with $n + k = |V|$. In this case, the heuristic to find a clique can be reused to find violated constraints.

### 4.3.1 Algorithm

We use an iterative *branch-and-cut* algorithm which consists of two steps that are repeated, in a similar fashion as the first Linear Program proposed by Caprara et al.. First, we find a currently valid assignment of edge lengths. Then, we search for violated constraints in the aforementioned fashion. Every violated constraint we find is added to the Linear Program.

The MaxLA approach could also be combined with betweenness variables. However, this would require even more variables which would further slow down the approach. The pre-tests of this algorithm already showed that the explained approach is not even competitive with the first Linear Program and thus, we did not implement a MaxLA algorithm with betweenness variables.

## 4.4 Linear Program with Community Detection

So far, this work aimed at improving the quality of lower bounds by introducing new constraints and developing a new approach based on the MaxLA problem with the goal of providing better bounds. However, the results show that there is not much room for improvement on the proposed lower bounds. Therefore, we focus on reducing the time complexity of the approaches without reducing the quality of the resulting lower bounds in the following sections.

The first step in this process is to improve the running time by reducing the number of different constraints. Caprara et al. already mentioned that the projected Star and Clique constraints in combination with Path constraints are the most important ones [CLS11]. Therefore, we will limit ourselves to these constraints and focus on speeding up the algorithms used to find violated constraints of these three types.

### 4.4.1 Path Constraints

To find violated path constraints, we originally solved the *All Pairs Shortest Paths (APSP)* problem in the first step by calculating the shortest path tree starting in each vertex. Since our graphs are sparse, we use *Dijkstra's algorithm* which results in a total running time in $\mathcal{O}(n \cdot (n+m)\log(n))$ to find all $n$ shortest path trees [Dij59]. Using the shortest paths to add violated path constraints takes at most $\mathcal{O}(m \cdot n)$ time and thus, the running time of this whole step is dominated by Dijkstra's algorithm.

The running time of this step can be reduced by limiting the total size of all calculated shortest path trees i.e. by limiting the output size of the first step. In our implementation, we compute all shortest path trees in parallel and stop after we found $c \cdot m$ shortest paths where $c$ is a constant which we set to 10. In total, the shortest path trees can contain at most $n + c \cdot m$ vertices instead of the $n^2$ vertices contained in the solution of the APSP problem.

The running time now strongly depends on the structure of the graph since the number of paths Dijkstra's algorithm considers does not directly depend on the output size. However, since we are mostly interested in real world graphs that are sparse, we expect this step to become faster. The main reason to reduce the output size is not to speed up this stage but to reduce the number of edges which have to be considered for star and clique constraints. Thus, speeding up this stage will help us to speed up later stages which are more time consuming in general. On the other hand, this can reduce the quality of the result since we do not look at all possible path constraints and also look at fewer constraints later.

### 4.4.2 Star Constraints

We simplify this step by just searching for the *largest* violated star constraint for each vertex while only considering edges which we found in the previous stage. Finding the biggest violated star constraint for a single vertex can be done in $\mathcal{O}(n \cdot \log(n))$, since we only need to sort all outgoing edges. The running time of this step is, however, dominated by the previous step, since each path only contributes to two stars and all paths were already built in the previous stage. Thus, this stage does not worsen our time complexity at all.

### 4.4.3 Clique Constraints

The heuristic to find violated clique constraints, proposed by Caprara et al., is the most time consuming part of the original algorithm. Adding a single clique constraint takes up to $\mathcal{O}(n^3)$ time since the clique contains $\mathcal{O}(n^2)$ projected edges, and each of these edges can consist of up to $n-1$ real edges. Therefore, we decided to try a completely new approach to find clique constraints based on *community detection.*

A community in a graph is a set of vertices which are densely connected internally and loosely connected to other parts of the graph. In case of a weighted graph, the edges inside a community should additionally be short while connections to other parts of the graph should contain edges with larger weights.

Our approach to find clique constraints is to use the weighted graph from the first step to detect communities within it and add a clique constraint for each community. The idea behind this approach is that the sum over all projected edges in the community are expected to be small since they are well connected. Thus, we assume that the gap between the current costs and the required costs to embed a clique is large which in turn should increase the lower bound. In our algorithm, we use the *Leiden Algorithm* proposed by Traag, Waltman and Van Eck [TWV19] and the implementation provided by *NetworKit* [SSM15]. We used the algorithm as a black box and with its default parameters set in NetworKit, the only change we applied was to set a flag to make its behavior deterministic and reproducible.

To actually build the constraint from a given community, we need to find shortest paths between all vertices within the community. To implement this step efficiently, we first build a shortest path tree for each vertex in the community which connects it to every other vertex in the community. This is done by applying Dijkstra's algorithm on the original graph and stopping as soon as all vertices from the community have been visited. Afterwards, we count on how many shortest paths each edge lies, since this is equal to the contribution of this edge to the clique constraint. This can be done in linear time on the shortest path tree with a simple depth-first search which accumulates the number of vertices below an edge in the shortest path tree. Thus, the complexity of this stage only depends on the complexity of the community detection algorithm and the time to build the shortest path trees.

Note that the computation of the shortest path trees could be sped up by restricting Dijkstra's algorithm to vertices within the same community. However, this would decrease the quality of the calculated constraint. We performed some initial experiments and observed that this optimization yields an insignificant speedup.

### 4.4.4 Algorithm

The resulting algorithm is an iterative *branch-and-cut* algorithm which works as described in this section. We generate an initial solution by adding a star constraint for each vertex with all its neighbors and solving the corresponding linear program. This initial constraints ensure that the result will be at least as good as the trivial degree lower bound from Petit [Pet03b]. After that we will iteratively search for violated constraints and then add them to the LP. We then solve the LP to generate a new solution for which we repeat the previous step. However, we additionally restrict the number of constraints which we add in each such iteration.

We noticed that the LP becomes much slower if we add too many constraints at once. Therefore, we only consider a limited amount of constraints to add. To be precise, we only add a constant amount $c_1$ of constraints. In our case we chose $c_1 = 1000$. We choose the $c_1$ *best* constraints ranked by the absolute difference of the left hand side and right hand side in the inequality of the constraint. This metric is chosen since it is simple to compute and corresponds to the maximum change of the LP bound if we add the given constraint. From the constraints which we did not add, we again choose a constant amount $c_2$ that we consider again in the next iteration, in our case we keep $c_2 = 4000$ constraints. Thus, in total, our algorithm only needs to store $c_1 + c_2 = 5000$ constraints during the first step of each iteration.

Further, we noticed that the bottleneck with the worst running time depending on the number of vertices is the computation of clique constraints. Therefore, we decided to limit the size of the clique constraints we compute. In our implementation, we ignore all communities found by the Leiden algorithm that contain more than $c_3 = 500$ vertices. This ensures that an iteration will not consume too much time.

# 5 Evaluation

In this chapter, we compare our algorithms with state-of-the-art algorithms for the MinLA problem. Our main focus lies on the lower bounds produced by the combination of the Linear Program with the Leiden algorithm for community detection.

### 5.0.1 Environment

All tests were performed on a machine operating on openSUSE Leap 15.2 (kernel 5.3.18). The machine has two Intel Xeon Gold 6144 CPUs, each with eight cores clocked at 3.5 GHz and $8 \times 64$ KB of L1 cache, $8 \times 1$ MB of L2 cache and 24.75 MB of shared L3 cache. Note that all our algorithms and libraries are implemented and used in a strict non parallel way, albeit that most parts allow trivial parallelization. We decided to do this to be more comparable to previous work. Further, the machine was used exclusively for one experiment at a time.

Our algorithms were implemented in C++ and compiled with g++ version 10.3.0 and optimization level 2. Further, we used the following third party libraries: Gurobi 9.5 to solve our Linear Programs [Gur22], NetworKit's implementation of the Leiden Algorithm to find communities [SSM15] and the maximum flow implementation from Bläsius, Friedrich and Weyand to compute Gomory-Hu trees [BFW20].

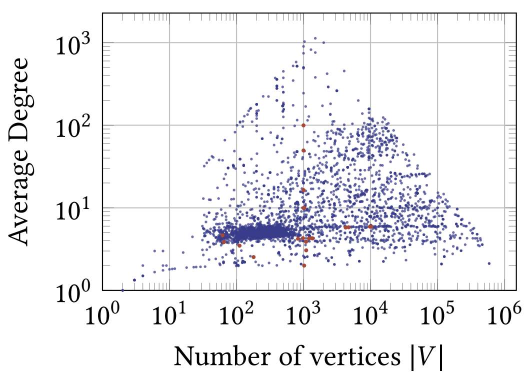
### 5.0.2 Benchmark Instances

We consider two sets of benchmark instances. The first was established by Petit [Pet03b] but contains just 22 graphs. However, these graphs have been used for evaluating MinLA algorithms since then and can therefore be used to compare our work with older results. More information about the actual graphs can be found in [Pet03b | Pet03a]. In Table 5.1, we give an overview over some basic graph properties of the benchmark instances. Further, we display the best known upper bounds and a more detailed list of various lower bounds computed so far. To the best of our knowledge, the table shows the results of all MinLA related papers published.

Additionally, we consider real world graphs published by Rossi and Ahmed [RA15]. Some of the basic properties of this set of graphs are visualized in Figure 5.1. These benchmark instances have been used for various algorithms but, to the best of our knowledge, not in the context of Minimum Linear Arrangements. Some graphs are part of both test cases, since the network repository is a collection of various graphs and Petit also included some graphs used in different fields. At least the graph drawing (gd) instances and the finite element graphs are also part of the network repository. Further, we did not use the network repository graphs as is, but pre-processed them by only considering the largest connected component and only including graphs with at most $10^6$ edges. Note that the MinLA problem is solved for connected components separately. Additionally, we also consider some geometric inhomogeneous random graphs generated by Bläsius et al. [Blä+19].

**Table 5.1:** This table contains the best known lower and upper bounds for the 22 benchmark instances introduced by Petit [Pet03b]. Note that the lower bounds obtained by Petit are the best results of various algorithms partially proposed by other authors [Pet03b | AH73 | JM92]. The best lower bound for each graph is highlighted in bold. The upper bound is the optimum chosen from various other works and should correspond to the best known upper bound [RHT08 | SRB06 | RHT05 | Pet03b | KH02].

| Name | $\|V\|$ | $\|E\|$ | LB[Pet03b] | LB[CLS11] | LB[Cap+11] | UB |
|---|---|---|---|---|---|---|
| randomA1 | 1 000 | 4 974 | **140 634** | - | - | 866 968 |
| randomA2 | 1 000 | 24 738 | **4 429 294** | - | - | 6 522 206 |
| randomA3 | 1 000 | 49 820 | **11 463 259** | - | - | 14 194 583 |
| randomA4 | 1 000 | 8 177 | **601 130** | - | - | 1 717 176 |
| randomG4 | 1 000 | 8 173 | **39 972** | - | - | 140 211 |
| bintree10 | 1 023 | 5 120 | 1 277 | **3 696** | - | 3 696 |
| hc10 | 1 024 | 2 112 | **349 525** | - | - | 523 776 |
| mesh33x33 | 1 089 | 1 022 | **31 680** | 20 042 | - | 31 680 |
| 3elt | 4 720 | 13 722 | **44 785** | - | - | 217 220 |
| airfoil1 | 4 253 | 12 289 | **40 221** | - | - | 272 931 |
| crack | 10 240 | 30 380 | **95 347** | - | - | 1 489 266 |
| whitaker3 | 9 800 | 28 989 | **144 854** | - | - | 1 143 645 |
| c1y | 828 | 1 749 | 14 101 | **59 971** | - | 62 230 |
| c2y | 980 | 2 102 | 17 842 | **76 253** | - | 78 757 |
| c3y | 1 327 | 2 844 | 23 417 | **113 801** | - | 123 145 |
| c4y | 1 366 | 2 915 | 21 140 | **106 942** | - | 114 936 |
| c5y | 1 202 | 2 557 | 19 217 | **88 741** | - | 96 850 |
| gd95c | 62 | 144 | 292 | 443 | **506** | 506 |
| gd96a | 1 076 | 1 676 | 5 155 | **77 860** | - | 95 242 |
| gd96b | 111 | 193 | 702 | 1 281 | **1 404** | 1 416 |
| gd96c | 65 | 125 | 241 | 402 | **519** | 519 |
| gd96d | 180 | 228 | 595 | **2 021** | 1 578 | 2 391 |



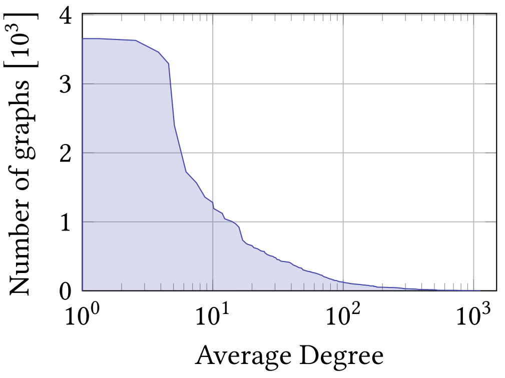


**Figure 5.1:** On the left, we see a point for each benchmark instance. The instances from Petit are highlighted in red. On the right, we can see the accumulated degree distribution. A point $(x, y)$ along the curve tells us that there are $y$ benchmark instances with an average degree not less than $x$. This shows that most of the instances have a small average degree.

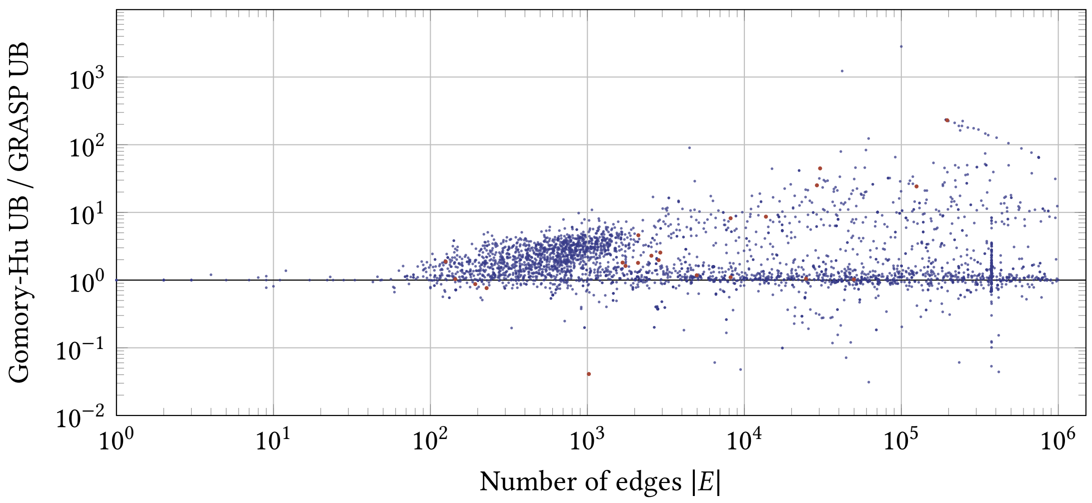


**Figure 5.2:** Results of the Gomory-Hu tree based upper bound compared with a GRASP based approach. A point $(x, y)$ corresponds to a result on an instance with $x$ edges where the ratio of the Gomory-Hu bound to the GRASP bound is $y$. The 22 instances provided by Petit are marked in red, all others are marked blue. For every point above the marked axis $y = 1$, the Gomory-Hu bound performs worse than the GRASP approach. Above the axis are 2877 out of 3559 instances. Additionally, 96 instances could not be solved and are not included in the plot.

## 5.1 Upper Bound

In this section, we evaluate the Gomory-Hu tree based upper bound. Note that more advanced heuristics are known [RHT08 | SRB06 | RHT05 | Pet03b | KH02]. However, no implementation of those algorithms is available and implementing them is outside the scope of this work. Thus, we implemented a simple algorithm: the *Greedy randomized adaptive search procedure (GRASP)* [FR89]. The results on random instances provided by Petit suggest that the GRASP approach does not perform too bad on random graphs compared to the best known algorithms as can be seen in Table 5.2. Thus, it is reasonable to compare the Gomory-Hu tree approach with the GRASP approach.

Our GRASP algorithm tries to find an initial solution by always appending some vertex to an existing solution which increases the costs as little as possible, starting with an empty solution. After that, it tries to swap single vertices inside the solution to locally improve the result. We did this local search on $10^5$ different initial solutions and took the optimum.

In Table 5.2 and Figure 5.2, we can see that even the simple GRASP algorithm yields better results in most cases on real world instances and the benchmark instances provided by Petit. Also, note that the GRASP method is faster in practice and has better asymptotic runtime than the Gomory-Hu based approach. Therefore, the Gomory-Hu bound for general graphs is only of theoretical interest and not competitive with the state-of-the-art. We can also see in Table 5.2 that the GRASP approach is not much worse than the best known upper bound (less than a factor of two) on the random graphs. Therefore, we expect it to perform well as well on the larger test set of real world graphs that is displayed in Figure 5.2.

However, the algorithm can still be useful to find solutions for weighted trees since to the best of our knowledge, no other algorithm was developed for this problem. Further, the instance *bintree10* shows that the algorithm performs quite well for this tree.

**Table 5.2:** This table contains the upper bounds known for the 22 benchmark instances introduced by Petit [Pet03b]. As before, the column labeled with *UB* is the optimum of various other works [RHT08 | SRB06 | RHT05 | Pet03b | KH02]. We can see that the Gomory-Hu tree approach does not perform well. The only case where it yields good results was the instance *bintree10* which is a binary tree.

| Name | $\|V\|$ | $\|E\|$ | UB | GRASP UB | Gomory-Hu UB |
|---|---:|---:|---:|---:|---:|
| randomA1 | 1 000 | 4 974 | **866 968** | 1 223 388 | 1 442 158 |
| randomA2 | 1 000 | 24 738 | **6 522 206** | 7 468 809 | 7 833 556 |
| randomA3 | 1 000 | 49 820 | **14 194 583** | 15 558 135 | 15 890 126 |
| randomA4 | 1 000 | 8 177 | **1 717 176** | 2 216 171 | 2 440 958 |
| randomG4 | 1 000 | 8 173 | **140 211** | 260 066 | 2 132 095 |
| bintree10 | 1 023 | 5 120 | **3 696** | 112 382 | 4 608 |
| hc10 | 1 024 | 2 112 | **523 776** | 2 396 811 | 551 141 435 |
| mesh33x33 | 1 089 | 1 022 | **31 680** | 45 766 | 210 345 |
| 3elt | 4 720 | 13 722 | **217 220** | 904 128 | 7 826 492 |
| airfoil1 | 4 253 | 12 289 | **272 931** | 19 064 719 | 461 500 798 |
| crack | 10 240 | 30 380 | **1 489 266** | 2 165 917 | 96 868 833 |
| whitaker3 | 9 800 | 28 989 | **1 143 645** | 1 335 544 | 33 581 891 |
| c1y | 828 | 1 749 | **62 230** | 220 600 | 357 054 |
| c2y | 980 | 2 102 | **78 757** | 280 590 | 504 157 |
| c3y | 1 327 | 2 844 | **123 145** | 442 458 | 872 796 |
| c4y | 1 366 | 2 915 | **114 936** | 377 936 | 964 572 |
| c5y | 1 202 | 2 557 | **96 850** | 330 250 | 757 007 |
| gd95c | 62 | 144 | **506** | 782 | 807 |
| gd96a | 1 076 | 1 676 | **95 242** | 250 909 | 454 936 |
| gd96b | 111 | 193 | **1 416** | 2 245 | 1 960 |
| gd96c | 65 | 125 | **519** | 688 | 1 294 |
| gd96d | 180 | 228 | **2 391** | 5 039 | 3 841 |

## 5.2 Lower Bound

In this section, we evaluate the Linear Program which utilizes community detection with Leiden's algorithm. In Table 5.3, we can see the results of our algorithm on the benchmark instances introduced by Petit [Pet03b]. The table shows that our algorithm performs well for real world graphs even though it was given only 10 minutes of computation time per instance. The lower bounds obtained by Petit had unlimited computation time [Pet03b] and the lower bounds obtained by Caprara et al. were calculated in up to 24 hours [CLS11 | Cap+11]. However, take these times with a grain of salt since the algorithms were executed on different hardware and with a different LP solver. Still, 10 minutes are much less computation time than in the previous works.

On the other hand, Table 5.3 also shows that for graphs like *hc10*, *mesh33x33* or the Erdős-Rényi random graphs, the results are quite poor. These graphs do not have a natural community structure like most real world graphs. Thus, the Leiden algorithm will not find communities which yield good clique constraints.

To compare our algorithm on all instances in our large test set, we re-implemented the *decorous lower bound* proposed by Caprara et al. [CLS11]. We did, however, exclude the tree constraints. We argue that this does not make a huge difference since Caprara et al. already

**Table 5.3:** This table contains the best lower bounds for the benchmark instances introduced by Petit [Pet03b]. Note that the lower bounds obtained by Petit are the best results of various algorithms partially proposed by other authors [Pet03b | AH73 | JM92]. The best lower bound for each graph is highlighted in bold. We can see that our algorithm improved the bound for the geometric random graph *randomG4* and is not far off for the real world graphs in comparison with the decorous lower bound by Caprara et al [CLS11], even though our algorithm had less computation time.

| Name | $\lvert V\rvert$ | $\lvert E\rvert$ | [Pet03b] | [CLS11] | [Cap+11] | Our |
|---|---|---|---|---|---|---|
| randomA1 | 1 000 | 4 974 | **140 634** | - | - | 82 336 |
| randomA2 | 1 000 | 24 738 | **4 429 294** | - | - | 1 610 485 |
| randomA3 | 1 000 | 49 820 | **11 463 259** | - | - | 2 465 981 |
| randomA4 | 1 000 | 8 177 | **601 130** | - | - | 197 591 |
| randomG4 | 1 000 | 8 173 | 39 972 | - | - | **64 250** |
| bintree10 | 1 023 | 5 120 | 1 277 | **3 696** | - | 2 847 |
| hc10 | 1 024 | 2 112 | **349 525** | - | - | 77 947 |
| mesh33x33 | 1 089 | 1 022 | **31 680** | 20 042 | - | 12 769 |
| 3elt | 4 720 | 13 722 | **44 785** | - | - | 42 090 |
| airfoil1 | 4 253 | 12 289 | **40 221** | - | - | 39 825 |
| crack | 10 240 | 30 380 | **95 347** | - | - | 67 415 |
| whitaker3 | 9 800 | 28 989 | **144 854** | - | - | 63 438 |
| c1y | 828 | 1 749 | 14 101 | **59 971** | - | 28 597 |
| c2y | 980 | 2 102 | 17 842 | **76 253** | - | 33 783 |
| c3y | 1 327 | 2 844 | 23 417 | **113 801** | - | 42 313 |
| c4y | 1 366 | 2 915 | 21 140 | **106 942** | - | 34 221 |
| c5y | 1 202 | 2 557 | 19 217 | **88 741** | - | 35 470 |
| gd95c | 62 | 144 | 292 | 443 | **506** | 417 |
| gd96a | 1 076 | 1 676 | 5 155 | **77 860** | - | 26 853 |
| gd96b | 111 | 193 | 702 | 1 281 | **1 404** | 1 258 |
| gd96c | 65 | 125 | 241 | 402 | **519** | 365 |
| gd96d | 180 | 228 | 595 | **2 021** | 1 578 | 1 965 |

reported that their effect is insignificant. For the comparison, both algorithms were only given 10 minutes per instance since the test set contains a huge number of graphs and we opted for a same computation time comparison. Note that we also reevaluated the Linear Program proposed by Caprara et al. on the instances introduced by Petit with the time limit.

The results of the equal time comparison are visualized in Figure 5.3. We can see that our approach performs better for almost all instances. Especially for larger graphs, our approach tends to perform better or is able to solve them at all. The main reason for this is that our approach is able to do much more iterations as seen in Figure 5.4. Even for the largest graphs in our test set, our algorithm is able to perform multiple iterations within the given time whereas the decorous Linear Program takes too long even for middle sized graphs.

We also observe that some design choices of our implementation can be seen in Figure 5.4. The plot shows that for graphs with more than roughly $10^4$ edges, the number of iterations starts to have a greater variance. We assume that around this point the Leiden algorithm starts to find communities that are larger than the threshold value $c_3$ for which we calculate clique constraints.

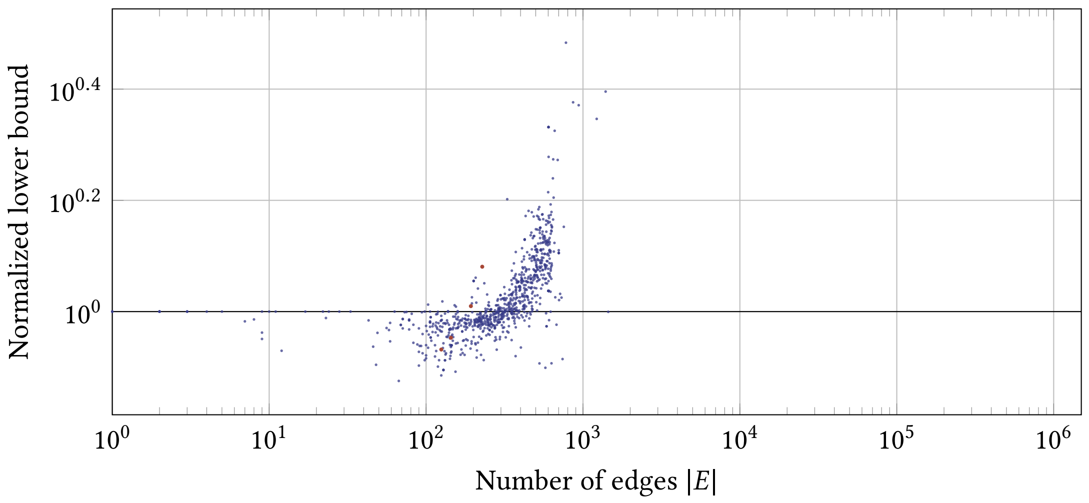


**Figure 5.3:** Results of our Linear Program compared with the Linear Program proposed by Caprara et al. [CLS11]. A point $(x, y)$ corresponds to a result on an instance with $x$ edges where the ratio of the decorous lower bound to our lower bound is $y$. The 22 instances provided by Petit are marked in red, all others are marked blue. For every point above the marked axis $y = 1$, our approach performs better than the decorous approach. Below the axis are only 403 out of 3559 instances. Additionally, 2636 instances could not be solved by the decorous Linear Program and are not included in the plot at all.

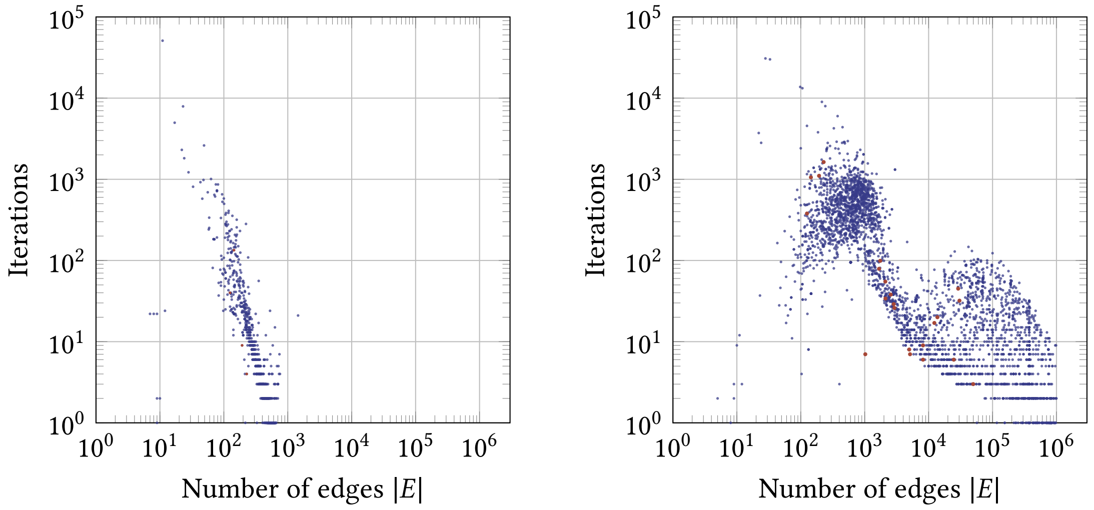
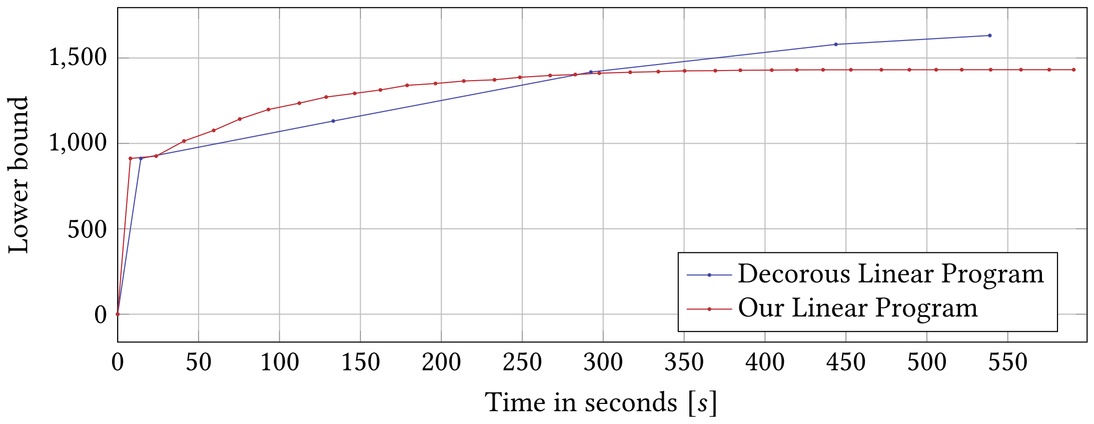


**Figure 5.4:** Results of our Linear Program compared with the Linear Program proposed by Caprara et al. [CLS11]. Number of iterations for both Linear Programs. On the left, we see the plot for the decorous lower bound by Caprara et al. [CLS11]. On the right, we see the plot for our approach. For graphs with few edges, we can observe that the Linear Program converges within a few iterations and then aborts. For larger graphs, we can see that our approach is able to do significantly more iterations. Note that the left plot only contains graphs where the decorous Linear Program finished at least one iteration. Therefore, 2636 graphs are omitted.



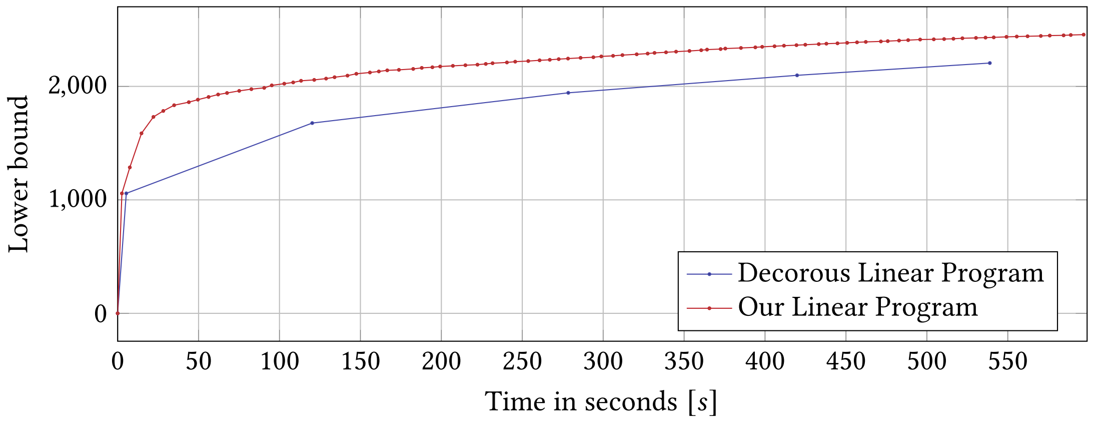


**Figure 5.5:** The results of the decorous lower bound algorithm by Caprara et al. [CLS11] in comparison to our algorithm, plotted over time. The first plot corresponds to the graph *gd96d* with 228 edges on 180 vertices. The second plot corresponds to the graph *reptilia-tortoise-network-bsv* with 373 edges on 134 vertices. We can see that our algorithm performs better initially since it can do more iterations in the same time. However, it will eventually get overtaken by the decorous lower bound if the later has enough time. In the second diagram, the decorous LP did not have enough time to overtake our approach since the iterations took too long for this graph.

Figure 5.5 also shows that our algorithm performs well if only a short amount of time is available while being worse in the long run. This is no big surprise since our main goal was to improve the speed of one iteration. In particular, we can see that our algorithm needs multiple iterations to improve the bound by the same amount as one iteration of the original LP does. For larger graphs, this results in better lower bounds by our algorithm since the decorous lower bound is not able to perform nearly enough iterations to converge. Note that the two graphs chosen for Figure 5.5 were chosen arbitrarily but the diagrams for other graphs of our test set look similar. Interestingly, our algorithm seems to perform better on the larger graph *reptilia-tortoise-network-bsv* than on the smaller graph *gd96d*. We assume that this comes from the fact that the second graph has a good *average clustering coefficient* i.e. it has a rather natural community structure. On one hand, this property improves the runtime of the Leiden algorithm but it also seems to improve the quality of our algorithm. Luckily, many real world graphs yield a community structure from which our algorithm benefits.

# 6 Conclusion

In this work, we took a closer look at the state-of-the-art algorithms on the MinLA problem with a focus on lower bounds. As mentioned before, the number of exact results is limited to very few graph classes which do not appear in practice. Therefore, there is still a need for algorithms which can be used on large real word graphs. For upper bounds, the current state-of-the-art algorithms are already able to solve graphs with millions of edges in short time. For lower bounds, on the other hand, the current algorithms need days for graphs with just a few thousand edges. In this work, we improved the runtime of one of the best lower bound algorithms while sacrificing only a little bit of quality of the bounds. This improved algorithm enabled us to compute lower bounds for graphs with millions of edges in just a few minutes. Especially promising is that we were able to improve the lower bound for the geometric random graph *randomG4* which was provided by Petit [Pet03a]. Since geometric random graphs share many properties with real world graphs, we expect our algorithm to perform well on real world graphs as well.

## 6.1 Future Work

We already made some claims in the previous chapters which need to be proven. Additionally, our work can be improved and extended in some ways. More precisely these are the parts which look most promising.

**Weighted Trees** How hard is the minimum linear arrangement problem on weighted trees? Is it $\mathcal{NP}$-hard as we suspect in Conjecture 3.4? Also, an interesting question is if the algorithm proposed by Adolphson and Hu is an approximation as formulated in Conjecture 3.4.

**Constraints** Can our approach be improved by including more constraints? Especially, are there more efficient ways to find constraints based on bipartite graphs or split graphs to include them in our algorithm without losing too much performance?

**Leiden algorithm** For now we used the Leiden algorithm as a black box. Which affect does the resolution parameter of the Leiden algorithm have on the quality of of the lower bound i.e. can we improve the results by finding communities of other sizes?

**Dynamic Program** Can the faster lower bound in combination with dynamic programming be used to build a branch-and-bound algorithm which solves the MinLA problem for real world graphs?

# Bibliography

[AH73] D. Adolphson and Tien Chung Hu. “Optimal linear ordering”. In: *SIAM Journal on Applied Mathematics* Volume 25.3 (1973), pp. 403–423. URL: *https://doi.org/10.1137/0125042*.

[AMS07] Christoph Ambuhl, Monaldo Mastrolilli, and Ola Svensson. “Inapproximability results for sparsest cut, optimal linear arrangement, and precedence constrained scheduling”. In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS07)*. IEEE. 2007, pp. 329–337. URL: *https://doi.org/10.1109/FOCS.2007.40*.

[BFW20] Thomas Bläsius, Tobias Friedrich, and Christopher Weyand. “Efficiently computing maximum flows in scale-free networks”. In: *arXiv preprint arXiv:2009.09678* (2020).

[BL76] Kellogg S Booth and George S Lueker. “Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms”. In: *Journal of computer and system sciences* Volume 13.3 (1976), pp. 335–379. URL: *https://doi.org/10.1016/S0022-0000(76)80045-1*.

[Blä+19] Thomas Bläsius, Tobias Friedrich, Maximilian Katzmann, Ulrich Meyer, Manuel Penschuck, and Christopher Weyand. “Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs”. In: *27th Annual European Symposium on Algorithms (ESA 2019)*. Vol. 144. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 21:1–21:14. URL: *https://doi.org/10.4230/LIPIcs.ESA.2019.21*.

[BS87] JAYARAM BHASKER and SARTAJ SAHNI. “Optimal linear arrangement of circuit components”. In: *Journal of VLSI and computer systems* Volume 2.1-2 (1987), pp. 87–109.

[Cap+11] Alberto Caprara, Marcus Oswald, Gerhard Reinelt, Robert Schwarz, and Emiliano Traversi. “Optimal linear arrangements using betweenness variables”. In: *Mathematical Programming Computation* Volume 3.3 (2011), p. 261. URL: *https://doi.org/10.1007/s12532-011-0027-7*.

[Chu84] Fan-Rong King Chung. “On optimal linear arrangements of trees”. In: *Computers & mathematics with applications* Volume 10.1 (1984), pp. 43–60. URL: *https://doi.org/10.1016/0898-1221(84)90085-3*.

[CLS11] Alberto Caprara, Adam N Letchford, and Juan-José Salazar-González. “Decorous lower bounds for minimum linear arrangement”. In: *INFORMS Journal on Computing* Volume 23.1 (2011), pp. 26–40. URL: *https://doi.org/10.1287/ijoc.1100.0390*.

[Coh+06] Johanne Cohen, Fedor Fomin, Pinar Heggernes, Dieter Kratsch, and Gregory Kucherov. “Optimal linear arrangement of interval graphs”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2006, pp. 267–279. URL: *https://doi.org/10.1007/11821069_24*.

[Cou16] David Coudert. “A note on Integer Linear Programming formulations for linear ordering problems on graphs”. PhD thesis. Inria; I3S; Universite Nice Sophia Antipolis; CNRS, 2016. URL: *https://hal.inria.fr/hal-01271838/*.

[Dij59] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* Volume 1.1 (1959), pp. 269–271. ISSN: 0945-3245. URL: *https://doi.org/10.1007/BF01386390*.

[DKSV06] Nikhil R Devanur, Subhash A Khot, Rishi Saket, and Nisheeth K Vishnoi. “Integrality gaps for sparsest cut and minimum linear arrangement problems”. In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 2006, pp. 537–546. URL: *https://doi.org/10.1145/1132516.1132594*.

[EF17] Juan Luis Esteban and Ramon Ferrer-i-Cancho. “A correction on Shiloach’s algorithm for minimum linear arrangement of trees”. In: *SIAM Journal on Computing* Volume 46.3 (2017), pp. 1146–1151. URL: *https://doi.org/10.1137/15M1046289*.

[Eve75] Shimon Even. “NP-completeness of several arrangement problems”. In: *Technical Report; Department of computer Science* Volume 43 (1975). URL: *http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1975/CS/CS0043.pdf*.

[FR89] Thomas A Feo and Mauricio GC Resende. “A probabilistic heuristic for a computationally difficult set covering problem”. In: *Operations research letters* Volume 8.2 (1989), pp. 67–71. URL: *https://doi.org/10.1016/0167-6377(89)90002-3*.

[GGJK78] Michael R. Garey, Ronald L. Graham, David S. Johnson, and Donald Ervin Knuth. “Complexity results for bandwidth minimization”. In: *SIAM Journal on Applied Mathematics* Volume 34.3 (1978), pp. 477–495. URL: *https://doi.org/10.1137/0134037*.

[GH61] Ralph E. Gomory and Tien Chung Hu. “Multi-terminal network flows”. In: *Journal of the Society for Industrial and Applied Mathematics* Volume 9.4 (1961), pp. 551–570. URL: *https://doi.org/10.1137/0109047*.

[GJ78] Michael R Garey and David S Johnson. ““strong”np-completeness results: Motivation, examples, and implications”. In: *Journal of the ACM (JACM)* Volume 25.3 (1978), pp. 499–508. URL: *https://doi.org/10.1145/322077.322090*.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 174. freeman San Francisco, 1979, pp. 90–92.

[GJS74] Michael R. Garey, David S. Johnson, and Larry Stockmeyer. “Some simplified NP-complete problems”. In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. 1974, pp. 47–63. URL: *https://doi.org/10.1145/800119.803884*.

[GK76] M. K. Goldberg and I. A. Klipker. “An algorithm for minimal numeration of tree vertices”. In Russian. In: *Sakharth. SSR Mecn. Akad. Moambe* Volume 81.3 (1976), pp. 553–556. URL: *https://www.cs.rpi.edu/~goldberg/publications/arrang.pdf*.

[Gur22] LLC Gurobi Optimization. *Gurobi Optimizer Reference Manual*. 2022. URL: *https://www.gurobi.com*.

[Gus90] Dan Gusfield. “Very simple methods for all pairs network flow analysis”. In: *SIAM Journal on Computing* Volume 19.1 (1990), pp. 143–155. URL: *https://doi.org/10.1137/0219009*.

[Har64] Lawrence Hueston Harper. “Optimal assignments of numbers to vertices”. In: *Journal of the Society for Industrial and Applied Mathematics* Volume 12.1 (1964), pp. 131–135. URL: *https://doi.org/10.1137/0112012*.

[JM92] Martin Juvan and Bojan Mohar. “Optimal linear labelings and eigenvalues of graphs”. In: *Discrete Applied Mathematics* Volume 36.2 (1992), pp. 153–168. URL: *https://doi.org/10.1016/0166-218X(92)90229-4*.

[KH02] Yehuda Koren and David Harel. “A multi-scale algorithm for the linear arrangement problem”. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer. 2002, pp. 296–309. URL: *https://doi.org/10.1007/3-540-36379-3_26*.

[MS88] Burkhard Monien and Ivan Hal Sudborough. “Min cut is NP-complete for edge weighted trees”. In: *Theoretical Computer Science* Volume 58.1-3 (1988), pp. 209–229. URL: *https://doi.org/10.1016/0304-3975(88)90028-X*.

[Pet03a] Jordi Petit. “Benchmark instances for the minimum linear arrangement problem”. In: 2003. URL: *https://www.cs.upc.edu/~jpetit/MinLA/Experiments/jpetit-extra.tar.gz*.

[Pet03b] Jordi Petit. “Experiments on the minimum linear arrangement problem”. In: *Journal of Experimental Algorithmics (JEA)* Volume 8 (2003). URL: *https://doi.org/10.1145/996546.996554*.

[Pet13] Jordi Petit. “Addenda to the survey of layout problems”. In: *Bulletin of EATCS* Volume 3.105 (2013).

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: *https://networkrepository.com*.

[RH08] Habib Rostami and Jafar Habibi. “Minimum linear arrangement of chord graphs”. In: *Applied Mathematics and Computation* Volume 203.1 (2008), pp. 358–367. URL: *https://doi.org/10.1016/j.amc.2008.04.051*.

[RHT05] Eduardo Rodriguez-Tello, Jin-Kao Hao, and Jose Torres-Jimenez. “Memetic algorithms for the MinLA problem”. In: *International Conference on Artificial Evolution (Evolution Artificielle)*. Springer. 2005, pp. 73–84. URL: *https://doi.org/10.1007/11740698_7*.

[RHT08] Eduardo Rodriguez-Tello, Jin-Kao Hao, and Jose Torres-Jimenez. “An effective two-stage simulated annealing algorithm for the minimum linear arrangement problem”. In: *Computers & Operations Research* Volume 35.10 (2008), pp. 3331–3346. URL: *https://doi.org/10.1016/j.cor.2007.03.001*.

[Shi79] Yossi Shiloach. “A minimum linear arrangement algorithm for undirected trees”. In: *SIAM Journal on Computing* Volume 8.1 (1979), pp. 15–32. URL: *https://doi.org/10.1137/0208002*.

[SRB06] Ilya Safro, Dorit Ron, and Achi Brandt. “Graph minimum linear arrangement by multilevel weighted edge contractions”. In: *Journal of Algorithms* Volume 60.1 (2006), pp. 24–41. URL: *https://doi.org/10.1016/j.jalgor.2004.10.004*.

[SSM15] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. *NetworKit: A Tool Suite for Large-scale Complex Network Analysis*. 2015. arXiv: *1403.3005* `[cs.SI]`. URL: *https://arxiv.org/abs/1403.3005*.

[TWV19] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. “From Louvain to Leiden: guaranteeing well-connected communities”. In: *Scientific reports* Volume 9.1 (2019), pp. 1–12. URL: *https://doi.org/10.1038/s41598-019-41695-z*.

[Yan85] Mihalis Yannakakis. “A polynomial algorithm for the min-cut linear arrangement of trees”. In: *Journal of the ACM (JACM)* Volume 32.4 (1985), pp. 950–988. URL: *https://doi.org/10.1145/4221.4228*.