The Min-Cost Flow Algorithm by Chen et al.

Bachelor Thesis of

Höfer Simon

At the Department of Informatics and Mathematics
Chair of Theoretical Computer Science

UNIVERSITÄT PASSAU

Reviewer: Prof. Dr. Ignaz Rutter

Time Period: 16th Mai 2023 – 21st August 2023

Statement of Authorship

Abstract

The minimum-cost flow problem is a classic optimization problem in graph/network theory. It describes a directed graph G where each vertex has a supply/demand that needs to be fulfilled by routing units of flow over the edges between them. Each edge has capacity constraints and associated costs. The goal is to find a flow on these edges that lies within the capacity constraint, satisfies all supply/demands of the vertices, and has a minimal cost.

In 2022, Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva proposed an algorithm that solves a minimum-cost flow problem by finding min-ratio cycles in $m^{1+o(m)}$ time. In theory, this is the fastest algorithm for solving min-cost flow problems.

In this thesis, we first motivate parts of the algorithm described by Chen et al. by examining the minimum-mean cycle canceling algorithm. Then, we will provide an overview of the algorithm by Chen et al. and look into certain aspects more detailed. Finally, we describe an algorithm partly implementing the algorithm by Chen et al. The core IPM will remain similar, but min-ratio cycles are found via classical algorithm. We then analyze the problems with this algorithm.

Deutsche Zusammenfassung

Das Min-Cost Flow Problem ist ein klassisches Optimierungsproblem der Graphentheorie . Hierbei ist auf einem gerichtetem Graphen G jeder Knoten mit einem Angebot oder Bedarf eines abstrakten Gutes assoziiert. Das Ziel ist einen Fluss über die Kanten zu finden, der Überschuss und Mangel ausgleicht. Die Kanten haben hierbei Kosten pro Einheit, deren Summe es gilt zu minimieren, während der Fluss gewisse Kapazitätsgrenzen nicht überschreiten darf.

In 2022 haben Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg and Sushant Sachdeva eine Arbeit veröffentlicht, in der sie einen Algorithmus vorstellen, der dieses Problem in $m^{1+o(m)}$ Zeit lösen. Damit ist dieser Algorithmus der schnellste, um Min-Cost Flow Probleme zu lösen.

In dieser Arbeit motivieren wir zuerst ein paar der Gedanken des Algorithmus von Chen et al. indem wir den klassischen Min-Mean Cycle Canceling Algorithmus erklären. Danach geben wir einen Überblick über die Kernaspekte des Algorithmus von Chen et al. und beschreiben ausgewählte Aspekte im Detail.

Am Ende bieten wir eine partielle Implementierung des Algorithmus an, in der wir das Finden von Min-Ratio Cycles durch klassische Algorithmen ersetzten haben. Anschließend analysieren welche Probleme dieser algorithmus in der Praxis hat.

Contents

1. Introduction

Imagine a company with n factories that either produce or need *some* product. In total, there are m direct paths that run from one factory to another. The company wants to find ways to transport this product from the factories that have a surplus to the ones that do not. However, on these direct paths, there are capacities that limit how many units can be transported or even force the minimum amount that needs to be transported. Most importantly, these paths have associated costs. Therefore, the company wants to find a way to deliver the product in the cheapest way possible. This is the essence of the min-cost flow problem. We model the factories on a network $G = (V, E)$ with n nodes and m edges. Because this problem is very generic, it can be used to solve different practical problems. Therefore, finding a fast algorithm to solve these problems is interesting.

The cost scaling algorithm is currently one of the most effective algorithms for solving min-cost flow problems. It runs in $\mathcal{O}(n^2m)$ ([KK12, p. 86, 113]). In 2022, Chen Li et al. constructed an algorithm that finds a min-cost flow with high probability in $m^{1+o(1)}$. It is worth mentioning that around the same time Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak found a deterministic algorithm for finding an $(1 - \varepsilon)$ approximate flow in $m^{1+o(1)}$ ([BGS21]).

This thesis is composed of three parts. In chapter 3 we describe the *minimum mean cycle canceling algorithm*. Even though it is a classical algorithm, it contains important concepts used by Chen et al. Thus, we can familiarize ourselves with these concepts and compare them with the newly proposed algorithm. Chapter 4 contains an overview of this algorithm followed by a more detailed explanation of the main data structure. In Chapter 5 we describe a C++ implementation of the algorithm of Chen et al., where we mainly replaced the data structure in charge of finding min-ratio cycles with a classical algorithm. This implementation fails rather quickly. We analyze why this is the case and whether this has any implications for the practicality of Chen et al. 's algorithm.

2. Preliminaries

It is assumed that definitions such as graphs, cycles, and spanning trees are known.

2.1 Notation and definitions

Definition 2.1 (Contraction)**.** *Given the graph G and a spanning forest $F \subseteq G$, then G/F is the graph created by contracting the vertices in $V(G)$ that belong to the same component in F.*

Most of the time G/F is a multigraph, with self-loops. In practice we can remove these self-loops.

Definition 2.2 (Concatenating paths)**.** *Given a graph G with $e_1 = (u, v), e_2 = (v, w) \in E(G)$, then*

$$e_1 \oplus e_2$$

is the walk traversing u, v, w via e_1 and e_2.

Definition 2.3 (Edge-Vertex Incidence Matrix)**.** *Given a directed graph $G = (V, E)$, we define the* edge-vertex incidence *matrix $\boldsymbol{B} \in \{-1, 0, 1\}^{E \times V}$ as*

$$\boldsymbol{B}_{e,v} = \begin{cases} 1 & \exists u \in V : e = (v, u) \\ -1 & \exists u \in V : e = (u, v) \\ 0 & v \notin e \end{cases}$$

Often we will refer to it as the incidence matrix.

General notation

As done by Chen et al., we denote vectors and matrices by boldface letters. For the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, we define $\boldsymbol{x} \circ \boldsymbol{y}$ to be the pairwise product and $|\boldsymbol{x}|$ to be the absolute value .

2.2 Landau notation

Often, we use notation extending the standard $\mathcal{O}$ notation. In particular in chapter 4 we use two more. $\widetilde{\mathcal{O}}$ is the soft-O notation, which is like big-O suppresses but also suppresses logarithmic factors. The other notation is more commonly used.

Definition 2.4 (small-O)**.** *Given* $g : \mathbb{N} \to \mathbb{R}, f : \mathbb{N} \to \mathbb{R}$*, then* $f = o(g)$ *if and only if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

We mainly use $o(1)$. In chapter 4, we regularly use the following interactions.

Lemma 2.5. *For* $n \in \mathbb{R}$ *:*

1. $\log(n) = n^{o(1)}$
2. $n^{o(1)}n^{o(1)} = n^{o(1)}$

Proof.

1. We can prove this by taking the logarithm of both sides and dividing by $\log(n)$ to get $\log(\log(n))/\log(n) \leq o(1)$. This is true (and can be proven using l'hopital rule).
2. We know $n^{o(1)}n^{o(1)} = n^{2o(1)}$. Now using $2o(1) = o(1)$, we prove that $n^{o(1)}n^{o(1)} = n^{o(1)}$.

□

Similar to how we proved that $n^{o(1)}$ suppresses logarithms, we can show that it also suppresses constants. The second point of the lemma allows us to loop over an operation taking $n^{o(1)}$ without the loop having an asymptotically higher runtime than the operation itself, as long as we iterate $n^{o(1)}$ times.

2.3 Networks and Flows

Ahuja et al. defined a directed *network* as directed graph G whose nodes[vertices] and/or arcs[edges] have associated numerical values ([AMO93, p. 24-25]). We can think about this association in two ways. The first, by defining a function, e.g. $h : E(G) \to \mathbb{R}$. The second one is by using vectors that we index with the edges/vertices. For the example above, we define the vector $\boldsymbol{h}$ by setting $\boldsymbol{h}_e = h(e)$ for all $e \in E(G)$. The order in which we index these edges is arbitrary, as long, as it is the same for all vectors and later on matrices. We use the second representation more commonly, especially in the implementation. Although sometimes we also apply standard function operations such as limiting the domain, e.g., for $G' \subseteq G$ we can define $h' = h_{|E(G')}$.

Examples of these associated vectors are capacities in computer networks, different outputs of factories (for node associated vectors), or simply distances between locations. We can model different problems like this. We will mainly look at the *Minimum Cost Flow Problem*/min-cost flow problem/MCF problem. However, we will also touch on some other problems along the way.

2.3.1 Minimum Cost Flow

Depending on the problem, a *flow* is defined differently. In this thesis, we define flow very liberally.

Definition 2.6. *Let G be a graph with the incidence matrix $\boldsymbol{B}$. A* flow *$\boldsymbol{f} \in \mathbb{R}^{E(G)}$ are edge - associated values that generate some demand $\boldsymbol{d} = \boldsymbol{B}^T \boldsymbol{f}$.*

Definition 2.7 (Circulation)**.** *On a graph G with edge-vertex incidence matrix $\boldsymbol{B}$, a $\boldsymbol{\Delta} \in \mathbb{R}^{E(G)}$ is a* circulation*, if $\boldsymbol{B}^T \boldsymbol{\Delta} = \boldsymbol{0}$.*

Thus, a circulation is equivalent to a flow that does not generate any demand. The simplest circulation is $\boldsymbol{0}$. Another example is a cycle in G where we route the same amount of flow over every edge.

Definition 2.8 (Minimum cost flow problem)**.** *An instance of the* minimum-cost flow *is a tuple $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ with $\boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^- \in \mathbb{Z}^E$ and $\boldsymbol{d} \in \mathbb{Z}^V$ and all integers bounded by U.*

In the introduction we motivated the min-cost flow as a network of factories n. For this example, we can interpret $\boldsymbol{d}$ as the demand/supply of the abstract good. Although we will use the two words interchangeably, the correct way to interpret $\boldsymbol{d}_v = 1$ for a $v \in V$ is to say that v has demand 1. $\boldsymbol{c}$ will be the transport cost of the edges and $\boldsymbol{u}^+, \boldsymbol{u}^-$ the capacities of how much we can/need to transport over an edge. Ignoring the cost for now, a flow that satisfies the demands and is within the constraints, is called a *feasible flow*.

Definition 2.9 (Feasible flow)**.** *Given an instance $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ of the min-cost flow problem, then $\boldsymbol{f} \in \mathbb{R}^E$ is a* feasible flow *on this instance, if:*

1. $\boldsymbol{u}^- \leq \boldsymbol{f} \leq \boldsymbol{u}^+$,
2. $\boldsymbol{B}^T \boldsymbol{f} = \boldsymbol{d}$ *where $\boldsymbol{B}$ is the edge-vertex incidence matrix*

Flows that are not feasible are called *pseudo flows*.

Definition 2.10 (Min-cost flow)**.** *Given an instance $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ of the min-cost flow problem, then $\boldsymbol{f}^* \in \mathbb{R}^E$ is a* minimum cost flow *on this instance, if:*

1. *$\boldsymbol{f}$ is a feasible flow,*
2. *$\boldsymbol{c}^T \boldsymbol{f}^* \leq \boldsymbol{c}^T \boldsymbol{f}$ for all feasible flows $\boldsymbol{f}$.*

2.3.1.1 Assumptions

For all MCF problem instance $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ in this thesis, we assume the following:

No parallel edges in G

In practice, parallel edges are not a problem. In theory, they pose notational challenges because we cannot differentiate $e = (u, v), e' = (u, v) \in E(G)$ by using only their vertices. We can reduce any problem with parallel edges e, e' as defined before, in the following manner: We replace e' by a new vertex u' and the edges $e'_1 = (u, u')$ and $e'_2 = (u', v)$. We set the $\boldsymbol{u}^+_{e'_1} = \boldsymbol{u}^+_{e'_2} = \boldsymbol{u}^+_{e'}$ and $\boldsymbol{c}_{e'_1} = 0, \boldsymbol{c}_{e'_2} = \boldsymbol{c}_{e'}$. In this way we can remove parallel edges by adding at most $m - 1$ edges and vertices, so we can find a min-cost flow in the same amount of time.

No self loops in G

Assuming G containsa self-loops, we know that routing any amount of flow over these loops does not change the generated demands. Therefore, we can simply solve the min-cost flow problem on G without the self-loops and then simply route the amount of flow over the self-loops, which reduces the cost the most. Thus, we can assume that G has no self-loops without loss of generality.

Integrality

As implicitly stated in the definitions, we only concern ourselves with problems where $\boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}$ only contain integers. If they were rational numbers we could simply scale up the values. Therefore, we can conveniently assume that the min-cost flow (assuming a min-cost flow exists) is an integer flow ([AMO93, p 318]).

2.3.1.2 Compact definition

We will look at a different representation of the min-cost flow that forms the bridge to linear programming.

For an instance of the MCF problem $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$, Ahuja et al. ([AMO93, p. 296]) describe the min-cost flow as the flow f^* satisfying the following properties:

$$\begin{aligned}
\text{minimize } c(\boldsymbol{f}) = \quad & \sum_{e\in E} \boldsymbol{c}_e \boldsymbol{f}_e \\
\text{subject to} \quad & \sum_{\substack{u\in V:\\(v,u)\in E}} \boldsymbol{f}_{(v,u)} - \sum_{\substack{u\in V:\\(u,v)\in E}} \boldsymbol{f}_{(u,v)} = \boldsymbol{d}_v \qquad \text{for all } v \in \text{V} \\
& \boldsymbol{u}^-_e \le \boldsymbol{f}_e \le \boldsymbol{u}^+_e \qquad \text{for all } e \in E
\end{aligned}$$

We can see how the first and the last line correspond to the second point of definition 2.10 and the first point of definition 2.9. A bit less straight forward is the part that captures the notion of this flow generating the correct demand . For this, let us focus on the line beginning with “subject to”. For ever $v \in V$, we can interpret $\sum_{\substack{u\in V:\\(v,u)\in E}} \boldsymbol{f}_{(v,u)}$ as all the units of flow coming into v while $\sum_{\substack{u\in V:\\(u,v)\in E}} \boldsymbol{f}_{(u,v)}$ as all the flow leaving it. We also now see that $(\boldsymbol{B}^T\boldsymbol{f})_v = \sum_{e\in E} \boldsymbol{B}^T(v,e)\boldsymbol{f}_e$. If we split this sum into the edges where $\boldsymbol{B}^T_{v,e} = 0$, $\boldsymbol{B}^T_{v,e} = -1$ and $\boldsymbol{B}^T_{v,e} = 1$, it is equivalent to the amount of flow coming into v minus the flow going out of v. In this way, the line beginning with “subject to” corresponds to the second point in the definition of a feasible flow.
With all this in mind, we can compactly define the min-cost flow to be

$$\boldsymbol{f}^* = \underset{\substack{\boldsymbol{B}^T\boldsymbol{f}=\boldsymbol{d}\\ \boldsymbol{u}^-\le\boldsymbol{f}\le\boldsymbol{u}^+}}{\arg\min}\, \boldsymbol{c}^T\boldsymbol{f}.$$

2.4 Linear Programming

A linear programming(LP) problem is an optimization problem in which the goal is to maximize or minimize a linear objective function under some linear constraints. These constraints are given as a collection of equalities and inequalities. Every LP problem can be transformed into the so called *standard form* ([PAN23, p. 14-17])

$$\begin{array}{llllll}
\min f = & \boldsymbol{c}_1\boldsymbol{x}_1 & +\boldsymbol{c}_2\boldsymbol{x}_2 & +\dots & +\boldsymbol{c}_m\boldsymbol{x}_m & \\
\text{subject to} & \boldsymbol{a}_{11}\boldsymbol{x}_1 & \boldsymbol{a}_{12}\boldsymbol{x}_2 & +\dots & \boldsymbol{a}_{1m}\boldsymbol{x}_m & = \boldsymbol{b}_1 \\
& \boldsymbol{a}_{21}\boldsymbol{x}_1 & \boldsymbol{a}_{22}\boldsymbol{x}_2 & +\dots & \boldsymbol{a}_{2m}\boldsymbol{x}_m & = \boldsymbol{b}_2 \\
& \vdots & \vdots & \ddots & \vdots & \vdots \\
& \boldsymbol{a}_{n1}\boldsymbol{x}_1 & \boldsymbol{a}_{n2}\boldsymbol{x}_2 & +\dots & \boldsymbol{a}_{nm}\boldsymbol{x}_m & = \boldsymbol{b}_n \\
& \boldsymbol{x}_1, & \boldsymbol{x}_2, & \dots & \boldsymbol{x}_m & \ge 0
\end{array}$$

or

$$\begin{aligned} \min f = \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned},$$

with $\boldsymbol{A} \in \mathbb{R}^{n\times m}, \boldsymbol{c}, \boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{b} \in \mathbb{R}^n$. The goal of this section is to establish, that the MCF problem can be represented as an LP, so that we can use the geometric properties of LPs to visualize what different MCF algorithms do.

2.4.1 Geometry of the feasibility region

In this section, we discuss some of geometric properties of the space of solutions satisfying the constraints of an LP. To visualize this, we also discuss how we can transform LPs into different representations.

Definition 2.11 (Feasibility region)**.** *The* feasibility region *of a linear program*

$$\begin{aligned} \min f = \quad & \boldsymbol{c}^T\boldsymbol{x} \\ \text{s.t.} \quad & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned},$$

is the set $\{x \in \mathbb{R}^m | Ax \leq b \wedge x \geq 0\}$ *of points that satisfy the constraints.*

Note that the linear program here is not in the standard form because we have $\boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$ instead of $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. This is an LP problem's canonical form ([NP17, p. 17]) and we can transform every LP problem in standard form into its canonical form and the other way around. To demonstrate this, let $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{im}\boldsymbol{x}_m = \boldsymbol{b}_i$ be some constraint of a LP problem in standard form. Then we can transform it into the two equivalent inequations $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{im}\boldsymbol{x}_m \leq \boldsymbol{b}_i$ and $-\boldsymbol{a}_{i1}\boldsymbol{x}_1 - \cdots - \boldsymbol{a}_{im}\boldsymbol{x}_m \leq -\boldsymbol{b}_i$. On the other hand let $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{im}\boldsymbol{x}_m \leq \boldsymbol{b}_i$ be some constraint of an LP problem in its canonical form, then it is equivalent to the equation $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{im}\boldsymbol{x}_m + \boldsymbol{x}_{m+1} = \boldsymbol{b}_i$ if we add the non-negativity constraint $\boldsymbol{x}_{m+1} \geq 0$. Also $\boldsymbol{c}_{m+1} = 0$ so that this new variable called a *slack variable*, does not contribute to the cost. The reason why we use two different forms is that the standard form is closer to the definition of the MCF problem and the canonical form makes it easier to motivate the shape of the feasibility region.

Let us first geometrically interpret the $x \in \mathbb{R}^m$ solving $\boldsymbol{a}^T\boldsymbol{x} = b$ for $x \in \mathbb{R}^m$ where $\boldsymbol{a}, \in \mathbb{R}^m$ and $b \in \mathbb{R}$. If there is no solution, then this is an empty set, therefore, let us assume that there is at least one solution $\boldsymbol{x}_0 = (x_1, x_2, \ldots, x_m)$. Then, for every other solution $\hat{\boldsymbol{x}} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_m)$ we know that $\boldsymbol{a}^T\hat{\boldsymbol{x}} = b = \boldsymbol{a}^T\boldsymbol{x}_0$. This is equivalent to $\boldsymbol{a}^T(\hat{\boldsymbol{x}} - \boldsymbol{x}_0) = 0$ This can be interpreted as the hyperplane orthogonal to $\boldsymbol{a}$ in $\mathbb{R}^n$, translated by $-\boldsymbol{x}_0$. Then $\boldsymbol{a}^T\boldsymbol{x} \leq b$ is just the half-space on one side of this hyperplane. In this way, the solution to $\boldsymbol{A}\boldsymbol{x} \leq b$ is just the intersection of n half spaces. The solutions for $\boldsymbol{x} \geq 0$ can be expressed as the intersection of the half-spaces $\bigcap i \in [m]\{\boldsymbol{x} \in \mathbb{R}^m | \boldsymbol{x}^T e_i \geq 0\}$ where e_i are the unit vectors. Therefore, overall the feasibility region is the intersection of half-spaces.
We will not prove the further characteristics of the feasibility region. An in-depth discussion can be found in the work of Robert J. Vanderbei [Van, chapter 2]. In summary, the feasibility region is a convex polytope with vertices(extreme points), edges and polygonal faces. All points not on a face are called *interior points*. Solutions in the feasibility set are called *feasible*. Since in our case these points will be flows, this will match our definition of feasible flows (as soon as we see that a MCF problem is a LP). It is important to note that in practice, there might not be any feasible solutions.

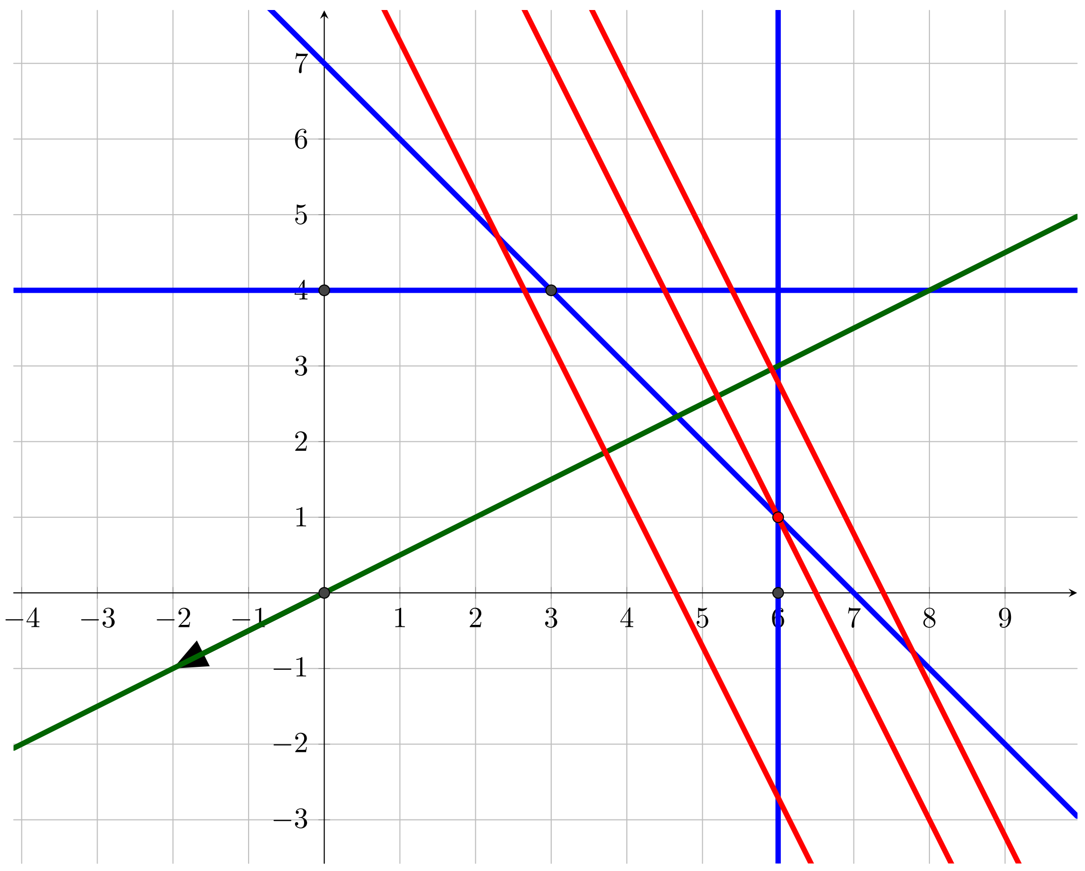

Figure 2.1: Two dimensional feasibility region

Figure 2.1 shows a 2 dimensional example an LP in canonical from where $\boldsymbol{c} = (-0.5, -1)$, $\boldsymbol{d} = (6, 4, 7)$ and

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 1 \end{pmatrix}$$

The blue lines correspond to the constraints. The arrow on the green line indicates the direction we must minimize. We now demonstrate why we can always find a vertex that is an optimal solution (if there is an optimal solution). We want to find a point $\boldsymbol{x} \in \mathbb{R}^n$ that minimizes the $\langle \boldsymbol{c}, \boldsymbol{x} \rangle$. For this, we can simply look at the orthogonal projection of $\boldsymbol{x}$ onto $\boldsymbol{c}$. In this way, we can decompose $\boldsymbol{x} = \beta \boldsymbol{c} + \boldsymbol{r}$, with $\beta \in \mathbb{R}$ and $\boldsymbol{r} \in \mathbb{R}^n$ and $\langle \boldsymbol{r}, \boldsymbol{c} \rangle = 0$. Thus, we can simply imagine slowly decreasing the β while always looking at the line orthogonal to $\boldsymbol{c}$ in which we would find $\boldsymbol{r}$. Visually speaking ,this pushes this line across the feasibility region. The red lines represent examples of these lines. One where β is too small, one too big and one correct. If the feasibility region were to continue infinitely in this direction, there would not be a minimal solution. If it does not, then we can find a optimal solution, where in the last point of the polytope intersects this line. This will either be a vertex or a line (wit two vertices). This is the idea behind the argument, why we can always find an optimal solution in a vertex of the polytope.

2.4.2 Min-cost flow as an LP-Problem

We now want to see how we can represent a MCF problem as an LP problem. For this, consider an instance of the min-cost flow problem as defined before.

$$\boldsymbol{f}^* = \underset{\substack{\boldsymbol{B}^T \boldsymbol{f} = \boldsymbol{d} \\ \boldsymbol{u}^- \leq \boldsymbol{f} \leq \boldsymbol{u}^+}}{\arg\min}\, \boldsymbol{c}^T \boldsymbol{f}$$

The only difference to the standard form are the capacity constraints $\boldsymbol{u}^- \leq \boldsymbol{f} \leq \boldsymbol{u}^+$. We can split this into two constraints $\boldsymbol{f} \leq \boldsymbol{u}^+$ and $\boldsymbol{f} \geq \boldsymbol{u}^-$. When it comes to $\boldsymbol{f} \leq \boldsymbol{u}^+$, we can use the same idea as used when transforming an LP-Problem from the canonical form to the standard form. We can do the same with $\boldsymbol{f} \geq \boldsymbol{u}^-$ after changing the constraint to the equivalent $-\boldsymbol{f} \leq -\boldsymbol{u}^-$.
We now need to ensure that $\boldsymbol{f} \geq \boldsymbol{0}$. We do this by replacing $\boldsymbol{f}_e$ by $\boldsymbol{f}_{e_1}$ and $\boldsymbol{f}_{e_2}$ for every $e \in E(G)$. Now we can add the constraint $\boldsymbol{f}_{e_1}, \boldsymbol{f}_{e_2} \geq 0$, to achieve the standard form. Therefore, we can visualize a MCF algorithm using the feasibility region. The algorithm we use, always satisfies the demands. Therefore, these algorithm leave the feasibility region when violating the capacity constraints.

2.4.3 Simplex Algorithm

There are multiple ways of solving an LP problem. The most well known is the *simplex algorithm* by George B. Dantzig in 1947 [PAN23, p. 57]. We will use the simplex as an example of how an algorithm traverses the feasibility region, without going into detail about its correctness or runtime. We assume the existence of an optimal solution of the LP/MCF problem. We also assume that we have found some vertex of the polytope, to use as an initial point. The canonical representation of an LP is helpfull while visualizing the simplex algorithm. When we are on a face of the polytope, there is some inequality $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{in}\boldsymbol{x}_n \leq \boldsymbol{b}_i$ in the LP problem that is at its maximum, so $\boldsymbol{a}_{i1}\boldsymbol{x}_1 + \cdots + \boldsymbol{a}_{in}\boldsymbol{x}_n = \boldsymbol{b}_i$. When we describe a point on an edge of the polytope, this is the case for two constraints. For a vertex, three or more inequations become equations. To traverse from one vertex to another, we must loosen one constraint and tighten another. The simplex algorithm decides which direction decreases the objective function and then traverses to the next vertex. This is repeated until a vertex is reached where the objective function can no longer be decreased by traversing. As we will see later, a classical min-cost flow algorithm uses a similar strategy.

3. Classical Min-Cost Flow Algorithms

In this chapter, we examine some basic ideas of MCF algorithms and motivate the ideas behind the *minimum-mean cycle canceling*/(MMCC) algorithm. Most proofs can be found in a different form in the work of Ahuja et al. ([AMO93, p. 306-382]). We specifically chose the MMCC algorithm because it uses the idea of minimum-mean cycles to iteratively get closer to an optimal solution. In chapter 4 we will see that Chen et al. use minimum-ratio cycles for this purpose, which is a generalized version of this concept.

For the rest of this chapter, we are working on an instance $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ with $\boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^- \in \mathbb{Z}^E$ of the min-cost flow problem as defined in definition 2.8. In this chapter only, we add the following three assumptions, which do not lead to a loss of generality: $\boldsymbol{u}^- = \boldsymbol{0}$ ([AMO93, p. 39]) , $\boldsymbol{c} \geq \boldsymbol{0}$ ([AMO93, p. 297]) and f.a. two vertices $v, u \in V$ there never both edges $(v, u) \in E$ and $(u, v) \in E$ at the same time([AMO93, p. 45]). This last assumption is once again only to avoid notational issues later on and can be ignored in practice.

3.1 Cycles and Circulations

All MCF algorithms mentioned in this thesis, do essentially the same thing. They find some flow on the polytope, and then continuously augment this flow to navigate the polytope while getting closer to the optimal solution. To stay inside the polytope, the flows always need to stay within the capacity constraints. The algorithms we look at achieve this in a fundamentally different ways. To be feasible, the flows must also always satisfy the demands. In this section we explain, why finding cycles is integral to augmenting flows in a way that keeps them feasible.

Lemma 3.1. *Given a graph G with incidence matrix $\boldsymbol{B}$. Let $\boldsymbol{\Delta}$ be a circulation and $\boldsymbol{f}$ a flow with $\boldsymbol{B}^T\boldsymbol{f} = \boldsymbol{d}$ for the demands $\boldsymbol{d}$, then:*

- $\boldsymbol{B}^T(\boldsymbol{f} + \boldsymbol{\Delta}) = \boldsymbol{B}^T(\boldsymbol{f}) = \boldsymbol{d}$
- $\boldsymbol{c}^T(\boldsymbol{f} + \boldsymbol{\Delta}) = \boldsymbol{c}^T\boldsymbol{f} + \boldsymbol{c}^T\boldsymbol{\Delta}$.

Proof. Since matrix multiplication is distributive and definition 2.7, we know that

$$\boldsymbol{B}^T(\boldsymbol{f} + \boldsymbol{\Delta}) = \boldsymbol{B}^T(\boldsymbol{f}) + \boldsymbol{B}^T(\boldsymbol{\Delta}) = \boldsymbol{B}^T(\boldsymbol{f}).$$

The second points directly follows the linearity of scalar products. □

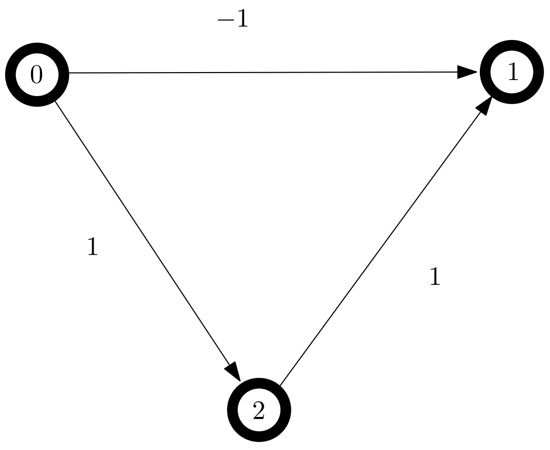

Figure 3.1: Augmentation cycle

Next is the important relationship between cycles and circulations. Here we introduce the definition of an augmentation cycle.

Definition 3.2 (Augmentation cycle)**.** *Given a graph G. Let $C \subseteq G$ by an undirected (ignoring edge directions) cycle. Then C is an* augmentation cycle*, if there is a corresponding circulation $\boldsymbol{\Delta} \neq \mathbf{0} \in \mathbb{R}^E$ only routing flow over edges in C.*

We will sometimes refer to this corresponding circulation without specifically mentioning the augmentation cycle, when we refer to an *augmentation cycle vector.*

Lemma 3.3. *Given a graph G with incidence matrix $\boldsymbol{B}$ and flow $\boldsymbol{f}$ on G then,*

1. *Every directed cycle C is also an augmentation cycle.*
2. *Not every augmentation cycle is a directed cycle.*
3. *For every augmentation cycle C with the corresponding augmentation vector $\boldsymbol{\Delta}$ there is a $\mu \in \mathbb{R} \setminus \{0\}$ with $|\boldsymbol{\Delta}_e| = \mu$ for all $e \in C$.*

The third point can be interpreted as augmentation circulation routing the same amount of flow in one direction along the augmentation cycle. Because these edges are sometimes directed in the opposite direction of the flow, the flow is negated on these edges.

Proof.

1. Let $\boldsymbol{\Delta}$ be the vector that routes 1 over every edge in C. Because every vertex $v \in V(G)$ in C receives one unit of flow and then passes it on, we know that $(\boldsymbol{B}^T\boldsymbol{\Delta})_v = 0$. For vertices that are not in the cycle, there is no flow routed in or out of them. Therefore $\boldsymbol{B}^T\boldsymbol{\Delta} = \mathbf{0}$ with $\boldsymbol{\Delta}$ only using edges of C. Therefore C is an augmentation cycle.
2. Figure 3.1 shows a graph not containing a directed cycle but an augmentation cycle.
3. We first show that $|\boldsymbol{\Delta}_e|$ is the same for all edges $e \in E(C)$. Here, let v be a vertex in $V(C)$, where $e_1, e_2 \in C$ are the two edges incident to v. We know that these are the only edges going in and/or out of v that route a value different from 0. Now, since $\boldsymbol{\Delta}$ is a circulation, we know that $(\boldsymbol{B}^T\boldsymbol{\Delta}) = 0$. So $\boldsymbol{B}_{e_1,v}\boldsymbol{\Delta}_{e_1} + \boldsymbol{B}_{e_2,v}\boldsymbol{\Delta}_{e_2} = 0$. Because e_1, e_2 are incident to v, we can conclude $\boldsymbol{B}_{e_1,v}, \boldsymbol{B}_{e_2,v} \in \{-1, 1\}$. Therefore $|\boldsymbol{\Delta}_{e_1}| = |\boldsymbol{\Delta}_{e_2}|$. Now we know that the value $\boldsymbol{\Delta}_e$ of some edge $e \in E(C)$ determines the absolute value of all other edges in C. We can now conclude that this value $\boldsymbol{\Delta}_e \neq 0$ because otherwise $\boldsymbol{\Delta} = 0$, which would be a contradiction to the definition of the augmentation cycle.

□

By definition, we know that all augmentation cycle vectors are circulations. We now show that we can decompose all circulations into augmentation cycle vectors.

Lemma 3.4. *We can decompose any circulation $\boldsymbol{\Delta}$ with $\boldsymbol{\Delta} \neq \mathbf{0}$ into augmentation cycle vectors $\boldsymbol{\Delta}_1, \dots, \boldsymbol{\Delta}_b$ so that $\sum_{i=1}^{b} \boldsymbol{\Delta}_i = \boldsymbol{\Delta}$.*

Proof. We prove this inductively by finding an augmentation cycle that is part of the circulation and then a corresponding augmentation vector. Subtracting this vector will reduce the number of edges in the circulation with non-zero flow, showing that this can only be done a finite amount of times.
We first show that there is an augmentation cycle $C_1 \subseteq G$ on the edges that $\boldsymbol{c}$ is routing flow over. To show this, assume that there is no such cycle. Since $\boldsymbol{\Delta} \neq 0$, there are edges that route some flow. These edges would not contain a cycle and would therefore be a forest with leaf vertices. Let v be such a leaf vertex with e being the only incident edge with $\boldsymbol{\Delta}_e \neq 0$. Then $|(\boldsymbol{B}^T\boldsymbol{\Delta})_v| = |\boldsymbol{\Delta}_e| \neq 0$ and thus $\boldsymbol{B}^T\boldsymbol{\Delta} \neq \mathbf{0}$. This is a contradiction to the definition of a circulation. If we take some edge $e \in C_1$ we can create the first augmentation circulation $\boldsymbol{\Delta}_1$ by setting $(\boldsymbol{\Delta}_1)_e = \boldsymbol{\Delta}_e$. From the third point of lemma 3.3 we know that this determines the values for the rest of the augmentation circulation. With this construction, it follows that $(\boldsymbol{\Delta} - \boldsymbol{\Delta}_1)_e = 0$. And since Δ_1 is a circulation itself, we know that that $\boldsymbol{B}^T(\boldsymbol{\Delta} - \boldsymbol{\Delta}_1) = \boldsymbol{B}^T\boldsymbol{\Delta} + \boldsymbol{B}^T\boldsymbol{\Delta}_1 = 0$ and is therefore once again a circulation. Note that this new circulation no longer routes any flow over edge e, and only edges that the original circulation used were changed. We have decreased the edges in the circulation with a non-zero value. We can inductively apply this step to the newly generated circulation until we have reach the circulation $\mathbf{0}$. □

Because the current flow and the optimal solution generate the same demand, there is always circulation changing the current flow into the optimal solution.

Definition 3.5 (Witness Circulation)**.** *Given an instance of the MCF problem with demand vector $\boldsymbol{d}$ and vertex incidence matrix $\boldsymbol{B}$. Let $\boldsymbol{f}^*$ be a optimal solution to this MCF problem. Given a feasible flow $\boldsymbol{f}$, then*

$$\boldsymbol{\Delta}(\boldsymbol{f}) = \boldsymbol{f}^* - \boldsymbol{f}$$

is the witness circulation*.*

With this, we now know that as long as we can find a circulation with negative cost that does not push the current flow outside the capacity bounds, we can decrease the cost of the flow. From a different perspective: we can build the witness circulation by adding up certain negative cost augmenting cycles.

3.2 Minimum-Mean Cost Canceling Algorithm

We now want to motivate the idea behind the MMCC algorithm. We first assume that for some instance of the MCF problem on the graph G, we have found a feasible flow $\boldsymbol{f}$ corresponding with some vertex of the feasibility region. Now, we want to find an augmenting cycle that takes us from the current vertex in the polytope to another while decreasing the cost of the flow, just as we did in the simplex algorithm 2.4.3. To ensure that, we do not leave the capacity constraints, we create a special network whose directed cycles correspond with augmentation cycles in G, which have this property. We can create such a network by looking at the possible changes or, in other words, the residual capacities.

We can either add $\boldsymbol{u}_e^+ - \boldsymbol{f}_e$ more flow with the cost of $\boldsymbol{c}_e$ or remove $\boldsymbol{f}_e - \boldsymbol{u}_e = \boldsymbol{f}_e$ flow by changing the cost by $-\boldsymbol{c}_e$ per unit. If we split the edge e into an edge that encodes the addition of flow and a reversed edge encoding the subtraction of flow, then the direct cycles will describe all augmentations inside the polytope. This network is called a *residual network* and is used by many different MCF algorithms.

Definition 3.6 (Residual network). *Given an instance $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ of the min-cost flow problem and a flow $\boldsymbol{f} \in \mathbb{Z}^E$. We define the* residual network $R(\boldsymbol{f}) = \{G_r = (V, E_r), \boldsymbol{r}, \boldsymbol{c}^r\}$. *We set $E_r = E(G) \cup \{(i,j) \mid (j,i) \in E(G)\}$, $\boldsymbol{r}_{(i,j)} = \boldsymbol{u}^+_{(i,j)} - \boldsymbol{f}_{(i,j)}, \boldsymbol{r}_{(j,i)} = \boldsymbol{f}_{(i,j)}, \boldsymbol{c}^r_{(i,j)} = \boldsymbol{c}_{(i,j)}, \boldsymbol{c}^r_{(j,i)} = -\boldsymbol{c}_{(i,j)}$ for every edge $(i,j) \in E(G)$.*

Note that this definition is notionally challenging, when we cannot have both an edge (i,j) and (j,i) in G. Because of this, we have removed these cases at the beginning of the section 3.

There is a direct relationship between flows in G and in $R(\boldsymbol{f})$. We will only examine the relationship between feasible directed cycles in $R(\boldsymbol{f})$ and augmentation cycles in G. We do not have capacity constraints or demands in $R(\boldsymbol{f})$, so we define a cycle vector $\boldsymbol{\Delta}$ to be feasible if $\boldsymbol{\Delta} \leq \boldsymbol{r}$.

Lemma 3.7. *Let $(G = (V, E), \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ be an instance of the min-cost flow problem with some feasible flow $\boldsymbol{f} \in \mathbb{Z}^E$ and the residual network $R(\boldsymbol{f})$. We can map any directed cycle vector $\boldsymbol{\Delta}$ of a cycle with length ≥ 3 in $R(\boldsymbol{f})$ to an augmentation cycle vector $\boldsymbol{\Lambda}$ in G by setting*

$$\boldsymbol{\Lambda}_{(i,j)^G} = \begin{cases} \mu & \text{if } \boldsymbol{\Delta}_{(i,j)} \neq 0 \\ -\mu & \text{if } \boldsymbol{\Delta}_{(j,i)} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

for all $(i,j)^G \in E(G)$. Furthermore if $\boldsymbol{\Delta}$ is feasible, then $\boldsymbol{f} + \boldsymbol{\Lambda}$ is feasible and $(\boldsymbol{c}^r)^T\boldsymbol{\Delta} = \boldsymbol{c}^T\boldsymbol{\Lambda}$.

Proof. Let $C \subseteq R(\boldsymbol{f})$ be the cycle of $\boldsymbol{\Delta}$ We first prove that the mapping is well defined. For this, we must show that every for edge $(i,j)^G \in E(G)$ at most only one of $(i,j), (j,i) \in E(R(\boldsymbol{f}))$ has a non-zero flow in $\boldsymbol{\Delta}$. If we assume that both have flow, then $(i,j), (j,i) \in E(C)$. This is only possible if these are the only two edges, which would result in a cycle of length 2, which is not allowed. Therefore $\boldsymbol{\Lambda}$ is well defined.
Let us now show that $\boldsymbol{\Lambda}$ is a circulation. For this let $\boldsymbol{B}$ be the incident matrix for G. We need to show that for all $v \in V(G)$ $(\boldsymbol{B}^T\boldsymbol{\Lambda})_v = 0$. Since $V(R(\boldsymbol{f})) = V(G)$ we will use $v \in G$ to refer to the vertex in both networks.
If $v \notin V(C)$ then for all edges $e \in E(R(\boldsymbol{f})$ incident to v, their flow is $\boldsymbol{\Delta}_e = 0$. Using the definition of the mapping, we know that for all edges $e^G \in E(G)$ incident to v $\boldsymbol{\Lambda}_e = 0$.
If $v \in V(C)$ then there are exactly two edges $(i,v), (v,j) \in E(R(\boldsymbol{f}))$ so that $\mu := \boldsymbol{\Delta}_{(i,v)} = \boldsymbol{\Delta}_{(v,j)} \neq 0$. Looking at the edges in G, there are two edges e_i, e_j that are incident to v and correspond with (i,v) and (v,j), which are the only edges with a non zero value, so $(\boldsymbol{B}^T\boldsymbol{\Lambda})_v = \boldsymbol{B}_{e_i,v}\boldsymbol{\Lambda}_{e_i} + \boldsymbol{B}_{e_j,v}\boldsymbol{\Lambda}_{e_j}$. If $e_i = (i,v)^G \in E(G)$, then $\boldsymbol{\Lambda}_{e_i} = \mu$ and $B_{e_i,v} = -1$ and therefore $B_{e_i,v}\boldsymbol{\Lambda}_{e_i} = -\mu$. If $e_i = (v,i)^G \in E(G)$, then $\boldsymbol{\Lambda}_{e_i} = -\mu$ and $B_{e_i,v} = 1$ and therefore $\boldsymbol{B}_{e_i,v}\boldsymbol{\Lambda}_{e_i} = -\mu$. So in any case $\boldsymbol{B}_{e_i,v}\boldsymbol{\Lambda}_{e_i} = -\mu$. The same argument can be made for e_j with $\boldsymbol{B}_{e_j,v}\boldsymbol{\Lambda}_{e_j} = \mu$. So together $(\boldsymbol{B}^T\boldsymbol{\Lambda})_v = \boldsymbol{B}_{e_i,v}\boldsymbol{\Lambda}_{e_i} + \boldsymbol{B}_{e_j,v}\boldsymbol{\Lambda}_{e_j} = 0$. Therefore $\boldsymbol{\Lambda}$ is a circulation. And if C is a directed circle, the corresponding edges in G are also an (undirected) circle.
Now, we show that if $\boldsymbol{\Delta} \leq \boldsymbol{r}$ then $\boldsymbol{f} + \boldsymbol{\Lambda}$ is feasible. Since $\boldsymbol{\Lambda}$ is a circulation and $\boldsymbol{f}$ is feasible, then $\boldsymbol{f} + \boldsymbol{\Lambda}$ must also satisfy the demands because of lemma 3.1. We

now only need to show that for each $(i,j)^G = e \in E(G) : 0 \le (\boldsymbol{f} + \boldsymbol{\Lambda})_e \le \boldsymbol{u}_e^+$. If $\boldsymbol{\Delta}_{(i,j)} = \boldsymbol{\Delta}_{(j,i)} = 0$, then $(\boldsymbol{f} + \boldsymbol{\Lambda})_e = \boldsymbol{f}_e$, is inside the capacity constraints since $\boldsymbol{f}$. If $\mu := \boldsymbol{\Delta}_{(i,j)} \neq 0$ then $\boldsymbol{r}_e \ge \mu > 0$ since $\boldsymbol{\Delta}$ is feasible. Therefore $\boldsymbol{\Lambda}_{(i,j)^G} = \mu \le \boldsymbol{r}_e = \boldsymbol{u}_e^+ - \boldsymbol{f}_e$, so $0 \le \boldsymbol{\Lambda}_{(i,j)^G} + \boldsymbol{f}_{(i,j)^G} \le \boldsymbol{u}^+_{(i,j)}$. If $\mu := \boldsymbol{\Delta}_{(j,i)} \neq 0$ then $\boldsymbol{r}_{(j,i)} \ge \mu > 0$ since $\boldsymbol{\Delta}$ is feasible. Therefore $0 > \boldsymbol{\Lambda}_{(i,j)^G} = -\mu \ge -\boldsymbol{r}_{(j,i)} = \boldsymbol{f}_{(i,j)}$, so $\boldsymbol{u}^+_{(i,j)^G} \ge \boldsymbol{\Lambda}_{(i,j)^G} + \boldsymbol{f}_{(i,j)^G} \ge 0$.
The last thing to show is that $(\boldsymbol{c}^r)^T\boldsymbol{\Delta} = \boldsymbol{c}^T\boldsymbol{\Lambda}$. For this let $(i,j)^G$ be an edge, in $E(G)$. If $\boldsymbol{\Delta}_{(i,j)} = \boldsymbol{\Delta}_{(j,i)} = 0$ then $\boldsymbol{\Lambda}_{(i,j)^G} = 0$. Thus the cost of the flow on these edges in both graphs is 0. If $\mu := \boldsymbol{\Delta}_{(i,j)} \neq 0$ then $\boldsymbol{\Delta}_{(j,i)} = 0$ and $\boldsymbol{\Lambda}_{(i,j)^G} = \mu$ so $\boldsymbol{\Delta}_{(i,j)}\boldsymbol{c}^r_{(i,j)} + \boldsymbol{\Delta}_{(j,i)}(-\boldsymbol{c}^r_{(i,j)}) = \mu\boldsymbol{c}^r_{(i,j)} = \boldsymbol{c}_{(i,j)^G}\boldsymbol{\Lambda}_{(i,j)^G}$. And if $\mu := \boldsymbol{\Delta}_{(j,i)} \neq 0$ then $\boldsymbol{\Delta}_{(i,j)} = 0$ and $\boldsymbol{\Lambda}_{(i,j)^G} = -\mu$ so $\Delta_{(i,j)}\boldsymbol{c}^r_{(i,j)} + \boldsymbol{\Delta}_{(j,i)}(-\boldsymbol{c}^r_{(i,j)}) = -\mu(-\boldsymbol{c}_{(i,j)^G}) = \boldsymbol{c}_{(i,j)^G}\boldsymbol{\Lambda}_{(i,j)^G}$. □

Now we can formalize the idea that we can also reduce the cost of a flow if we find the right negative cost augmentation cycle.

Lemma 3.8 (Negative cycle optimality condition)**.** *A feasible flow $\boldsymbol{f}$ is a min-cost flow if and only if $R(\boldsymbol{f})$ does not contain a cycle with negative cost.*

Proof. First assume that $\boldsymbol{f}$ is a feasible flow and $R(\boldsymbol{f})$ contains a cycle with negative cost. With lemma 3.7 we can find a augmentation cycle vector $\boldsymbol{\Delta} \subseteq G$ with a negative cost. With Lemma 3.1 we conclude that $\boldsymbol{c}^T(\boldsymbol{f} + \boldsymbol{\Delta}) < \boldsymbol{c}^T\boldsymbol{f}$. Therefore $\boldsymbol{f}$ is not a min-cost flow. Now let us assume $\boldsymbol{f}$ is a feasible flow and that there is another feasible flow $\boldsymbol{f}^*$ with $\boldsymbol{c}^T\boldsymbol{f}^* < \boldsymbol{c}^T\boldsymbol{f}$. Then $\boldsymbol{c}^T(\boldsymbol{f}^* - \boldsymbol{f}) < 0$. As shown after the definition 3.5, $\boldsymbol{\Lambda} := \boldsymbol{f}^* - \boldsymbol{f}$ is a circulation. With lemma 3.4 we can decompose $\boldsymbol{\Lambda}$ in augmentation circulation vectors. At least one of these augmentation cycles must have a negative cost, otherwise we can prove that the witness circulation has a positive cost using lemma 3.1. We can map this augmentation cycle back into the residual network as a directed cycle with negative cost using the reversed mapping of the one used in lemma 3.7. We do not prove that the reverse of this mapping works as we claim, but it is commonly used by Ahuja et al. □

We now have a simple way of finding a min-cost flow. By searching for negative cycles in the residual network of the current flow and adding the corresponding augmentation cycles to the current flow, we can decrease the cost of the flow until we can no longer find a negative cycle. In theory, we could find a min-cost flow faster by using cycles that give us the largest improvement. For this, we do not only want a negative cycle, but also one with a large residual capacity, so that we can scale the augmentation cycle before adding it. Therefore, we need to find a cycle C that minimizes

$$\sum_{(i,j)\in C} \boldsymbol{c}^r_{(i,j)} \min_{(i,j)\in C} \boldsymbol{r}_{(i,j)}.$$

This is an NP-complete problem ([AMO93, p. 319]). By instead looking for cycles that still effectively reduce cost, but are faster to find, it is possible to create an algorithm that runs in polynomial time.
One way of finding good negative cycles, is by looking for minimum mean cycles.

Definition 3.9. *Given a network with associated edge costs* $\boldsymbol{c}$*, then the* minimum mean cycle C *is the cycle that minimizes*

$$\frac{\sum_{(i,j)\in C} \boldsymbol{c}_{(i,j)}}{|C|}.$$

It is possible to identify a min-mean cost cycle in $\mathcal{O}(nm)$ [Kar78, p. 311]. In chapter 5 we also describe an algorithm for finding min-ratio cycles, which are generalized min-mean cycles. The MMCC algorithm searches for these cycles in the residual network. Every time a min-mean cycle is found, the corresponding augmentation cycle is scaled to its maximum so that some edge will reach its capacity constraints. With this, we can see, that this algorithm functions somewhat like the simplex algorithm 2.4.3, in the sense that it traverses the edges of the polytope until no edge/circulation can further decrease the cost. The algorithm runs in $\mathcal{O}(n^2m^2\log(mC)$ time. A full proof of this can be found in the work of Ahuja et al. ([AMO93, p 377-381]).

4. The Min-Cost Flow Algorithm by Li Chen et al.

Chen et al. describe the main parts of their algorithm in more than 45 pages ([AMO93, p.24-71]. Therefore, we cannot look at all of those parts in the same detail. Going a step further, the primary goal of this chapter is to give an overview of the different ideas and moving parts used in this algorithm. For this, we often leave out details or push it back to later sections where they might be easier to motivate.
For this, we first look at the general class of algorithms under which we can categorize the algorithm by Chen et al. In the second part of this chapter, we present the main ideas of their work. In the third part, we will go deeper into the details of a few select aspects of the algorithm to prove why the main concepts work.

Even so, it is important to highlight that practically all the ideas of this chapter and most of the proofs were originally done by Chen Li and his colleagues. We simply reorder and motivate these ideas, thereby trading the complete proof given by Chen et al. to hopefully make it more comprehensive. At some points, we also look at some extra lemmas that we use to explore the ideas in more detail or add some more detail to the proofs by Chen et al.

4.1 Interior Point Methods

With the simplex algorithm (2.4.3, we have already looked at a concept for solving an LP. The *interior point methods* (IPM) represent an entirely different approach. They are an umbrella term used for algorithms that share the same concept. Instead of traversing the edges and vertices of the polytope, they traverse the interior points of the feasibility region. The most well-known algorithm of this type is the Karmarkar Algorithm([Kar84]). This algorithm and the algorithm given by Chen et al. are both potential function algorithms. This means that there is a potential function Φ on which we perform some form of gradient decent. For this to be effective, the function must satisfy the two properties:

1. At the minimum of the function there must be an optimal solution. The further the input of the function is a way from this optimum, the higher the value of the function.
2. The function must punish extreme points close to violating some constraint. Visually speaking, it wants to keep the gradient descent away from the faces of the polytope as this might limit the step size in certain directions. Therefore, the function must return higher values when the input is close to a constraint.

We assume that the LP has a solution. Then, we can find an optimal point on a vertex. Keeping this in mind, the two properties seemingly work against each other. To ensure that this does not cause an issue during the gradient decent, the term corresponding to the second property is scaled by an α value.

The algorithm by Chen et al. uses the potential function

$$\Phi(f) \overset{def}{=} 20m \log(c^T f - F^*) + \sum_{e \in E}((u_e^+ - f_e)^{-\alpha} + (f_e - u_e^-)^{-\alpha})), \tag{4.1}$$

with $\alpha = \frac{1}{1000 \log mU}$ and F^* being the cost of a minimum flow. Of course, we do not know the value of F^*. To solve this, we simply use the algorithm in a binary search, where we guess the value of F^*. The first summand of this function $(20m \log(c^T f - F^*))$ returns a large value, if the current flow f has a large cost. This corresponds to property 1. The second summand $\sum_{e \in E}((u_e^+ - f_e)^{-a} + (f_e - u_e^-)^{-\alpha}))$ returns a large value if the flow is near edge capacities. This is what achieves property 2. So intuitively it makes sense that we can find a min-cost flow by using gradient decent on this function.

4.2 Reductions

The interior point method described by Chen et al. makes a few assumptions.

Initial Flow

We have explained that in the IPM, we want to decrease the value of the potential function. To do this, we first need to find an initial flow $\boldsymbol{f}^{(init)}$ to use as a starting point. This starting point must be within the capacity constraints, satisfy the demands and not be too far from the optimal solution. We go in depth into this in section 4.4. The idea behind it is, that we use the IPM on a modified instance of the min-cost flow. We obtain this modified version by adding a vertex, and edges from every other vertex to this vertex. In this way, we can take an arbitrary flow within the capacity constraints and fix the unfulfilled demands by routing flow over these added edges.

Minimum cost

To create the potential function (4.1), we need the cost F^* of a min-cost flow. The way we find this is by simply guessing it and running the IPM with that guess. Depending on the results, we guess smaller or bigger and run the IPM again. In this way, we perform a binary search for a value that the IPM needs with the IPM itself. The search range is $[-mU^2, mU^2]$ since U is the limit for cost and capacity and we have m edges. And because the min-cost flow is an integer flow and all costs are integers, F^* is an integer. Therefore the binary search takes $\log(2mU^2)m^{1+o(1)}$ under the assumption, that the IPM runs in $m^{1+o(1)}$ the overall algorithm runs in the same asymptotical time because of lemma 2.5, $\log(2mU^2)m^{1+o(1)} = m^{1+o(1)}$.

Rounding Flow

Chen et al. also assumed that the flow found by the IPM can be rounded to the correct integer flow. They prove the reason for this in Lemma 4.11 [CKL+22, Lemma 4.11].

4.3 Overview

The algorithm given by Chen et al. can be separated into two parts. One is the main IPM reducing the potential via min-ratio cycles. The other part explains, how these min-ratio cycles are found. This section serves as an overview of the algorithm by Chen et al., establishing the different ideas and lemmas used without going into to much detail why they work. Selected aspects will be discussed in more detail in the other sections.

4.3.1 Reducing the potential

Definition 4.1. *Let $(G, \boldsymbol{d}, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, U)$ be an instance of the mcf-problem as usual as well as a flow $\boldsymbol{f} \in \mathbb{R}^E$, then*

$$\boldsymbol{g}(\boldsymbol{f})_e = (\nabla\Phi(\boldsymbol{f}))_e = 20m(\boldsymbol{c}^T\boldsymbol{f} - F^*)^{-1}\boldsymbol{c}_e + \alpha(\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-1-\alpha} - \alpha(\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-1-\alpha}$$

and is the gradient *of the $\boldsymbol{f}$ on the edge $e \in E(G)$ and*

$$\boldsymbol{l}(\boldsymbol{f})_e = (\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-1-\alpha} + (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-1-\alpha}$$

is the length*.*

If there is only one flow in question, we often only use $\boldsymbol{g}$ for $\boldsymbol{g}(\boldsymbol{f})$ and $\boldsymbol{l}$ for $\boldsymbol{l}(\boldsymbol{f})$. To augment the current flow without changing the generated supply, we need to use circulations, as shown in Lemma 3.1. The algorithm uses the gradients and lengths to determine a good augmentation cycle.

Definition 4.2. *Given $G = (V, E)$ be a graph with the edge-vertex incidence matrix $\boldsymbol{B}$ as well as lengths $\boldsymbol{l} \in \mathbb{R}^E_{>0}$, and costs $\boldsymbol{g} \in \mathbb{R}^E \setminus \{\boldsymbol{0}\}$. The minimum ratio cycle is the augmentation cycle $\boldsymbol{\Delta} \in \mathbb{R}^{E(G)} \setminus \{\boldsymbol{0}\}$ which minimizes the ratio of the gradient $\boldsymbol{g}^T\boldsymbol{\Delta}$ in relationship to the length $\boldsymbol{l}^T\boldsymbol{\Delta}$ ([AMO93, 150]). Or formally*

$$\min_{\boldsymbol{B}^T\boldsymbol{\Delta}=0} \frac{\boldsymbol{g}^T\boldsymbol{\Delta}}{||\boldsymbol{L}\boldsymbol{\Delta}||_1},$$

where $\boldsymbol{L} = diag(\boldsymbol{l})$.

In the literature a name that is commonly used for this concept, is *minimum cost-to-time ratio cycle*. One important thing to note here is that the direction of the edges of the augmentation cycle only interacts with the gradients, like in a residual network where we used the gradients instead of the costs. A result of this is the following lemma.

Lemma 4.3. *Given a graph G with the gradients $\boldsymbol{g} \in \mathbb{R}^{E(G)} \setminus$ and lengths $\boldsymbol{l} \in \mathbb{R}^{E(G)}_{>0}$ and the min ratio cycle $\boldsymbol{\Delta} \in \mathbb{R}^{E(G)}$, then*

$$\frac{\boldsymbol{g}^T\boldsymbol{\Delta}}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} \leq 0.$$

Proof. Assume the Δ has a positive ratio. Then

$$\frac{\boldsymbol{g}^T\boldsymbol{\Delta}}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} > -\frac{\boldsymbol{g}^T\boldsymbol{\Delta}}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} = \frac{\boldsymbol{g}^T(-\boldsymbol{\Delta})}{||\boldsymbol{L}(-\boldsymbol{\Delta})||_1}.$$

This is a contradiction since $\boldsymbol{\Delta}$ has the smallest ratio □

An interesting parallel to the MMCC algorithm is that the concept of min-ratio cycles is a generalization of the minimum-mean cycle. If we solve a min-ratio cycle problem where for all edges $e \in E(G)$ $\boldsymbol{l}_e = 1$, it is the same as looking for a min-mean cycle with $\boldsymbol{c} = \boldsymbol{g}$ (3.9).

Visual representation

Compared to the MMCC algorithm we no longer traverse the edges of the polytope but stay inside the interior. We can visualize this idea by extending the polytope by another dimension, which we can imagine as a height, given by the value of the potential function for each point/flow. We can interpret circulations as pointing in some kind of direction in the polytope. We can now ask if we see a decrease in height by moving the flow in one of these directions. By using the gradient, we can determine the directions in which we the height decreases the fastest. We might imagine this by finding a way down into a valley by going down the steepest slopes. When looking at the runtime and correctness, this of course depends on both the shape of the mountains around the value (are there any plateaus, or false minima?) and the size of the steps we take (we want to make sure to not step outside the polytope). The second part is partially achieved by scaling the min-ratio cycles depending on their lengths. A full proof that this works in $m^{o(1)}$ iterations can be found in [CKL+22, Lemma 4.4].

4.3.1.1 Dynamic trees

The current flow of the IPM is stored in dynamic trees. In section 4.3.2.2 we see the motivation behind using multiple dynamic trees. This data structure is used in other parts of the algorithm, as well and we will not explain the operations of these trees, because they are the standard implementation as described by Daniel D. Sleator and Robert Endre Trajan [SE83] with one extra functionality. In summary, almost all operations that the algorithm by Chen et al. performs on the dynamic trees, run in an amortized $\widetilde{\mathcal{O}}(1)$. These operations include edge insertions/deletions and updates to gradients, lengths and flow. The only operation that differs, is the added functionality called DETECT(). The Motivation behind the function is that the values of that gradients and lengths will only change slowly over the course of the algorithm. In this way, they only need to be recalculated periodically. Detect returns a subset S of edges of the dynamic tree, whose flow has changed significantly, since the last time it was returned in $\widetilde{\mathcal{O}}(s)$ time, where [CKL+22, Lemma 3.3].

4.3.2 How to find min-ratio cycles

The part of the algorithm that finds the min-ratio cycle is the HSFC data structure. We interpret the HSFC data structure as a wrapper around the (B)-branching tree-chain. They contain most of the complexity of the algorithm by Chen et al. We will first take a superficial look at this data structure and then break down the components from which it is composed.

4.3.2.1 HSFC data structure as a black box

Let us first explain the name. HSFC is an abbreviation of *hidden stable flow chasing.* This is used to describe updates on the graph, gradients, and lengths that satisfy certain constraints. These constraints are in place so that we can prove some properties of the *width* later on. The HSFC data structure receives these updates but it also generates these updates in a recursive manner to maintain its internal structure.

The HSFC data structure, as defined by Chen et al., maintains a set of dynamic trees and supports two operations ([CKL+22, Theorem 6.2]). Both exactly once during every iteration of the IPM. We assume that the algorithm runs for τ cycles with t as the current cycle. In our explanation however, we try to remove references to the current time t. At a point, the defined operations refer to the *width* $\boldsymbol{w}^{(t)}$ which are an overestimate for $\boldsymbol{\Delta}^{*,(t)} \circ \boldsymbol{l}^{(t)}$ where $\boldsymbol{\Delta}^{*,(t)}$ is the current witness circulation as defined in 3.5.

- *UPDATE*($U^{(t)}, \boldsymbol{g}^{(t)}, \boldsymbol{l}^{(t)}$): Updates gradients and lengths to $\boldsymbol{g}^{(t)}$ and $\boldsymbol{l}^{(t)}$. $U^{(t)}$ only contains edge insertions/deletions, and together with $\boldsymbol{g}^{(t)}, \boldsymbol{l}^{(t)}$ these update must satisfy the HSFC constraints. It also must be the case, that for $\alpha = 1/1000 \log(n)$,

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{||\boldsymbol{w}^{(t)}||_1} \leq -\alpha.$$

- *QUERY*() : Returns a tree T_i corresponding to one of the dynamic trees and cycle $\boldsymbol{\Delta}$ represented as $m^{o(1)}$ off tree edges. This cycle fulfills the following property for $\kappa = \exp(-\mathcal{O}(\log^{7/8} m \cdot \log\log m))$,

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{L}^{(t)} \boldsymbol{\Delta}||_1} \leq -\kappa\alpha.$$

The runtime of these operations summed up over all cycles is $m^{o(1)}(m + Q)$ with $Q = \sum_{t \in [\tau]} |U^{(t)}|$ the overall amount of individual updates applied to the edges. Without going into too much detail $Q = m^{o(1)}$ so together the run time is $m^{1+o(1)}$ over all iterations.

In the extra condition for the update, the $-\alpha$ puts an upper bound to the ratio of the witness circulation. In the query we see, that the returned cycle can push this bound up to a factor of κ. So κ can be somewhat interpreted as an approximation factor.

The tree returned, corresponds with one of the trees we described in section 4.3.1.1.

4.3.2.2 Tree-chain motivation

In the overview, Chen et al. describe a static algorithm that finds an $m^{o(1)}$ approximate min-ratio cycle [CKL+22, p. 7,8]. They use a probabilistic low stretch spanning tree T of G. Let us first look at what probabilistic low stretch means for a spanning tree. The parameter that we stretch is the length $\boldsymbol{l}_{(u,v)}$ of an edge $(u, v) \in E(G)$ when routing one unit of flow from u to v inside T. Thus, the stretch of (u, v) is

$$\mathbf{str}^{T,\boldsymbol{l}}_{(u,v)} = 1 + \frac{\sum_{e \in T[u,v]} \boldsymbol{l}_e}{\boldsymbol{l}_{(u,v)}}, \tag{4.2}$$

where $T[u, v]$ is the unique u-v path in a tree. In a probabilistic stretch tree, this stretch is expected to be $\widetilde{\mathcal{O}}(1)$. We can find such a tree in $\widetilde{\mathcal{O}}(m)$ time [AN19, p. 227]. Next, they want to find a good approximation of the witness circulation of the fundamental tree cycles. A fundamental tree cycle is created when combining an edge $(u, v) \notin E(T)$ with the unique path $T[u, v]$. Chen et al. now argue, that with a probability of at least $1/2$ that a random tree like this does not stretch the witness circulation too much. They conclude, that with a probability greater or equal than $1/2$, we are able to find a cycle that is a $\widetilde{\mathcal{O}}(1) = m^{o(1)}$ approximation to witness circulation. We can boost this probability by simply sampling more independent trees. The issue with this algorithm is that the lengths change every time, a circulation is added to the flow. Therefore, the trees might not be low stretch anymore and rebuilding even a single tree each iteration would lead to an overall runtime of $\Omega(m^2)$.

Chen et al. solved this problem by using so called *tree-chains*. A tree-chain is a sequence of graphs $G_0, \ldots, G_d$, where a graph G_i recursively constructed by taking a specific spanner $\mathcal{S}(G_{i-1}, F)$ of $\mathcal{C}(G_{i-1}, F) = G_{i-1}/F$ where $F \subseteq G_i$ is a spanning forest. Before even defining the tree-chain formally, we want to establish the relationship between a specific spanning tree T of G and the sequence of forests $F_1, \ldots, F_d$ used in a tree-chain. For now, let us look a tree-chain of length two, with one forest and one tree.

Lemma 4.4. *Given a graph $G = (V, E)$ with a spanning forest $F \subseteq G$. Let $\mathcal{S}(G, F)$ be some a spanner on G/F, removing at least all self-loops. Then we can map every spanning trees on $\mathcal{S}(G, F)$ to a unique and the spanning trees of G whose edges are all in $F \cup \mathcal{S}(G, F)$.*

To make sense of the $F \cup \mathcal{S}(G, F)$, it is important to mention, that when needed, we identify edges in $\mathcal{S}(G, F)$ with their preimages in G.

Proof. Let $\overline{T} \subseteq \mathcal{S}(G, F)$ be a spanning tree. Then, we can map it to $T = (V, E(F) \cup E(\overline{T}))$.

First, we show that T is a spanning tree. Assume T is not connected. Then, there must be two components in F which cannot be connected by adding edges $E(\overline{T})$. Since these components correspond with vertices in $G\backslash F$, we know $\overline{T}$ would not be connected and therefore not a spanning tree, which is not the case. Now lets assume T contains a circle. We have two different cases. First assume the circle only contains vertex of one component in F. Since $\mathcal{S}(G, F)$ removed all self-loops of G/F, the cycle cannot be inside a single component of F. Thus the cycle must pass through multiple components each corresponding to a vertex in $V(\mathcal{S}(G, F)$. These vertices must form a cycle in $\overline{T}$ as well. This is a contradiction. Therefore T must be a tree. Now to show that T is spanning G, let k be the amount of components in F. Since F is a forest, we can use the basic graph theory result $|E(F)| = |V(F)| - k = |V(G)| - k$ together with $|E(\overline{T})| = k - 1$ and that fact that T is a tree, to conclude that T spans $|E(T)| + 1 = |E(F)| + |E(\overline{T})| + 1 = |V(G)| - k + k - 1 + 1 = |V(G)|$ vertices.

To prove that this mapping is unique, we prove the injectivity of it. Let $\overline{T}_1, \overline{T}_2 \subseteq \mathcal{S}(G, F)$ be spanning trees with $\overline{T}_1 \neq \overline{T}_2$. Then since there must be at least one edge that the trees do not have in common, $T_1 = (V, E(F) \cup E(\overline{T}_1)) \neq (V, E(F) \cup E(\overline{T}_2)) = T_2$.

□

If we now look at a tree-chain of length d, then by applying the lemma above recursively, we know, that we can identify a spanning tree of G by the unification of the forests used for the core graph construction (By building a tree from F_d and F_{d-1}, which we can then use to build a tree with $F_{d-2}, \ldots$). This will allows us to maintain the low stretch property on different levels, where lower levels can be rebuilt without impacting the upper levels. We will also generalize the concept of fundamental tree cycles to *fundamental chain cycles* in section 4.7.1. Of course for this to be possible, the forest and spanners must be selected and maintained in a very specific way.
As before, we do only guarantee the probabilistic stretch of the witness circulation in these trees. Instead of using multiple independent tree-chains, we take a singular chain and branch it at every level. So $B = \mathcal{O}(\log n)$ forests for every graph in the preceding level. This way the branching tree-chain itself has once again a tree structure with $B^d = s$ leaf vertices. And with every one of these leafs we have a corresponding tree, on which the IPM maintains a part of the current flow.

4.3.2.3 Definitions

In the section we will formally define the branching tree-chain. We also look at how the gradients and lengths are mapped through the different levels. We also refer to a lot of trees and rooted forests with an associated stretch in this section. We give a formal definition of these in section 4.3.2.4 dots as well look at how they are created.

Definition 4.5 (Core Graph). *Given a graph G and a rooted spanning forest and tree $F \subseteq T$ on G, with stretch overestimates $\widetilde{\boldsymbol{str}}_e$. The* core graph $\mathcal{C}(G, F)$ *is the contraction*

G/F, with lengths $\boldsymbol{l}^{\mathcal{C}(G,F)} \in \mathbb{R}_{>0}^{E(\mathcal{C}(G,F))}$ and gradients $\boldsymbol{g}^{\mathcal{C}(G,F)} \in \mathbb{R}^{E(\mathcal{C}(G,F))}$. For every $\hat{e} \in E(G/F)$ with preimage $(u,v) = e \in E(G)$, the length is $\boldsymbol{l}_{\hat{e}}^{\mathcal{C}(G,F)} \overset{def}{=} \widetilde{\boldsymbol{str}}_e \boldsymbol{l}_e$ and the gradient is$\boldsymbol{g}_{\hat{e}}^{\mathcal{C}(G,F)} \overset{def}{=} \boldsymbol{g}_e + \langle \boldsymbol{g}, \boldsymbol{p}(T[v,u]) \rangle$.

$\boldsymbol{p}$ is the function mapping a sequence of edges to the flow, routing one unit overe these edges.

We will treat the sparsification process as a block box. For completion sake, we will list the same definition for the sparsified core graph as given by Chen et al. even though, we have not defined some of the conecpts and are mostly not using them in this thesis. We will informally explain them below. Up front one funtion, that is going to come up again is the embedding $\boldsymbol{\Pi}_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}$ which maps every edge from $E(\mathcal{C}(G,F))$ to a path in $\mathcal{S}(G,F)$. This includes the self-loops, which are mapped to empty paths.

Definition 4.6 (Sparsified core graph)**.** *Given a graph G, forest F and a parameter reduction parameter $k \in \mathbb{R}_{>0}$ A subgraph $\mathcal{S}(G,F) \subseteq \mathcal{C}(G,F)$ is a $(\gamma_s, \gamma_c, \gamma_l)$-*sparsified core graph *with embedding as a subgraph and embedding $\Pi_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}$, if it satisfies*

1. *For any $\hat{e} \in E(\mathcal{C}(G,F))$, all edges $\hat{e} \in \Pi_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}(\hat{e'})$ satisfy $\boldsymbol{l}_{\hat{e}}^{\mathcal{C}(G,F)} \approx_2 \boldsymbol{l}_{\hat{e'}}^{\mathcal{C}(G,F)}$.*
2. *length$(\Pi_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)} \leq \gamma_l$ and econg$(\Pi_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}) \leq k\gamma_c$.*
3. *$\mathcal{S}(G,F)$ has at most $m\gamma_s/k$ edges.*
4. *The lengths and gradients of edges in $\mathcal{S}(G,F)$ are the same as in $\mathcal{C}(G,F)$*

In summary, the sparsified core graph is a spanner of the core graph depending on the values $\gamma_s, \gamma_c, \gamma_l \in \mathbb{R}_{>0}$. γ_s is self-explanatory part of the of the upper bound $m\gamma_s/k$ of the edge amount. k is the reduction factor which well be used at global level. Here it will be set to $k = m_0^{1/d}$ where m_0 is the amount of edges of top level graph and $d = \mathcal{O}(\log n)$ being the depth of the branching tree-chain, which we are yet to formally define. γ_l is an upper bound for maximum amount of edges in a path that $\Pi_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}$ maps to. γ_c together with the reduction factor forms an upper bound for the maximal edge congestion in $\mathcal{S}(G,F)$. The congestion of an edge $e \in \mathcal{S}(G,F)$ is the amount of edges in $\mathcal{C}(G,F)$ mapped to a path containing e. The other points are straight forward.

Definition 4.7 (Tree-chain)**.** *Let G be graph. Then the sequence of graphs $G_0, \ldots, G_d$ a* tree-chain *with the corresponding forests $F_1, \ldots, F_d$ is a* tree-chain *if $G_0 = G$ and $G_i = \mathcal{S}(G_{i-1}, F)$ for $i \in \{1, \ldots, d\}$.*

Note, that to make sure, we can construct a spanning tree of G out of the spanning forests, the forest on the lowest level must be a tree.

With this we can now finally define the *B-branching tree chain*

Definition 4.8 (B-branching tree-chain)**.** *For a graph G, parameter k, and branching factor B a B-*branching tree-chain *consists of collections of graphs $\{\mathcal{G}_i\}_{0\leq i\leq d}$, such that $\mathcal{G}_0 = \{G\}$ and recursive definition of $\mathcal{G}_i$ $i \in \{1, \ldots, d\}$ inductively as follows.*

1. *For each $G_i \in \mathcal{G}_i, i < d$ is a collection of B trees $\mathcal{T}^{G_i} = \{T_1, T_2, \ldots, T_B\}$ and a collection of B forests $\mathcal{F}_i^G = \{F_1, F_2, \ldots, F_B\}$ such that $E(F_j) \subseteq E(T_j)$ which F_j, T_j being a LSD of G_i.*

2. *For each $G_i \in \mathcal{G}_i$ and $F \in \mathcal{F}^{G_i}$ there are $\gamma_s, \gamma_c, \gamma_l$-sparsified core graphs and embedding $\mathcal{S}(G_i, F)$ and $\Pi_{\mathcal{C}(G_i,F)\to\mathcal{S}(G_i,F)}$.*
3. *Set $G_{i+1} = \{\mathcal{S}(G_i, F) : G_i \in \mathcal{G}_i, F \in \mathcal{F}^{G_i}\}$.*

For every $G_d \in \mathcal{G}_d$ we maintain a low-stretch tree T.

At this point we have not yet introduced the LSD or *low stretch decomposition.* This will be explained in chapter 4.3.2.4.

4.3.2.4 Low stretch decomposition

The algorithm we described to motivate the HSFC data structure used a low stretch tree to find a good witness circulation approximation. With the HSFC data structure, we are no longer working with trees but with forests, which will once again have a low stretch. With forests, we cannot guarantee that there is a path between every two vertices. Therefore, we must find a different way of thinking about the stretch. Chen et al. do this by introducing *rooted spanning forests.*

Definition 4.9 (Rooted Spanning Forrest)**.** *Given a graph G, a* rooted spanning forest *$F \subseteq G$ is a forest spanning G in which every component of F has a specified vertex called the root. For $v \in V(G)$, $root_v^F$ denotes the root of the component to which v belongs in F.*

In other words, a rooted spanning forest is a forest of rooted trees.

Definition 4.10 (Stretch of a forest)**.** *Given a rooted spanning forest F of the graph G with lengths $\boldsymbol{l} \in \mathbb{R}_{>0}^{E(G)}$. The* stretch *of an edge $e = (u, v) \in E(G)$ is*

$$\boldsymbol{str}_e^{F,\boldsymbol{l}} = \begin{cases} 1 + \langle \boldsymbol{l}, |\boldsymbol{p}(F[u,v])| \rangle / \boldsymbol{l}_e & \text{if } root_u^F = root_v^F \\ 1 + \left\langle l, |\boldsymbol{p}(F[u,\ root_u^F\ ])| + |\boldsymbol{p}(F[v,\ root_v^F\ ])| \right\rangle / \boldsymbol{l}_e & \text{if } root_u^F \neq root_v^F \end{cases}.$$

If u, v are in the same component of F, this definition coincides with the definition of stretch for a tree (4.2). The reason why the path to the roots is important becomes clear once we discuss how the min-ratio cycles found in the lower levels of the branching tree-chain are brought to the upper levels, in section 4.7.2. The basic idea is that when lifting a cycle, the preimages of its edges might not form a cycle (their end and starting vertex might not be the same, when not contracted). Hence, we add a path over the roots to create a pre-image cycle. We can now consider what this forest stretch of an edge (u, v) is in the context of the branching tree chain. If (u, v) are not in the same component int F, then e might reappear in the next level. There it is stretched again. At some point (assuming e is not deleted by the sparsifier), u and v will be in the same component. Therefore, we can simply use the tree stretch. If we now lift up the cycle using the mapping described earlier, then we obtain a path that is in the tree corresponding to the tree-chain, because we only use edges in F. In this way, the stretch of a forest, can be described as part of the stretch of edge e, which needs to reconnect cycles from the core graph.

Chen et al. explain the creation and maintenance of forests in two lemmas. Lemma 6.5 describes how a singular spanning forest (as a subgraph of a spanning tree) with certain properties is created and maintained. Lemma 6.6 expands lemma 6.5 by using a multiplicative weight update procedure to effectively create a set of spanning forests (and spanning trees) whose expected stretch is bounded by a certain value. Chen et al. use this set to sample independent forest, to improve the probability to get a low stretch forest. Now, let us take a closer look at the properties defined in lemma 6.5.

Lemma 4.11 (Dynamic low stretch decomposition (incomplete))**.** *There is a deterministic algorithm with a total runtime of $\widetilde{\mathcal{O}}(m)$ that, on a graph $G = (V, E)$ with lengths $\boldsymbol{l} \in R^E_{>0}$ and reduction factor k, can compute a tree T spanning V and a rooted spanning forest $F \subseteq T$ with $\mathcal{O}(m/k)$ components, and stretch overestimates $\widetilde{\boldsymbol{str}}$. Furthermore, the algorithm maintains F decrementally against batches of updates, with $\widetilde{\boldsymbol{str}}_e = 1$ for any new edge that is added by vertex split or edge insertion.*

This is only the first part of lemma 6.5 ([CKL+22, lemma 6.4]. We now informally describe more properties of the algorithm at the center of the lemma. The stretch overestimates will always be greater or equal than the real stretch for each edge. At the same time, they are bounded by a global constant. At the initialization of the LSD, F has $\mathcal{O}(m/k)$ components. With every update, the number of components increases by $\widetilde{\mathcal{O}}(1)$. This is because we want to keep the stretch below some constant, and we can simply do this by deleting the edges of F. In an empty tree, the stretch of every edge would be 1. Of course, this does not help us, but it helps us to think about how deleting edges reduces the stretch by handing it to the lower levels of the tree-chain. Chen et al. describe how this is achieved in their appendix [CKL+22, p. 94-98].

4.3.2.5 Algorithm stack

This section links the different parts of the HSFC data structure to the lemmas by Chen et al., describing how they are maintained. Every paragraph in this section describes a different level of maintenance, where the lower paragraphs are part of the maintenance process of the upper paragraphs.

Maintaining the HSFC data structure

We have already discussed the interface of the HSFC data structure in section 4.3.2.1. Internally, updates to the gradients and lengths are simply passed to the branching tree-chain. After updating, the branching tree-chain returns its best guess for the min-ratio cycle. It might be the case, that the ratio of this cycle is too large. The result of this, is that parts of the branching tree-chain need to be rebuilt, until the cycle found has a small enough ratio and is returned in the QUERY function of the HSFC data structure. We describe parts of this process in section 4.8. Updating, rebuilding and finding the min-ratio cycle takes cumulative time $m^{1+o(1)}$ during the IPM.

Maintain the branching tree-chain

The branching tree-chain can be updated, queried for a min-ratio cycle, and prompted to rebuild all levels below some level i. The update recursively generates and applies HSFC updates throughout the levels. The min-ratio cycle is found by continuously maintaining a maximizer that maximizes the ratio of the gradient and a length overestimate for cycles corresponding to off-tree edges of the trees corresponding to the s tree-chains. More on this in section 4.7.1. The rebuilding functionality simply takes the level i of the branching tree-chain to rebuilds by reinitializing all forest, trees, and core graphs and sparsified core graphs in and below. This rebuild are made in done in $m^{1+o(1)}/k^i$ and are not only triggered at initialization and through the rebuilding game but also periodically after a certain amount of updates, because the forests have deleted to many edges. Most of these ideas can be found in Lemma 7.9, algorithm 5, and Theorem 7.1 of Chen et al. [CKL+22]. The cumulative time for updates, queries, and rebuilds is $m^{1+o(1)}$ during the IPM.

Maintain a set of sparsified core graphs

This process maintains a set of sparsified core graphs. Here, we examine some graph G inside the branching tree-chain. When the branching tree-chain is initialized or rebuilt, this process creates the sparsified core graphs $\mathcal{S}(G, F_1), \dots, \mathcal{S}(G, F_B)$ (and their embeddings) with the forest described in section 4.3.2.4. It also describes what handles updates to G. This process then generates the corresponding updates that need to be applied to $\mathcal{S}(G, F_1), \dots, \mathcal{S}(G, F_B)$ and their embedding. As mentioned, we will not analyze the sparsified core graphs. The detailed algorithm can be found in Lemma 7.8 in the work of Chen et al. ([CKL+22, Lemma 7.8]) and has a run time of $\widetilde{\mathcal{O}}(mk)$.

Maintain a set of core graphs

As with the above process, the core graphs $\mathcal{C}(G, F_1), \dots, \mathcal{C}(G, F_B)$ are generated from some graph G in the branching tree-chain. It handles updates by interfacing with the algorithm described in the lemma 4.11. For example, whenever an edge is updated, it is deleted in the forests $F_1, \dots F_B$. To maintain the corresponding core graphs as contractions, they undergo vertex splits. The detailed algorithm can be found in Lemma 7.5 in the work of Chen et al. ([CKL+22]) and has a runtime of $\widetilde{\mathcal{O}}(mk)$

4.4 Initial Point

For the gradient decent, we need some initial point/flow $\boldsymbol{f}_e^{(init)} \in \mathbb{R}^{E(G)}$. We also want to ensure that this initial point is not potentially in the polytope vertex that is the furthest away from the optimal solution. An initial idea might be to place the initial flow in the center of the polytope. For this, we set $\boldsymbol{f}_e^{(\text{init})} = \frac{\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-}{2}$ for every edge $e \in E(G)$. This, of course, does not guarantee that the demands are met. We can circumvent this by solving the MCF problem on a modified version of the network $\widetilde{M}$, in which there is a new vertex v^* as well as new edges between v^* and all of the vertices with an incorrect supply. In this way, we can fix the supply by routing surplus/deficiency over v^* and create an initial feasible flow on the modified network. If we set the costs on the added edges sufficiently high, an MCF algorithm will avoid routing over these added edges. If the calculated min-cost flow $\widetilde{\boldsymbol{f}}$ on $\widetilde{M}$ does not route flow over any edges not found in G, it is easy to reconstruct a minimum cost flow f in the original network. We now give a more formal construction of $\widetilde{M}$:

Given an instance $M = (G, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$ of the MCF problem with $G = (V, E)$, we can construct $\widetilde{M} = (\widetilde{G}, \widetilde{\boldsymbol{c}}, \widetilde{\boldsymbol{u}^+}, \widetilde{\boldsymbol{u}^-}, \widetilde{\boldsymbol{d}}, \widetilde{U})$ and the initial flow $\boldsymbol{f}^{(\text{init})}$ in the following manner:

1. We first define the pseudo-flow $\boldsymbol{f'}$ on M, by setting $\boldsymbol{f}'_e = \frac{\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-}{2}$ for all $e \in G$.
2. Next we compute the supply $\overline{\boldsymbol{d}}$ generated by the pseudo-flow $\overline{\boldsymbol{d}} = \boldsymbol{B}^T \boldsymbol{f}'_{|E(G)}$ as well as the difference to the original demands $\hat{\boldsymbol{d}} = \overline{\boldsymbol{d}} - \boldsymbol{d}$.
3. We create $\widetilde{G}$ by setting $V(\widetilde{G}) = V \cup \{v^*\}$ and $E(\widetilde{G}) = E \cup E_{\text{new}}$ where $E_{\text{new}} = \{(v, v^*) \mid v \in V \wedge \hat{\boldsymbol{d}}_v > 0\} \cup \{(v^*, v) \mid v \in V \wedge \hat{\boldsymbol{d}}_v < 0\}$.
4. Now we set $\widetilde{\boldsymbol{u}}_e^- = 0, \widetilde{\boldsymbol{u}}_e^+ = 2|\hat{\boldsymbol{d}}_e|, \widetilde{\boldsymbol{c}}_e = 4mU^2$ for every $e \in E_{\text{new}}$ as well as $\widetilde{\boldsymbol{d}}_{v^*} = 0$. For all edges $e \in E(G)$ we keep the values by setting $\widetilde{\boldsymbol{u}}_e^- = \boldsymbol{u}_e^-, \widetilde{\boldsymbol{u}}_e^+ = \boldsymbol{u}_e^+, \widetilde{\boldsymbol{d}}_e = \boldsymbol{d}_e, \widetilde{\boldsymbol{c}}_e = \boldsymbol{c}_e$.
5. We create the initial
$$f^{(\text{init})}_{(u,w)} = \begin{cases} |\hat{d}_u| & \text{if } w = v^* \\ |\hat{d}_w| & \text{if } u = v^* \\ \boldsymbol{f}'_{(u,v)} & \text{otherwise} \end{cases}$$
for all $e \in E(\widetilde{G}$.

Lemma 4.12. *In reference to the construction above, the following is true:*

1. $\boldsymbol{f}^{(init)}$ *is a feasible flow on* $\widetilde{M}$.
2. *Let* $\boldsymbol{f}^*$ *be a MCF on* $\widetilde{M}$ *then* $\boldsymbol{f}^*_{|E(G)}$ *is a MCF on* M *if and only if,* $\boldsymbol{f}^*_e = 0$ *for all* $e \in E_{new}$.
3. $\Phi(\boldsymbol{f}^{(init)}) \leq 200m\log(mU)$.

Proof:

1. If we assume that $\boldsymbol{f}^*$ is a min-cost flow on $\widetilde{M}$ and $\boldsymbol{f}^*_e = 0$ for all $e \in E_{\text{new}}$, then $\boldsymbol{f}^*_{|E(G)}$ is also feasible on M. The reason for this is that we did not change any capacities from $E(G)$ so $\boldsymbol{f}^*_{|E(G)}$ must be inside the capacity constraints. We also did not change the demands of the vertices $V(G)$, and if there is no flow on any edge $e \in E_{\text{new}}$, then for every vertex $v \in V(G) : (\widetilde{\boldsymbol{B}}^T \boldsymbol{f}^*)_v = (\boldsymbol{B}^T \boldsymbol{f}^*_{|E(G)})_v = \boldsymbol{d}_v$. We now show that $\boldsymbol{f}^*_{|E(G)}$ is also minimal. Assume that there is a flow $\boldsymbol{f}'$ on M with a cost less than $\boldsymbol{f}^*_{|E(G)}$. Then, the flow is defined by

$$\boldsymbol{f}_{(u,w)} = \begin{cases} 0 & \text{if } v^* \in (u,w) \\ \boldsymbol{f}'_{(u,v)} & \text{otherwise} \end{cases},$$

   is feasible on $\widetilde{M}$ with a cost equal to $\boldsymbol{f}'$, which is lower than the cost of $\boldsymbol{f}^*$. This is a contradiction.

   Conversely let $\boldsymbol{f}^*_{|E(G)}$ be a minimum cost flow on M. Then, the total costs are bounded by mU^2 since there are m edges with capacity and cost both being bound by U. We know that setting $\boldsymbol{f}_e = 0$ for all $e \in E_{\text{new}}$ results in a feasible flow with costs smaller or equal to mU^2. We now only need to show that setting $\boldsymbol{f}_e \neq 0$ for these edges results in a flow of higher cost. For this, note that the lower constraints force $\boldsymbol{f}_e \geq 0$. As mentioned, there exists an integer min-cost flow for our given problem. Therefore, any potential min-cost flow using $e \in E_{\text{new}}$ would need to route at least one whole unit of flow over e, thereby adding $4mU^2$ to the cost. This is higher than setting all $\boldsymbol{f}_e = 0$ for the new edges.

   A direct consequence of this is, that given a min-cost flow $\boldsymbol{f}^*$ on $\widetilde{M}$ with at least one edge $e \in E_{\text{new}}$ with $\boldsymbol{f}^*_{(\text{init})} > 0$, we can conclude that there is no minimum or feasible flow on M.

2. We know $|E(\widetilde{G})| \leq 2m$ as well as that the maximum cost of an edge is $4mU^2$. The maximum amount of flow routed over an edge is bounded by mU. So we can conclude that $\boldsymbol{c}^T \boldsymbol{f}^{(\text{init})} \geq 2m \cdot 4mU^2 \cdot mU$. We also know that $F^* \geq -mU^2$. Therefore, we can say the following about the first summand of the potential (4.1):

$$\begin{aligned} 20m\log(\boldsymbol{c}^T \boldsymbol{f}^{(init)} - F^*) &\leq 20m\log(8m^3U^3 + mU^2) \\ &\leq 20m\log(9m^3U^3) \\ &= 20m(\log(9) + 3\log(mU)) \\ &\leq 80\log(mU) \end{aligned}$$

   For the second summand of the potential, we know that $\alpha > 0$. Therefore, we can set an upper bound for $\left(\frac{1}{2}\right)^{-\alpha}$ by using the minimum value that $(\boldsymbol{u}^+_e - \boldsymbol{f}_e)$ is going to be, as an upper bound. Since $\boldsymbol{f}_e$ is half-integer and $\boldsymbol{u}_e^+ > \boldsymbol{u}_e^-$, we know that $(\boldsymbol{u}_e^+ - \boldsymbol{f}_e) = (\boldsymbol{f}_e - \boldsymbol{u}_e^-) \leq \left(\frac{1}{2}\right)$ and therefore $(\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-\alpha} = (\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-\alpha} \leq \left(\frac{1}{2}\right)^{-\alpha}$. Now, we can now conclude the following:

$$\begin{aligned}\sum_{e\in E}((\boldsymbol{u}_e^+ - \boldsymbol{f}_e^{(init)})^{-\alpha} + (\boldsymbol{f}_e^{(init)} - \boldsymbol{u}_e^-)^{-\alpha}) &\leq 2m(\left(\frac{1}{2}\right)^{-\alpha} + \left(\frac{1}{2}\right)^{-\alpha}) \\ &\leq 4m2^{\alpha} \\ &\leq 8mU \qquad\qquad \leq \log(8mU)\end{aligned}$$

Together, it follows that

$$\begin{aligned}\Phi(\boldsymbol{f}^{(\text{init})}) &= 20m\log(\boldsymbol{c}^T\boldsymbol{f}^{(init)} - F^*) + \sum_{e\in E}((\boldsymbol{u}_e^+ - \boldsymbol{f}_e^{(init)})^{-\alpha} + (\boldsymbol{f}_e^{(init)} - \boldsymbol{u}_e^-)^{-\alpha}) \\ &\leq 88\log(mU) \leq 100\log(mU).\end{aligned}$$

4.5 Interpreting gradients and lengths

In section 4.3.1 we have introduced the gradients and lengths of a flow $\boldsymbol{f}$ on an instance of the MCF problem $(G, \boldsymbol{c}, \boldsymbol{u}^+, \boldsymbol{u}^-, \boldsymbol{d}, U)$. We have seen that the gradient of an edge can be used to determine whether the flow on an edge must increase or decrease. We show that we need more than the gradients to effectively reduce the potential and explain why the lengths fill this role. As a reminder, the gradient of and edge $e \in E(G)$ is

$$\boldsymbol{g}_e = 20m(\boldsymbol{c}^T\boldsymbol{f} - F^*)^{-1}\boldsymbol{c}_e + \alpha(\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-1-\alpha} - \alpha(\boldsymbol{f}_e - \boldsymbol{u}_e^-)^{-1-\alpha}.$$

We have introduced the potential function with two properties: 1. A minimum at the optimal solution. 2. Punishing flows coming close to the capacity constraints. Assume that $\boldsymbol{f}_e$ is coming close to $\boldsymbol{u}_e^+$, then $\lim_{\boldsymbol{f}_e\to\boldsymbol{u}_e^+}(\boldsymbol{u}_e^+ - \boldsymbol{f}_e)^{-1-\alpha} = \infty$. The same is the case if $\boldsymbol{f}_e$ is coming close to $\boldsymbol{u}_e^-$. Assuming that $\boldsymbol{c}^T\boldsymbol{f}$ is also close to F^* this is not a problem. If this is not the case, we cannot guarantee to not get a very large gradient. If we search for cycles with large absolute gradients, the flow of the edge might be increase even though it is already close to the edge. Of course, this is a hypothetical scenario, but it motivates the need for a variable representing the distance to the capacity constraints. Consider the length of an edge as a function

$$l_e : (\boldsymbol{u}_e^-, \boldsymbol{u}_e^+) \to \mathbb{R}_{>0}, f_e \mapsto (\boldsymbol{u}_e^+ - f_e)^{-1-\alpha} + (f_e - \boldsymbol{u}_e^-)^{-1-\alpha}.$$

We can see that compared to the second summand of the gradient, the length does not contain subtractions. We always have positive values. In the gradient, this is important because if an edge has a positive cost, subtraction makes sense to regulate it. If it has a negative cost, an addition is needed to counteract moving closer to the constraints. The lengths, in this sense do note differentiate between upper and lower bounds. We now differentiate the length to gain more insight into how it behaves:

$$l_e' : (\boldsymbol{u}_e^-, \boldsymbol{u}_e^+) \to \mathbb{R}, f_e \mapsto (-1-\alpha)((\boldsymbol{u}_e^+ - f_e)^{-2-\alpha} - (f_e - \boldsymbol{u}_e^-)^{-2-\alpha}).$$

For $f_e = (\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-)/2$ we get $l_e'(f_e) = (-1-\alpha)(((\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-)/2)^{-2-\alpha} - ((\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-)/2)^{-2-\alpha}) = 0$. For $f_e < (\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-)/2$ we find $l_e'(f_e) < 0$ and for $f_e > (\boldsymbol{u}_e^+ + \boldsymbol{u}_e^-)/2$, $l_e'(f_e) > 0$. The length is at its minimum at the middle of the polytope and increases towards capacity constraints. Using the same argument used to explore the asymptotic behavior of the gradients, we can see that the length of e goes towards infinity at the capacity constraints. Therefore, we can somewhat interpret the inverse of the length as the distance from the edge. Chen et al. use this fact once the HSFC data structure found a min-ratio cycle $\boldsymbol{\Delta}$. Before adding $\boldsymbol{\Delta}$ to the current flow, they scale it by $\eta = -\kappa^2\alpha^2/(800\langle \boldsymbol{g}, \boldsymbol{\Delta}\rangle$, and therefore remove the gradients and make progress toward the capacity constraint entirely dependent on the lengths.

4.6 Updates in the HSFC data structure

In this section, we go into more detail as to how updates are handled and what problems come with them.
At the top level, we do not need to update the gradients and lengths in every iteration. Chen et al. show in lemma 4.9 and 4.10 [CKL$^+$22], that gradients and lengths are relatively stable when the residual capacity does not change much. They even go a step further and show that we can use an old value of $\boldsymbol{c}^T f^* - F^*$ to calculate the gradients, provided that it does not change more than a certain ε factor. Thus, only $m^{o(1)}$ edges need to be updated per iteration of the IPM. We have seen that the core graphs and sparsified core graphs both have properties that need to be maintained during updates. For instance, if the length of e increases, then we need to ensure that in every branching tree chain $G_0, \dots, G_d$, with forest $F_0, \dots, F_d$, these forests do not stretch the length $\boldsymbol{l}_e$ too much. This is done by deleting e from F_0 this intern triggers a vertex split in $\mathcal{C}(G, F_0)$, which propagates to $\mathcal{S}(G, F_0)$ and so on. Thus, a singular update triggers multiple updates inside the branching tree-chain. Chen et al. showed that on every level, these updates have a low recourse , so they trigger at most $\gamma_r = m^{o(1)}$ additional updates ([CKL$^+$22, Lemma 7.5, Lemma 7.8]). Thus, the total number of updates per singular update on the top-level graph is $\mathcal{O}(B\gamma_r)^d = m^{o(1)}$. Because we have $m^{o(1)}$ top level updates at every iteration, this results in $m^{o(1)}$ total updates in the branching tree-chain. However, these updates change the forests used in the branching tree-chain. This is a problem because these forests were sampled independently 4.3.2.4 to ensure that with high probability the witness circulation is mapped through the core graphs. Because of the updates, the forests are not longer independent. Therefore, the branching tree-chain is no longer guaranteed to find a good min-ratio cycle. A simple fix would be to rebuild the entire branching tree-chain. This takes too much time. Therefore, we need a strategy that only rebuilds parts of the data structure. We explain this strategy in section 4.8. To prove that this strategy works, we need to introduce a new concept called the *width*.

Definition 4.13 (Width). *Given an instance of the min-cost flow problem on graph G, with current lengths $\boldsymbol{l}$ and witness circulation $\boldsymbol{\Delta}$,*

the width $\boldsymbol{w} \geq |\boldsymbol{c} \circ \boldsymbol{l}|$ *is as an upper bound of the witness circulation times the length.*

In practice, they are set to $\boldsymbol{w}_e = 50 + |\boldsymbol{l}_e, \boldsymbol{\Delta}_e|$ for every $e \in E(G)$.

In the branching tree-chain $\boldsymbol{w}^{(prev_i)}$ is the width at level i at the time it was last rebuilt.

This is a good time to revisit the HSFC updates. Every updated given and generated inside the branching tree chain is an HSFC update and G (and every other tree in the branching tree-chain) satisfy the following properties after every update:

1. The witness circulation $\boldsymbol{\Delta}^*$ is always a circulation.
2. The width $\boldsymbol{w}$ are always an upper bound for the length of the witness circulation: $|\boldsymbol{l} \circ \boldsymbol{\Delta}| \leq \boldsymbol{w}$.

3. The current width of an edge is upper bounded by two times the width of the edge, after it was last updated.
4. Both widths and lengths are quasipolynomially lower and upper-bounded.

This is an informal description of some properties of the HSFC updates ([CKL$^+$22, Defintion 6.1]). With our definitions so far, the first two points are self-explanatory. The second two points are essential for the rebuilding strategy. With the third point, we link the current width to the time width when a layer of the branching tree-chain is rebuilt.

4.7 Min-ratio cycles in the branching tree-chain

In this section, we focus on the min-ratio cycles returned by the branching tree chain. Note that when we refer to min-ratio cycles, they are at best a $m^{o(1)}$ approximation to the min-ratio cycle. This would suffice, since Chen et al. have shown that the ratio simply needs to be smaller than some $\kappa \in (0,1)$ [1] during the whole IPM, for the runtime to be $m^{1+o(1)}$ [CKL$^+$22, Theorem 4.3]. As we have seen, this is not necessarily the case because the branching tree-chain is not guaranteed to return a good approximation as soon as updates are applied. For this, Chen et al. prove in Theorem 7.1 that during every iteration (t) of the IPM, the branching tree-chain returns a cycle $\boldsymbol{\Delta}$ with

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}\rangle}{||\boldsymbol{l}^{(t)} \circ \boldsymbol{\Delta}||_1} \leq \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)}\rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}^{(\mathrm{prev}_i^{(t)})}||_1}. \tag{4.3}$$

This is the key idea that will be leveraged to prove that the rebuilding strategy works. For the rest of this chapter, we explore how to find these min-ratio cycles in the branching tree chain and on they full fill (4.3).

In this section, we often use the same symbols on different levels of the branching tree chain. We avoid conflicting notation by adding the associated graph to the symbol. For example, the gradients become $\boldsymbol{g}^G$ if it is not clear whether they belong to G.

In (4.3) we also used $X^{(t)}$ to denote a variable X at iteration t. We often include this because Chen et al. use it to formally argue, especially about the width at rebuild times. For simplicity we sometimes remove this from our proofs motivations.

4.7.1 Finding min-ratio cycles

To motivate the HSFC data structure, we have described a short algorithm that finds a min-ratio cycle by creating a probabilistic low-stretch spanning tree and then looking at the fundamental tree cycles to find the best solution. In the sections 4.3.2.2 and 4.3.2.3, we have that the branching tree-chain corresponds to s spanning trees. In this section, we explore how to find a minimum-ratio cycle on these trees using a similar concept.

When returning, the min-ratio cycle is represented as $m^{o(1)}$ off-tree edges on some tree T_i $i \in [s]$. So for the off-tree edges $e_1 = (u_1, v_1), \ldots, e_L = (u_L, v_L) \in \mathrm{E}(G) \setminus E(T_1)$ the min-ratio cycle is

$$e_1 \oplus T_i[v_1, u_2] \oplus \cdots \oplus e_L \oplus T[v_L, u_1].$$

This cycle is found by looking and maintaining the ratio of cycles, created by the off-tree edges of all the trees $T_1, \ldots, T_s$ corresponding to the branching tree chain. Every one of these edges has a lowest level, where it is removed by the in the sparsified core graph.

[1]in practice this is $\kappa = \alpha\kappa'/8$, where $\kappa' = \exp(-\mathcal{O}(\log^{7/8} m \log\log m))$ ([CKL$^+$22, Lemma 9.3]).

Definition 4.14 (Level of an edge in a tree-chain)**.** *Given a tree-chain $G_0 = G, \dots, G_d$, then edge $e^G \in E(G)$ is at* level *$level_{e^G} = i$ if its image e is in $E(\mathcal{C}(G_i, F_i) \setminus E(\mathcal{S}(G_i, F_i))$.*

It is important to note that when contracting G_i to $\mathcal{C}(G_i, F_i)$, we retain the self-loops. However, for a self-loop $e \in \mathcal{C}(G_i, F_i)$, the embedding $\Pi_{\mathcal{C}(G_i,F_i)\to\mathcal{S}(G_i,F_i)}(e)$ maps e to an empty path. Therefore, all edges in G_i that connect two vertices inside the same component of F_i have level i. At level d, we maintain the forest F_d, which is a tree, as discussed in section 4.3.2.3. Even though we create neither a core graph nor a sparsified core graph, all edges left would become self-loops. Therefore, we can define their level as d.

If we have an edge e at level i, we know that, as described in paragraph 4.3.2.5, that we maintain an embedding $\Pi_{\mathcal{C}(G_i,F_i)\to\mathcal{S}(G_i,F_i)}$, embedding e into some path in $\mathcal{S}(G_i, F_i)$. With this, we define the *sparsifier cycle.*

Definition 4.15 (Sparsifier cycle)**.** *Given a tree-chain $G_0 = G, \dots, G_d$, the* sparsifier cycle *$a(e)$ of such an edge $e = e_0 \in \mathcal{C}(G_i, F_i)$ with level i is the cycle $a(e) = e_0 \oplus rev(\Pi_{\mathcal{C}(G_i,F_i)\to\mathcal{S}(G_i,F_i)}(e_0)) = e_0 \oplus e_1 \oplus \dots \oplus e_L$, where ref revers the path. $\boldsymbol{a}(e)$ is the corresponding circulation routing one unit of flow over $a(e)$.*

With the we can now define the equivalent to a fundamental tree cycle for a tree-chain.

Definition 4.16 (Fundamental chain cycle)**.** *Given a tree-chain $G_0 = G, \dots, G_d$ and an edge $e \in \mathcal{C}(G_i, F_i)$ at level i, then the* fundamental chain cycle *$a^G(e^G)$ is the preimage in G of the sparsifier cycle $a(e) = e \oplus e_1 \oplus \dots \oplus e_L$ with $e_j = (u_j, v_j)$ for $j \in [L]$ and $u_{L+1} = u_0$*

$$a^G(e^G) = e_0^G \oplus T[v_0^G, u_1^G] \oplus e_1^G \oplus T[v_1^G, u_2^G,] \oplus \dots \oplus e_L^G \oplus T[v_L^G, u_{L+1}^G],$$

$\boldsymbol{a}^G(e^G)$ is the corresponding circulation routing one unit of flow over $a^G(e^G)$.

Definition 4.17 (Lifted cycle)**.** *Given the cycle $C \subseteq \mathcal{C}(G_i, F_i)$ with the edges $e_1^{\mathcal{C}(G_1,F_i)} \oplus \dots \oplus e_L^{\mathcal{C}(G_1,F_i)}$ with $e_j^{G_i} = (u_j, v_j)$, then the* lift *of C into G_i is*

$$e_1^{G_i} \oplus F_i[v_1, u_2] \oplus \dots \oplus e_L^{G_1} \oplus F_i[v_L, u_1].$$

Lemma 4.18. *Let $C^{G_i} \subseteq G_i$ be a lifted cycle of the cycle $C \subseteq \mathcal{C}(G_i, F_i)$. $C = e_1 \oplus \dots \oplus e_L$ where $e_j^{G_i} = (u_j, v_j)$, then*

1. *C^{G_i} is also a cycle.*
2. *$\sum_{j=1} \boldsymbol{p}(T_i[v_j, u_j]) = \sum_{j=1} \boldsymbol{p}(F_i[v_j, u_{j+1}])$ where T_i was the tree that together with F_i was generated by the LSD in lemma 4.11.*

Proof. 1. The way C^{G_i} is defined, it is a walk starting and ending at the same vertex and is therefore a circulation. We now only need to show that every vertex in G_i is incident to at most two edges. To prove this, let $v \in V(C^{G_i})$ be some vertex of the circulation. v belongs to a component $k \subseteq F_i$ corresponding to some vertex $v^{\mathcal{C}(G_i,F_i)} \in \mathcal{C}(G_i, F_i)$. Since $v^{\mathcal{C}(G_i,F_i)}$ is part of C, it is incident to two edges of this cycle. Therefore, there are two edges that connect k to the other components. Let $(u_j, v_j), (u_{j+1}, v_{j+1}) \in E(G_i)$ be those edges, then $v^{\mathcal{C}(G_i,F_i)}$ must be on the path $F_i[v_j, u_{j+1}]$ and is therefore incident to two edges of this circulation. From this, we can conclude that every vertex in the circulation has exactly one ingoing and one outgoing edge and that C^{G_i} is therefor a circle.

2. We show that $\sum_{j=1} \boldsymbol{p}(T_i[v_j, u_j]) = \sum_{j=1} \boldsymbol{p}(F_i[v_j, u_{j+1}])$ by showing that they generate the same supply of T_i. This uniquely determines the flow because we cannot augment any flow on T_i by adding circulations (except $\mathbf{0}$) because there are no cycle in T_i. For $j_0 \in \{1, \dots L\}$ we know from the construction that $T_i[v_{j_0}, u_{j_0+1}]$ is completely in one component of F_i. There for
$$(\boldsymbol{B}^T \boldsymbol{p}(F[v_{j_0}, u_{j_0+1}]))_v = \begin{cases} -1 & \text{if } v = v_{j_0} \\ 1 & \text{if } v = u_{j_0+1} \\ 0 & \text{otherwise} \end{cases}.$$
We now want to prove something similar for $\boldsymbol{p}(T[v_{j_0}, u_{j_0}])$. Here, we use the fact that $T[v_{j_0}, u_{j_0}] \oplus (u_{j_0}, v_{j_0})$ is a cycle in G_i. Note that (u_{j_0}, v_{j_0}) exists in G_i because it is one of the edges lifted from $\mathcal{C}(G_i, F_i)$ to G_i. Therefore, we know that $\boldsymbol{B}^T \boldsymbol{p}(T[v_{j_0}, u_{j_0}] \oplus (u_{j_0}, v_{j_0})) = \mathbf{0}$. This is equivalent to $\boldsymbol{B}^T \boldsymbol{p}(T[v_{j_0}, u_{j_0}]) + \boldsymbol{B}^T \boldsymbol{p}((u_{j_0}, v_{j_0})) = \mathbf{0}$. So
$$(\boldsymbol{B}^T \boldsymbol{p}(T[v_{j_0}, u_{j_0}]))_v = -\boldsymbol{B}^T \boldsymbol{p}((u_{j_0}, v_{j_0})) = \begin{cases} -1 & \text{if } v = v_{j_0} \\ 1 & \text{if } v = u_{j_0} \\ 0 & \text{otherwise} \end{cases}.$$
We can now conclude that
$$\boldsymbol{B}^T \sum_{j=1} \boldsymbol{p}(T_i[v_j, u_j]) = \boldsymbol{B}^T \sum_{j=1} \boldsymbol{p}(T_i[v_j, u_{j+1}])$$
Because in both iterations, for every $j \in \{1, \dots, L\}$, let $e_j = (u_j, v_j)$, we set the supply generated at u_j to -1 and the supply generated at v_j to 1. And with this $\sum_{j=1} \boldsymbol{p}(T_i[v_j, u_j]) = \boldsymbol{B}^T \sum_{j=1} \boldsymbol{p}(T_i[v_j, u_{j+1}])$.

□

When we lift a cycle C from $\mathcal{C}(G, F)$ to G and then contract G using F, all the forest edges that intersperse the cycle C are contracted again. Therefore, C is the image of the contracted cycle. If we apply this fact inductively, it shows that lifted cycles are fundamental chain cycles.

We have now defined a method by which we can create a cycle from a single off-tree edge. We now link the ratio of a sparsifier cycle to the ratio of its lifted cycle.

Lemma 4.19 (Lifted cycle ratio)**.** *Given a tree chain $G_0 = G, \dots, G_d$ and an edge $e \in \mathcal{C}(G_i, F_i)$ at level i, then*
$$\frac{\langle \boldsymbol{g}^G, \boldsymbol{a}^G(e^G) \rangle}{||\boldsymbol{L}^G \boldsymbol{a}^G(e^G)||_1} \leq \frac{\langle \boldsymbol{g}^{\mathcal{C}(G_i,F_i)}, \boldsymbol{a}(e) \rangle}{||\boldsymbol{L}^{\mathcal{C}(G_i,F_i)} \boldsymbol{a}(e)||_1},$$
where $\boldsymbol{g}^{\mathcal{C}(G_i,F_i)}, \boldsymbol{L}^{\mathcal{C}(G_i,F_i)}$ are the gradients mapped to the core graph $\mathcal{C}(G_i, F_i)$.

Together with the embedding $\Pi_{\mathcal{C}(G_i,F_i) \to \mathcal{S}(G_i,F_i)}(e)$ we can also store the total gradient of this mapping. Therefore, we can maintain the total gradient of every sparsifier cycle with a constant overhead. In the next lemma, we see that this gradient is the same as the total gradient of the lifted cycle.

Lemma 4.20 (Gradient correctness)**.** *Let $e^G \in E(G) \setminus E(T)$ be an edge, with $level_{e^G} = i$ and e its image in $\mathcal{C}(G_i, F_i)$. Then*
$$\langle \boldsymbol{g}^{\mathcal{C}(G_i,F_i)}, \boldsymbol{a}(e) \rangle = \langle \boldsymbol{g}^G, \boldsymbol{a}^G(e^G) \rangle.$$

Proof. We only prove that lifting the cycle $\boldsymbol{\Delta}^{\mathcal{C}(G_i,F_i)} \in \mathbb{R}^{E(\mathcal{C}(G_i,F_i))}$ as in definition 4.17, keeps the gradient the same. Let $\boldsymbol{\Delta}$ be on $e_1^{\mathcal{C}(G_i,F_i)}, \dots, e_L^{\mathcal{C}(G_i,F_i)}$, with $e_j^{\mathcal{C}(G_i,F_i)} = (u_j, v_j)$ and $v_L = u_1$.

$$\begin{aligned}
\langle \boldsymbol{g}^{\mathcal{C}(G_i,F_i)}, \boldsymbol{\Delta}^{\mathcal{C}(G_i,F_i)} \rangle &= \sum_{j=1}^{L} \boldsymbol{g}_{e_j^{\mathcal{C}(G_i,F_i)}} \\
&= \sum_{j=1}^{L} \boldsymbol{g}_{e_j^{G_i}} + \langle \boldsymbol{g}^{G_i}, \boldsymbol{p}(T_i[v_j, u_j]) \rangle \\
&\overset{\text{Lemma 4.18}}{=} \sum_{j=1}^{L} \boldsymbol{g}_{e_j^{G_i}} + \langle \boldsymbol{g}^{G_i}, \boldsymbol{p}(F_i[v_j, u_{j+1}]) \rangle \qquad = \langle \boldsymbol{g}^{G_i}, \boldsymbol{\Delta}^{G_i} \rangle
\end{aligned}$$

The rest follows by induction. □

Similar to the gradients of an edge e at level i, we can maintain the length of the sparsifier cycle $\widetilde{\mathbf{len}}_{e^G} = \langle \boldsymbol{l}^{\mathcal{C}(G_i,F_i)}, |\boldsymbol{a}(e)| \rangle$ together with the embedding with a constant overhead. Chen et al. prove that this is an overestimate for the total length of the lifted cycle, so $\langle \boldsymbol{l}^G, |\boldsymbol{a}^G(e^G)| \rangle \leq \widetilde{\mathbf{len}}_{e^G}$ ([CKL+22, Lemma 7.14].

Lemma 4.19. Follows directly from lemma 4.20 and lemma 7.14 of Chen et al., when considering, that the cycle is negative. □

Lemma 4.19. This follows directly from lemma 4.20 and lemma 7.14 of Chen et al., when considering that the cycle is negative. □

With this we can now find a cycle with a ratio smaller than some value by finding an edge whose associated sparsifier cycle has a ratio small enough. In fact, this is done very effectively by maintaining the maximizer

$$\max_{e^G \in E(G) \setminus E(T_i)} \frac{|\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G) \rangle|}{\widetilde{\text{len}}_{e^G}}.$$

In the proof of theorem 7.1 Chen et al. state that this can be done with an overhead of $\widetilde{\mathcal{O}}(1)$ for every tree.

4.7.2 Limiting the min-ratio cycles via the width

In this section, we discuss the core aspects required to prove lemma 7.1 of Chen et al. First, we establish how to map the witness circulation through a tree-chain.

Definition 4.21 (Mapping witness circulation). *Given a graph G with the spanning forest F and the witness circulation $\boldsymbol{\Delta}^* \in \mathbb{R}^{E(G)}$ then the witness circulation in $\mathcal{C}(G,F)$ is*

$$\boldsymbol{\Delta}^{*,\mathcal{C}(G,F)} = \boldsymbol{\Delta}_{|E(\mathcal{C}(G,F)}$$

and in $\mathcal{S}(G,F)$

$$\boldsymbol{\Delta}^{*,\mathcal{S}(G,F)} = \sum_{\hat{e} \in E(\mathcal{C}(G,F)} \boldsymbol{c}_{\hat{e}}^{\mathcal{C}(G,F)}(\hat{e}) \boldsymbol{\Pi}_{\mathcal{C}(G,F) \to \mathcal{S}(G,F)}(\hat{e}).$$

Not that $|\mathbf{\Pi}_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}(\hat{e})|$ is not the amount of edges in the embedding, but the absolute flow of routing one unit over this embedding.

With the we can map the witness circulation through a branching tree-chain. We now want to go a step further and decompose the witness circulation into fundamental chain cycles using these mappings.

Lemma 4.22 (Decomposition of the witness circulation, Lemma 7.16 [CKL+22])**.** *Given a tree-chain $G = G_0, \dots, G_d$, with the witness circulation $\mathbf{\Delta}^*$ mapped through this chain, as defined in 4.21 with $\mathbf{\Delta}^i$ as the circulation in G_i. Then*

$$\mathbf{\Delta}^* = \sum_{i=0}^{d} \sum_{e^G : level_{e^G} = i} \mathbf{\Delta}_e^{G_i} \boldsymbol{a}^G(e^G).$$

The proof can be found in the work of Chen et al. [CKL+22, p.59f].

As we have established, we will find the min-ratio cycle through these fundamental chain cycles. To prove that we can find a good enough cycle , we need to first establish a relationship between the gradient of the witness circulation and the gradients of the decomposed cycles.

Lemma 4.23. *Given a tree-chain $G = G_0, \dots, G_d$, gradients $\boldsymbol{g} \in \mathbb{R}^E(G)$ with $\mathbf{\Delta}^*$ the current witness circulation mapped through this chain as defined in 4.21 to $\mathbf{\Delta}^{G_1}, \dots, \mathbf{\Delta}^{G_d}$, then*

$$|\langle \boldsymbol{g}, \mathbf{\Delta}^* \rangle| \leq \sum_{i=0}^{d} \sum_{e^G : level_{e^G} = i} |\mathbf{\Delta}_e^{G_i}| |\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G) \rangle|.$$

Proof. Follows from lemma 4.22 and the triangle inequation. ☐

This is the reason why we do not maintain a minimizer but a maximizer to find the min-ratio cycle in section 4.7.1. all cycles in which the witness circulation is decomposed have a negative gradient. We do not prove this here. A way to think about this being true is that if the witness circulation could be decomposed into a cycle with a positive gradient, then we could find a circulation that is even smaller than the witness circulation by subtracting this cycle. Visually speaking, we find a slope on the polytope steeper than the slope of the witness circulation, which leads us to the optimal solution. Without further analysis of the potential function this could be the case. Then again, Chen et al. use this fact without proof.

We now discuss how we can do something similar with the lengths, but instead of showing that the length total length of the decomposed witness circulation is bounded by the length of the witness circulation, we use the width.

For this, let us first define how we map the width throughout the branching tree-chain.

Definition 4.24 (Mapping of the width)**.** *Given a graph G with a spanning tree F, the respective stretch overestimate $\widetilde{\boldsymbol{str}}$ as well as the current width $\boldsymbol{w}$. The width of an edge e in the core graph is*

$$\boldsymbol{w}_e^{\mathcal{C}(G,F)} = \widetilde{\boldsymbol{str}}_e \boldsymbol{w}_e.$$

And the width in the sparsified core graph is

$$\boldsymbol{w}^{\mathcal{S}(G,F)} = 2 \sum_{\hat{e} \in E(\mathcal{C}(G,F))} \boldsymbol{w}_{\hat{e}}^{\mathcal{C}(G,F))} |\mathbf{\Pi}_{\mathcal{C}(G,F)\to\mathcal{S}(G,F)}(\hat{e})|.$$

At this point, it is important to note that with the mapping for the widths and witness circulations defined in definitions 4.24, 4.21, the mapped witness circulations are always circulations and the mapped widths are upper bound to the lengths of the respective circulations [CKL+22, Lemma 7.4, 7.7].

Lemma 4.25. *Given a tree-chain $G = G_0, \dots, G_d$ with $\boldsymbol{\Delta}^*$ the current witness circulation mapped through this chain as defined in 4.21 to $\boldsymbol{\Delta}^{G_1}, \dots, \boldsymbol{\Delta}^{G_d}$. Then with $\widetilde{\boldsymbol{len}}_{e^G}$ as the fundamental chain cycle length overestimate for edge $e^G \in E(G)$,*

$$\sum_{i=0}^{d} \sum_{e^G : level_{e^G} = i} |\boldsymbol{\Delta}_e^{G_i}| \widetilde{\boldsymbol{len}}_{e^G} \leq \widetilde{\mathcal{O}}(k) \sum_{i=0}^{d} ||\boldsymbol{w}^{G_i}||_1$$

Proof. The proof of this is describe in the proof for lemma 7.17 of Chen et al. We comment its steps to give a better insight.

$$\begin{aligned}
\sum_{i=0}^{d} \sum_{e^G:\text{level}_{e^G}=i} |\boldsymbol{\Delta}_e^{G_i}||\widetilde{\mathbf{len}}_{e^G} &\overset{(i)}{=} |\sum_{i=0}^{d} \sum_{e^G:\text{level}_{e^G}=i} \left(\widetilde{\mathbf{str}} \boldsymbol{l}_e^{G_i} |\boldsymbol{\Delta}_e^{G_i}| + \sum_{e' \in \boldsymbol{\Pi}_{\mathcal{C}(G_i,F_i) \to \mathcal{S}(G_i,F_i)}(e)} \boldsymbol{l}_e^{G_{i+1}} |\boldsymbol{\Delta}_e^{G_i}| \right) \\
&\overset{(ii)}{\leq}) \sum_{i=0}^{d} \sum_{e^G:\text{level}_{e^G}=i} \left(\widetilde{\mathcal{O}}(k) \boldsymbol{w}_e^{G_i} + \boldsymbol{w}_e^{G_i} \right) \\
&\overset{(iii)}{\leq} \widetilde{\mathcal{O}}(k) \sum_{i=0}^{d} ||\boldsymbol{w}^{G_i}||_1.
\end{aligned}$$

In step (i), they take apart the definition of $\widetilde{\mathbf{len}}_{e^G} = \langle \boldsymbol{l}^{\mathcal{C}(G_i,F_i}, |\boldsymbol{a}(e)| \rangle$ by splitting the sparsifier cycle into the edge $e \in \mathcal{C}(G_i, F_i)$ and the reverse path in the sparsifier $\text{rev}(\Pi_{\mathcal{C}(G_i,F_i) \to \mathcal{S}(G_i,F_i)})$. Of course, the fact that is reversed does not matter. In step (ii), they substitute $\widetilde{\mathbf{str}}$ with $\widetilde{\mathcal{O}}(k)$ since this is the certain stretch upper bound, we have munitioned in 4.3.1.1. Then $\boldsymbol{l}_e^{G_i}|\boldsymbol{\Delta}_e^{G_i}| \leq \boldsymbol{w}_e^{G_i}$ is true, because as discussed above, the property of the width being an upper witness circulation length bound is kept during the mapping through the branching tree chain. For the second summand representing the length of the path in the sparsifier embedding, Chen et al. refer to the mapping of the width through the branching tree-chain. Step (iii) is correct, since $||\boldsymbol{w}^{G_i}||_1$ sums up the width over every edge in level i. With the iteration of the witness circulation decomposition, we only hit each of these widths at most 2 times. This constant factor is suppressed by $\widetilde{\mathcal{O}}(k)$. □

We can now start putting together the proof of lemma 7.1 of Chen et al. We want to show, the cycle $\boldsymbol{\Delta}$ maintained in the maximizer of section 4.7.1 fullfills (4.3). For this, we first show

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{l}^{(t)} \circ \boldsymbol{\Delta}||_1} \leq \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}||_1}.$$

as done by Chen et al. ([CKL+22, Lemma 7.17]). This can be achieved by combining lemma 4.23 with 4.25 of Chen et al. to show

$$\begin{aligned}
\frac{1}{\widetilde{\mathcal{O}}(k)} \frac{|\langle \boldsymbol{g}, \boldsymbol{\Delta}^* \rangle|}{\sum_{i=0}^{d} ||\boldsymbol{w}||_1} &\leq \frac{\sum_{i=0}^{d} \sum_{e^G:\text{level}_{e^G}=i} |\boldsymbol{\Delta}_e^{G_i}||\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G) \rangle|}{\sum_{i=0}^{d} \sum_{e^G:\text{level}_{e^G}=i} |\boldsymbol{\Delta}_e^{G_i}|\widetilde{\mathbf{len}}_{e^G}} \\
&\overset{(i)}{\leq} \max_{e \in E(G) \setminus E(T)} \frac{|\langle \boldsymbol{g}, \boldsymbol{a}^G(e^G) \rangle|}{\widetilde{\mathbf{len}}_{e^G}}
\end{aligned}$$

Where T is the tree of the tree-chain and (i) uses

$$\max_{i\in[n]} \frac{\boldsymbol{x}_i}{\boldsymbol{y}_i} \geq \frac{\sum_{i\in[n]} \boldsymbol{x}_i}{\sum_{i\in[n]} \boldsymbol{y}_i},$$

for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n_{\geq 0}$.

We now now, that the maximizer and with lemma 4.19 we can now put a upper bound on the min-ratio size, using a scaled ratio of the witness circulation. We now only need to make sure that for some tree-chain,

$$\frac{1}{\widetilde{\mathcal{O}}(k)} \frac{|\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^* \rangle|}{\sum_{i=0}^{d} ||\boldsymbol{w}||_1} \leq \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}^{(\text{prev}_i^{(t)})}||_1} \tag{4.4}$$

to ensure that (4.3) is satisfied. The existence of a tree-chain like this exists, is proven in lemma 7.9 by Chen et al. [CKL+22]. We can motivate parts of it by looking at the properties of the HSFC updates 4.6. The third point states that the width of an edge at most doubles since the last time the edge is updated. We now describe how updates affect the widths inside the branching tree chain. As always, we focus on the core graphs, but similar methods are used for the sparsified core graphs. Let G be somewhere in a tree-chain, with F being the forest used to create $\mathcal{C}(G, F)$, and $\boldsymbol{w}^G$ be the current witness circulation in G. If an edge is updated in G, the stretch is set to 1. By definition 4.24 the $\boldsymbol{w}_e^{\mathcal{C}(G,F)} = 1 \cdot \boldsymbol{w}_e^G$. If an edge is not updated, then $\boldsymbol{w}_e^{\mathcal{C}(G,F)} \leq 2\boldsymbol{w}_e^{\mathcal{C}(G,F),\text{prev}}$ where $\boldsymbol{w}_e^{\mathcal{C}(G,F)}$ is the width of $\mathcal{C}(G, F)$ when it was (re-)built. Because at that time the forests were used to construct the core graphs of G, there is with a high probability a forest (which we assume is our forest F) that does not stretch the witness circulation by too much. With this, it is possible to bound the core graphs width

$$||\boldsymbol{w}^{\mathcal{C}(G,F)}||_1 \leq \gamma ||\boldsymbol{w}^{G,\text{prev}}||_1 + ||\boldsymbol{w}^G||_1,$$

([CKL+22, Lemma 7.5]). γ is a stand-in for a factor that we do not discuss. With a similar reason and an inductive approach, Chen et al. show that with high probability, there is a tree-chain satisfying (4.4) ([CKL+22, Theorem 7.1]. With this , the maximizer finds a cycle that when lifted produces $\boldsymbol{\Delta}$ satisfying

$$\frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{L}^{(t)} \boldsymbol{\Delta}||_1} \leq \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}^{(\text{prev}_i^{(t)})}||_1}. \tag{4.5}$$

4.8 The Rebuilding Game

As we have established, the branching tree-chain is not guaranteed to return a min-ratio cycle small enough. Since these cycles are returned in the form of off-tree edges of a dynamic tree, we can simply test, if this is the case. If it is not, we need to rebuild parts of the branching tree-chain. The strategy behind it, is rather simple. How ever to prove that this strategy works, Chen et al. abstract the so called rebuilding game [CKL+22, chapter 8]. We now establish the motivation behind this game.

As we have seen, the branching tree-chain find a min-ratio cycle $\boldsymbol{\Delta}$ with

$$\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} \leq \kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}^{(\text{prev}_i^{(t)})}||_1}. \tag{4.6}$$

for $\kappa = \exp(-\mathcal{O}(\log^{7/8} m \log\log m))$. If $\boldsymbol{\Delta}$ does not a have a small enough ratio, then $\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} \geq -\kappa' \alpha$ for an $\kappa' \in (0, 1)$ of our choosing. By setting $\kappa' = \kappa/(2d + 2)$, the following becomes true:

$$\kappa \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{\sum_{i=0}^{d} ||\boldsymbol{w}^{(\text{prev}_i^{(t)})}||_1} \geq -\kappa' \alpha \overset{(i)}{\geq} \frac{\kappa}{2d+2} \frac{\langle \boldsymbol{g}^{(t)}, \boldsymbol{\Delta}^{*,(t)} \rangle}{||\boldsymbol{w}^{(t)}||}$$

For (i) we used that $\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta}^{*,(t)} \rangle}{||\boldsymbol{w}||} \leq -\alpha$ as we have already seen in section 4.3.2.1 and is show to be correct by Chen et al. [CKL$^+$22, theorem 6.2]. With this we can use

$$\sum_{i=0}^{d} ||\boldsymbol{w}^{\mathrm{prev}_i^{(t)}}||_1 \geq 2(d+1)||\boldsymbol{w}||_1 \tag{4.7}$$

to abstract the notion that we need to rebuild the branching tree-chain to obtain a better cycle. At this point, it is important to mention, that when rebuilding a level i of the branching tree-chain, we also rebuild all levels $j > i$ with $j \in [d]$.

The rebuilding game is set up with an adversary and a player. The adversary picks values simulating a width unknown to the player. If certain conditions are met, the adversary can force the player to rebuild the branching tree-chain. The first way the adversary can force the player to rebuild is when a certain number of rounds have passed without a rebuild. This makes sense because in every update, edges are deleted from the spanning forests. After $\frac{m^{1-o(1)}}{k^i}$ updates a core graph has grown to large [CKL$^+$22, p. 16]. This aspect of the rebuilding game is rather simple. Here, the strategy is to simply maintain when we last rebuild every level of the branching tree-chain[2]. Once a level has not been updated long enough, we rebuild it together with the levels below. This takes $\frac{m^{1+o(1)}}{k^i}$ time and is effective when amortized over the runtime of the algorithm [CKL$^+$22, p 67]. The second situation in which the adversary can force the player to update, has to do with the width. This is where the game aspect really comes into play. Instead of proving that the strategy works on the widths, Chen et al. show that it works for the adversary picking some values W substituting $||\boldsymbol{w}||_1$. They do this by making the second condition for a rebuild essential (4.7). If we now show that the rebuilding strategy works well for W chosen by an adversary, then we can conclude that it is also an effective way of rebuilding with the real widths. And that therefore, we can also effectively resolve situations in which we do not find a good enough min-ratio cycle.

Rebuilding strategy

The full strategy can be found in the work of Chen et al. [CKL$^+$22, p. 62]. We have already established that the first condition for the adversary being able to force a rebuild is circumvented, by us always rebuild when a level of the branching tree chain becomes too old. For the second condition, every level i maintains a fix counter fix_i, that is reset every time a level is rebuilt because it it self or a level above became too old to fulfill condition one. If condition two is met, Chen et al. look for the smallest level, where fix_i is smaller than a specific bound. This level is then rebuilt, incrementing $\mathrm{fix}_i{+}{+}$. If this solves condition two, the counter is reset and the min-ratio cycle is returned. If not, the level is rebuilt until $\mathrm{fix}_i{+}{+}$ is no longer smaller than the specific bound. Then, the same process starts with the level above.

This is a very superficial view of the rebuilding game. However, we can motivate some of the ideas. Because we have seen that a rebuild of level i takes $\frac{m^{1+o(1)}}{k^i}$, which is, of course, much faster than for level $i-1$. The idea behind the bound of the fix-counter comes from the fact that the width (and W) are quasipolynomially bound, as described in the fourth point of 4.6. This allows Chen et al. to show that after we have rebuilt often enough, without fixing condition two, decreasing W^{prev_i} does no significantly decrease $\sum_{j=0}^{d} = ||W^{\mathrm{prev}_j}||_1$ ([CKL$^+$22, p.63, correctness condition]). Therefore, rebuilding this level will not solve condition two, even after we have rebuilt $i-1$. In this way, the level

[2]Outside the rebuilding game this not done via the amount of updates, but the cumulative encoding size of these updates

we are currently fixing starts at d and progressively approaches 0 until condition two is resolved. Chen et al. show, that a certain upper bound for fix_0 is never reached, and that therefore this process even works ([CKL$^+$22, lemma 8.5]). When amortized over the runtime of the algorithm, this strategy is fast enough [CKL$^+$22, lemma 8.3].

5. A partial Implementation

Given the complexity of the algorithm by Chen et al, a full implementation is beyond the scope of this thesis. A low effort way of getting initial insights into practical aspects of this algorithm, is by replacing the HSFC data structure with slower classical min-ratio cycle algorithms. The implementation of this algorithm is realized in C++ with the addition of the Eigen library for linear algebra. Also utilized is the Lemon library to check the result of the algorithm as well as to compare the runtime to that of established min-cost flow algorithms. As we will have shown by the end of this chapter, a serious analysis of runtime is not possible, since the implementation of the algorithm fails quickly.

5.0.1 The IPM

The initial point generation 4.4 and the IPM 4.3.1 are the parts we have implemented. Due to the issues hinted at before, we did not implement the binary search for the cost of the min-cost flow, since there is not much insight to be gained with the current algorithm. To the rest of the algorithm given by Chen et al. we have also made some small changes. First of all, we maintain the current flow in a singular vector since we are not using multiple spanning trees to find the min-ratio cycle. We also maintain the gradients and lengths explicitly and recalculate them in every iteration. One major change that we have made is that we let the IPM terminate once the cost of current flow $\boldsymbol{c}^T\boldsymbol{f}$ is smaller than $F^* + 1/(12m^2U^3)$ where m and U are the edges and upper bounds of the min-cost flow problem instance we have created to find the initial flow. We do this, to significantly decrease how precise the IPM must approximate the min-cost flow. Therefore practically reducing the runtime and and postponing numerical problems. This has the effect, that when rounding the flow at the end of the algorithm, we using Lemma 8.10 of Jan van den Brand et al. [vdBLN$^+$21, lemme 8.10] we can only prove to get the correct min cost flow with a probability of at least 1/2 if we consider our the min-cost flow problem as the slightly perturbed instance.

5.1 Finding minimum ratio cycles

In this section we will describe a classical way of finding min-ratio cycles. We will reduce this problem to a series of negative cycle detection problems and one shortest cycle problem. This in itself is nothing new as a similar approach is used in other works [AMO93], [BDHK17]. The difference is that these algorithms run on integer gradients and lengths. In our case we will have to use floating point numbers but can also return an approximated min-ratio cycle.

Theorem 5.1. *Given a graph G with gradients $\boldsymbol{g}^{E(G)}$ and lengths $\boldsymbol{l}^{E(G)}$. There is a algorithm that finds a ε approximation of the min-ratio cycle $\boldsymbol{\Delta}^*$ in $\mathcal{O}(\log(U)mn + n^3)$.*

Before we describe this algorithm we will first define the length.

Definition 5.2. *Given a graph G and some edge associated values $\boldsymbol{w} \in \mathbb{R}^E$ the* length *$len_{\boldsymbol{w}}$ of a walk $W \subseteq G$ is $len_{\boldsymbol{w}}(W) = \sum_{e \in E(W)} \boldsymbol{w}_(e)$.*

With this we now use length to describe to different contexts. We will sometimes continue using length to describe the sum of weights of some walk. Most of the time we will simply refer to it as the associated values we use, the *weight.*

Definition 5.3. *Given a min-ratio cycle problem $G, \boldsymbol{g}, \boldsymbol{l}$ and the scalar $\mu \in \mathbb{R}$, the* weight *of and edge $e \in E(G)$ is $\boldsymbol{w}_e = \boldsymbol{g}_e - \mu \boldsymbol{l}_e$.*

This way, when refer the the weight of a cycle we refer to the length of a cycle with the weight being the associated edge values. A *min-weight cycle* or *min-cycle* is therefore a cycle of minimum weight for some pair $G, \boldsymbol{w}$ and a *negative cycle* is a cycle of negative weight. When using words such as “longer” or “shortest”, this does not transver as well.

We can now describe the main idea behind finding a min-ratio cycle.

Lemma 5.4. *Given a min-ratio cycle problem $G, \boldsymbol{g}, \boldsymbol{l}$, with the unknown min-ratio μ_0. Let μ be the scalar assiociated with the weight $\boldsymbol{w}$, then if:*

$$\begin{aligned}
\textit{There is a negative cycle in } (G, \boldsymbol{w}) &\Rightarrow \mu > \mu_0 \\
\textit{There is no negative cycle in } (G, \boldsymbol{w}) &\Rightarrow \mu \leq \mu_0 \\
\textit{There is no negative cycle but a zero weight cycle in } (G, \boldsymbol{w}) &\Rightarrow \mu = \mu_0
\end{aligned}$$

Proof. First assume there is a negative cycle C in $(G, \boldsymbol{w})$. Then we know:

$$\sum_{e \in E(C)} \boldsymbol{w}_e = \sum_{e \in E(C)} (\boldsymbol{g}_e - \mu \boldsymbol{l}_e) < 0$$

This is equivalent to

$$\mu > \frac{\sum_{e \in E(C)} \boldsymbol{g}_e}{\sum_{e \in E(C)} \boldsymbol{l}_e} \overset{(i)}{\geq} \mu_0$$

Where (i) is a direct consequence of μ_0 being the minimum ratio of a cycle and C being some cycle.

On the contrary, if we assume that there is no negative cycle, so $\sum_{e \in E(C)} \boldsymbol{w}_e \geq 0$ for every cycle C, then as above, this is equivalent to

$$\frac{\sum_{e \in E(C} g_e}{\sum_{e \in E(C)} l_e} \geq \mu.$$

And since the min-ratio cycle is among these cycles, we get $\mu \leq \mu_0$. If there is now a cycle C^* with weight zero, this is equivalent to

$$\frac{\sum_{e \in E(C^*} g_e}{\sum_{e \in E(C)} l_e} = \mu.$$

□

Because we are working with floating point number, we are not using the third point of this lemma directly. With the first and second point we can run a binary search where we maintain the bounds $\mu_s \leq \mu_0 \leq \mu_b$ for μ_0 being the correct min-ratio. Let $\boldsymbol{w}^s$ be the weights corresponding the μ_s and $\boldsymbol{w}^b$ to μ_b. In every step we search for a negative cycle with the weight generated by $\mu = (\mu_s + \mu_b)/2$ if we find a negative cycle, we use lemma 5.4 and set $\mu_b := \mu$ or if we did not, then $\mu_s = \mu$. With this we can slowly approximate the min-ratio, but not yet the cycle itself. An intuitive idea would be to simply take the minimum weight cycle found in $G, \boldsymbol{w}^s$. Now we the question becomes how close μ_s and μ_b must be so that we find a ε of the min-ratio cycle with μ_s

Lemma 5.5. *Given a graph G with gradients and lengths $\boldsymbol{g}, \boldsymbol{l}$, where μ_0 is the min-ratio. Let $\mu_s \leq \mu_0 \leq \mu_b$ and $C_s \subseteq G$ be the min-weight cycle found using weights $\boldsymbol{w}^s$. If $\mu_b - \mu_s \leq \delta = \left(\frac{\sum_{e\in E(G)} \boldsymbol{l}_e}{2\min_{e\in E(G)} \boldsymbol{l}_e} - 1\right)^{-1} \varepsilon$ then*

$$\frac{\sum_{e\in C_s} \boldsymbol{g}_e}{\sum_{e\in C_s} \boldsymbol{l}_e} - \mu_0 < \varepsilon$$

Proof. We will do this proof in two steps. First we will show that weight a of the min-weight cycle $C_s \subseteq G$ in $G, \boldsymbol{w}^s$ can be made arbitrarily small by scaling δ. Then we will show that the difference between the ratio of the min-ratio cycle and the ratio of C_s only depends on a and δ.

For this let $C_b \subseteq G$ be the min-weight cycle of $G, \boldsymbol{w}^b$. Since C_s is the cycle with the smallest ratio in $G, \boldsymbol{w}^s$,

$$a = \sum_{e\in C_s} \boldsymbol{w}^s_e \leq \sum_{e\in C_b} \boldsymbol{w}^s_e = \sum_{e\in C_b} \boldsymbol{g}_e - \mu_s \boldsymbol{l}_e$$

We know that C_b is a negative cycle, so $\sum_{e\in C_b} \boldsymbol{g}_e - \mu_b \boldsymbol{l}_e < 0$ which is equivalent to $\sum_{e\in C_b} \boldsymbol{g}_e < \mu_b \sum_{e\in C_b} \boldsymbol{l}_e$. With this we can bound

$$\begin{aligned} a &\leq \sum_{e\in C_b} \boldsymbol{g}_e - \mu_s \boldsymbol{l}_e \\ &< \mu_b \sum_{e\in C_b} \boldsymbol{l}_e - \mu_s \sum_{e\in C_b} l_e \qquad\qquad \leq \delta \sum_{e\in C_b} \boldsymbol{l}_e \end{aligned}$$

With this result we can bound the ratio of C_s to be

$$\frac{\sum_{e\in C_s} \boldsymbol{g}_e}{\sum_{e\in C_s} \boldsymbol{l}_e} = \frac{a}{\sum_{e\in C_s} \boldsymbol{l}_e} + \mu_s < \frac{\delta \sum_{e\in C_b} bmg_e}{\sum_{e\in C_s} \boldsymbol{l}_e} + \mu_s$$

Utilizing the fact that $\mu_s > \mu_0 - \delta$ we can conclude this proof by rearranging the inequation above:

$$\frac{\sum_{e\in C_s} \boldsymbol{g}_e}{\sum_{e\in C_s} \boldsymbol{l}_e} - \mu_0 < \frac{\delta \sum_{e\in C_b}}{\sum_{e\in C_s} \boldsymbol{l}_e} - \delta \overset{(i)}{\leq} \delta \left(\frac{\sum_{e\in E(G)} \boldsymbol{l}_e}{2\min_{e\in E(G)} \boldsymbol{l}_e} - 1\right) = \varepsilon.$$

(i) follows since the cycle C_s uses at least 2 edges which have a length of at least $\min_{e\in E(G)}$ and the lengths of C_b can at maximum be the sum of all lengths, since they are greater than 0. □

In practice, we can even show that $\sum_{e\in C_b} \boldsymbol{l}_e \leq \frac{1}{2} \sum_{e\in E(G)}$ since we have added the reverse of every edge to G. Traversing an edge and the reverse of an egde cancels out the gradient and therefore increases the length of the min-ratio cycle. So C_b uses at most half of the edges in $E(G)$. Further more we can imagine, that the closer μ_s and μ_b are, the more edges they might have in common. So in practice this bound could probably be tightened.

5.2 Theoretical approximation value

There is a discrepancy between what we theoretically need for the approximation value δ and what is feasible. We only need to prove that our algorithm can find an min-ratio cycle with a ratio $-\kappa$ for some $\kappa \in (0,1)$. To make sure this is the case, we use the Theorem 6.2 by Chen et al. [CKL^{+}22, Theorem 6.2] where they state, that their data structure can find an cycle $\boldsymbol{\Delta}$ with

$$\frac{\langle \boldsymbol{g}, \boldsymbol{\Delta} \rangle}{||\boldsymbol{L}\boldsymbol{\Delta}||_1} \leq -\kappa' \alpha$$

for $\kappa' = \exp(-\mathcal{O}(\log^{7/8} m \cdot \log\log m))$. Because of this, we know that the min-ratio cycle has a ratio smaller than $-\kappa'\alpha$. We can now set our $\kappa = 0.9\kappa'\alpha$ and this way the min-ratio cycle will always have a ratio small enough to reduce the potential in a sufficient way. We of course need to make sure that the approximation we find is good enough. We can force this by setting $\delta = 0.1\kappa'\alpha$, since the ratio of the cycle must be smaller than 0

A different problem will appear in the implementation of the algorithm described above. In practice for every edge $e = (i,j)$, $\boldsymbol{w}_{(i,j)} \neq \boldsymbol{w}_{(j,i)}$. Therefore we construct some sort of residual network where we double the edges by adding them reversed with the negated gradient. If we now found a min-weight cycle on this network, it might be the case that the cycle traverses e and the reverse of e , resulting in the min-ratio cycle found being $\mathbf{0}$. We see that the weight of this cycle is $\boldsymbol{g}_e - \mu_s \boldsymbol{l}_e + (-\boldsymbol{g}_e) - \mu_s \boldsymbol{l}_e = -2\mu_s \mathrm{l}_e$. So we also need to set δ in a way, the makes sure we are always $\delta < 2\mu_s \boldsymbol{l}_e$ to make sure $\mathbf{0}$ is not a valid approximated min-ratio cycle. Since we have set the approximation value to a very small value, this is not an issue in practice.

5.3 Bellman Ford Algorithm

To detect negative cycles, we use the Bellman-Ford algorithm. This algorithm is used for determining the length/weight of the shortest path of each vertex $v \in V(G)$ to a specific source vertex $s \in V(G)$ inside a graph G. This includes graphs with negative edges.
The idea behind detecting negative cycles, is that the bellman ford algorithm finds the shortest path in mn iterations, under the assumption that there is no negative cycle. If there is a negative cycle, then for some vertex, we can always find shorter walks, by traversing the negative cycle as often as needed. Therefore the distance between some vertices will continue to decrease if we continue to apply the core operation behind the Bellman-Ford Algorithm.

Definition 5.6. *Given the graph G and the weights $\boldsymbol{w}$ we define the distance $d(v,u)$ of two vertices v, u as the sum of the weight of the shortest walk $W = (v, v_1) = e_1 \oplus \cdots \oplus e_k = (v_k, u)$ so that for ever other walk W' from v to u,*

$$len_{\boldsymbol{w}}(W) \leq len_{\boldsymbol{w}}(W').$$

The Bellman-Ford algorithm maintains a vector $\boldsymbol{d} \in \mathbb{R}^{E(G)}$ of upper bounds for these distances from some edge $s \in V(G)$ to all other vertices in G. Each iteration these upper bounds are reduced by applying the idea of the *shortest path optimality condition* [AMO93][p. 136].

Theorem 5.7 (Shortest path optimaltiy condition)**.** *Given a graph G with weights $\boldsymbol{w}$ without negative cycles. Then for an arbitrary source vertex $s \in V(G)$,*

$$d(s,v) \leq d(s,u) + \boldsymbol{w}_{(u,v)},$$

for all $(u,v) \in E(G)$.

Algorithm 5.1: BELLMANN-FORD

```
  Input: Graph $G$, weights $\boldsymbol{w}$
1 $\boldsymbol{d}[s] = 0$ for $v \in V(G)$ do
2     if $v \neq s$ then
3         $\boldsymbol{d}[v] = \infty$
4 for $i = 0; i < n - 1; i + +$ do
5     for $(u, v)$: $E(G)$ do
6         if $\boldsymbol{d}[v] > d[u] + \boldsymbol{w}_{(u,v)}$ then
7             $\boldsymbol{d}[v] = d[u] + \boldsymbol{w}_{(u,v)}$
```

Proof. Assume for some vertex $v \in V(G)$ and for the edge $(u, v) \in E(G)$, $d(s, v) > d(s, u) + \boldsymbol{w}_{(u,v)}$. Then shortest walk from s to u concatenated with (u, v) would be shorter than the shortest walk from s to v. This is a contradiction. □

In our implementation we represent we select the vertex with index 0 as the source s. For the initialisation we set $\boldsymbol{d}[s] = 0$ and for all $v \in V(G) \setminus \{s\}$, we set $\boldsymbol{d}[v] = \infty$. The rest of the Bellmann-Ford algorithm is described in algorithm 5.1.

Lemma 5.8. *Given a graph G with weights $\boldsymbol{w}$. Then $G, \boldsymbol{w}$ has no negative cycle if and only if after running the Bellman-Ford algorithm*

$$\forall (u, v) \in E(G) : \boldsymbol{d}[v] \leq \boldsymbol{d}[u] + \boldsymbol{w}_{(u,v)}.$$

This way it is possible to detect a negative cycle in $\mathcal{O}(mn)$.

Proof. If we assume that $G, \boldsymbol{w}$ does not contain a negative cycle, this follows directly from the proof of the Bellman-ford algorithm [Bel58, p. 87-90]

Let us assume that $G, \boldsymbol{w}$ contains a negative cycle. The runtime of the bellman-ford algorithm only depends on m and n and therefore terminates after $\mathcal{O}(mn)$ even though we have negative cycles. Now since every vertex is reached in this shortest path algorithm in our graphs, there is a path from s to a vertex v in the negative cycle. And what ever value $\boldsymbol{d}(v)$ is, we can find a walk with a smaller weight by adding the negative as often as needed. □

5.4 Finding min-weight cycles

Once we know our μ_s will give us a ε approximated min-ratio cycle, we only need to find this minimum weight cycle. In the implementation we have used a modified version of Floyd-Warshall algorithm running in $\mathcal{O}(n^3)$. This is once again an algorithm calculating shortest paths, but this time between every two vertices. For this we extend definition 5.6 by $d(i, k)$ as the shortest distance between $i, k \in V(G)$. The Floy-Warshall algorithm is based on the following optimality condition.

Theorem 5.9 (All-Pairs Shortest Path Optimality Condition)**.** *Given a graph G with weights $\boldsymbol{w}$ without negative cycles, then $d[i, j] \leq d[i, k] + d[k, j]$ for all $i, j, k \in V(G)$.*

Proof. See proof [AMO93][p. 146] □

While initializing the initial distances, we set the distance from each distance to itself to ∞, thus we shortest paths we find are cycles. For this the graph $G, \boldsymbol{w}^s$ cannot contain any negative cycles, which we have made sure with Bellman-Ford. It is worth mentioning, that there are faster algorithms for the min-weight cycle such as Orlins algorithm [OSN].

A full description and proof of the Floyd-Warshall algorithm is given by Ahuja et al. [AMO93, p. 148]

5.5 Analysis

In this section we analyse why this algorithm runs into problems.

5.5.1 Practical analysis

We ran our tests on the following system

- Operating System: Linux Solus 4.4
- CPU: AMD Ryzen 7 3800X 8-Core Processor
- Ram: 16 GiB

We can successfully ran the algorithm on a minimal example consisting of two vertices and a singular edge in about 26 seconds. In this time we have 85310 IPM iterations. As soon as we create a slightly bigger graph (4 vertices and 6 edges), the algorithm fails. The min-cost flow has a ratio of 9 while the flow we found had a cost smaller than 9.00002343 which is still far away from being safely roundable to an integer flow.

The reason for this is, that the Floyd-Warshall receives an graph with a negative cost cycle. Resulting in the construction of a path using infinite edges. Therefore, the program never terminates. This is interesting, since the Bellman-Ford algorithm was supposed to ensure that the weights used in the Floyd-Warshall algorithm do not contain negative cycles, and it does so correctly on the number representation we have used: double long.

5.5.2 Theoretical analysis

The reason why the Bellman-Ford algorithm does not detect a negative cycle while Floyd-Warsahll does, is numerical nature. To understand why, we look at how the lengths behave once the current flow gets close to the capacity constraints.

Lemma 5.10. *Let $G, \boldsymbol{c}, \boldsymbol{d}, \boldsymbol{u}^+, \boldsymbol{u}^-, U$ be an instance of the min-cost flow problem. Let $\widetilde{G}, \widetilde{\boldsymbol{c}}, \widetilde{\boldsymbol{d}}, \widetilde{\boldsymbol{u}^+}, \widetilde{\boldsymbol{u}^-}, \widetilde{U}$ be the moddified version on which we find the initial flow. Then there is an edge e for which the length $\boldsymbol{l}_e \geq 128m^6U^8$ a during the execution of the algorithm.*

Proof. At the end, we will need to round the current flow f on $\widetilde{G}$. For this we have the condition that $\langle \widetilde{\boldsymbol{c}}^T, \boldsymbol{f} \rangle \leq F^* + \frac{1}{12\widetilde{m}^2, \widetilde{U}^3}$. Then at some point t we have a flow for which: $\langle \widetilde{\boldsymbol{c}}^T, \boldsymbol{f} \rangle < F^* + \frac{1}{8\widetilde{m^2, \widetilde{U}^3}} = 1/(8m^2(4mU^2)^3$ Let $e \in E(\widetilde{G})$ be an edge added at the initial flow generation with $\boldsymbol{u}_e^- = 0$ and cost $4mU^2$. Both of this combinde we can bound the amount of flow rounted of e at t by $\frac{1}{32m^5U^6} > \boldsymbol{f}_e \widetilde{\boldsymbol{c}_e} = \boldsymbol{f}_e 4mU^2$. So at some point $f_e < \frac{1}{128m^6U^8}$. We can now also put a bound on the length of e:

$$\boldsymbol{l}_e = (\widetilde{\boldsymbol{u}_e^+} - \boldsymbol{f}_e)^{-1-\alpha} + (\boldsymbol{f}_e - \widetilde{\boldsymbol{u}_e^-})^{-1-\alpha} \geq f_e^{-1} > 128m^6U^8$$

□

In the example, that did not terminate we had $m = 6$ and $U = 10$. Thus we have an edge that at some point $\boldsymbol{l}_e > 128 \cdot 6^6 10^8 = 5,971968 \cdot 10^{14}$. In the floating point representation we lose some precision because of this. We have defined δ to be very small. We have seen that this puts an upper limit on the weight of our min-weight cycles in $G, \boldsymbol{w}^s$. In the same manner it puts a lower bound on the min-weight cycles in $G, \boldsymbol{w}^b$. So we can assume that $G, \boldsymbol{w}^b$ contains a very small negative cycle.

We can now describe what we assume to be the reason, why the Bellman-Ford algorithm does not detect the negative cycle, while the Floyd-Warshall algorithm does. In Bellman-Ford, we start with the source vertex s and maintain the weight of paths to other vertices. These paths depend on the on the choice of the source an can be very big in the worst case. Once the Bellman-Ford algorithm is done and we see if we can decrease the length of a path by using a negative cycle. But we have seen that the weights of these cycle is very small. So when adding the small cycle weights to the big paths, resulting value might be rounded to the original value of the path. So therefore no negative cycle is detected. In the Floyd-Warshall, we maintain the shortest cycles for every vertex individually. Thus we might not run into the same issue, because we do not need the potentially large path of some source vertex to these cycles. Therefore, Floyd-Warshall detects the negative cycle and fails.

The way this algorithm fails is mainly depended on the choice of number representation and the algorithms used for negative cycle detection and min-cycle reconstruction. So we can not conclude that this problem exists in the algorithm algorithm proposed by Chen et al. In fact, they show that the lengths are quasipolynomially bounded for 4.6 point 4. The question remains, if the size representations need to run this algorithm on a bigger graph, is practically feasible.

6. Conclusion

In this thesis, we established the basic concepts of the MCF algorithm. We described the MMCC algorithm and compared it to the algorithm proposed by Chen et al. We then gave an overview of this algorithm and motivated how approximated min-ratio cycles are found. Finally, we describe a min-cost flow algorithm running in C++ that utilizes both parts of the IPM of Chen et al. and the classical algorithm. We then saw that while this works in theory, because of the precision needed for finding min-ratio cycles and the large values the lengths take on during the IPM, we run into numerical problems with our choice of number representation and min-ratio cycle detection.

Bibliography

[AMO93] Ravindra Ahuja, Thomas Magnanti, and James Orlin. *NETWORK FLOWS*. PRENTICE HALL, 1993.

[AN19] Ittai Abraham and Ofer Neiman. Using petal-decompositions to build a low stretch spanning tree. *SIAM Journal on Computing*, 48(2):227–248, 2019.

[BDHK17] Karl Bringmann, Thomas Dueholm Hansen, and Sebastian Krinninger. Improved algorithms for computing the cycle of minimum cost-to-time ratio in directed graphs. 2017.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BGS21] Aaron Bernstein, Maximilian Probst Gutenberg, and Thatchaphol Saranurak. Deterministic decremental sssp and approximate min-cost flow in almost-linear time, 2021.

[CKL$^+$22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time, 2022.

[Kar78] Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discret. Math.*, 23:309–311, 1978.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 302–311, New York, NY, USA, 1984. Association for Computing Machinery.

[KK12] Z. Király and P. Kovács. Efficient implementations of minimum-cost flow algorithms, 2012.

[NP17] Nikolaos Samaras Nikolaos Ploskas. *Linear Programming Using MATLAB*, volume 127 of *Springer Optimization and Its Applications.* Springer International Publishing AG, 2017.

[OSN] James B. Orlin and Antonio Sedeño-Noda. *An O(nm) time algorithm for finding the min length directed cycle in a graph*, pages 1866–1879.

[PAN23] Ping-Qi PAN. *Linear Programming Computation.* 2023.

[SE83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[Van] Robert J Vanderbei. *Linear Programming Foundations and Extensions.* Fifth edition.

[vdBLN$^+$21] Jan van den Brand, Yin-Tat Lee, Danupon Nanongkai, Richard Peng, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Bipartite matching in nearly-linear time on moderately dense graphs, 2021.