Universität Heidelberg

Institut für Informatik
Mathematische Logik und Theoretische Informatik

Bachelorarbeit

A Simple and Efficient Algorithm for Minimum Cuts of Weighted Graphs

Name: Martin Koloseus
Matrikelnummer: 4084758
Betreuer: Dr. Wolfgang Merkle
Datum der Abgabe: 29.12.2023

Abgabedatum: 29.12.2023

Zusammenfassung

Wir betrachten einen ungerichteten, gewichteten Graphen mit nicht-negativen Kantenwerten. Das Minimaler-Schnitt-Problem besteht darin, eine Partition der Menge der Knoten V in zwei Teilmengen X und $V \setminus X$ zu finden, sodass die Summe der Gewichte aller Kanten mit einem Endpunkt in X und einem Endpunkt in $V \setminus X$ minimal ist.
Wie von Bhardwaj et al. [4] dargelegt, ist das Minimaler-Schnitt-Problem ein grundlegendes Problem in der Graphentheorie und hat mehrere Anwendungen in Themen wie Netzwerksicherheit [7], Clusteranalyse [5] und der Lösung des Handlungsreisendenproblems [2].
Ein besonders schneller Algorithmus zur Lösung des Minimaler-Schnitt-Problems wurde von Karger [8] entwickelt. Er liefert einen Minimum-Cut in $O(m \log^3 n)$ Zeit mit hoher Wahrscheinlichkeit (d.h. mit einer Wahrscheinlichkeit von $1 - O(1/n^d)$ für eine Konstante d), wobei n die Anzahl der Knoten und m die Anzahl der Kanten des Graphen ist. Dieser Algorithmus ist jedoch recht komplex und schwierig zu implementieren. Um dieses Problem zu lösen, entwickelte Bhardwaj et al. [4] eine vereinfachte Version des Algorithmus mit derselben Laufzeit von $O(m \log^3 n)$. Während der vereinfachte Algorithmus eine Top-Tree-Datenstruktur [1] verwendet, die immer noch relativ komplex ist, stellen die Autoren der Publikation fest, dass die Verwendung einer so ausgefeilten Datenstruktur vermieden werden kann, jedoch auf Kosten eines zusätzlichen Logarithmus-Faktors, was zu einer Laufzeit von $O(m \log^4 n)$ führt.
Die Publikation, die den vereinfachten Algorithmus einführt [4], enthält keine umfassende Erklärung bestimmter Aspekte. Insbesondere lässt es Teile des Algorithmus mit Verweisen auf andere Arbeiten aus. Das Ziel dieser Arbeit ist es, dieses Problem zu lösen, indem der Algorithmus detaillierter erklärt wird, insbesondere durch die Einführung und Erläuterung aller Algorithmen und Datenstrukturen, die in der orginalen Publikation weggelassen wurden. Daher sollte es möglich sein, den Algorithmus allein auf Grundlage dieser Arbeit zu verstehen und zu implementieren, ohne auf andere Arbeiten zurückgreifen zu müssen.

Abstract

We consider an undirected, weighted graph with non-negative edge weights. The minimum cut problem is to find a partition of the set of vertices V into two subsets X and $V \setminus X$, so that the sum of the weights of all edges with one endpoint in X and one endpoint in $V \setminus X$ is minimal.
As stated by Bhardwaj et al. [4], the minimum cut problem is a fundamental problem in graph theory and has several applications in topics such as network reliability [7], cluster analysis [5], and for solving the traveling salesman problem [2].
A particularly fast algorithm that solves the minimum cut problem was developed by Karger [8]. It returns a minimum cut in $O(m \log^3 n)$ time with high probability (meaning with probability $1 - O(1/n^d)$ for some constant d) where n is the number of vertices and m is the number of edges of the graph. This algorithm, however, is quite complex and difficult to implement. To solve this problem, Bhardwaj et al. [4] developed a simpler version of the algorithm with the same runtime of $O(m \log^3 n)$. While the simplified algorithm employs a top tree data structure [1] which is still relatively complex, the authors of the paper note that the usage of such a sophisticated data structure can be avoided at the cost of an additional log-factor resulting in a runtime of $O(m \log^4 n)$.
The paper introducing the simplified algorithm [4] lacks a comprehensive explanation of certain aspects. In particular, it omits parts of the algorithm with references to other papers. The goal of this thesis is to solve this issue by explaining the algorithm in more detail, specifically by introducing and explaining all algorithms and data structures that were omitted in the original paper. Thus, it should be possible to understand and implement the algorithm based on this thesis alone, without having to resort to other work.

Contents

1 Introduction

We consider an undirected, weighted graph with non-negative edge weights. The minimum cut problem is to find a partition of the set of vertices V into two subsets X and $V \setminus X$, so that the sum of the weights of all edges with one endpoint in X and one endpoint in $V \setminus X$ is minimal.
As stated by Bhardwaj et al. [4], the minimum cut problem is a fundamental problem in graph theory and has several applications in topics such as network reliability [7], cluster analysis [5], and for solving the traveling salesman problem [2].
A particularly fast algorithm that solves the minimum cut problem was developed by Karger [8]. It returns a minimum cut in $O(m \log^3 n)$ time with high probability (meaning with probability $1 - O(1/n^d)$ for some constant d) where n is the number of vertices and m is the number of edges of the graph. This algorithm, however, is quite complex and difficult to implement. To solve this problem, Bhardwaj et al. [4] developed a simpler version of the algorithm with the same runtime of $O(m \log^3 n)$. While the simplified algorithm employs a top tree data structure [1] which is still relatively complex, the authors of the paper note that the usage of such a sophisticated data structure can be avoided at the cost of an additional log-factor resulting in a runtime of $O(m \log^4 n)$.
The paper introducing the simplified algorithm [4] lacks a comprehensive explanation of certain aspects. In particular, it omits parts of the algorithm with references to other papers. The goal of this thesis is to solve this issue by explaining the algorithm in more detail, specifically by introducing and explaining all algorithms and data structures that were omitted in the original paper. Thus, it should be possible to understand and implement the algorithm based on this thesis alone, without having to resort to other work.

In Chapter 2, we first introduce some basic definitions that we will need throughout this thesis. Chapter 3 explains the general approach of the algorithm. It reveals, among other things, that the algorithm has two stages. In Chapter 4, we present and explain the first stage. In Chapter 5, we cover the second stage.

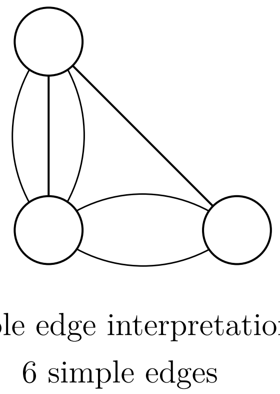
2 Preliminaries

Definition 2.1 (Weighted graph)**.** *We define a (positively) weighted graph as a triple $G = (V, E, w)$ of*
a set of vertices $V \neq \emptyset$,
a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ and
a weight function $w : E \to \mathbb{R}_{>0}$.
We write $V_G := V$, $E_G := E$, $w_G := w$.

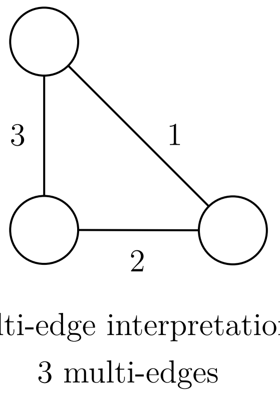
Definition 2.2 (Multigraph)**.** *We define an (unweighted) multigraph as a triple $G = (V, E, m)$ of*
a set of vertices $V \neq \emptyset$,
a set of edges $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ and
an edge-multiplicity function $m : E \to \mathbb{N}$.
We refer to the edges $e = \{u, v\} \in E$ as multi-edges and say that G has $m(e)$ many simple edges (or parallel edges) that connect u and v.
Furthermore, we say that G has $|E|$ many multi-edges and $\sum_{e \in E} m(e)$ many simple edges.
We write $V_G := V$, $E_G := E$, $m_G := m$.

We can interpret a multigraph as a graph that can have more than one edge between two vertices or as a weighted graph with integer-weights. In the context of this thesis, both interpretations are useful. To differentiate the two, we call the edges "simple edges" in the first case and "multi-edges" in the second case.

Figure 2.1: Multigraph interpretations



Simple edge interpretation
6 simple edges



Multi-edge interpretation
3 multi-edges

For sets of vertices $X, Y \subset V_G$ of a graph G, we define

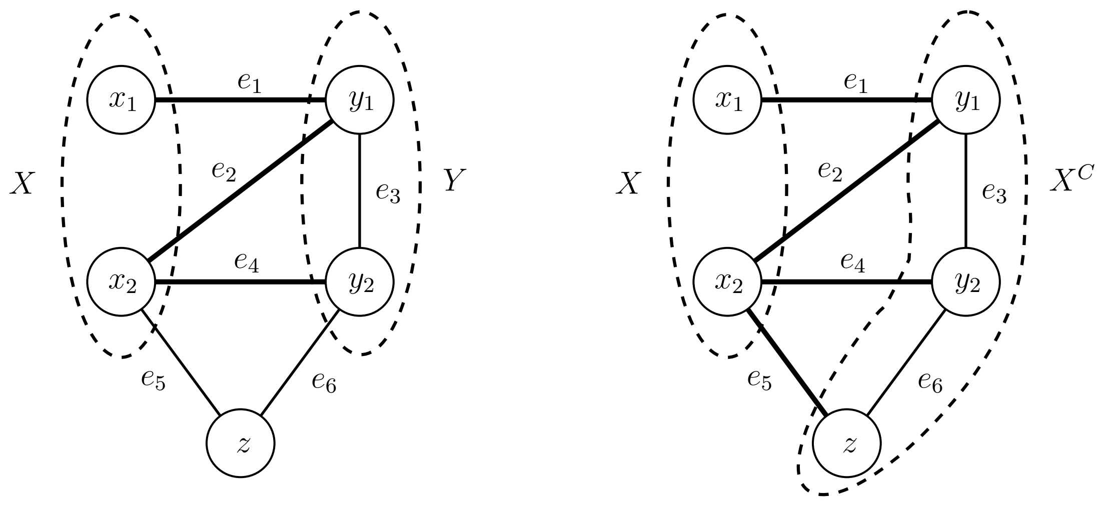
$$E_G(X, Y) := \{\{x, y\} \in E_G \mid x \in X, y \in Y\}$$

as the set of edges with one endpoint in X and one endpoint in Y.
We further define $E_G(X) := E_G(X, X^C)$ for $X^C := V_G \setminus X$ as the set of edges with one endpoint in X and one endpoint outside of X.
For a weighted graph with weight function w_G, we define

$$w_G(X,Y) = \sum_{e \in E_G(X,Y)} w_G(e) \quad \text{and} \quad w_G(X) = \sum_{e \in E_G(X)} w_G(e)$$

as the sum of the weights of edges between X and Y, and between X and $V_G \setminus X$ respectively.
For multigraphs, we define $m_G(X,Y)$ and $m_G(X)$ in a manner analogous to $w_G(X,Y)$ and $w_G(X)$. In this case, $m_G(X,Y)$ is the number simple edges with one endpoint in X and one in Y.

Figure 2.2: Edges between sets of vertices



$X = \{x_1, x_2\}, \quad Y = \{y_1, y_2\}$
$E(X) = \{e_1, e_2, e_4\}$
$w(X) = w(e_1) + w(e_2) + w(e_4)$

$X = \{x_1, x_2\}, \quad X^C = \{y_1, y_2, z\}$
$E(X) = \{e_1, e_2, e_4, e_5\}$
$w(X) = w(e_1) + w(e_2) + w(e_4) + w(e_5)$

Definition 2.3. *Let G be a weighted graph and $X \subset V_G$ be a subset of vertices of G with $X \neq \emptyset$ and $X^C \neq \emptyset$.*
We call $\mathcal{C} = \{X, X^C\}$ a cut of G. We define $E_G(\mathcal{C}) := E_G(X, X^C) = E_G(X)$ and say that an edge e of G crosses the cut if $e \in E_G(\mathcal{C})$.
The weight of the cut $\mathcal{C}$ is defined as $w_G(\mathcal{C}) := w_G(X, X^C) = w_G(X)$.

Definition 2.4. *Let G be a weighted graph. A cut $\mathcal{C}$ of G called a minimum cut of G if its weight $w_G(\mathcal{C})$ is minimal among all cuts of G i.e. if for all other cuts $\mathcal{C}'$ of G:*

$$w_G(\mathcal{C}) \leq w_G(\mathcal{C}')$$

We define $\mu_G := w_G(\mathcal{C})$ as the weight of a minimum cut of G.

This definition can be extended to multigraphs by interpreting the edge-multiplicities as integer-weights. The weight of the cut then amounts to the number of simple edges that cross it.

Definition 2.5. *Let $P = \langle e_1, \ldots, e_k \rangle$ be a sequence of edges and $v_1, \ldots, v_{k+1}$ be vertices so that $e_i = \{v_i, v_{i+1}\}$.*
We call P a walk from v_1 to v_{k+1} of length k.
We call P a trail if all edges $e_1, \ldots, e_k$ are distinct.
We call P a path if all vertices $v_1, \ldots, v_{k+1}$ (and thus all edges) are distinct.
We call a trail P a cycle if only v_1 and v_{k+1} are equal.
Instead of $P = \langle e_1, \ldots, e_k \rangle$, we also write $P = v_1 \ldots v_{k+1}$.

Definition 2.6. *A tree is a connected graph that contains no cycle.*
A spanning tree of a graph G is a tree T with the same vertices as G and with T's edges forming a subset of G's edges.
In this situation, if the edges of G are weighted by some function $w(\cdot)$, we define $\sum_{e \in E_T} w(e)$ as the weight of the tree T, where E_T are the edges of T. We call T a minimum (maximum) spanning tree of G with respect to $w(\cdot)$ if T has the smallest (the largest) weight among all spanning trees of G.

3 Minimum-Cut Algorithm idea

In this chapter, we explain the main idea behind the minimum cut algorithm from Karger [8] and give a high level overview of the two subsequent chapters. We start with the following insight.

Lemma 3.1. *Let G be a weighted graph, $\mathcal{C} = \{X, X^C\}$ be a cut of G and T be a spanning tree of G. Let v, w be vertices of G and $P_{v,w} = vu_1 \dots u_k w$ be the unique path from the vertex v to the vertex w in the tree T.*
Then v and w are on opposite sides of the cut if and only if the number of crossing edges in the path $P_{v,w}$ is odd.
In particular, an edge $e = \{v, w\}$ crosses $\mathcal{C} = \{X, X^C\}$ if and only of the number of crossing edges in the path $P_{v,w}$ is odd.

Proof. We prove the statement by induction over k.

Base case $k = 0$
In this case, $P_{v,w} = vw$, meaning $\{v, w\}$ is the only edge of the path $P_{v,w}$. Because of this

$$\begin{aligned} &P_{v,w} \text{ has an odd number of crossing edges} \\ \Longleftrightarrow &\{v, w\} \text{ crosses the cut} \\ \Longleftrightarrow &v, w \text{ are on opposite sides of the cut} \end{aligned}$$

Induction hypothesis (IH) Let the lemma be true for some $k \in \mathbb{N}_0$
Induction step
Let $P_{v,w} = vu_1 \dots u_{k+1} w$ and $e = \{u_{k+1}, w\}$. Let w.l.o.g. $v \in X$. Then

$$\begin{aligned} &vu_1 \dots u_{k+1} w \text{ has an odd number of crossing edges} \\ \Longleftrightarrow &(e \text{ crossing} \Longleftrightarrow vu_1 \dots u_{k+1} \text{ has an even number of crossing edges}) \\ \overset{(IH)}{\Longleftrightarrow} &(e \text{ crossing} \Longleftrightarrow u_{k+1} \in X) \\ \Longleftrightarrow &w \in X^C \end{aligned}$$

□

A consequence of this lemma is that every cut $\mathcal{C}$ of a graph G with spanning tree T is completely determined by its crossing edges in T. This motivates the following definition / lemma:

Definition / Lemma 3.2. *Let G be a weighted graph and T be a spanning tree of G. Then for every set of pairwise distinct edges $e_1, \dots, e_k$ of T, there is exactly one cut $\mathcal{C}$ of G, such that $e_1, \dots, e_k$ are the crossing edges of $\mathcal{C}$ in T.*
We define $\mathcal{C}_T(e_1, \dots, e_k) := \mathcal{C}$.

Proof.
Existence
Let v be an arbitrary vertex and let $e_1, \dots, e_k$ be pairwise distinct edges of T. Then $\mathcal{C}_T(e_1, \dots, e_k) = \{Y, Y^C\}$ for

$$\begin{aligned} Y &= \{w \in V_G \mid P_{v,w} \text{ contains an even number of the edges } e_1, \dots, e_k\} \text{ and} \\ Y^C &= \{w \in V_G \mid P_{v,w} \text{ contains an odd number of the edges } e_1, \dots, e_k\} \end{aligned}$$

Uniqueness
Let $\mathcal{C} = \{X, X^C\}$ and $\mathcal{C}' = \{Y, Y^C\}$ be two cuts that both have exactly the crossing edges $e_1, \dots, e_k$ in T.
Let v be a vertex of G and w.l.o.g. $v \in X$, $v \in Y$. According to Lemma 3.1, we get

$$\begin{aligned} w \in X &\iff \mathcal{P}_{v,w} \text{ has an even number of edges that cross } \mathcal{C} \\ &\iff \mathcal{P}_{v,w} \text{ contains an even number of the edges } e_1, \dots, e_k \\ &\iff \mathcal{P}_{v,w} \text{ has an even number of edges that cross } \mathcal{C}' \\ &\iff w \in Y \end{aligned}$$

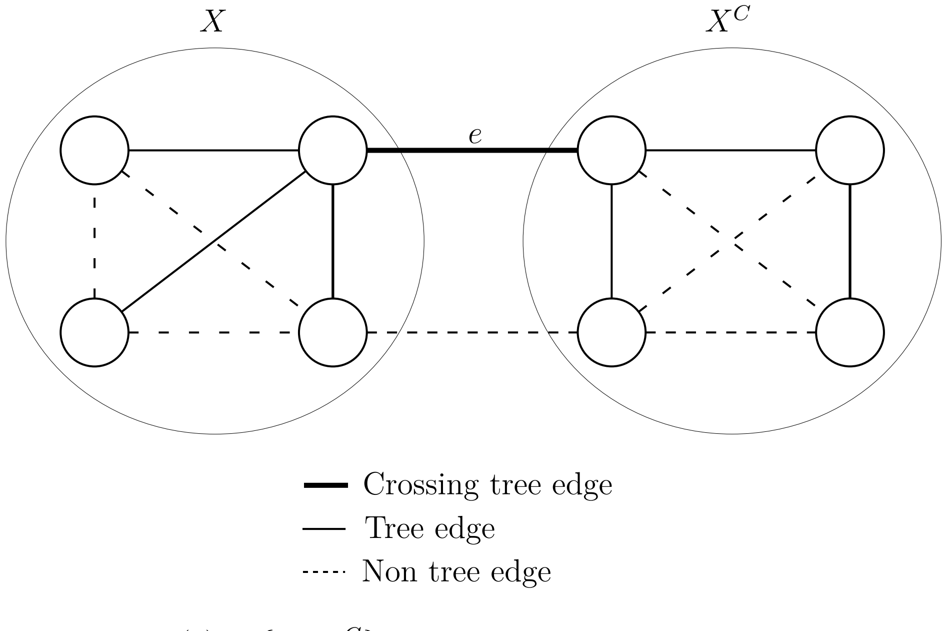
Hence $X = Y$ and thus $\mathcal{C} = \mathcal{C}'$. □

The approach behind the minimum cut algorithm from Karger [8] is motivated by the subsequent definition and lemma:

Definition 3.3 (Karger [8]). *Let T be a spanning tree of a weighted graph G and $\mathcal{C}$ be a cut of G. We say that $\mathcal{C}$ k-respects T if T contains at most k crossing edges of $\mathcal{C}$. We say that $\mathcal{C}$ strictly k-respects T if T contains exactly k crossing edges of $\mathcal{C}$.*

Lemma 3.4 (Karger [8]). *Let G be a weighted graph and $\mathcal{C}$ be a minimum cut of G. Then $\mathcal{C}$ 2-respects at least one spanning tree of G.*

Figure 3.1: Example of a minimum cut that 1-respects a spanning tree T



$C_T(e) = \{X, X^C\}$ 1-respects the tree T

Thus, we can find a minimum cut of G with the following strategy:

1. Find a set of $O(\log n)$ spanning trees $T_1, \ldots, T_N$ of G of which a minimum cut of G 2-respects at least one.

2. For each tree T_i, find the smallest cut of G that 2-respects T_i and return the smallest cut found across all trees.

In chapter 4, we will prove Lemma 3.4 and show how such a set of trees can be found in near-linear-time with high probability.
In chapter 5, we will show how the smallest cut that 2-respects a given spanning tree can be found in near-linear-time.

4 Finding suitable spanning trees

4.1 Definitions and the proof of Lemma 3.4

We start with a definition about multigraphs:

Definition 4.1. *Let G be a multigraph and $\mathcal{P} = \{T_1, \dots T_k\}$ be a multiset of spanning trees of G. We call $\mathcal{P}$ a tree packing of G if and only if every simple edge of G is contained in at most one tree of $\mathcal{P}$.*
We define the weight of the tree-packing $w_{\mathcal{P}} := |\mathcal{P}|$ as the number of trees in it.

Since a multi-edge e of a multigraph G consists of $m_G(e)$ many simple edges, the definition above can be restated as follows: Each multi-edge e is present in at most $m_G(e)$ distinct trees.
Nash-Williams [13] proved the following statement about tree-packings of multigraphs.

Theorem 4.2 (Nash-Williams [13])**.** *Let G be a multigraph. Then there exists a tree-packing $\mathcal{P}$ of G with weight $|\mathcal{P}| = w_{\mathcal{P}} \geq \mu_G/2$.*

This theorem is sufficient to prove a version Lemma 3.4 for multigraphs.

Proposition 4.3 (Karger [8], Bhardwaj et al. [4])**.** *Let G be a multigraph and $\mathcal{C}$ be a minimum cut of G. Then there is a tree packing $\mathcal{P}$ of G so that $\mathcal{C}$ 2-respects at least half of the trees in the packing.*

Proof. According to Theorem 4.2, there exists a tree-packing $\mathcal{P}$ of G with weight $|\mathcal{P}| \geq \mu_G/2$.
Let $N_i \in \mathbb{N}_0$ be the number of trees in $\mathcal{P}$ that contain exactly i simple crossing edges of $\mathcal{C}$. Since every simple edge can only be contained in at most one tree, the average number of simple crossing edges per tree is at most $\mu_G/(\mu_G/2) = 2$. Therefore

$$2\sum_{i=1}^{\infty} N_i = 2|\mathcal{P}| \geq \left(\frac{1}{|\mathcal{P}|}\sum_{i=1}^{\infty} iN_i\right)|\mathcal{P}| = \sum_{i=1}^{\infty} iN_i$$

Hence

$$0 \geq \sum_{i=1}^{\infty} iN_i - 2\sum_{i=1}^{\infty} N_i = \sum_{i=1}^{\infty}(i-2)N_i = -N_1 + \sum_{i=3}^{\infty}(i-2)N_i \geq -N_1 + \sum_{i=3}^{\infty} N_i$$

Thus

$$N_1 + N_2 \geq N_1 \geq \sum_{i=3}^{\infty} N_i$$

□

The goal is to adapt Proposition 4.3 to weighted graphs. For this purpose, we introduce a generalization of Definition 4.1 and Theorem 4.2.

Definition 4.4. *Let G be a weighted graph and $\mathcal{P} = (\mathscr{T}, w_{\mathscr{T}})$ be a pair of a multiset $\mathscr{T} = \{T_1, \dots T_k\}$ of spanning trees of G and a weight function $w_{\mathscr{T}} : \mathscr{T} \to \mathbb{R}$. We call $\mathcal{P}$ a weighted tree-packing of G if*

$$\forall e \in E_G : \sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq w_G(e)$$

We define the weight of the tree-packing $w_{\mathcal{P}} := \sum_{T \in \mathscr{T}} w_{\mathscr{T}}(T)$ as the sum of the weights of the trees.

This generalized definition can also be applied to multigraphs if we interpret the edge-multiplicities as weights. Further, every tree packing $\mathcal{P}$ from Definition 4.1 can be turned into a weighted tree packing if we set the weight of each tree to 1. With this Definition, we can prove a version of Theorem 4.2 for weighted graphs.

Lemma 4.5 (Karger [8], Bhardwaj et al. [4])**.** *Let G be a weighted graph. Then there exists a weighted tree-packing $\mathcal{P}$ of G with weight $w_{\mathcal{P}} \geq \mu_G/2$.*

Proof. To reach a contradiction, assume that there is a weighted graph G with n vertices and m edges, so that all tree packings of G have a weight smaller than $(1-\epsilon)\mu_G/2$ for some $\epsilon > 0$.
Let $\delta = \epsilon\mu_G/m$. Since $\mathbb{Q}$ is dense subset of $\mathbb{R}$, for every edge e of G, there are rational numbers a_e/b_e ($a_e, b_e \in \mathbb{N}$) such that $w_G(e) - \delta < a_e/b_e \leq w_G(e)$.
Let $d = \prod_{e \in E_G} b_e$ and let G' be a multigraph with the same vertices and (multi-)edges as G but with edge-multiplicities

$$m_{G'}(e) = a_e \prod_{e' \in E_G, e' \neq e}^{m} b_{e'} = \frac{a_e}{b_e} d$$

for every multi-edge $e \in E_{G'}$. Let $\mathcal{C}'$ be the minimum cut of G'. Since G and G' have the same vertices, $\mathcal{C}'$ can also be interpreted as a cut in G with weight $\sum_{e \in E_G(\mathcal{C}')} w_G(e) = w_G(\mathcal{C}') \geq \mu_G$. Hence

$$\begin{aligned}
\mu_{G'} = \sum_{e \in E_G(\mathcal{C}')} m_{G'}(e) \quad & = \sum_{e \in E_G(\mathcal{C}')} \frac{a_e}{b_e} d \\
\geq \sum_{e \in E_G(\mathcal{C}')} (w_G(e) - \delta) d \quad & = \left(\sum_{e \in E_G(\mathcal{C}')} w_G(e) - \sum_{e \in E_G(\mathcal{C}')} \delta \right) d \\
\geq (\mu_G - |E_G(\mathcal{C}')|\delta)\, d \quad & = \left(\mu_G - |E_G(\mathcal{C}')| \frac{\epsilon\mu_G}{m} \right) d \\
\geq (\mu_G - \epsilon\mu_G)\, d \quad & = \mu_G(1-\epsilon) d
\end{aligned}$$

Furthermore, according to Theorem 4.2, G' has a tree packing $\mathcal{P}' = \{T_1, \dots, T_k\}$ with weight $w_{\mathcal{P}'} \geq \mu_{G'}/2$.
Let $\mathcal{P} = (\mathscr{T}, w_{\mathscr{T}})$ with $\mathscr{T} = \{T_1, \dots, T_k\} = \mathcal{P}'$ and $w_{\mathscr{T}} : \mathscr{T} \to \mathbb{R}_{>0}$ with $w_{\mathscr{T}}(T) = 1/d$ for all $T \in \mathscr{T}$. $\mathcal{P}$ is a tree packing of G, since for all edges e of G:

$$\begin{aligned}
\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) = \frac{1}{d} \sum_{T \in \mathscr{T} \text{ with } e \in E_T} 1 \quad & = \frac{1}{d} |\{T \in \mathscr{T} \mid e \in E_T\}| \\
\leq \frac{1}{d} m_{G'}(e) \quad & = \frac{1}{d} \frac{a_e}{b_e} d \\
\leq w_G(e) &
\end{aligned}$$

$\mathcal{P}$ has weight

$$w_{\mathcal{P}} = \sum_{T \in \mathscr{T}} w_{\mathscr{T}}(T) = |\underbrace{\mathscr{T}}_{=\mathcal{P}'}|\frac{1}{d} = w_{\mathcal{P}'}\frac{1}{d} \geq \frac{\mu_{G'}}{2d} \geq \frac{\mu_G(1-\epsilon)d}{2d} = \frac{\mu_G(1-\epsilon)}{2}$$

which contradicts our assumption! □

In order to use Lemma 4.5 to prove Lemma 3.4, we need the following auxiliary lemma:

Lemma 4.6 (Karger [8], Bhardwaj et al. [4])**.** *Let G be a weighted graph. Let furthermore $\mathcal{C}$ be a cut of G with weight $w_G(\mathcal{C}) = \alpha\mu_G$ and let $\mathcal{P} = (\mathscr{T}, w_{\mathscr{T}})$ be a weighted tree packing of G with weight $w_{\mathcal{P}} = \beta\mu_G$.*
Let $X := \{T \in \mathscr{T} \mid \mathcal{C}$ 2-respects $T\}$ and $Y := \{T \in \mathscr{T} \mid \mathcal{C}$ does not 2-respect $T\}$. Then

$$\frac{1}{w_{\mathcal{P}}} \sum_{T \in X} w_{\mathscr{T}}(T) \geq \frac{1}{2}\left(3 - \frac{\alpha}{\beta}\right)$$

meaning at least a fraction of $\frac{1}{2}(3 - \alpha/\beta)$ of the trees by weight 2-respect the cut.

Proof. Let

$$x := \sum_{T \in X} w_{\mathscr{T}}(T),\ y := \sum_{T \in Y} w_{\mathscr{T}}(T)$$

Since $X \cup Y = \mathscr{T}$ we get

$$x + y = \sum_{T \in \mathscr{T}} w_{\mathscr{T}}(T) = w_{\mathcal{P}}$$

We define

$$\mathbb{1}_{E_T}(e) := \begin{cases} 1 & \text{if } e \in E_T \\ 0 & \text{if } e \notin E_T \end{cases}$$

Since

$$\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq w_G(e)$$

for all edges $e \in E_G(\mathcal{C})$, we also get

$$\begin{aligned} w_G(\mathcal{C}) &= \sum_{e \in E_G(\mathcal{C})} w_G(e) \\ &\geq \sum_{e \in E_G(\mathcal{C})} \sum_{T \in \mathscr{T}} \mathbb{1}_{E_T}(e) w_{\mathscr{T}}(T) \\ &= \sum_{T \in \mathscr{T}} \left(\sum_{e \in E_G(\mathcal{C})} \mathbb{1}_{E_T}(e) \right) w_{\mathscr{T}}(T) \\ &= \sum_{T \in X} \underbrace{\left(\sum_{e \in E_G(\mathcal{C})} \mathbb{1}_{E_T}(e) \right)}_{\geq 1} w_{\mathscr{T}}(T) + \sum_{T \in Y} \underbrace{\left(\sum_{e \in E_G(\mathcal{C})} \mathbb{1}_{E_T}(e) \right)}_{\geq 3} w_{\mathscr{T}}(T) \\ &\geq x + 3y \end{aligned}$$

Hence

$$\begin{aligned}
\frac{1}{w_\mathcal{P}}\sum_{T\in X} w_\mathscr{T}(T) &= \frac{1}{w_\mathcal{P}}x &&= \frac{1}{w_\mathcal{P}}\frac{1}{2}(3(x+y)-(x+3y)) \\
&\geq \frac{1}{w_\mathcal{P}}\frac{1}{2}(3w_\mathcal{P} - w_G(\mathcal{C})) &&= \frac{1}{2}\left(3 - \frac{w_G(\mathcal{C})}{w_\mathcal{P}}\right) \\
&= \frac{1}{2}\left(3 - \frac{\alpha\mu_G}{\beta\mu_G}\right)
\end{aligned}$$

□

Thus, we get ...

Lemma 3.4 (Karger [8]). *Let G be a weighted graph and $\mathcal{C}$ be a minimum cut of G. Then $\mathcal{C}$ 2-respects at least one spanning tree of G.*

Proof. According to Lemma 4.5, there is a weighted tree packing $\mathcal{P} = (\mathscr{T}, w_\mathscr{T})$ of G with weight $w_\mathcal{P} \geq \mu_G/2 = \beta\mu_G$ for $\beta = 1/2$. Moreover, the cut $\mathcal{C}$ has weight $\mu_G = \alpha\mu_G$ for $\alpha = 1$. Let $X := \{T \in \mathscr{T} \mid \mathcal{C}$ 2-respects $T\}$. Lemma 4.6 implies, that

$$\frac{1}{w_\mathcal{P}}\sum_{T\in X} w_\mathscr{T}(T) \geq \frac{1}{2}\left(3 - \frac{\alpha}{\beta}\right) = \frac{1}{2}\left(3 - \frac{1}{1/2}\right) = \frac{1}{2}$$

Hence $X \neq \emptyset$, meaning there exists $T \in \mathscr{T}$ so that $\mathcal{C}$ 2-respects T. □

Lemma 4.6 also guarantees that cuts with weight sightly bigger than μ_G respects a fraction of the trees of the packing if $\alpha < 3\beta$ since

$$\frac{1}{2}\left(3 - \frac{\alpha}{\beta}\right) > 0 \iff 3\beta > \alpha$$

At first glance, the consideration of cuts larger than the minimum might seem unnecessary, since we are primarily interested in the graph's minimum cuts. However, it turns out that we can simplify and speed up the process of finding a tree packing by making slight modifications to the weights of the graph's edges.
These weight modifications influence the cut-weights, potentially resulting in some minimum cuts loosing their minimum cut status. Yet, with careful adjustments, it is possible to ensure that the weight of the original graph's minimum cut remains very close to the minimum cut weight of the modified graph. Because of this, the property of 2-respecting a fraction of the trees, as specified in Lemma 4.6, is retained.

4.2 Finding a tree packing

To find a weighted tree packing, we use the proposed strategy from Karger [8] which has been concretized by Bhardwaj et al. [4]: Instead of finding a sufficiently large tree packing for the original weighted graph directly, we first approximate the weighted graph G as a multigraph G' and use an algorithm for finding large weighted tree packings of multigraphs on G'. We do this by first normalizing the weights, so that the smallest weight has value 1. Then, we multiply the weights by some constant ϵ^{-1} with $\epsilon > 0$ and round the weights up to the next largest integer. The resulting graph is our multigraph G' with the integer-weights being interpreted as its edge multiplicities.
The weights of Gs minimum cuts have the following upper bound in G':

Lemma 4.7. *Let G be a weighted graph with minimum weight $w_{min} = \min_{e \in E} w_G(e)$ and minimum cut $\mathcal{C}$. Let $0 < \epsilon$ and $x := \epsilon^{-1} w_{min}^{-1}$.*
Let G' be the approximation of G as a multigraph with $m_{G'}(e) = \lceil x w_G(e) \rceil$. Furthermore, let $\mathcal{C}'$ be the minimum cut of G' and $w_{G'}(\mathcal{C})$ be the (integer-)weight of the cut $\mathcal{C}$ in G'.
Then $w_{G'}(\mathcal{C}) \leq (1+\epsilon)\mu_{G'}$.

Proof. We get

$$m_{G'}(e) = \lceil x w_G(e) \rceil = \left\lceil \frac{1}{\epsilon} \frac{1}{w_{\min}} w_G(e) \right\rceil \leq \frac{1}{\epsilon} \frac{w_G(e)}{w_{\min}} + 1 \leq \frac{1}{\epsilon} \frac{w_G(e)}{w_{\min}} + \frac{w_G(e)}{w_{\min}}$$
$$= \left(\frac{1}{\epsilon} + 1\right) \frac{w_G(e)}{w_{\min}} = (1+\epsilon) \frac{1}{\epsilon} \frac{w_G(e)}{w_{\min}} = (1+\epsilon) x w_G(e)$$

Since $\mathcal{C}'$ is a (not necessarily minimum) cut in G, we get $w_G(\mathcal{C}') = \sum_{e \in E_G(\mathcal{C}')} w_G(e) \geq \mu_G$. Hence

$$\mu_{G'} = \sum_{e \in E_{G'}(\mathcal{C}')} m_{G'}(e) = \sum_{e \in E_G(\mathcal{C}')} \lceil x w_G(e) \rceil \geq x \sum_{e \in E_G(\mathcal{C}')} w_G(e) \geq x \mu_G$$

Therefore

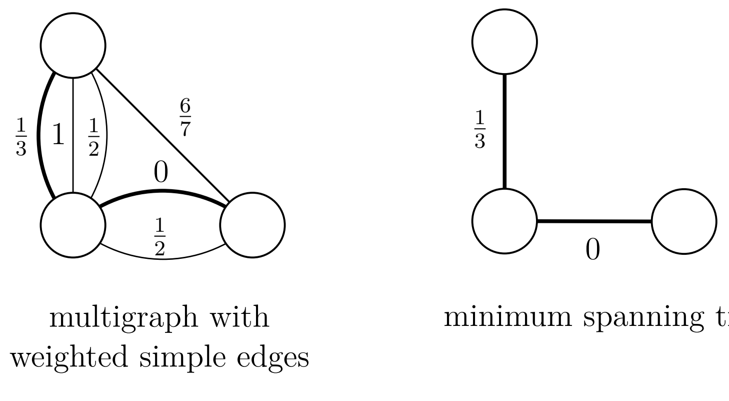
$$w_{G'}(\mathcal{C}) = \sum_{e \in E_{G'}(\mathcal{C})} m_{G'}(e) \leq (1+\epsilon) x \sum_{e \in E_G(\mathcal{C})} w_G(e) = (1+\epsilon) x \mu_G \leq (1+\epsilon)\mu_{G'}$$

□

Note that the number of edges of the weighted graph G is the same as the number of multi-edges in the multigraph G', since we only change the weights (edge-multiplicities) of the edges. The number of simple edges, however, can become arbitrarily large depending on the weights of G.

To find a sufficiently large tree packing for a multigraph, we use the algorithm that is proposed by Bhardwaj et al. [4]. The algorithm was first explicitly stated by Thorup and Karger [16] with references to Plotkin et al. [14] and Young [17].
In the context of this algorithm, we also assign real-valued weights to the simple edges of the multigraph. These weights are independent from the edge-multiplicity (integer weights) of the multi-edges. When we talk about the minimum spanning tree in this context, we are referring to minimality in terms of the weights of the simple edges.

Figure 4.1: Example of a multigraph with weighted simple edges



Algorithm 1 Find weighted tree packing of a multigraph

Let G' be a multigraph with $\widetilde{m}$ simple edges.

```
 1: Initialize $l(\widetilde{e}) \leftarrow 0$ for all simple edges $\widetilde{e}$
 2: Initialize $W \leftarrow 0$ and initialize multiset $P \leftarrow \emptyset$
 3: while True do
 4:     Find a minimum spanning tree $T$ of $G'$ with respect to $l(\cdot)$.
 5:     for all simple edges $\widetilde{e}$ in $T$ do
 6:         Update $l(\widetilde{e}) \leftarrow l(\widetilde{e}) + \epsilon^2/(3\ln\widetilde{m})$
 7:         if $l(\widetilde{e}) > 1$ then
 8:             return $P, W$
 9:         end if
10:     end for
11:     Add $T$ to $P$
12:     Update $W \leftarrow W + \epsilon^2/(3\ln\widetilde{m})$
13: end while
```

Lemma 4.8 ([4], [16], [14], [17]). *Let $0 < \epsilon < 1$ and G' be a multigraph. Then Algorithm 1 returns a weighted tree packing of G' of weight at least $(1-\epsilon)\mu_{G'}/2$.*

Proof. Let τ be the weight of the maximally weighted tree packing of G' and W be the weight of the returned packing from Algorithm 1. Thorup and Karger [16] state with references to Plotkin et al. [14] and Young [17] that $(1-\epsilon)\tau \leq W \leq \tau$. Lemma 4.5 states that G' has a weighted packing with weight at last $\mu_{G'}/2$.
Thus $\tau \geq \mu_{G'}/2$ and hence $W \geq (1-\epsilon)\tau \geq (1-\epsilon)\mu_{G'}/2$. □

To find the minimum spanning tree, we use Kruskal's minimum spanning tree algorithm. This algorithm is presented and explained in Appendix A. Since all simple edges are assigned a load, Kruskal's algorithm needs to check all $\widetilde{m}$ simple edges, which results in a runtime of $O(\widetilde{m}\log\widetilde{m})$ (Lemma A.4). We can reduce this runtime to $O(m\log n)$ by working with multi-edges instead of simple edges. This can be accomplished by using an implementation trick from Gawrychowski et al. [6]:
Note that among all simple edges $\widetilde{e}_1, \ldots, \widetilde{e}_k$ of a multi-edge e, the only simple edge of significance for the minimum spanning tree is the edge $\widetilde{e}_i$ with the minimum load $l(\widetilde{e}_i) = \min_{j=1,\ldots,k} l(\widetilde{e}_j)$.
Firstly, the spanning tree can contain at most one of the simple edges. If it would contain two simple edges $\widetilde{e}_i, \widetilde{e}_j$, then $\langle\widetilde{e}_i, \widetilde{e}_j\rangle$ would form a cycle.
Secondly, if the tree would contain an edge $\widetilde{e}_j$ with a bigger load $l(\widetilde{e}_j) > l(\widetilde{e}_i)$, it would not be a minimum spanning tree with respect to $l(\cdot)$, since we could obtain a smaller tree by replacing $\widetilde{e}_j$ with $\widetilde{e}_i$.
Finding the edge with the minimal load can be done in $O(1)$ time as follows:
Since the load of a simple edge always gets increased by a fixed amount for every spanning tree that contains it, the edge with the minimum load it the edge that is contained in the fewest spanning trees. Thus, we only need to cycle through the simple edges of e where we move to the nearest simple edge whenever the current simple edge is part of a new spanning tree.
With this idea in mind, we can write a more efficient version of Algorithm 1:

Algorithm 2 Find weighted tree packing of a multigraph using multi-edges

Let G' be a multigraph with m multi-edges and $\widetilde{m}$ simple edges.

```
 1: Initialize $l(e) \leftarrow 0$ and $k(e) \leftarrow 0$ for all multi-edges $e$
              $\triangleright$ $k(e)$ represents the index of the current simple edge and $l(e)$ its load
 2: Initialize $W \leftarrow 0$ and initialize multiset $P \leftarrow \emptyset$
 3: while True do
 4:     Find a minimum spanning tree $T$ of $G'$ with respect to $l(\cdot)$.
 5:     for all multi-edges $e$ in $T$ do
 6:         if $l(e) + \epsilon^2/(3\ln\widetilde{m}) > 1$ then
 7:             return $P, W$
 8:         end if
 9:         Update $k(e) \leftarrow k(e) + 1$
10:         if $k(e) \geq m_{G'}(e)$ then
11:             Set $k(e) \leftarrow 0$ and $l(e) \leftarrow l(e) + \epsilon^2/(3\ln\widetilde{m})$
12:         end if
13:     end for
14:     Add $T$ to $P$
15:     Update $W \leftarrow W + \epsilon^2/(3\ln\widetilde{m})$
16: end while
```

Lemma 4.9 (Bhardwaj et al. [4], Gawrychowski et al. [6])**.** *Let* $0 < \epsilon < 1$, $\epsilon = \Theta(1)$ *and* G' *be a multigraph with* n *vertices,* m *multi-edges,* $\widetilde{m}$ *simple edges, and a minimum cut of weight* $\mu_{G'}$.
Then Algorithm 2 runs for at most $\mu_{G'} 3\ln\widetilde{m}/\epsilon^2 = O(\mu_{G'}\log\widetilde{m})$ *iterations that have a runtime of at most* $O(m\log n)$ *each. Therefore, the total runtime of Algorithm 2 is* $O(m\mu_{G'}(\log n)(\log\widetilde{m}))$.

To prove Lemma 4.9, more specifically the runtime of Algorithm 2, we need an upper bound for the number of iterations. To achieve this, we use the following statement about the size of a tree packing.

Lemma 4.10. *Let* G *be a weighted graph. Then all tree packings of* G *have weight at most* μ_G.

Proof. Let $\mathcal{P}$ be a tree packing of G with weight $w_{\mathcal{P}} = \beta\mu_G$ and $\mathcal{C}$ be a minimum cut of G with weight $\mu_G = \alpha\mu_G$ for $\alpha = 1$.
Let furthermore $X := \{T \in \mathscr{T} \mid \mathcal{C} \text{ 2-respects } T\} \subseteq \mathscr{T}$. Using Lemma 4.6, we get

$$1 = \frac{w_{\mathcal{P}}}{w_{\mathcal{P}}} \geq \frac{1}{w_{\mathcal{P}}}\sum_{T\in X} w_{\mathscr{T}}(T) \geq \frac{1}{2}\left(3 - \frac{\alpha}{\beta}\right) = \frac{3}{2} - \frac{1}{2}\frac{1}{w_{\mathcal{P}}/\mu_G} = \frac{3}{2} - \frac{1}{2}\frac{\mu_G}{w_{\mathcal{P}}}$$

Also

$$1 \geq \frac{3}{2} - \frac{1}{2}\frac{\mu_G}{w_{\mathcal{P}}} \iff -\frac{1}{2} \geq -\frac{1}{2}\frac{\mu_G}{w_{\mathcal{P}}} \iff 1 \leq \frac{\mu_G}{w_{\mathcal{P}}} \iff w_{\mathcal{P}} \leq \mu_G$$

□

We can now proceed with the proof of Lemma 4.9.

Proof of Lemma 4.9.
Correctness

We argue that Algorithm 2 is functionally identical to Algorithm 1. The correctness then follows from Lemma 4.8.
For a multi-edge e, $k(e)$ represents the index of the current and minimally loaded simple edge and $l(e)$ its load. Whenever an edge is part of a spanning tree, we cycle to the next simple edge by incrementing $k(e)$. If $k(e) \geq m_{G'}(e)$, we have cycled through all simple edges and go back to the first simple edge ($k(e) \leftarrow 0$). After that, the loads of all simple edges have been increased by $\epsilon^2/(3\ln\widetilde{m})$, meaning the minimal load $l(e)$ increases by $\epsilon^2/(3\ln\widetilde{m})$.

Time complexity
Let N be the number of iterations. In each iteration, the total weight of the packing increases by $\epsilon^2/(3\ln\widetilde{m})$. Thus, the total weight of the packing is $W = N\epsilon^2/(3\ln\widetilde{m})$. According to Lemma 4.10, the total weight of any weighted tree packing can be at most $\mu_{G'}$. Hence

$$N\frac{\epsilon^2}{3\ln\widetilde{m}} = W \leq \mu_{G'} \Longrightarrow N \leq \mu_{G'}\frac{3\ln\widetilde{m}}{\epsilon^2} = O(\mu_{G'}\log\widetilde{m})$$

The most time consuming part of each iteration is finding the minimum spanning tree. Since Kruskal's algorithm only needs to check the m multi-edges, it has a runtime of $O(m\log m) = O(m\log n)$ (Lemma A.4). □

4.3 Achieving near-linear-time

Since Algorithm 2 takes $O(m\mu_{G'}(\log n)(\log\widetilde{m}))$ time (Lemma 4.9), we would need to require $\mu_{G'}\log\widetilde{m} = O(\log^3 n)$ for the algorithm to run in $O(m\log^4 n)$ time on G'. Both requirements are not satisfied, since both $\mu_{G'}$ and $\widetilde{m}$ can become arbitrarily large for fix n and m.

To fix this problem for $\mu_{G'}$, Bhardwaj et al. [4] propose to use a random sampling technique from Karger [9].

Lemma 4.11 (Karger [9]). *Let G' be a multigraph. Let $0 < d, \epsilon = \Theta(1)$ and let $p = 2(d+2)(\ln n)/(\epsilon^2\gamma\mu_{G'}) \leq 1$ for some $\gamma \leq 1$. Sample each simple edge of G' independently at random with probability p and let H be the resulting multigraph. Then H has the following properties with probability $1 - O(1/n^d)$:*

(a) Every cut $\mathcal{C}$ of G' with weight $w_{G'}(\mathcal{C})$ in G' has weight

$$(1-\epsilon)pw_{G'}(\mathcal{C}) \leq w_H(\mathcal{C}) \leq (1+\epsilon)pw_{G'}(\mathcal{C})$$

in H.

(b) For the weight μ_H of the minimum cut of H, we get

$$(1-\epsilon)p\mu_{G'} \leq \mu_H \leq (1+\epsilon)p\mu_{G'}$$

and in particular

$$\mu_H \leq \frac{2(1+\epsilon)(d+2)}{\epsilon^2\gamma}(\ln n)$$

If $\gamma = \Theta(1)$, then $\mu_H = O(\log n)$.

(c) *If $\mathcal{C}$ is a cut with weight $w_{G'}(\mathcal{C}) \leq \alpha\mu_{G'}$ in G' and with weight $w_H(\mathcal{C})$ in H, then $w_H(\mathcal{C}) \leq \alpha(1+\epsilon)(1-\epsilon)^{-1}\mu_H$*

Proof.

(a) Let $\epsilon' = \epsilon\sqrt{\gamma} \leq \epsilon$. Then $2(d+2)(\ln n)/(\epsilon'^2\mu_{G'}) = 2(d+2)(\ln n)/(\epsilon^2\gamma\mu_{G'}) = p$. Corollary 2.4 from Karger [9] implies that every cut $\mathcal{C}$ of G' has weight

$$\underbrace{(1-\epsilon')}_{\geq 1-\epsilon} p w_{G'}(\mathcal{C}) \leq w_H(\mathcal{C}) \leq \underbrace{(1+\epsilon')}_{\leq 1+\epsilon} p w_{G'}(\mathcal{C})$$

in H.

(b) Let $\mathcal{C}$ be the minimum cut of G'.

Using (a), we get that $\mathcal{C}$ has weight $w_H(\mathcal{C}) \leq (1+\epsilon)p\mu_{G'}$ in H.
Hence $\mu_H \leq w_H(\mathcal{C}) \leq (1+\epsilon)p\mu_{G'}$.

Now, let $\mathcal{C}$ be an arbitrary cut of H with weight $w_{G'}(\mathcal{C})$ in G'. Using (a), we get that $\mathcal{C}$ has weight $w_H(\mathcal{C}) \geq (1-\epsilon)pw_{G'}(\mathcal{C}) \geq (1-\epsilon)p\mu_{G'}$ in H. Therefore, all cuts in H have weight at least $(1-\epsilon)p\mu_{G'}$, which implies $\mu_H \geq (1-\epsilon)p\mu_{G'}$.

Further

$$\mu_H \leq (1+\epsilon)p\mu_{G'} = \frac{2(1+\epsilon)(d+2)(\ln n)}{\epsilon^2\gamma}$$

If $\gamma = \Theta(1)$, then

$$\mu_H \leq \frac{2(1+\epsilon)(d+2)(\ln n)}{\epsilon^2\gamma} = O(\log n)$$

since $d, \epsilon, \gamma = \Theta(1)$.

(c) $w_H(\mathcal{C}) \overset{(a)}{\leq} (1+\epsilon)pw_{G'}(\mathcal{C}) \leq (1+\epsilon)(1-\epsilon)^{-1}(1-\epsilon)p\alpha\mu_{G'} \overset{(b)}{\leq} \alpha(1+\epsilon)(1-\epsilon)^{-1}\mu_H$

□

According to (b) of Lemma 4.11, the graph H has a minimum cut of size at most $O(\log n)$ with high probability for $\gamma = \Theta(1)$, which would ensure a runtime of Algorithm 2 of $O(m(\log^2 n)(\log \widetilde{m}))$ on H where m is the number of multi-edges and $\widetilde{m}$ is the number of simple edges of H.

To find an upper bound for $\widetilde{m}$, we make use the following lemma:

Lemma 4.12. *Let G be a weighted graph and let F be a copy of G with adjusted weights $w_F(e) = \min\{w_G(e), \hat{\mu_G}\}$ for some $\hat{\mu_G} \geq \mu_G$. Then G and F have the same tree packings and $\mu_G = \mu_F$.*

Proof. Since G and F have the same edges, we get $E_G = E_F$.
$\mu_G = \mu_F$ because

- For any cut $\mathcal{C}$, we get

$$w_F(\mathcal{C}) = \sum_{e \in E_F(\mathcal{C})} w_F(e) \leq \sum_{e \in E_G(\mathcal{C})} w_G(e) = w_G(\mathcal{C})$$

Hence $\mu_F \leq \mu_G$.

- Let $\mathcal{C}$ be a minimum cut of F. If $\exists e' \in E_F(\mathcal{C})$ with $w_F(e') = \hat{\mu_G}$, then

$$\mu_F = \sum_{e \in E_F(\mathcal{C})} w_F(e) \geq w_F(e') = \hat{\mu_G} \geq \mu_G$$

  Otherwise $w_F(e) = w_G(e)$ for all $e \in E_{\mathcal{C}'}$ and thus

$$\mu_F = \sum_{e \in E_F(\mathcal{C})} w_F(e) = \sum_{e \in E_G(\mathcal{C})} w_G(e) \geq \mu_G$$

G and F have the same tree packings because

- Let $\mathcal{P} = (\mathscr{T}, w_{\mathscr{T}})$ be a tree packing of G. According to Lemma 4.10, the weight of the packing $w_{\mathcal{P}} = \sum_{T \in \mathscr{T}} w_{\mathscr{T}}(T)$ is at most μ_G.
  Since

$$\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq w_G(e) \text{ for all edges } e \quad \text{and}$$
$$\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq \sum_{T \in \mathscr{T}} w_{\mathscr{T}}(T) \leq \mu_G \leq \hat{\mu_G}$$

  we get

$$\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq \min\{w_G(e), \hat{\mu_G}\} = w_F(e)$$

  for all edges e, meaning $\mathcal{P}$ is also a tree packing of F.

- Let $\mathcal{P} = (\mathscr{T}, w_{\mathscr{T}})$ be a tree packing of F. Then $\mathcal{P}$ is also a tree packing of F since

$$\sum_{T \in \mathscr{T} \text{ with } e \in E_T} w_{\mathscr{T}}(T) \leq w_F(e) \leq w_G(e)$$

  for all edges e.

□

If we interpret H as an integer-weighted graph, we can use Lemma 4.12 to get a graph H' with the same tree packings as H but with the edge-multiplicities capped to an upper bound $\hat{\mu_H}$ of μ_H. As upper bound, we use $\hat{\mu_H} := 2(1+\epsilon)(d+2)(\ln n)/(\epsilon^2\gamma) = O(\log n)$ from Lemma 4.11 (b).
Thus, H' has at most $\widetilde{m}' = O(m\hat{\mu_H}) = O(m \log n)$ many simple edges in total. Since $m \log n \leq n^2 n = n^3$, we get $O(\log \widetilde{m}') = O(\log n)$. Hence, the runtime of Algorithm 2 on H' is $O(m\mu_H(\log n)(\log \widetilde{m}')) = O(m \log^3 n)$.

In practice, we never construct H but construct H' directly from G' by already considering the upper bound of edge-multiplicities $\hat{\mu_H}$ during sampling. The sampling is then done as proposed by Bhardwaj et al. [4]:
For every multi-edge e of G', let X_e be the random variable denoting the number of simple edges sampled from e, meaning $m_H(e) = X_e$. Since all edges are sampled independently at random with probability p, X_e follows the binomial-distribution $Bin(m_{G'}(e), p)$. One possible method to draw from the binomial distribution is to use inverse transform sampling [4]: For every multi-edge e, we draw numbers $U_e \sim U[0, 1]$ uniformly at random from 0 to 1 and compute
$Y_e := \min\{x \in \mathbb{N}_0 \mid U_e < \mathbb{P}(X_e \leq x)\}$.

Then $\mathbb{P}(Y_e = k) = \mathbb{P}(X_e = k)$, since for all k

$$\begin{aligned}\mathbb{P}(Y_e > k) &= \mathbb{P}(k \notin \{x \in \mathbb{N}_0 \mid \mathbb{P}(X_e \le x) > U_e\}) \\ &= \mathbb{P}(\mathbb{P}(X_e \le k) \le U_e) \\ &= \mathbb{P}(U_e \in [\mathbb{P}(X_e \le k), 1]) \\ &= 1 - \mathbb{P}(X_e \le k) \\ &= \mathbb{P}(X_e > k)\end{aligned}$$

Since we want every multi-edge to have multiplicity at most $\hat{\mu_H}$, it is sufficient to compute $\mathbb{P}(X_e \le k)$ for $k \in \{0, \dots, M_e\}$ with $M_e = \min\{m_{G'}(e), \hat{\mu_H}\}$ and set

$$m_{H'}(e) := \begin{cases} \min\{x \in \mathbb{N}_0 \mid U_e < \mathbb{P}(X_e \le x)\} & \text{if } U_e < \mathbb{P}(X_e \le M_e) \\ M_e & \text{if } U_e \ge \mathbb{P}(X_e \le M_e) \end{cases}$$

Since $M_e \le \hat{\mu_H} = O(\log n)$, the sampling can be done in $O(m \log n)$.

4.4 Estimating the minimum cut size

There is one last obstacle that we need to overcome. To compute sampling probability $p = 2(d+2)(\ln n)/(\epsilon_2^2 \gamma \mu_{G'})$ we need some lower bound $\hat{\mu_{G'}}$ of $\mu_{G'}$ with $\gamma := \hat{\mu_{G'}}/\mu_{G'} \le 1$ so that $\gamma = \Theta(1)$.
The approach of Bhardwaj et al. [4] is as follows: We start with some known upper bound U of $\mu_{G'}$ and attempt to run Algorithm 2 on the resulting graph H'. As long as the algorithm "fails", we cut the estimate in half and try again. We repeat this process until the algorithm "succeeds". For this approach to work, we need an upper bound U, a criterion to reject a false estimate of $\mu_{G'}$, and a criterion to accept a true estimate of $\mu_{G'}$ based on the tree packing that Algorithm 2 returns.
The following lemma, that is attributed to Karger [9], gives us an upper bound U:

Lemma 4.13 (Karger [9]). *Let G be a weighted graph with n vertices, T be a maximum spanning tree of G, and $w_{min} := \min_{e \in E_T} w_G(e)$ be the minimum edge-weight in T.*
Then $w_{min} \le \mu_G \le n^2 w_{min}$.

Proof.

- $w_{\min} \le \mu_G$ because ...
  Let $\mathcal{C} = \{X, X^C\}$ be a minimum cut of G and $v \in X$, $u \in X^C$.

  Let $P_{v,u} = \langle e_1, \dots, e_k \rangle$ be the path from v to u in T. Since v and u are on opposite sides of the cut, Lemma 3.1 implies that the number of crossing edges in $P_{v,u}$ is odd and thus $\neq 0$.

  Let $e_i \in E_G(\mathcal{C})$ be a crossing tree-edge. Then

$$\mu_G = \sum_{e \in E_G(\mathcal{C})} w_G(e) \ge w_G(e_i) \ge w_{\min}$$

- $\mu_G \le n^2 w_{\min}$ because ...
  Let $e_0 \in E_T$ be a minimal tree edge ($w_G(e_0) = w_{\min}$) and let $\mathcal{C} = \mathcal{C}_T(e_0)$ be the

cut who's only crossing edge in T is e_0. To reach a contradiction, assume that there is a crossing edge $e_1 \in E_G(\mathcal{C})$ with $w_G(e_1) > w_{\min}$. Then $T' := T - e_0 + e_1$ is a spanning tree of G and

$$\sum_{e \in E_{T'}} w_G(e) = \sum_{e \in E_T} w_G(e) + \underbrace{w_G(e_1)}_{> w_{\min}} - \underbrace{w_G(e_0)}_{= w_{\min}} > \sum_{e \in E_T} w_G(e)$$

which contradicts the maximality of T.
Therefore $w_G(e) \leq w_{\min}$ for all $e \in E_G(\mathcal{C})$ and thus

$$\mu_G \leq \sum_{e \in E_G(\mathcal{C})} w_G(e) \leq |E_G(\mathcal{C})| w_{\min} \leq |E_G| w_{\min} \leq n^2 w_{\min}$$

□

Thus, we can set $U = n^2 w_{\min}$ as the upper bound of $\mu_{G'}$.
On a side note, we cannot just use $\hat{\mu_{G'}} := w_{\min}$ as lower bound for $\mu_{G'}$. If, for example, $\mu_{G'} = n w_{\min}$, then $\gamma = \hat{\mu_{G'}}/\mu_{G'} = w_{\min}/(n w_{\min}) = 1/n \neq \Theta(1)$. But we need to require $\gamma = \Theta(1)$ to ensure that $\mu_H = O(\log n)$ as stated by Lemma 4.11 (b).

The following lemma gives us a criterion for a rejection of a false estimate:

Lemma 4.14. *Let $0 < \epsilon_2, \epsilon_3$, $(1-\epsilon_2)(1-\epsilon_3) \geq 2/3$ and let the sampling probability be $p = 2(d+2)(\ln n)/(\epsilon_2^2 \gamma \mu_{G'}) \leq 1$ for $\gamma \leq 1$.*
Let $\mathcal{P}$ be a tree packing of H with weight $w_{\mathcal{P}} \geq (1-\epsilon_3)\mu_H/2$.
Then $w_{\mathcal{P}} \geq \frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$ with probability $1 - O(1/n^d)$.

Proof. According to Lemma 4.11 (b), $\mu_H \geq (1-\epsilon_2)p\mu_{G'}$ with probability $1-O(1/n^d)$. Thus

$$\begin{aligned} w_{\mathcal{P}} &\geq \frac{(1-\epsilon_3)\mu_H}{2} \\ &\geq \frac{(1-\epsilon_3)(1-\epsilon_2)p\mu_{G'}}{2} \\ &= (1-\epsilon_3)(1-\epsilon_2)\frac{2(d+2)(\ln n)}{2\epsilon_2^2 \gamma \mu_{G'}}\mu_{G'} \\ &\geq (1-\epsilon_3)(1-\epsilon_2)\frac{(d+2)(\ln n)}{\epsilon_2^2} \\ &\geq \frac{2}{3}\frac{(d+2)(\ln n)}{\epsilon_2^2} \end{aligned}$$

□

According to Lemma 4.9, running Algorithm 2 on H' with approximation ϵ_3 yields a tree packing $\mathcal{P}$ of H' of weight at least $(1-\epsilon_3)\mu_{H'}/2$. Lemma 4.12 implies, that $\mathcal{P}$ is also a tree packing of H and that $(1-\epsilon_3)\mu_{H'}/2 = (1-\epsilon_3)\mu_H/2$. Thus, if Algorithm 2 returns a tree packing of weight smaller than $\frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$ for an estimate $\hat{\mu_{G'}}$ of $\mu_{G'}$, then $\hat{\mu_{G'}} > \mu_{G'}$ with probability $O(1/n^d)$ (Lemma 4.14). We then reject the estimate $\hat{\mu_{G'}}$ and try again with $\hat{\mu_{G'}} \leftarrow \hat{\mu_{G'}}/2$.
The remaining question is, under which conditions we can safely accept a tree packing. While a correct estimate $\hat{\mu_{G'}}$ will most likely result in a tree packing of weight at least $\frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$, there is no guarantee that a tree packing of such size comes from a correct estimate. There is, however, a (likely) limit on the factor by which the estimate $\hat{\mu_{G'}}$ can surpass $\mu_{G'}$ in such a case.

Lemma 4.15 (Bhardwaj et al. [4])**.** *Let $0 < \epsilon_2 \leq 1/3$ and $p = 2(d+2)(\ln n)/(\epsilon_2^2\gamma\mu_{G'})$ but with $\gamma \geq 6$. Let G' be a multigraph and H be the multigraph obtained from G' by sampling each simple edge with probability p.*
Then H has the following properties with probability at least $1 - 1/n^{d+2}$:

(a) $\mu_H < \frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$

(b) All tree packings $\mathcal{P}$ of H have weight $w_{\mathcal{P}} < \frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$

To prove this lemma, we make use of the following chernoff bound:

Lemma 4.16 (Chernoff bound)**.** *Let $n \in N, 0 \leq p \leq 1$.*
Let $X \sim Bin(n,p)$ be a random variable that follows the binomial-distribution.
Then $\mathbb{P}(X \geq (1+\delta)np) \leq e^{-\frac{1}{3}\delta np}$ for all $\delta \geq 1$.

Proof of Lemma 4.15.

(a) Let $\mathcal{C}$ be the minimum cut of G' and $w_H(\mathcal{C})$ be the random variable denoting is weight in H. Since H and G' have the same multi-edges (potentially with multiplicity 0 in H), we get $E_{G'} = E_H$. Therefore

$$w_H(\mathcal{C}) = \sum_{e\in E_H(\mathcal{C})} \underbrace{m_H(e)}_{\sim Bin(m_{G'}(e),p)} \sim Bin(\sum_{e\in E_{G'}(\mathcal{C})} m_{G'}(e), p) = Bin(\mu_{G'}, p)$$

and according to Lemma 4.16,

$$\mathbb{P}\left(w_H(\mathcal{C}) \geq (1+\delta)\mu_{G'}p\right) \leq \exp\left(-\frac{1}{3}\delta\mu_{G'}p\right)$$

for all $\delta \geq 1$ and $\exp(x) := e^x$.
With $\delta := \gamma/3 - 1 \geq 6/3 - 1 = 1 \implies (1+\delta) = \gamma/3$, we get

$$\begin{aligned}
\mathbb{P}\left(w_H(\mathcal{C}) \geq \frac{2}{3}\frac{(d+2)(\ln n)}{\epsilon_2^2}\right) &= \mathbb{P}\left(w_H(\mathcal{C}) \geq \frac{\gamma}{3}\mu_{G'}\frac{2(d+2)(\ln n)}{\epsilon_2^2\gamma\mu_{G'}}\right) \\
&= \mathbb{P}\left(w_H(\mathcal{C}) \geq (1+\delta)\mu_{G'}p\right) \\
&\leq \exp\left(-\frac{1}{3}\delta\mu_{G'}p\right) \\
&= \exp\left(-\frac{1}{3}\left(\frac{\gamma}{3} - 1\right)\mu_{G'}\frac{2(d+2)(\ln n)}{\epsilon_2^2\gamma\mu_{G'}}\right) \\
&= \exp\left(-\frac{1}{3}\left(\frac{2}{3} - \frac{2}{\gamma}\right)\frac{(d+2)}{\epsilon_2^2}(\ln n)\right) \\
&\leq \exp\left(-\frac{1}{3}\left(\frac{2}{3} - \frac{2}{6}\right)\frac{(d+2)}{\left(\frac{1}{3}\right)^2}(\ln n)\right) \\
&= \exp\left(-(d+2)(\ln n)\right) \\
&= n^{-(d+2)}
\end{aligned}$$

Since $\mu_H \leq w_H(\mathcal{C})$, we get

$$\mathbb{P}\left(\mu_H < \frac{2}{3}\frac{(d+2)(\ln n)}{\epsilon_2^2}\right) \geq \mathbb{P}\left(w_H(\mathcal{C}) < \frac{2}{3}\frac{(d+2)(\ln n)}{\epsilon_2^2}\right) \geq 1 - n^{-(d+2)}$$

(b) Let $\mathcal{P}$ be a tree packing of H. According to Lemma 4.10

$$w_{\mathcal{P}} \leq \mu_H \overset{(a)}{<} \frac{2}{3}\frac{(d+2)(\ln n)}{\epsilon_2^2}$$

□

Therefore, if Algorithm 2 returns a tree packing of weight at least $\frac{2}{3}(d+2)(\ln n)/\epsilon_2^2$, we can assume $\hat{\mu_{G'}} < 6\mu_{G'}$. We then run the algorithm one last time for $\hat{\mu_{G'}}/6$ and return the resulting tree packing.

4.5 Putting it all together

Putting it all together, we get the following algorithm:

Algorithm 3 Obtain $\Theta(\log n)$ spanning trees for the 2-respect Algorithm

```
Let $G$ be a weighted graph.
Let $d > 0$ be the exponent in the probability of success $1 - O(1/n^d)$.
Let $\epsilon_1, \epsilon_2, \epsilon_3 > 0$ such that $f := 3/2 - (1+\epsilon_1)(1+\epsilon_2)(1-\epsilon_2)^{-1}(1-\epsilon_3)^{-1} > 0$ and
$(1-\epsilon_2)(1-\epsilon_3) \geq 2/3$ and $\epsilon_2 \leq 1/3$.
Let $b = (d+2)(\ln n)/\epsilon_2^2$.
 1: Compute $G$s minimum weight $w_{\min}^G = \min_{e \in E_G} w_G(e)$
 2: Approximate $G$ as a multigraph $G'$ with multiplicities
        $m_{G'}(e) = \lceil \epsilon_1^{-1}(w_{\min}^G)^{-1} w_G(e) \rceil$
 3: Compute a maximum spanning tree $T$ of $G'$ with
        $G'$'s edge-multiplicities being interpreted as weights.
 4: Compute $T$s minimum edge weight $w_{\min}^T = \min_{e \in E_T} m_{G'}(e)$
 5: Initialize $c \leftarrow n^2 w_{\min}^T$ and $l \leftarrow False$          ▷ $l$ stands for "last round"
 6: while True do
 7:     Construct $H'$ from $G'$ by randomly choosing for each multi-edge $e$:
            $m_{H'}(e) \sim Bin(m_{G'}(e), p)$ with $p = \min\{2b/c, 1\}$ while
            capping $m_{H'}(e)$ to $\lceil 24(1+\epsilon_2)b \rceil$
 8:     Run Algorithm 2 on $H'$ with approximation $\epsilon_3$ and
            let $\mathcal{P}$ be the returned tree packing.
 9:     if $p = 1$ or $l = True$ then
10:         return $\lceil -d \ln n / \ln(1-f) \rceil$ trees randomly sampled from $\mathcal{P}$
                proportional to their weights
11:     end if
12:     if $w_{\mathcal{P}} \geq \frac{2}{3}b$ then
13:         $c \leftarrow c/6$
14:         $l \leftarrow True$
15:     else
16:         $c \leftarrow c/2$
17:     end if
18: end while
```

Bhardwaj et al. [4] propose to use the constants $\epsilon_1 = 1/100$, $\epsilon_2 = 1/6$, $\epsilon_3 = 1/5$.

Lemma 4.17 (Bhardwaj et al. [4]). *Let G be a weighted graph and $\mathcal{C}$ be a minimum cut of G. Then, running Algorithm 3 on the graph G returns $\Theta(\log n)$ many spanning trees of G in $O(m \log^3 n)$ time such that $\mathcal{C}$ 2-respects at least one of the trees with probability $1 - O(1/n^d)$.*

Proof.
Correctness
We denote the graph H' without capped edge-multiplicities as H. Since H's edge-multiplicities are capped to $\lceil 24(1+\epsilon_2)b \rceil$, we get $\mu_{H'} = \mu_H$ and H and H' have the same tree packings if $\mu_H \leq \lceil 24(1+\epsilon_2)b \rceil$ (Lemma 4.12).
The loop can be in four distinct states with a high probability. We aim to demonstrate that, in each of these states, the algorithm will succeed with a probability of $1 - O(1/n^d)$.

1. Assume that ($\mu_{G'}/12 \leq c \leq \mu_{G'}$ and $l = True$) or ($\mu_{G'}/12 \leq c$ and $p = 1$).
   Since G is approximated by G' as described in Lemma 4.7,
   we get $w_{G'}(\mathcal{C}) \leq (1+\epsilon_1)\mu_{G'}$.
   Let $\alpha, \beta > 0$ such that $w_H(\mathcal{C}) = \alpha \mu_H$ and $w_{\mathcal{P}} = \beta \mu_H$.
   We get $\alpha \leq (1+\epsilon_1)(1+\epsilon_2)(1-\epsilon_2)^{-1}$ and $\mu_H \leq \lceil 24(1+\epsilon_2)b \rceil$, because ...

   (a) Assume $p < 1$ and $c \leq \mu_{G'}$.
   $\min\{2b/c, 1\} = p < 1$ implies $p = 2b/c = 2(d+2)(\ln n)/(\epsilon_2^2 \gamma \mu_{G'})$ for $\gamma := c/\mu_{G'}$. As $1/12 \leq \gamma \leq 1$, we get $\gamma = \Theta(1)$.
   Using Lemma 4.11 (c) and $w_{G'}(\mathcal{C}) \leq (1+\epsilon_1)\mu_{G'}$, we get

   $$w_H(\mathcal{C}) \leq (1+\epsilon_1)(1+\epsilon_2)(1-\epsilon_2)^{-1}\mu_H$$

   and Lemma 4.11 (b) yields

   $$\mu_H \leq \frac{2(1+\epsilon_2)(d+2)(\ln n)}{\epsilon_2^2 \gamma} \leq 24(1+\epsilon_2)\frac{(d+2)(\ln n)}{\epsilon_2^2} \leq \lceil 24(1+\epsilon_2)b \rceil$$

   with both bounds being true with probability $1 - O(1/n^d)$.

   (b) Assume $p = 1$. Then $G' = H$. Therefore

   $$w_H(\mathcal{C}) = w_{G'}(\mathcal{C}) \leq (1+\epsilon_1)\underbrace{\mu_{G'}}_{=\mu_H} \leq (1+\epsilon_1)\underbrace{(1+\epsilon_2)(1-\epsilon_2)^{-1}}_{>1}\mu_H$$

   Since $2b/c \geq p = 1$, we get $\lceil 24(1+\epsilon_2)b \rceil \geq 24b \geq 12c \geq \mu_{G'} = \mu_H$.

   Since $\mu_H \leq \lceil 24(1+\epsilon_2)b \rceil$, $\mathcal{P}$ is also a tree packing of H and $\mu_H = \mu_{H'}$.
   Since running Algorithm 2 on H' with approximation ϵ_3 returns a tree packing $\mathcal{P}$ of weight at least $(1-\epsilon_3)\mu_{H'}/2 = (1-\epsilon_3)\mu_H/2$ (Lemma 4.9), we get
   $\beta \geq (1-\epsilon_3)/2$.

   Applying Lemma 4.6 to the cut $\mathcal{C}$ and the tree packing $\mathcal{P}$ in H, we get that $\mathcal{C}$ 2-respects a fraction of at least

   $$\frac{1}{2}\left(3 - \frac{\alpha}{\beta}\right) \geq \frac{3}{2} - \frac{1}{2}\frac{(1+\epsilon_1)(1+\epsilon_2)(1-\epsilon_2)^{-1}}{(1-\epsilon_3)/2} = \frac{3}{2} - \frac{(1+\epsilon_1)(1+\epsilon_2)}{(1-\epsilon_2)(1-\epsilon_3)} = f$$

trees by weight.

Since $p = 1$ or $l = True$, we return $t := \lceil -d \ln n / \ln(1-f) \rceil$ randomly sampled trees of $\mathcal{P}$ proportional to their weights and the algorithm terminates.
The probability that the cut $\mathcal{C}$ 2-respects none of the sampled trees is

$$(1-f)^t \leq (1-f)^{-d \ln n / \ln(1-f)} = n^{-d}$$

Because of this and since case 1a fails with probability $O(1/n^d)$, the overall probability of failure of case 1 is at worst $O(1/n^d) + n^{-d} = O(1/n^d)$.

2. Assume that $\mu_{G'}/2 < c \leq \mu_{G'}$ and $l = False$ and $p < 1$.

   Analogous to case 1a, we get $\mu_H \leq \lceil 24(1+\epsilon_2)b \rceil$ with probability $1 - O(1/n^d)$, meaning $\mathcal{P}$ is also a tree packing of H and $\mu_H = \mu_{H'}$.
   Since $w_{\mathcal{P}} \geq (1-\epsilon_3)\mu_{H'}/2 = (1-\epsilon_3)\mu_H/2$ (Lemma 4.9) and since the sampling probability is $p = 2b/c = \frac{2}{3}(d+2)(\ln n)/(\epsilon_2^2 \gamma \mu_{G'})$ for $\gamma := c/\mu_{G'} \leq 1$, Lemma 4.14 implies that $w_{\mathcal{P}} \geq \frac{2}{3}b$ with probability $1 - O(1/n^d)$.
   In this case, Algorithm 3 will proceed to the next iteration with $c \leftarrow c/6$ and $l \leftarrow True$. Thus, we run the next iteration with $\mu_{G'}/12 < c \leq \mu_{G'}/6 < \mu_{G'}$ and $l = True$ which is case 1.
   Since case 1 has a probability of failure of $O(1/n^d)$, the overall probability of failure of this case is at most $O(1/n^d) + O(1/n^d) = O(1/n^d)$.

3. Assume $\mu_{G'} < c < 6\mu_{G'}$ and $p < 1$.

   (a) If $\mathcal{P}$ has weight $w_{\mathcal{P}} \geq \frac{2}{3}b$, Algorithm 3 will proceed to the next iteration with $c \leftarrow c/6$ and $l \leftarrow True$. Thus, we run the next iteration with $\mu_{G'}/6 < c < \mu_{G'}$ and $l = True$ which is case 1.

   (b) If $\mathcal{P}$ has weight $w_{\mathcal{P}} < \frac{2}{3}b$, Algorithm 3 will proceed to the next iteration with $c \leftarrow c/2$. This can happen at most 3 times, since $c < 2^{-3}6\mu_{G'} = 3\mu_{G'}/4 < \mu_{G'}$ ($\rightarrow$ case 2) after the third time. Alternatively, this can lead to case 1 if $p = 1$ or to case 3a if $w_{\mathcal{P}} \geq \frac{2}{3}b$ in the next round.

   Since both scenarios will eventually lead to case 1 or case 2 with certainty and since case 1 and case 2 have a probability of failure of at most $O(1/n^d)$ each, the probability of failure of this case is at most $O(1/n^d)$.

4. Assume $c \geq 6\mu_{G'}$ and $p < 1$.
   Since we sample edges with probability $p = 2b/c = \frac{2}{3}(d+2)(\ln n)/(\epsilon_2^2 \gamma \mu_{G'})$ for $\gamma := c/\mu_{G'} \geq 6$, Lemma 4.15 implies

$$\mu_H < \frac{2}{3} \frac{(d+2)(\ln n)}{\epsilon_2^2} = \frac{2}{3}b < \lceil 24(1+\epsilon_2)b \rceil$$

   and $w_{\mathcal{P}'} < \frac{2}{3}(d+2)(\ln n)/\epsilon_2^2 = \frac{2}{3}b$ for all tree packings $\mathcal{P}'$ of H with probability at least $1 - 1/n^{d+2}$. Since H and H' have the same tree packings, we get $w_{\mathcal{P}} < \frac{2}{3}b$ and Algorithm 3 will proceed to the next iteration with $c \leftarrow c/2$.
   Since $c \leq n^2 w^T_{\min}$ and $\mu_{G'} \geq w^T_{\min}$ (Lemma 4.13), this can happen at most $\lceil \log_2(n^2 w^T_{\min}/w^T_{\min}) \rceil = O(\log n)$ times before we reach case 3 ($c < 6\mu_{G'}$) or alternatively case 1 ($p = 1$), which both have failure probabilities of $O(1/n^d)$.
   Therefore, the overall probability of failure for this case is at most

$$O(n^{-(d+2)} \log n) + O(1/n^d) \leq O(1/n^d) + O(1/n^d) = O(1/n^d)$$

The remaining cases

- $c < \mu_{G'}/12$
- $c \leq \mu_{G'}/2$ and $l = False$
- $c > \mu_{G'}$ and $l = True$

can only occur if the algorithm fails at some stage: These cases cannot be initial states since $c = n^2 w^T_{\min} \geq \mu_{G'}$ and $l = False$ before the start of the loop, and no other case leads to them, assuming everything goes as planned.

Time-complexity
The initialization can be done in $O(m \log n)$ time:
Computing Gs minimum weight and approximating G as G' takes $O(m)$ time.
Computing G's maximum spanning tree T using Kruskal's algorithm takes $O(m \log n)$ time (Lemma A.4). Computing Ts minimum edge weight takes $O(m)$ time.

The total runtime of the loop is $O(m \log^3 n)$ with high probability:

Since case 4 can only occur at most $O(\log n)$ times, case 3b at most 3 times, and all other cases at most once, there are at most $O(\log n)$ iterations of the main-loop. As sampling can be done in $O(m \log n)$ time as explained in section 4.3, the algorithm spends at most $O(m \log^2 n)$ time on sampling across all iterations.

The expected runtime of Algorithm 2 across all iterations is $O(m \log^3 n)$ for the following reason:
Let k be the number of iterations of the main loop. Let c_i, p_i, H_i, H'_i be the values c, p, H, H' from iteration i.
Since the algorithm terminates if $p = 1$, we know that $p_i < 1$ and thus $p_i = 2b/c_i$ for $i = 1, \dots, k-1$. Because $c_{i+1} \leq c_i/2$, we get $p_{i+1} = 2b/c_{i+1} \geq 4b/c_i = 2p_i$ for $i = 1, \dots, k-2$, which implies

$$\sum_{i=1}^{k} p_i = p_k + \sum_{i=1}^{k-1} p_i \leq p_k + \sum_{i=1}^{k-1} p_{k-1} \frac{1}{2^{(k-1)-i}} = p_k + \underbrace{p_{k-1}}_{\leq p_k} \underbrace{\sum_{i=0}^{k-2} \frac{1}{2^i}}_{\leq 2} \leq 3p_k$$

Let $\mathcal{C}'$ be a minimum cut of G'. Since $m_{H_i}(e) \sim Bin(m_{G'}(e), p_i)$, we get $\mathbb{E}[m_{H_i}(e)] = p_i m_{G'}(e)$, and therefore

$$\mathbb{E}[\mu_{H'_i}] \leq \mathbb{E}[w_{H'_i}(\mathcal{C}')] = \mathbb{E}\left[\sum_{e \in E_{H'_i}(\mathcal{C}')} m_{H'_i}(e)\right] \leq \mathbb{E}\left[\sum_{e \in E_{H_i}(\mathcal{C}')} m_{H_i}(e)\right]$$
$$= p_i \sum_{e \in E_{G'}(\mathcal{C}')} m_{G'}(e) = p_i \mu_{G'}$$

Let N_i be the number of iterations for which Algorithm 2 ran in iteration i and let $N = \sum_{i=1}^{k} N_i$ be the total number of iterations. Further, let $\widetilde{m}_i$ be the number of simple edges of H'_i and $\widetilde{m}_{\max} := \max_{i=1,\dots,k} \widetilde{m}_i$. According to Lemma 4.9,

$$N_i \leq \frac{3 \ln \widetilde{m}_i}{\epsilon^2} \mu_{H'_i} \leq \frac{3 \ln \widetilde{m}_{\max}}{\epsilon^2} \mu_{H'_i}$$

Therefore

$$\begin{aligned}
\frac{\epsilon^2}{3\ln \widetilde{m}_{\max}}\mathbb{E}[N] &= \frac{\epsilon^2}{3\ln \widetilde{m}_{\max}}\sum_{i=1}^{k}\mathbb{E}[N_i] && \leq \sum_{i=1}^{k}\mathbb{E}[\mu_{H_i'}] && \leq \sum_{i=1}^{k} p_i\mu_{G'} \\
&\leq 3p_k\mu_{G'} && \leq 3\frac{2b}{c_k}\mu_{G'} && \leq 3\frac{2b}{\mu_{G'}/12}\mu_{G'} \\
&= 72b
\end{aligned}$$

Since the edge-multiplicities of the graphs H_i' are capped to $\lceil 24(1+\epsilon_2)b\rceil$ and as $b = (d+2)(\ln n)/\epsilon_2^2 = O(\log n)$, we get $\ln \widetilde{m}_{\max} \leq \ln(m\lceil 24(1+\epsilon_2)b\rceil) = O(\log n)$. Therefore $\mathbb{E}[N] \leq 216b(\ln \widetilde{m}_{\max})/\epsilon_2^2 = O(\log^2 n)$. Since one iteration of Algorithm 2 takes $O(m\log n)$ time (Lemma 4.9), the total expected runtime of Algorithm 2 across all iterations of Algorithm 3 is $O(m\log^3 n)$.
Using an argument similar to Lemma 4.15, this can be turned into a high probability statement.

Thus, the total expected runtime of Algorithm 3 is

$$O(m\log n) + O(m\log^2 n) + O(m\log^3 n) = O(m\log^3 n)$$

□

5 Finding the smallest 2-respecting cut of a spanning tree

Let G be a weighted graph with n vertices and m edges and let T be a spanning tree of G.

Definition 5.1. *A rooted tree T is a tree that has some vertex r designated as root vertex.*
Let v be a vertex and $P_{v,r} = vw_1 \dots w_k r$ be the path from v to the root vertex r.
We define

- *the depth of v as the length of the path $P_{v,r}$*
- *$v^{\uparrow} := \{v, w_1, \dots, w_k, r\}$ as the set of ancestors of v*
- *$v^{\downarrow} := \{u \in V_T \mid v \in u^{\uparrow}\}$ as the set of descendants of v.*

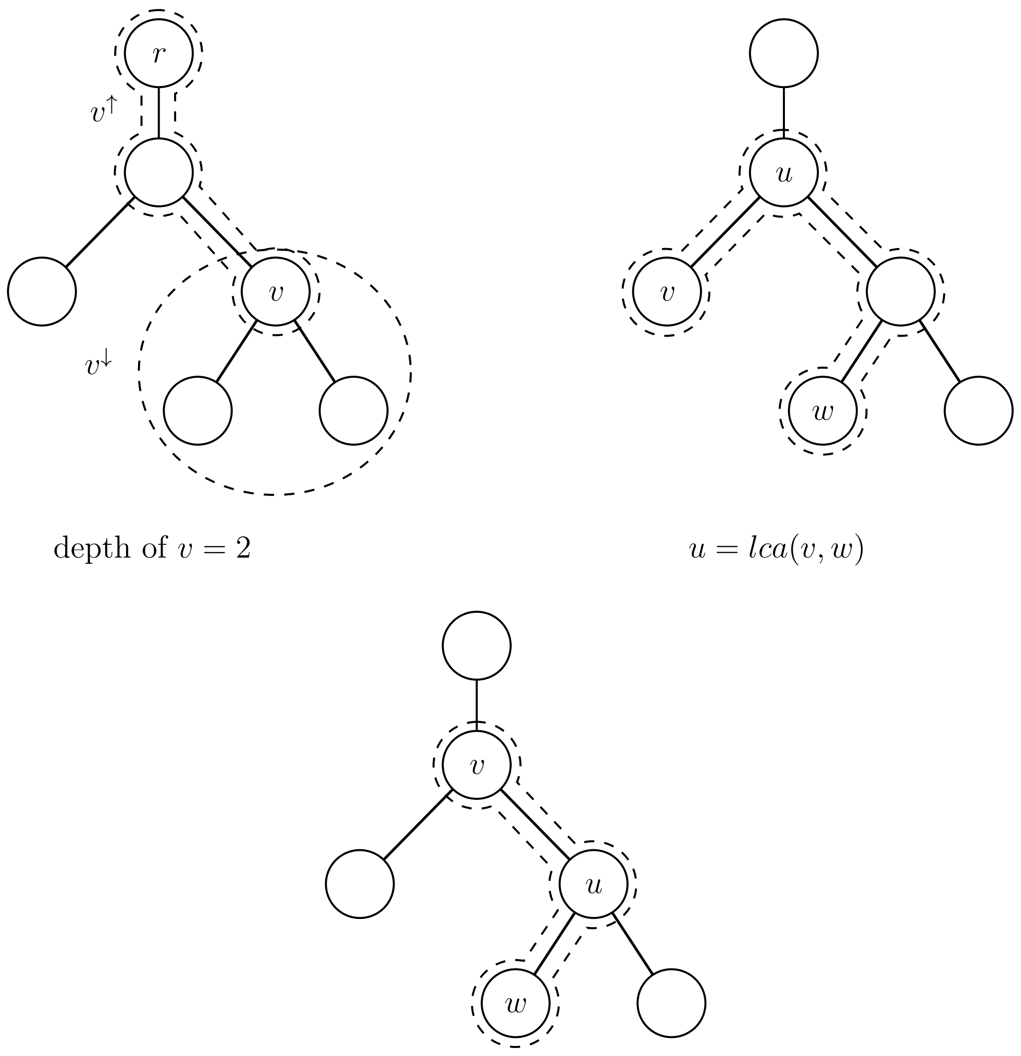
The depth of the tree is the depth of the deepest vertex.
If two vertices v and w are connected by an edge $\{v, w\}$ and $w \in v^{\downarrow}$, we call w a child of v and v the parent of w.
For two vertices v, w, we define the lowest common ancestor of v and w as the deepest vertex that is an ancestor of both v and w.
We write $lca(v, w) :=$ lowest common ancestor v of w.

Definition 5.2. *Let T be a rooted tree. We call a path $P = v_1 \dots v_k$ a downward path if $v_{i+1} \in v_i^{\downarrow}$ for all $i = 1, \dots, k-1$.*

We further define $[a..b] = \{a, a+1, \dots, b-1, b\}$ for all $a, b \in \mathbb{Z}$ with $a \leq b$.

Let T be rooted tree with an arbitrarily chosen root vertex r. The goal of this chapter is to find the smallest cut of G that 2-respects T in $O(m \log^3 n)$ time. Throughout this chapter, we denote $\mathcal{C}(e_1, \dots, e_k) := \mathcal{C}_T(e_1, \dots, e_k)$ for tree-edges $e_1, \dots, e_k$.
To find the smallest 2-respecting cut, we search for the smallest 1-respecting cut and the smallest strictly 2-respecting cut separately.

Figure 5.1: Depth, ancestors, descendants, and lca of vertices, and a downward path



depth of $v = 2$ $\quad$ $u = lca(v, w)$

$P = vuw$ is a downward path

5.1 Finding the smallest 1-respecting cut

Let $e_1, \dots, e_{n-1}$ be the edges of T.
We can find the smallest 1-respecting cut by computing $w_G(\mathcal{C}(e_j))$ for all $j = 1, \dots, n-1$ and finding e_i such that $w_G(\mathcal{C}(e_i)) = \min_{j=1,\dots,n-1} w_G(\mathcal{C}(e_j))$. Since T only has $n-1$ edges, we only need to iterate over $O(n)$ many cuts. The main problem is to efficiently calculate the weights of the cuts. Each cut can have $O(m)$ many crossing edges, meaning that computing their weights individually by summing over the weights of their respective crossing edges would take $O(nm)$ time in total. Instead, Bhardwaj et al. [4] propose the following strategy: We only compute the weight of the first cut $\mathcal{C}(e_1)$ by summing all of its edge-weights together and then compute the differences

$$\delta_i = w_G(\mathcal{C}(e_i)) - w_G(\mathcal{C}(e_{i-1})) \text{ for } i = 2, \dots, n-1$$

Let $S_i^+ = E_G(\mathcal{C}(e_i)) \setminus E_G(\mathcal{C}(e_{i-1}))$ be the set of edges that cross $\mathcal{C}(e_i)$ but not $\mathcal{C}(e_{i-1})$ and $S_i^- = E_G(\mathcal{C}(e_{i-1})) \setminus E_G(\mathcal{C}(e_i))$ be the set of edges that cross $\mathcal{C}(e_{i-1})$ but not

$\mathcal{C}(e_i)$. Then

$$\delta_i = \sum_{e \in E_G(\mathcal{C}(e_i))} w_G(e) - \sum_{e \in E_G(\mathcal{C}(e_{i-1}))} w_G(e) = \sum_{e \in S_i^+} w_G(e) - \sum_{e \in S_i^-} w_G(e)$$

Thus, to calculate δ_i for some i, we need to find the sets S_i^+ and S_i^- and perform $|S_i^+| + |S_i^-|$ many additions. Then we can easily compute
$w_G(\mathcal{C}(e_i)) = w_G(\mathcal{C}(e_{i-1})) + \delta_i$.
This approach is not necessarily fast for an arbitrary arrangement of the tree-edges. But it turns out that we can reach a runtime of $O(m \log n)$ if we arrange the tree edges in a particular order, so that the differences between the adjacent cuts are relatively small.
To reach this goal, we first introduce an equivalent definition for the sets S_i^+ and S_i^-:

Lemma 5.3.
$S_i^+ = \{\{v, w\} \in E_G \mid P_{v,w}$ *contains* e_i *but not* $e_{i-1}\}$ *and*
$S_i^- = \{\{v, w\} \in E_G \mid P_{v,w}$ *contains* e_{i-1} *but not* $e_i\}$

Proof. According to Lemma 3.1, $e = \{v, w\}$ crosses the cut $\mathcal{C}(e_j)$ for some tree edge e_j if and only if the path $P_{v,w}$ from v to w contains an odd number of edges that cross $\mathcal{C}(e_j)$. Since $\mathcal{C}(e_j)$s only crossing edge is e_j, e crosses $\mathcal{C}(e_j)$ if and only if $P_{v,w}$ contains e_j. □

We can make the sets S_i^+ and S_i^- small by ordering the tree-edges $e_1, \dots, e_{n-1}$ in such a way, that the edges of all tree-paths are mostly contiguous in the order. To understand how this can be achieved, we take a step back and introduce the concept of the heavy-light-decomposition from Sleator and Endre Tarjan [15].

Definition 5.4. *Let* $e = \{v, u\}$ *be an edge of a rooted tree* T *and* v *be the parent of* u.
We call e *heavy if* $|u^\downarrow| \geq |v^\downarrow|/2$. *Otherwise we call* e *light.*
We call u *a heavy (light) child of* v *if* e *is heavy (light).*

Note that every vertex can have at most one heavy child: If there would be a vertex v with two heavy children u, w, then

$$|v^\downarrow| \geq |\{v\} \cup u^\downarrow \cup w^\downarrow| = 1 + |u^\downarrow| + |w^\downarrow| \geq 1 + 2|v^\downarrow|/2 > |v^\downarrow|$$

contradiction!

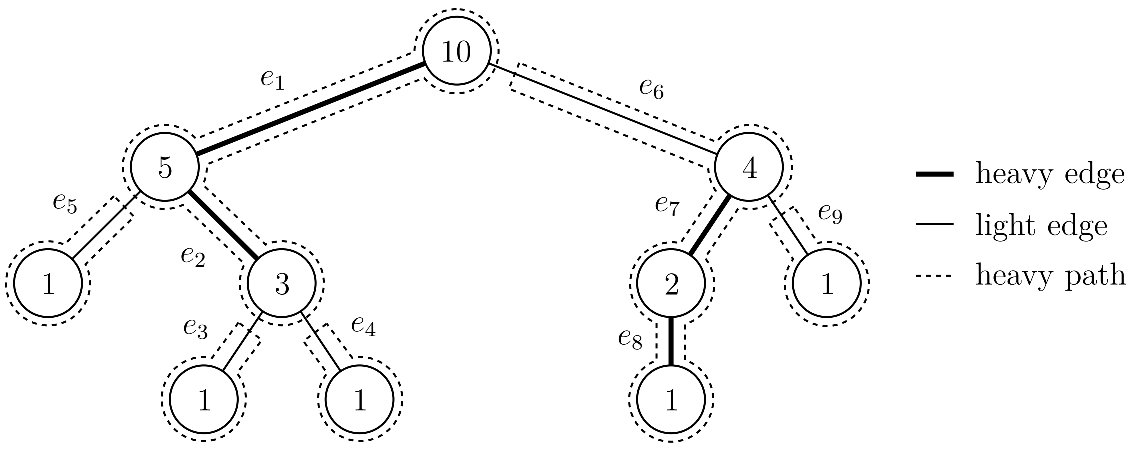
We order the tree edges $e_1, \dots, e_{n-1}$ and split the ordering into contiguous segments with the following algorithm:

Algorithm 4 Heavy-light decomposition

1. Compute $|v^{\downarrow}|$ for all vertices v using a post-order traversal: For every vertex v with children $w_1, \dots, w_k$, recursively compute $|w_i^{\downarrow}|$ for all i and then calculate $|v^{\downarrow}| = 1 + |w_1^{\downarrow}| + \dots + |w_k^{\downarrow}|$
2. Traverse the tree with depth-first search while always visiting the heavy child of a vertex first if it exists. Order the edges in the order in which there are traversed.
3. Split the obtained ordering $e_1, \dots, e_{n-1}$ into contiguous segments with a new segment starting right before every light edge. For every edge e, store the first edge of the segment that e belongs to.

The segments are called the heavy-paths of G.

Figure 5.2: Example of a heavy-light decomposition



$$P_1 = \langle e_1, e_2 \rangle, \quad P_2 = \langle e_3 \rangle, \quad P_3 = \langle e_4 \rangle, \quad P_4 = \langle e_5 \rangle, \quad P_5 = \langle e_6, e_7, e_8 \rangle, \quad P_6 = \langle e_9 \rangle$$

Lemma 5.5. *Algorithm 4 takes $O(n)$ time.*

Proof.

1. During the post-order traversal, every vertex is processed exactly once. Processing a vertex v with children $w_1, \dots, w_k$ takes constant time if we attribute the summation $|w_1^{\downarrow}| + \dots + |w_k^{\downarrow}|$ to its children: For every node w_i, one addition $\cdot + |w_i|$ needs to be performed for its parent v.

   Thus, this step takes $O(n)$ time.
2. During the depth-first search, every vertex is processed exactly once. For every vertex v with children $w_1, \dots, w_k$, we need to check if $|w_i^{\downarrow}| \geq |v^{\downarrow}|/2$ for any i and visit v's children starting with w_i if such an i exists. This takes a constant time per vertex if we attribute the search of heavy children to v's children: For every none w_i, one check $|w_i^{\downarrow}| \geq |v^{\downarrow}|/2$ needs to be performed for its parent v.

   Thus, this step takes $O(n)$ time.

3. This can be done by iterating over $i = 1, \dots, n-1$, which takes $O(n)$ time.

□

The ordering, that Algorithm 4 creates, has the following property:

Lemma 5.6. *Let the tree-edges $e_1, \dots, e_{n-1}$ be ordered by Algorithm 4.*
Let $P = \langle f_1, f_2, \dots, f_k \rangle$ be a downward path so that the edges $f_2, \dots, f_k$ are heavy.
Then, the edges $f_1, f_2, \dots, f_k$ form a contiguous subsequence of the order, meaning $f_1 = e_i,\ f_2 = e_{i+1},\ \dots,\ f_k = e_{i+k-1}$ for some i.

Proof. We prove the statement by showing that for all $j = 1, \dots, k-1$: $f_j = e_i$ and $f_{j+1} = e_{i+1}$ for some i.
Let $j \in [1..k-1]$ and $f_j = \{u, v\}, f_{j+1} = \{v, w\}$, so that u is the parent of v and v is the parent of w. By the definition of P, f_{j+1} must be a heavy edge. Thus, w is a heavy child of v.
Let i be the position of f_j in the order, meaning $f_j = e_i$. After traversing e_i, Algorithm 4 visits v's heavy child w first and thus traverses f_{j+1} immediately after traversing f_j. Therefore, $f_{j+1} = e_{i+1}$. □

We now prove that $\sum_{i=2}^{n-1}(|S_i^+| + |S_i^-|) = O(m \log n)$ for this ordering. We start by proving the following lemma:

Lemma 5.7. *Let v be an arbitrary vertex and $P_{r,v}$ be the downward path form the root-vertex r to v. Further, let l_P be the number of light edges of $P_{r,v}$ and n be the number of vertices of the tree.*
Then $|v^{\downarrow}| \leq n/2^{l_P}$.

Proof. We prove the statement by induction over l_P

- Base Case $l_P = 0$
  Then $2^{l_P}|v^{\downarrow}| = |v^{\downarrow}| \leq n$
- Induction Step
  Let $l_P \geq 1$ and let the statement be true for all paths with fewer than l_P light edges. Let $P_{r,v} =: \langle f_1, \dots, f_k \rangle = u_1 \dots u_{k+1}$ ($u_1 = r$, $u_{k+1} = v$) and $f_i = \{u_i, u_{i+1}\}$ be the last light edge of $P_{r,v}$.

  Then $P_{r,u_i} = \langle f_1, \dots, f_{i-1} \rangle = u_1 \dots u_i$ has $l_G - 1$ light edges. By the induction hypothesis, $n \geq 2^{l_P - 1}|u_i^{\downarrow}|$. Since f_i is a light edge, we get $|u_{i+1}^{\downarrow}| < |u_i^{\downarrow}|/2$.

  Therefore, $n \geq 2^{l_P-1}|u_i^{\downarrow}| \geq 2^{l_P}|u_{i+1}^{\downarrow}| \geq 2^{l_P}|u_{i+2}^{\downarrow}| \geq \dots \geq 2^{l_P}|u_{k+1}^{\downarrow}| = 2^{l_P}|v^{\downarrow}|$.

□

This implies …

Lemma 5.8. *Let $P_{r,w}$ be a downward path of T from the root-vertex r to w and let the edges $e_1, \dots, e_{n-1}$ of T be ordered by Algorithm 4.*
Then P consists of a union of at most $(\log_2 n)+1 = O(\log n)$ contiguous subsequences of the order.
The start- and endpoints of these subsequences can be computed in $O(\log n)$ time.

Proof. Let r be the root vertex and $P_{r,w} = \langle f_1, \dots, f_k \rangle$ be the downward path from r to w. Let l be the number of light edges of $P_{r,w}$.
Lemma 5.7 implies $n \geq 2^l |w^{\downarrow}| \geq 2^l$. Thus $l \leq \log_2 n$.
Let $f_{i_1}, \dots, f_{i_l}$ be the light edges of $P_{r,w}$ so that $i_j < i_{j+1}$. We split $P_{r,w}$ into $l+1$ paths

$$\begin{aligned} P_1 &= \langle f_1, \dots, f_{i_1 - 1} \rangle, \\ P_2 &= \langle f_{i_1}, \dots, f_{i_2 - 1} \rangle, \ \dots, \ P_l = \langle f_{i_{l-1}}, \dots, f_{i_l - 1} \rangle, \\ P_{l+1} &= \langle f_{i_l}, \dots, f_k \rangle \end{aligned}$$

(if $l_1 = 1$, then P_1 is empty). Since the paths $P_1, \dots, P_{l+1}$ can only contain heavy edges with the exception of the first edge, they each from a contiguous subsequence of the order of the tree edges as stated by Lemma 5.6. Thus, $P_{r,w}$ consists of a union of at most $(\log_2 n) + 1 = O(\log n)$ contiguous subsequences of the order.

To find the first and last edges of the paths $P_1, \dots, P_{l+1}$, we start at w and move up to r while storing the first edge of the old path and the last edge of the new path whenever we switch paths (= when we encounter a light edge). This process can be sped up by using the following insight: Let f_i be the first and f_j be the last edge of a path P_p ($p \in [1..l+1]$). Then f_i is the first edge of f_j's heavy path.
This is the case for the following reason:
By the construction of the heavy-paths from Algorithm 4, no new heavy path starts between f_i and f_j, as there is no light edge between f_i and f_j. Therefore, f_i and f_j are part of the same heavy path. f_i is the first edge of the heavy-path, as

- in the case of $p \geq 2$, f_i is a light edge. Therefore, the heavy path begins right before f_i.
- in the case of $p = 1$ (assuming P_1 is non-empty), because $f_1 = e_1$: If P_1 is non-empty, then $l_1 > 1$, meaning the f_1 is a heavy edge. Since f_1 is the first edge of the path $P_{r,w}$, f_1 is adjacent to the root-vertex r. As Algorithm 4 starts at the root-vertex and traverses the heavy edges first, f_1 is the first edge of the ordering and thus also the first edge of its heavy-path.

As Algorithm 4 stores the first edges of the heavy paths, we can jump directly from the last edge f_j to the first edge f_i of the path P_p in $O(1)$ time by looking up the first edge of f_j's heavy path.
Doing this for all $l+1$ paths takes $(l+1)O(1) = O(\log n)$ time in total. □

Lemma 5.8 can be generalized to all paths in T.

Lemma 5.9. *Let $P_{v,w}$ be any path in T with endpoints v and w and let the edges $e_1, \dots, e_{n-1}$ of T be ordered by Algorithm 4.*
Then $P_{v,w}$ consists of a union of at most $2\log_2 n + 2 = O(\log n)$ contiguous subsequences of the order.
The start- and endpoints of these subsequences can be computed in $O(\log n)$ time.

Proof. Let u be the lowest common ancestor of v and w. Then $P_{u,v}$ and $P_{u,w}$ are downward paths and $P_{v,w}$ is the union of $P_{v,u}$ and $P_{u,w}$. Furthermore, $P_{r,v}$ is the union of $P_{r,u}$ and $P_{u,v}$ and $P_{r,w}$ is the union of $P_{r,u}$ and $P_{u,w}$.

According to Lemma 5.8, $P_{r,v}$ and $P_{r,w}$ each consist of a union of at most $(\log_2 n) + 1 = O(\log n)$ contiguous subsequences of the order whose start- and endpoints can be computed in $O(\log n)$ time.
Thus, the walk $P_{v,r}P_{r,w} = P_{v,u}P_{u,r}P_{r,u}P_{u,w}$ consist of a union of at most $2(\log_2 n) + 2 = O(\log n)$ contiguous subsequences of the order. The path $P_{v,w}$ can be obtained by removing the middle part $P_{u,r}P_{r,u}$ from the walk $P_{v,r}P_{r,w}$. Therefore, $P_{v,w}$ consists of at most as many contiguous subsequences as the walk $P_{v,r}P_{r,w}$.

To compute these subsequences, we first compute the subsequences of $P_{r,v}$ and $P_{r,w}$. Since both of these paths start with $P_{r,u}$ and differ after u, we need to find the first edge at which the paths differ and remove all previous ones. This can be done in $O(\log n)$ time by comparing the start- and endpoints of the subsequences of $P_{r,v}$ and $P_{r,w}$. □

With this knowledge, we can return to the original problem.
We find the sets S_i^+ and S_i^- as well as $E_G(\mathcal{C}(e_1))$ with the following algorithm:

Algorithm 5 Compute the intervals and find S_i^+, S_i^-

```
 1: Initialize sets $T_i^+ \leftarrow \emptyset$ and $T_i^- \leftarrow \emptyset$ for $i = 1, \dots, n$
 2: for all edges $e = \{v, w\}$ of $G$ do
 3:     Compute $O(\log n)$ pairwise disjoint intervals $I_1^e, \dots, I_k^e$ so that
            $\{j \in [1..n-1] \mid P_{v,w}$ contains $e_j\} = \bigcup_{i=1}^k I_i^e$
 4:     for all $I = [i..j] \in \{I_1^e, \dots, I_k^e\}$ do
 5:         Add $e$ to $T_i^+$ and to $T_{j+1}^-$
 6:     end for
 7:     for all $I = [i..j] \in \{I_1^e, \dots, I_k^e\}$ do
 8:         if $e$ is in both $T_i^+$ and $T_i^-$ then
 9:             Remove $e$ from $T_i^+$ and from $T_i^-$
10:         end if
11:     end for
12: end for
13: return $T_i^+$, $T_i^-$ for all $i = 2, \dots, n-1$, and $T_1^+$, $T_n^-$,
            as well as the intervals $I_i^e$ for all edges $e$
```

Lemma 5.10.
Algorithm 5 returns the sets

$$T_i^+ = \{\{v,w\} \in E_G \mid P_{v,w} \textit{ contains } e_i \textit{ but not } e_{i-1}\} = S_i^+ \textit{ and}$$
$$T_i^- = \{\{v,w\} \in E_G \mid P_{v,w} \textit{ contains } e_{i-1} \textit{ but not } e_i\} = S_i^-$$

for $i = 2, \dots, n-1$, and

$$T_1^+ = \{\{v,w\} \in E_G \mid P_{v,w} \textit{ contains } e_1\} = E_G(\mathcal{C}(e_1)) \textit{ and}$$
$$T_n^- = \{\{v,w\} \in E_G \mid P_{v,w} \textit{ contains } e_{n-1}\} = E_G(\mathcal{C}(e_{n-1}))$$

and the intervals I_i^e in $O(m \log n)$ time.
We also get

$$\sum_{i=2}^{n-1}(|S_i^+| + |S_i^-|) = O(m \log n)$$

Proof.
Correctness
Let $I_1, \dots, I_k$ be the computed intervals from line 3 for the edge $e = \{v, w\}$. Let $I := \{j \in [1..n-1] \mid P_{v,w} \text{ contains } e_j\}$.
Then for $i = 2, \dots, n-1$

$$\begin{aligned} e \in S_i^+ &\iff i \in I \text{ and } i-1 \notin I \iff \exists j \in [1..k] : i \in I_j \text{ and } \forall l \in [1..k] : i-1 \notin I_l \\ &\iff \exists j \in [1..k] : i \text{ is the first number in } I_j \text{ and no interval contains } i-1 \\ e \in S_i^- &\iff i \notin I \text{ and } i-1 \in I \iff \forall l \in [1..k] : i \notin I_l \text{ and } \exists j \in [1..k] : i-1 \in I_j \\ &\iff \exists j \in [1..k] : i-1 \text{ is the last number in } I_j \text{ and no interval contains } i \end{aligned}$$

The first inner loop adds e to T_i^+ if i is the first number in some interval I_j, and adds e to T_i^- if i is the last number in some interval I_j.
This can, however, cause issues if an interval I_j starts immediately after another interval I_j ends. In this case, e would be wrongly added to both T_i^+ and T_i^- if $i-1$ is the last number in I_j and i is the first number in I_j.
To address this, the second inner loop removes any wrongly added edges.

Since all intervals have values between 1 and $n-1$, we get

$$\begin{aligned} P_{v,w} \text{ contains } e_1 &\iff 1 \in I \iff \exists j \in [1..k] : 1 \text{ is the first number in } I_j \\ &\iff e \in T_1^+ \\ P_{v,w} \text{ contains } e_{n-1} &\iff n-1 \in I \iff \exists j \in [1..k] : n-1 \text{ is the last number in } I_j \\ &\iff e \in T_n^- \end{aligned}$$

Time complexity
The computation of the $O(\log n)$ intervals can be done in $O(\log n)$ as explained by Lemma 5.9. With $\{I_1, \dots, I_k\}$ containing $O(\log n)$ intervals, the subsequent inner loops can be performed in $O(\log n)$ time. Consequently, the overall time complexity of Algorithm 5 is $O(m \log n)$.

Size of S_i^+ and S_i^-
As each edge is added to T_i^+ and T_i^- at most once per interval and since $T_i^+ = S_i^+$ and $T_i^- = S_i^-$, the each edge is in at most $O(\log n)$ of the sets S_i^+ and S_i^- for $i = 2, \dots, n-1$. Consequently

$$\sum_{i=2}^{n-1} (|S_i^+| + |S_i^-|) = O(m \log n)$$

□

The algorithm for finding the smallest cut that 1-respects T is as follows:

Algorithm 6 Find the smallest cut that 1-respects T

```
1: Order the tree-edges $e_1, \dots, e_{n-1}$ with Algorithm 4
2: Determine the sets $S_i^+$, $S_i^-$ and $E_G(\mathcal{C}(e_1))$ with Algorithm 5
3: Compute $w_G(\mathcal{C}(e_1)) = \sum_{e \in E_G(\mathcal{C}(e_1))} w_G(e)$
4: for $i = 2, \dots, n-1$ do
5:     Compute $\delta_i = \sum_{e \in S_i^+} w_G(e) - \sum_{e \in S_i^-} w_G(e)$ and $w_G(\mathcal{C}(e_i)) = w_G(\mathcal{C}(e_{i-1})) + \delta_i$
6: end for
7: return the smallest weight $w_{\min}$ and $\{e \in \{e_1, \dots, e_{n-1}\} \mid w_G(\mathcal{C}(e)) = w_{\min}\}$
```

Lemma 5.11 (Bhardwaj et al. [4]). *Algorithm 6 returns the crossing edges of all smallest 1-respecting cuts, as well as their weight in $O(m \log n)$ time.*

Proof. The weights of the cuts are computed correctly as by definition

$$w_G(\mathcal{C}(e_1)) = \sum_{e \in E_G(\mathcal{C}(e_1))} w_G(e)$$

and

$$\delta_i = w_G(\mathcal{C}(e_i)) - w_G(\mathcal{C}(e_{i-1})) = \sum_{e \in S_i^+} w_G(e) - \sum_{e \in S_i^-} w_G(e)$$

Algorithm 4 can be done in $O(n)$ time as stated by Lemma 5.5. According to Lemma 5.10, Algorithm 5 takes $O(m \log n)$ time and $\sum_{i=2}^{n-1}(|S_i^+|+|S_i^-|) = O(m \log n)$. Therefore, the for-loop takes $O(m \log n)$ time in total. □

5.2 Finding the smallest strictly 2-respecting cut

In this section, we want to extend the approach of the previous section to 2-respecting cuts. Let the tree-edges $e_1, \dots, e_{n-1}$ be ordered as in the section 5.1. Let further

$$S_i^+ = \{\{v, w\} \in E_G \mid P_{v,w} \text{ contains } e_i \text{ but not } e_{i-1}\} \text{ and}$$
$$S_i^- = \{\{v, w\} \in E_G \mid P_{v,w} \text{ contains } e_{i-1} \text{ but not } e_i\}$$

as above. We still want to iterate over the edges $e_1, \dots, e_{n-1}$ but we now also have to find a second edge e_j for every edge e_i that minimizes $w_G(\mathcal{C}(e_i, \cdot))$.

To explain the approach of Bhardwaj et al. [4] to do this efficiently, we start with the following definition:
For all edges e of G, $i = 2, \dots, n-1$, and $j = 1, \dots, n-1$, let

$$\delta_{i,j;e} = \begin{cases} w_G(e) & \text{if } e \text{ crosses } \mathcal{C}(e_i, e_j) \text{ but not } \mathcal{C}(e_{i-1}, e_j) \\ -w_G(e) & \text{if } e \text{ crosses } \mathcal{C}(e_{i-1}, e_j) \text{ but not } \mathcal{C}(e_i, e_j) \\ 0 & \text{otherwise} \end{cases}$$

Then, the following equality holds:

$$w_G(\mathcal{C}(e_i, e_j)) - w_G(\mathcal{C}(e_{i-1}, e_j)) = \sum_{e \in E_G} \delta_{i,j;e}$$

Note that we usually require $e_i \neq e_j$, since $\mathcal{C}(e_i, e_i)$ is not a valid cut. To make our definitions work, we assume that the imaginary cuts $\mathcal{C}(e_i, e_i)$ have no crossing edges and thus have weight 0. This assumption is justified by the following observation: Let $e = \{v, w\}$ be an edge of G and $e_i \neq e_j$ be tree-edges. According to Lemma 3.1

$$\begin{aligned} e \text{ crosses } \mathcal{C}(e_i, e_j) &\iff P_{v,w} \text{ contains an odd number of edges that cross } \mathcal{C}(e_i, e_j) \\ &\iff P_{v,w} \text{ contains either } e_i \text{ or } e_j \text{ but not both} \end{aligned}$$

When dealing with invalid cuts $\mathcal{C}(e_i, e_i)$, we can use the equivalency as a definition which means that $\mathcal{C}(e_i, e_i)$ has no crossing edges. As we will only encounter crossing edges in the context of this equivalency for the rest of this section, we can allow $e_i = e_j$ from now on.

The above observation further implies that for all i

$$\begin{aligned} e_j \text{ is not in } P_{v,w} &\iff (e \text{ crosses } \mathcal{C}(e_i, e_j) \iff P_{v,w} \text{ contains } e_i) \\ e_j \text{ is in } P_{v,w} &\iff (e \text{ crosses } \mathcal{C}(e_i, e_j) \iff P_{v,w} \text{ does not contain } e_i) \end{aligned}$$

To make efficient use of $\delta_{i,j;e}$, we show its following properties:

Lemma 5.12. *Let $e = \{v, w\}$ be an edge of G.
Then $\delta_{i,j;e} = 0$ if and only if $e \notin S_i^+ \cup S_i^-$.*

Proof. If e_j is in $P_{v,w}$, we get

$$\begin{aligned} \delta_{i,j;e} = 0 &\iff (e \text{ crosses } \mathcal{C}(e_i, e_j) \iff e \text{ crosses } \mathcal{C}(e_{i-1}, e_j)) \\ &\iff (P_{v,w} \text{ does not contain } e_i \iff P_{v,w} \text{ does not contain } e_{1-1}) \\ &\iff e \notin S_i^+ \cup S_i^- \end{aligned}$$

If e_j is not in $P_{v,w}$, we get

$$\begin{aligned} \delta_{i,j;e} = 0 &\iff (e \text{ crosses } \mathcal{C}(e_i, e_j) \iff e \text{ crosses } \mathcal{C}(e_{i-1}, e_j)) \\ &\iff (P_{v,w} \text{ contains } e_i \iff P_{v,w} \text{ contains } e_{1-1}) \\ &\iff e \notin S_i^+ \cup S_i^- \end{aligned}$$

□

We further show the following:

Lemma 5.13. *Let $e = \{v, w\} \in S_i^+ \cup S_i^-$.
If $e \in S_i^+$, then*

$$\delta_{i,j;e} = \begin{cases} -w_G(e) & \text{if } e_j \text{ is in } P_{v,w} \\ w_G(e) & \text{if } e_j \text{ is not in } P_{v,w} \end{cases}$$

If $e \in S_i^-$, then

$$\delta_{i,j;e} = \begin{cases} w_G(e) & \text{if } e_j \text{ is in } P_{v,w} \\ -w_G(e) & \text{if } e_j \text{ is not in } P_{v,w} \end{cases}$$

Proof. Since $e \in S_i^+ \cup S_i^-$, we get $\delta_{i,j;e} \neq 0$ (Lemma 5.12).
If $e \in S_i^+$, then $P_{v,w}$ contains e_i but not e_{i-1}. Hence

$$\begin{aligned} \delta_{i,j;e} = w_G(e) &\iff e \text{ crosses } \mathcal{C}(e_i, e_j) \text{ but not } \mathcal{C}(e_{i-1}, e_j) \\ &\iff (e \text{ crosses } \mathcal{C}(e_k, e_j) \iff P_{v,w} \text{ contains } e_k) \text{ for } k = i-1, i \\ &\iff e_j \text{ is not in } P_{v,w} \end{aligned}$$

If $e \in S_i^-$, then $P_{v,w}$ contains e_{i-1} but not e_i. Hence

$$\begin{aligned} \delta_{i,j;e} = -w_G(e) &\iff e \text{ crosses } \mathcal{C}(e_{i-1}, e_j) \text{ but not } \mathcal{C}(e_i, e_j) \\ &\iff (e \text{ crosses } \mathcal{C}(e_k, e_j) \iff P_{v,w} \text{ contains } e_k) \text{ for } k = i-1, i \\ &\iff e_j \text{ is not in } P_{v,w} \end{aligned}$$

□

Let $R_1^+ = \{\{v,w\} \in E_G \mid P_{v,w} \text{ contains } e_1\}$. Using Lemma 5.12 and Lemma 5.13, we can give the first draft of the algorithm:

Algorithm 7 Find the smallest cut that strictly 2-respects T

```
 1: Order the tree-edges $e_1, \dots, e_{n-1}$ with Algorithm 4
 2: Compute the sets $S_i^+$, $S_i^-$ for $i = 2, \dots, n-1$ and $R_1^+$ with Algorithm 5
 3: Initialize $w_j \leftarrow 0$ as weights of $e_j$ for all $j = 1, \dots, n-1$
                  ▷ the weight $w_j$ is independent from $e_j$'s weight $w_G(e_j)$ in $G$
 4: for all edges $e = \{v,w\}$ of $G$ do
 5:     if $e \in R_1^+$ then
 6:         Update $w_j \leftarrow w_j + w_G(e)$ for all edges $e_j$ not in $P_{v,w}$
 7:     else
 8:         Update $w_j \leftarrow w_j + w_G(e)$ for all edges $e_j$ in $P_{v,w}$
 9:     end if
10: end for
11: Find the smallest $w_j$ with $j \neq 1$ and store $c_1 \leftarrow (e_1, e_j)$ and $y_1 \leftarrow w_j$
12: for $i = 2, \dots, n-1$ do
13:     for all $e = \{v,w\} \in S_i^+$ do
14:         Update $w_j \leftarrow w_j - w_G(e)$ for all edges $e_j$ in $P_{v,w}$
15:         Update $w_j \leftarrow w_j + w_G(e)$ for all edges $e_j$ not in $P_{v,w}$
16:     end for
17:     for all $e = \{v,w\} \in S_i^-$ do
18:         Update $w_j \leftarrow w_j + w_G(e)$ for all edges $e_j$ in $P_{v,w}$
19:         Update $w_j \leftarrow w_j - w_G(e)$ for all edges $e_j$ not in $P_{v,w}$
20:     end for
21:     Find the smallest $w_j$ with $j \neq i$ and store $c_i \leftarrow (e_i, e_j)$ and $y_i \leftarrow w_j$
22: end for
23: Find the smallest $y_i$ and let $c := c_i$ and $y := y_i$
24: return $c$, $y$
```

Lemma 5.14 (Bhardwaj et al. [4]). *Algorithm 7 returns the weight of the smallest strictly 2-respecting cut, as well as its crossing edges in T.*

Proof. According to Lemma 5.10, Algorithm 5 returns the sets $T_i^+ = S_i^+$, $T_i^- = S_i^-$ for $i = 2, \dots, n-1$ and $T_1^+ = R_1^+$.

We first prove that $w_j = w_G(\mathcal{C}(e_1, e_j))$ for all $j = 1, \dots, n-1$ after the first for-loop: The weight of an edge $e = \{v,w\}$ gets added to w_j if and only if $P_{v,w}$ contains e_1 ($\iff e \in R_1^+$) but not e_j or if $P_{v,w}$ doesn't contain e_1 ($\iff e \notin R_1^+$) but e_j. As e crosses $\mathcal{C}(e_1, e_j)$ if and only if $P_{v,w}$ contains either e_1 or e_j but not both, the weight of an edge e gets added to w_j if and only if e crosses $\mathcal{C}(e_1, e_j)$. Thus $w_j = w_G(\mathcal{C}(e_1, e_j))$.
Furthermore, $y_1 = \min_{j=1,\dots,n-1} w_j = \min_{j=1,\dots,n-1} w_G(\mathcal{C}(e_1, e_j))$ and $c_1 = (e_1, e_j)$ with $w_G(\mathcal{C}(e_1, e_j)) = y_1$ after the first for-loop.

Next, we prove that $w_j = w_G(\mathcal{C}(e_i, e_j))$ for all $j = 1, \dots, n-1$ at the end of iteration i of the second for-loop by induction over $i = 2, \dots, n-1$:

Assume $w_j = w_G(\mathcal{C}(e_{i-1}, e_j))$ (induction hypothesis). Lemma 5.12 yields

$$w_G(\mathcal{C}(e_i, e_j)) - w_G(\mathcal{C}(e_{i-1}, e_j)) = \sum_{e \in E_G} \delta_{i,j;e} = \sum_{e \in S_i^+ \cup S_i^-} \delta_{i,j;e} = \sum_{e \in S_i^+} \delta_{i,j;e} + \sum_{e \in S_i^-} \delta_{i,j;e}$$

According to Lemma 5.13, we get

$$\delta_{i,j;e} = \begin{cases} -w_G(e) & \text{if } e_j \text{ is in } P_{v,w} \\ w_G(e) & \text{if } e_j \text{ is not in } P_{v,w} \end{cases}$$

for $e = \{v, w\} \in S_i^+$ and thus

$$w_j = w_G(\mathcal{C}(e_{i-1}, e_j)) + \sum_{e \in S_i^+} \delta_{i,j;e}$$

after the first inner loop. Since

$$\delta_{i,j;e} = \begin{cases} w_G(e) & \text{if } e_j \text{ is in } P_{v,w} \\ -w_G(e) & \text{if } e_j \text{ is not in } P_{v,w} \end{cases}$$

for $e = \{v, w\} \in S_i^-$, we get

$$w_j = w_G(\mathcal{C}(e_{i-1}, e_j)) + \sum_{e \in S_i^+} \delta_{i,j;e} + \sum_{e \in S_i^-} \delta_{i,j;e} = w_G(\mathcal{C}(e_i, e_j))$$

after the second inner loop.
Furthermore, $y_i = \min_{j=1,\dots,n-1} w_j = \min_{j=1,\dots,n-1} w_G(\mathcal{C}(e_i, e_j))$ and $c_i = (e_i, e_j)$ with $w_G(\mathcal{C}(e_i, e_j)) = y_i$ at the end of iteration i of the outer-loop.

Thus $y = \min_{i=1,\dots,n-1} y_i = \min_{i,j=1,\dots,n-1} w_G(\mathcal{C}(e_i, e_j))$ is the weight of the smallest 2-respecting cut of T. Furthermore, the edges $c_i = (e_i, e_j)$ with $w_G(\mathcal{C}(e_i, e_j)) = y$ are the crossing edges of a smallest 2-respecting cut of T. □

While Algorithm 7 returns the desired result, it doesn't run in near-linear time. Namely, the updates to the weights w_j and the search for the minimum w_j (with $i \neq j$) are problematic, since they need $O(n)$ time each. To speed this up, we store the w_j in a data structure that allowes us to perform these computations in poly-logarithmic time.

Lemma 5.15 (Bhardwaj et al. [4]). *There is a data structure that can perform the following operations in $O(\log^2 n)$ time on a weighted tree:*

- *PathAdd(u, v, x) := Add weight x to the all edges on the path $P_{u,v}$*
- *NonPathAdd(u, v, x) := Add weight x to the all edges that are not on the path $P_{u,v}$*
- *GetMinimumExcept(i) := Retrieve (e_j, w_j) for the edge e_j, $i \neq j$ with minimum weight w_j among all edges except e_i.*

Bhardwaj et al. [4] propose two different structures: The first is a top tree, which can achieve the specified operations of Lemma 5.15 in $O(\log n)$ time [1, 4]. Top trees are, however, difficult to implement. To keep it simpler, we opt for the second proposal by Bhardwaj et al. [4] — a segment tree.

Lemma 5.16 (Berg et al. [3]). *Let L be an ordered list of real numbers. Then there is a data structure that can perform the following operations in $O(\log n)$ time on L:*

- *IntervalAdd(l, r, x) := Add x to all elements in the range $[l..r]$*
- *GetMinimum() := Get the value and the index of an element with the smallest value.*

We can use this data structure to construct the data structure from Lemma 5.15.

Proof of Lemma 5.15.
We use the data structure from Lemma 5.16 on the ordered list $L = [w_1, \dots, w_{n-1}]$.

As explained in Lemma 5.9, the edges of every path consist of at most $O(\log n)$ intervals if the edges were ordered by Algorithm 4. Thus, we can perform PathAdd(u, v, x) in $O(\log^2 n)$ time by using IntervalAdd(a_i, b_i, x) on the $O(\log n)$ intervals $I_1 = [a_1..b_1], \dots, I_k = [a_k..b_k]$ of the path $P_{u,v}$.

We can perform NonPathAdd(u, v, x) by keeping a global value g and performing $g \leftarrow g + x$ and PathAdd$(u, v, -x)$.

To perform GetMinimumExcept(i), we call

$$\text{IntervalAdd}(i, i, W),\ \text{GetMinimum}(),\ \text{IntervalAdd}(i, i, -W)$$

for $W = \sum_{e \in E_G} w_G(e) + 1$. As

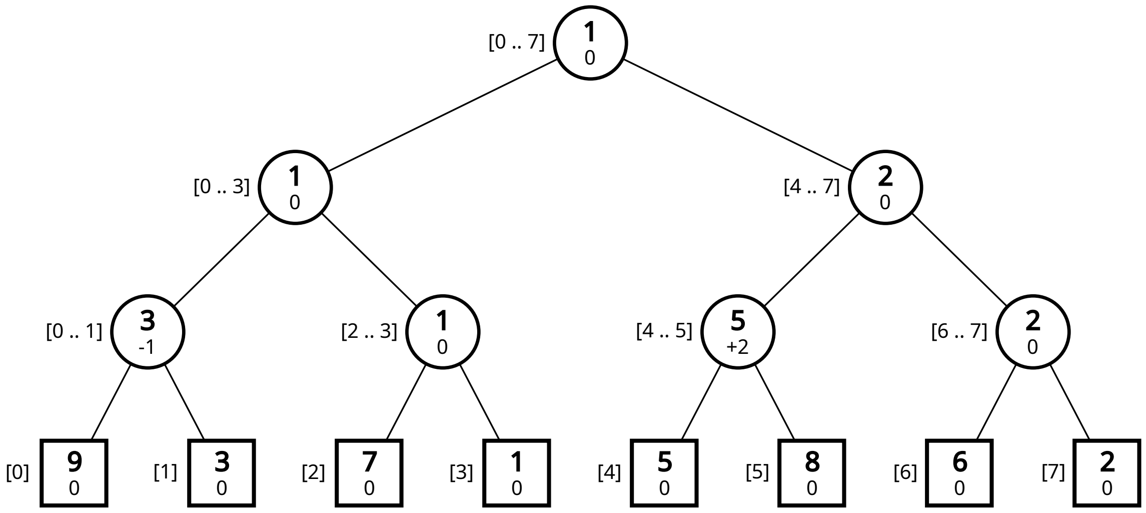
$$w_G(\mathcal{C}) = \sum_{e \in E_G(\mathcal{C})} w_G(e) \leq \sum_{e \in E_G} w_G(e) < W$$

for all cuts $\mathcal{C}$, GetMinimum() will effectively exclude item i. If i and w are the index and value returned by GetMinimum(), we return $(e_i, w + g)$. □

To prove Lemma 5.16, we introduce and explain segment trees, the invention of which is credited to Berg et al. [3]. A segment tree is an augmented binary search tree that is structured as follows:
Each node within the tree corresponds to a segment of the list and stores the minimum value within that segment. While the root node represents the entire list, the leaf nodes represent the elements of the list (list-elements = segments of size 1). The leaf-nodes are thereby ordered as in the list that the tree represents, i.e. the leftmost leaf represents the first element of the list, the second leftmost leaf represents the second element, and so on. The relationship between an internal node and its two children is defined as follows:
Consider a node responsible for elements with indices in the range $[a..b]$, where $a < b$, and let $m := \lfloor (a + b)/2 \rfloor$ represent the midpoint of the interval. The left child of this node is responsible for the range $[a..m]$, and the right child is responsible for $[m + 1..b]$.
A node that manages only a single element is referred to as a leaf node. This hierarchical structure ensures that the segment tree of a list containing N elements has a depth of $O(\log N)$. This is the case because, at each level of depth, the size of the intervals for which the nodes are responsible is halved.

To perform the operation $IntervalAdd$ in $O(\log n)$ time, we further need to equip each node in the tree with a lazy-value. Lazy-values store pending updates for the elements in the node's range: If all elements in the node's range are supposed to be incremented by a value v, we instead add v to the lazy value of the node. The actual addition of v to the elements is postponed until the respective values are needed, namely, until the node is accessed or read.

Figure 5.3: Segment tree for the list $[9, 3, 7, 1, 5, 8, 6, 2]$ with pending updates for the intervals $[0..1]$ and $[4..5]$



Lemma 5.17. *A segment tree representing a list of length N can be initialized from the list in $O(N)$ time.*

Proof. To initialize the segment tree, we begin by assigning the leaf-nodes the corresponding elements of the list. This can be done in $O(N)$ time.
For each internal node n, let $l(n)$ and $r(n)$ be its left and right child respectively. We compute the minimum values for the internal nodes of the segment tree through a postorder traversal: For each internal node n, we first compute the minimums of its left and right child. Then, the minimum value of n is the smaller of the two minimums.
Every node only needs a constant time to compute its minimum. This is because the minimum of its children has already been computed and stored during the traversal. Since the tree has N leaf nodes and every internal node has exactly two children, the tree has at most $(1 + \frac{1}{2} + \frac{1}{4} + \dots)N = 2N$ nodes. Therefore, the overall time complexity of the postorder traversal is $O(N)$. □

The operation $IntervalAdd$ can be implemented as follows:

Algorithm 8 *IntervalAdd*(s, t, x)

For a node n, we define $[a_n..b_n]$ as its range, n_m as its minimum, n_l as its lazy-value and $l(n), r(n)$ as its left and right children.

```
1:  procedure IntervalAddRec($n, s, t, x$)        ▷ $n$ node, $s, t$ indices, $x$ real-value
2:      if $n_l \neq 0$ then                          ▷ There is a pending update for $n$
3:          Update $n_m \leftarrow n_m + n_l$
4:          if $n$ is not leaf node then
5:              Add $n_l$ to $l(n)_l$ and $r(n)_l$     ▷ propagate $n_l$ to lazy-values of children
6:          end if
7:          Set $n_l \leftarrow 0$
8:      end if
9:      If $[s..t] \cap [a_n..b_n] = \emptyset$, abort!
10:     if $[a_n..b_n] \subseteq [s..t]$ then         ▷ query range completely covers $n$s range
11:         Update $n_m \leftarrow n_m + x$
12:         if $n$ is not leaf node then
13:             Add $x$ to $l(n)_l$ and to $r(n)_l$    ▷ propagate $x$ to lazy-values of children
14:         end if
15:     else
16:         Perform IntervalAddRec($l(n), s, t, x$) and
                IntervalAddRec($r(n), s, t, x$)
17:         Update $n_m \leftarrow \min\{l(n)_m, r(n)_m\}$
18:     end if
19: end procedure
20: Perform IntervalAddRec($n_0, s, t, x$) with $n_0$ being the root-vertex
```

Lemma 5.18. *Let T_S be a segment tree representing a list of length N. Then IntervalAdd(s, t, x) from Algorithm 8 adds the value x to all values in the range $[s..t]$ with some of the additions being postponed using lazy values. Algorithm 8 runs in $O(\log N)$ time.*

Proof.

Correctness

We prove that after IntervalAddRec(n, s, t, x) is performed on any node n, x is added to all elements in the range $[s..t] \cap [a_n..b_n]$ while allowing potential postponement with lazy-values. Additionally, n_m contains the minimum of all elements in the range $[a_n..b_n]$ with no pending updates ($n_l = 0$).

The first if-block ensures that n_m is the correct minimum of the elements in the range $[a_n..b_n]$ before adding x. If n has a non-zero lazy-value n_l, all elements in $[a_n..b_n]$ still need to be incremented by n_l. We add n_l to n's minimum n_l and, once again, postpone the updates of n's children by adding n_l to their respective lazy-values.

We now prove that x gets added to the elements in $[s..t] \cap [a_n..b_n]$ by induction:

Base case: If $[s..t] \cap [a_n..b_n] = \emptyset$, then there is no overlap between the query-range and the current node's range, so we abort. Now, assume $[a_n..b_n] \subseteq [s..t]$. Since $[a_n..b_n] \cap [s..t] = [a_n..b_n]$, x is to be added to all elements in the range that n represents. The new minimum of these elements is $n_m + x$ and we can postpone the updates of n's children by adding x to their lazy-values.

Note that the base case applies to all leaf nodes n, since $a_n = b_n =: i$ for all leaf nodes n, meaning $[a_n..b_n] = \{i\}$. Thus, either

$$\underbrace{[a_n..b_n]}_{=\{i\}} \cap [s..t] = \emptyset \text{ if } i \notin [s..t] \text{ or}$$
$$\underbrace{[a_n..b_n]}_{=\{i\}} \subseteq [s..t] \qquad \text{if } i \in [s..t]$$

Inductive Step: Let n be an internal node with children $l(n), r(n)$, and $[s..t] \cap [a_n..b_n] \neq \emptyset$, as well as $[a_n..b_n] \not\subseteq [s..t]$. Further, let $m := \lfloor (a_n + b_n)/2 \rfloor$, meaning $[a_n..m]$ is the range of n's left child $l(n)$ and $[m + 1..b_n]$ is the range of n's right child $r(n)$. Assume that IntervalAddRec works on $l(n)$ and $r(n)$ as intended (induction hypothesis).
Then, IntervalAddRec(n, s, t, x) will perform IntervalAddRec on $l(n)$ and $r(n)$. After this, $l(n)_m$ will contain the minimum of the elements in $[a_n..m]$, and $r(n)_m$ will contain the minimum of the elements in $[m + 1..b_n]$. Therefore, the minimum of the elements in $[a_n..b_n]$ is $\min\{l(n)_m, r(n)_m\}$.

Since Algorithm 8 performs IntervalAddRec(n_0, s, t, x) for the root-vertex n_0 and the root-vertex represents the entire list, x gets added to all elements in the range $[s..t] \cap [a_{n_0}..b_{n_0}] = [s..t]$.

Runtime complexity
Let L_i be the number of nodes that the algorithm visits at depth i. We prove by induction that $L_i \leq 4$ for all depths i:
Base case $i = 0$: Since the root vertex is the only vertex with depth 0, we get $L_0 = 1$.
Induction step: Let $L_i \leq 4$ for some depth i.
Assume at first $L_i \leq 2$. As every node visits at most 2 of its children, we get $L_{i+1} \leq 4$.
Now, let $L_i = 3$ and n_1, n_2, n_3 be the nodes visited at depth i and let their respective ranges be $[a_1..b_1], \ldots, [a_3..b_3]$. Let the nodes be ordered from left to right, meaning $b_j < a_{j+1}$. If the range of one of the nodes does not cross the interval $[s..t]$ at all, then the node does not visit any of its children and we can reduce the case to $L_i \leq 2$. Thus, we can assume w.l.o.g. that the ranges $[a_1..b_1], \ldots, [a_3..b_3]$ all cross the interval. Hence $s \leq b_1 < a_2 \leq b_2 < a_3 \leq t$. Since $[s..t]$ completely covers n_2's range, n_2 doesn't visit any of its children. As only the children of n_1 and n_3 can get visited, we get $L_{i+1} \leq 4$.
The case $L_i = 4$ with nodes $n_1, \ldots, n_4$ is analogous to the case $L_i = 3$ with $[s..t]$ completely covering the range of the nodes n_2 and n_3.

Let d be the depth of the tree. Since $d = O(\log n)$, the algorithm visits at most $\sum_{i=0}^{d-1} L_i \leq 4d = O(\log n)$ nodes. □

The operation *GetMinimum*() can also be implemented to run in $O(\log n)$ time.

Lemma 5.19. *In a segment tree, the smallest value of all its elements can be found in $O(1)$ time. The index of the element with the smallest value can be found in $O(\log n)$ time.*

Proof. The smallest value is the minimum $(n_0)_m$ of the root vertex n_0 and can be read in $O(1)$ time.
Finding the index of the element with the smallest value can be done as follows: Start at the root-vertex and update its children's minimums if they have pending updates. This can be done as in the first if-block of Algorithm 8. Then, travel down to the child with the smaller minimum and repeat this process until a leaf-node is reached. Return the index of the element that this leaf-node is responsible for. This can be done in $O(\log n)$ time, since the tree has depth $O(\log n)$. □

This proves the existence of our desired data structure.

Proof of Lemma 5.16. Follows directly from Lemma 5.18 and Lemma 5.19. □

The algorithm for finding the smallest cut that strictly 2-respects T is as follows:

Algorithm 9 Find the smallest cut that strictly 2-respects T

```
1: Order the tree-edges $e_1, \dots, e_{n-1}$ with Algorithm 4
2: Compute the sets $S_i^+$, $S_i^-$ for $i = 2, \dots, n-1$, the set $R_1^+$, and
        the intervals $I_i^e$ of $P_{v,w}$ for every edge $e = \{v, w\}$ with Algorithm 5
3: Initialize the segment tree on the list $L = [0, \dots, 0]$ with length $n-1$,
        representing the weights $w_1, \dots, w_{n-1}$.
4: for all edges $e = \{v, w\}$ of $G$ do
5:     if $e \in R_1^+$ then
6:         Perform NonPathAdd$(v, w, w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
7:     else
8:         Perform PathAdd$(v, w, w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
9:     end if
10: end for
11: Use GetMinimumExcept$(1)$ to find the smallest $w_j$ with $j \neq 1$ and
        store $c_1 \leftarrow (e_1, e_j)$ and $y_1 \leftarrow w_j$
12: for $i = 2, \dots, n-1$ do
13:     for all $e = \{v, w\} \in S_i^+$ do
14:         Perform PathAdd$(v, w, -w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
15:         Perform NonPathAdd$(v, w, w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
16:     end for
17:     for all $e = \{v, w\} \in S_i^-$ do
18:         Perform PathAdd$(v, w, w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
19:         Perform NonPathAdd$(v, w, -w_G(e))$ using the intervals $I_i^e$ of $P_{v,w}$
20:     end for
21:     Use GetMinimumExcept$(i)$ to find the smallest $w_j$ with $j \neq i$ and
            store $c_i \leftarrow (e_i, e_j)$ and $y_i \leftarrow w_j$
22: end for
23: Find the smallest $y_i$ and let $c := c_i$ and $y := y_i$
24: return $(c, y)$
```

Lemma 5.20 (Bhardwaj et al. [4]). *Algorithm 9 returns the weight of the smallest strictly 2-respecting cut, as well as its crossing edges in T in $O(m \log^3 n)$ time.*

Proof. Correctness follows from Lemma 5.14.

Initializing the segment tree takes $O(m)$ time according to Lemma 5.17.
As stated by Lemma 5.15, the operations PathAdd, NonPathAdd, and GetMinimumExcept take $O(\log^2 n)$ time each. The first loop takes $O(m \log^2 n)$ time, since it runs for $O(m)$ iterations and each iteration performs PathAdd or NonPathAdd. GetMinimumExcept(i) gets called exactly once for every $i = 1, \ldots, n-1$ taking $O(n \log^2 n)$ time in total.
The operations PathAdd and NonPathAdd get called $\sum_{i=2}^{n-1}(|S_i^+| + |S_i^-|)$ times each during the second outer loop. As $\sum_{i=2}^{n-1}(|S_i^+| + |S_i^-|) = O(m \log n)$ (Lemma 5.10), these operations take $O(m \log^3 n)$ time in total. □

5.3 Putting it all together

Putting it all together, we can obtain a minimum cut $\mathcal{C}$ of a weighted graph G in $O(m \log^4 n)$ time with high probability as follows:
First, we run Algorithm 3 on G. Algorithm 3 returns a set of $\Theta(\log n)$ spanning trees of G of which $\mathcal{C}$ 2-respects at least one with high probability. Thus, if we run Algorithm 6 and Algorithm 9 on all $\Theta(\log n)$ trees, we find a minimum cut with high probability.
Algorithm 3 has a runtime of $O(m \log^3 n)$ (Lemma 4.17) and the Algorithms 6 and 9 have runtimes of $O(m \log n)$ and $O(m \log^3 n)$ respectively (Lemma 5.11 and Lemma 5.20). Since we run Algorithm 6 and 9 on $\Theta(\log n)$ trees, we get a total runtime of $O(m \log^4 n)$.

This runtime is slightly longer than the one achieved by Bhardwaj et al. [4] who attained a total runtime of $O(m \log^3 n)$. The reason for this is, that they opted to use a top-tree instead of a segment-tree. This makes the version explained in this thesis easier to implement but their proposed version slightly faster.

6 Conclusion

In this thesis, we presented the simplified minimum cut algorithm from Bhardwaj et al. [4].
In chapter 3, we explained the general idea behind the algorithm: that for every minimum cut $\mathcal{C}$ of a weighted graph G, there is a spanning tree T of G that $\mathcal{C}$ 2-respects, and that the minimum cut can be found by determining the smallest cut that 2-respects T.
In chapter 4, we showed how we can find a set of spanning trees of which the minimum cut 2-respects at least one by approximating the weighted graph as a multigraph and using a tree packing algorithm for multigraphs given by Thorup and Karger [16]. We further explained how this approach can be made efficient by employing a random sampling technique on the simple edges of the multigraph and capping the edge-multiplicities to a sufficiently large upper bound.
In chapter 5, we showed how we can determine the smallest cut that 2-respects a given spanning tree of the graph. We proved that the weights of the 2-respecting cuts can be computed efficiently by ordering the cuts using heavy-light decomposition and computing the weight differences between adjacent cuts. We also showed how a segment tree data structure can be used to find the smallest strictly 2-respecting cut and in $O(m \log^3 n)$ time despite there being $\Theta(n^2)$ such cuts. In particular, we explained how and why the heavy-light decomposition and the segment tree data structure work - an explanation that was omitted in the original paper.

A remaining question for future research is, whether the simplification from Bhardwaj et al. [4] of the algorithm from Karger [8] can be effectively applied to other algorithms. A promising candidate for the application of such simplification appears to be an algorithm presented by Karger and Panigrahi [10]. This algorithm is designed to construct a cactus representation of the minimum cuts in a weighted graph, providing a representation from which *all* minimum cuts can be easily derived. Notably, this algorithm is based on the approach of Karger [8] and inherits its complexity.

Bibliography

[1] Stephen Alstrup et al. *Maintaining Information in Fully-Dynamic Trees with Top Trees.* 2003. arXiv: `cs/0310065 [cs.DS]`.

[2] David L. Applegate et al. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics).* USA: Princeton University Press, 2007. ISBN: 0691129932.

[3] Mark Berg et al. *More Geometric Data Structures.* In: *Computational Geometry: Algorithms and Applications.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 219–241. ISBN: 978-3-540-77974-2. DOI: `10.1007/978-3-540-77974-2_10`.

[4] Nalin Bhardwaj, Antonio Molina Lovett, and Bryce Sandlund. *A Simple Algorithm for Minimum Cuts in Near-Linear Time.* 2020. arXiv: `1908.11829 [cs.DS]`.

[5] Rodrigo A. Botafogo. *Cluster Analysis for Hypertext Systems.* In: *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.* SIGIR '93. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1993, pp. 116–125. ISBN: 0897916050. DOI: `10.1145/160688.160704`.

[6] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. *Minimum Cut in $O(m \log^2 n)$ Time.* 2020. arXiv: `1911.01145 [cs.DS]`.

[7] David R. Karger. *A Fully Polynomial Randomized Approximation Scheme for the All Terminal Network Reliability Problem.* 1998. arXiv: `cs/9809012 [cs.DS]`.

[8] David R. Karger. *Minimum Cuts in Near-Linear Time.* 1998. arXiv: `cs/9812007 [cs.DS]`.

[9] David R. Karger. *Random Sampling in Cut, Flow, and Network Design Problems.* In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing.* STOC '94. Montreal, Quebec, Canada: Association for Computing Machinery, 1994, pp. 648–657. ISBN: 0897916638. DOI: `10.1145/195058.195422`.

[10] David R. Karger and Debmalya Panigrahi. *A Near-Linear Time Algorithm for Constructing a Cactus Representation of Minimum Cuts.* In: *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '09. New York, New York: Society for Industrial and Applied Mathematics, 2009, pp. 246–255.

[11] Jon Kleinberg and Éva Tardos. *The Minimum Spanning Tree Problem.* In: *Algorithm Design.* Boston, MA: Pearson/Addison-Wesley, 2006, pp. 142–151. ISBN: 978-0-321-29535-4.

[12] Joseph B. Kruskal. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* In: *Proceedings of the American Mathematical Society* 7.1, pp. 48–50, pp. 48–50. ISSN: 00029939, 10886826.

[13] C. St.J. A. Nash-Williams. *Edge-Disjoint Spanning Trees of Finite Graphs.* In: *Journal of the London Mathematical Society* s1-36.1, pp. 445–450, pp. 445–450. DOI: `https://doi.org/10.1112/jlms/s1-36.1.445`.

[14] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. *Fast Approximation Algorithms for Fractional Packing and Covering Problems.* In: *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991.* IEEE Computer Society, 1991, pp. 495–504. DOI: `10.1109/SFCS.1991.185411`.

[15] Daniel D. Sleator and Robert Endre Tarjan. *A data structure for dynamic trees.* In: *Journal of Computer and System Sciences* 26.3, pp. 362–391, pp. 362–391. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(83)90006-5`.

[16] Mikkel Thorup and David R Karger. *Dynamic Graph Algorithms with Applications.* In: *Scandinavian Workshop on Algorithm Theory.* 2000.

[17] Neal E. Young. *Randomized Rounding without Solving the Linear Program.* In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms.* SODA '95. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 1995, pp. 170–178. ISBN: 0898713498.

A Kruskal's algorithm for minimum (maximum) spanning trees

Let G be a connected graph and let its edges be weighted by some function $w(\cdot)$. The minimum spanning tree of G with respect to $w(\cdot)$ can be found with an algorithm from Kruskal [12]. A more detailed explanation of the algorithm, in particular with regards to an efficient implementation, is given by Kleinberg and Tardos [11].

Algorithm 10 Obtain a minimum spanning tree

```
1: Initialize the set of tree edges as $E_T \leftarrow \emptyset$.
2: Sort the edges of the graph by weight in ascending order
3: for all edges $e$ of $G$ ordered by weight $w(e)$ do
4:     if adding $e$ to $E_T$ would not create a cycle then
5:         Add $e$ to $E_T$
6:     end if
7: end for
8: return $E_T$
```

A maximum spanning tree can by found by sorting the edges in descending order.

Lemma A.1 (Kruskal [12], Kleinberg and Tardos [11])**.** *Algorithm 10 returns the edges of a minimum spanning tree of G with respect to $w(\cdot)$.*

Proof. Let T be the graph with the same vertices as G and with its edges being the returned edges E_T. By construction, T cannot contain a cycle since no edge is added if doing so would create a cycle.
To reach a contradiction, assume T is not connected and let X, Y be the vertex-sets of two disconnected components of T. Since G is connected, G must contain at least one edge with one endpoint in X and one endpoint in Y. But the first encountered edge that connects X and Y would have been added to E_T, contradiction! Thus, T is a spanning tree of G.

We prove the minimality by showing the following: At every stage of the algorithm, there exists a minimum spanning tree S of G that contains all edges in E_T and none of the edges that have been rejected because adding them would create a cycle. We prove the statement by induction over the number of iterations of the for-loop $N \in \mathbb{N}_0$:

Base case $N = 0$
The statement is clearly true before the start of the loop. Any minimum spanning

tree S fulfills the requirements.

Induction step
Assume that there is a spanning tree S that fulfills the requirements after N iterations of the loop. We prove that such a spanning tree must also exist after $N + 1$ iterations. Let E_T be the current set of edges and $e = \{v, w\}$ be the next edge that the algorithm inspects.

- Assume adding e to E_T would create a cycle. Then the algorithm rejects e. Since S contains all edges in E_T, S cannot contain e. Otherwise, S would contain a cycle.

- Assume adding e to E_T would not create a cycle. If e is in S, then S also fulfills the requirements for $E_T \cup \{e\}$.

  Now assume that S does not contain $e = \{v, w\}$. Since S is connected, there must be a path $P = \langle e_1, \ldots, e_k \rangle$ from w to v in S. Thus, adding e to S would create a cycle $C = \langle e_1, \ldots, e_k, e \rangle$ in S. Since adding e to E_T does not create a cycle, there must be some edge e_i of C that is not contained in E_T. Therefore, $S + e - e_i$ contains all edges in $E_T \cup \{e\}$. Since removing e_i brakes the cycle C, $S + e - e_i$ is spanning tree of G.

  We still need to prove that $S - e_i + e$ is also a minimum spanning tree of G. If e_i was rejected by the algorithm, S could not contain e_i either. Thus, e_i has not jet been examined by the algorithm and therefore $w(e_i) \geq w(e)$. Hence, the weight of $S - e_i + e$ is at most as large as the weight of S. Since S has minimum weight, $S - e_i + e$ must also have minimum weight.

$\square$

For this algorithm to run efficiently, we need a fast method to determine whether adding an edge would result in a cycle or not. Note that adding an edge $e = \{v, w\}$ creates a cycle if and only if v and w are already connected: If there already is a path $P = \langle e_1, \ldots, e_k \rangle$ from w to v then $C = \langle e_1, \ldots, e_k, e \rangle$ would form a cycle. On the other hand, if $C = \langle e_1, \ldots, e_k, e \rangle$ is a cycle then $P = \langle e_1, \ldots, e_k \rangle$ is a path from w to v.
Thus, we need a fast way to check, whether two vertices are connected. This can be accomplished employing a union-find data structure on the set of vertices as proposed by Kleinberg and Tardos [11].
A union-find data structure manages a collection of disjoint sets. Every set X has one element $x \in X$ assigned as a representative. The data structure supports the following operations:

- $\textsc{Init}(A)$:= Create a set $\{a\}$ with representative a for every $a \in A$

- $\textsc{Find}(x)$:= Find the representative of the set that contains x

- $\textsc{Union}(x, y)$:= Replace the set X containing x and the set Y containing y with their union $X \cup Y$. If r is the representative of X and s is the representative of Y, then the representative of $X \cup Y$ is either r or s.

There are several ways to implement such a data structure. We use the version that Kleinberg and Tardos [11] propose. This is not the most efficient version, but it is fast enough for our purposes.
We store every set as a tree. Every node of the tree stores one element of the set and its parent node. The root node of the tree is the representative of the set and further stores the size of the set. The operations are implemented as follows:

- $\textsc{Init}(A)$
  For every $a \in A$, initialize a tree of size 1 that only consists of the node a.

- $\textsc{Find}(x)$
  If x is the root of its tree, return x. Otherwise return $\textsc{Find}(x.\mathit{parent})$.

- $\textsc{Union}(x, y)$
  Compute $r \leftarrow \textsc{Find}(x)$ and $s \leftarrow \textsc{Find}(y)$. Ensure that the size of r's set is bigger or equal to the size of s's set, otherwise swap r and s. Set $s.\mathit{parent} \leftarrow r$ (make s a child of r, r is the representative of the union) and update
  $r.\mathit{size} \leftarrow r.\mathit{size} + s.\mathit{size}$ (r now stores the size of the union).

To prove the efficiency of this implementation, we make use of the following lemma:

Lemma A.2. *For any set X stored in the union-find data structure, let T_X be its tree, s_X be its size, r_X be its representative, and d_X be the depth of T_X, where the depth of T_X denotes length of the longest path from the root of T_X to one of its leafs. Then $s_X \geq 2^{d_X}$ for all sets X.*

Proof. The statement is true after the initialization, since every set has size 1 and every tree has depth 0. Hence $s_X = 1 = 2^0 = 2^{d_X}$ for all sets X.
It stays true after the unionization of two arbitrary sets X and Y for the following reason: Assume w.l.o.g. $s_X \geq s_Y$. Then r_Y becomes a child of r_X. Thus, the resulting tree of $X \cup Y$ has depth $d_{X\cup Y} = \max\{d_X, d_Y + 1\}$.
If $d_Y \geq d_X$, then $d_{X\cup Y} = d_Y + 1$ and $s_{X\cup Y} = s_X + s_Y \geq 2s_Y \geq 2^{d_Y+1} = 2^{d_{X\cup Y}}$.
If $d_Y < d_X$, then $d_{X\cup Y} = d_X$ and $s_{X\cup Y} = s_X + s_Y \geq s_X \geq 2^{d_X} = 2^{d_{X\cup Y}}$ □

Thus, we get ...

Lemma A.3. *For a set A with size $|A|$, the operation $\textsc{Init}(A)$ takes $O(|A|)$ time and the operations $\textsc{Find}(x)$ and $\textsc{Union}(x, y)$ take $O(\log |A|)$ time each.*

Proof. $\textsc{Init}(A)$ initializes $|A|$ many trees. Initializing a tree can be done in $O(1)$ time, since all trees only consist of one node.
$\textsc{Find}(x)$ (with $x \in X$) walks from x to the root of the tree T_X. Since the longest path to T_X's root has length d_X, $\textsc{Find}(x)$ traverses at most $d_X + 1$ many nodes (x included). According to Lemma A.2, $d_X \leq \log_2 s_X \leq \log_2 |A|$, meaning $\textsc{Find}(x)$ takes $O(\log |A|)$ time.
$\textsc{Union}(x, y)$ performs $\textsc{Find}(x)$ and $\textsc{Find}(y)$ which takes $O(\log |A|)$ time. Merging the smaller tree with the larger tree and computing the size of the union takes $O(1)$ time. □

We can use this result provide an efficient version of Kruskal's algorithm.

Algorithm 11 Obtain a minimum spanning tree using a union-find data structure

```
 1: Initialize the set of tree edges as $E_T \leftarrow \emptyset$
 2: Sort the edges of the graph by weight in ascending order
 3: Perform INIT($V$) on the set of vertices $V$
 4: for all edges $e = \{v, w\}$ of $G$ ordered by weight $w(e)$ do
 5:     if FIND($v$) $\neq$ FIND($w$) then
 6:         Add $e$ to $E_T$
 7:         Perform UNION($v, w$)
 8:     end if
 9: end for
10: return $E_T$
```

Lemma A.4 (Kruskal [12], Kleinberg and Tardos [11])**.** *Algorithm 11 returns the edges of a minimum spanning tree of G with respect to $w(\cdot)$ in $O(m \log m)$ time, where m is the number of edges in G.*

Proof.
Correctness
The algorithm uses the union-find data structure to keep track of vertices that are already connected to each other by some path: Two vertices are connected if they are part of the same set. Thus, we can check whether two vertices are connected by examining their representatives. Since every set has a unique representative, both vertices share the same set if and only if they have the same representative.
As explained above, adding an edge $e = \{v, w\}$ to E_T creates a cycle if and only if v and w are already connected. Thus, Algorithm 11 produces the same result as Algorithm 10 and the correctness follows from Lemma A.1.

Time complexity
Sorting the edges can be done in $O(m \log m)$ time with a standard sorting algorithm like heap sort.
Since the graph is connected, we get $|V| = O(m)$. Thus, INIT(V) takes $O(m)$ time and FIND(v), FIND(w), and UNION(v, w) take $O(\log m)$ time each. Since the loop iterates over m edges, the loop has a total runtime of $O(m \log m)$. □