Contents

1. Abstract

In this Master's Thesis, we investigate the approximative maxflow algorithm from [She13] in the context of unit-capacity multidimensional grid graphs. We start with the mathematical background, take a look at the basics of maximum flow problems, provide some proofs from [She13] in more detail, give an approximator for our graph structure and provide an implementation with additional adjustments together with an experimental evaluation of the algorithm, showing that our approximator can be used with $\alpha = 3$ or even less.
For our graph structure, we analyse some properties and give efficient implementations for operations needed for the algorithm from [She13].
The repository is located at `https://github.com/js97/approximate-flows`.

2. Introduction

The paper [She13] introduces an algorithm for finding nearly maximum flows in nearly linear time. It uses approximators to estimate the maximum congestion of a graph, and also shows how to construct such approximators. Additionally, it introduces a potential function $\Phi(f)$ and a derivable alternative $\phi(f)$, which uses this approximator, where a gradient descent algorithm is used to find the minimum of this potential. The minimum of $\Phi(f)$ and $\phi(f)$ is also a solution for the maximum flow problem.

Unit-capacity multidimensional undirected grid graphs are a simple class of graphs that is suitable for further investigation, as efficient congestion approximators lie at hand. At this point, it is interesting to evaluate the performance of the algorithm proposed in [She13] on this graph class. An evaluation also serves as a test of theories against actual runs, i.e. the proofs in [She13] about runtimes rely on models, which might or might not be hugely suboptimal in predicting the performance, even though they might only use individually tight bounds where inequalities are used. Many proofs are far from trivial, but understanding is required in order to review the validity of theorems that are affected by optimizations. As many proofs in [She13] are not very detailed, we also try to increase the understandability of those proofs in this thesis by adding more details.

For the congestion approximator, α is a metric about the upper limit of the approximator's underestimation of the optimal congestion. Since this is not trivial to find for our graph class, we aim to find an estimate by making observations about the data structure and use an empiric search.

3. Mathematics

This chapter introduces some mathematical functions used in this thesis, and serves as collection of tedious proofs, outsourcing them from other chapters with different focus.

3.1. Symmetric Softmax

The *symmetric softmax* serves as alternative to the standard maximum function. Its advantage over the maximum function is its derivability, enabling the possibility to use gradient descent for approximate minimum searches.

3.1.1. Definition

Let $\vec{x} = (x_1, ..., x_n)^T \in \mathbb{R}^n$ be a vector over $\mathbb{R}$. Then, the Symmetric Softmax Function is defined for $\vec{x}$ as follows:

$$lmax(\vec{x}) \coloneqq \ln\left(\sum_{i=1}^{n} e^{x_i} + e^{-x_i}\right) \qquad \text{(Definition 1)}$$

3.1.2. Derivative

The derivative of $lmax(\vec{x})$ w.r.t. x_j can be calculated from Definition 1 by applying well-known derivatives and the chain rule:

$$\frac{\partial}{\partial x}\ln(x) = \frac{1}{x} \qquad \text{(Derivative of } ln(x)\text{)}$$

$$\frac{\partial}{\partial x}e^x = e^x \qquad \text{(Derivative of } e^x\text{)}$$

$$\frac{\partial}{\partial x}\sum_{i=1}^{n} f(x) = \sum_{i=1}^{n}\frac{\partial}{\partial x}f(x) \qquad \text{(Sum rule)}$$

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g)}{\partial g}\cdot\frac{\partial g(x)}{\partial x} \qquad \text{(Chain rule)}$$

$$\frac{\partial x_i}{\partial x_j} = \delta_{ij} := \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \qquad \text{(Derivative of variables)}$$

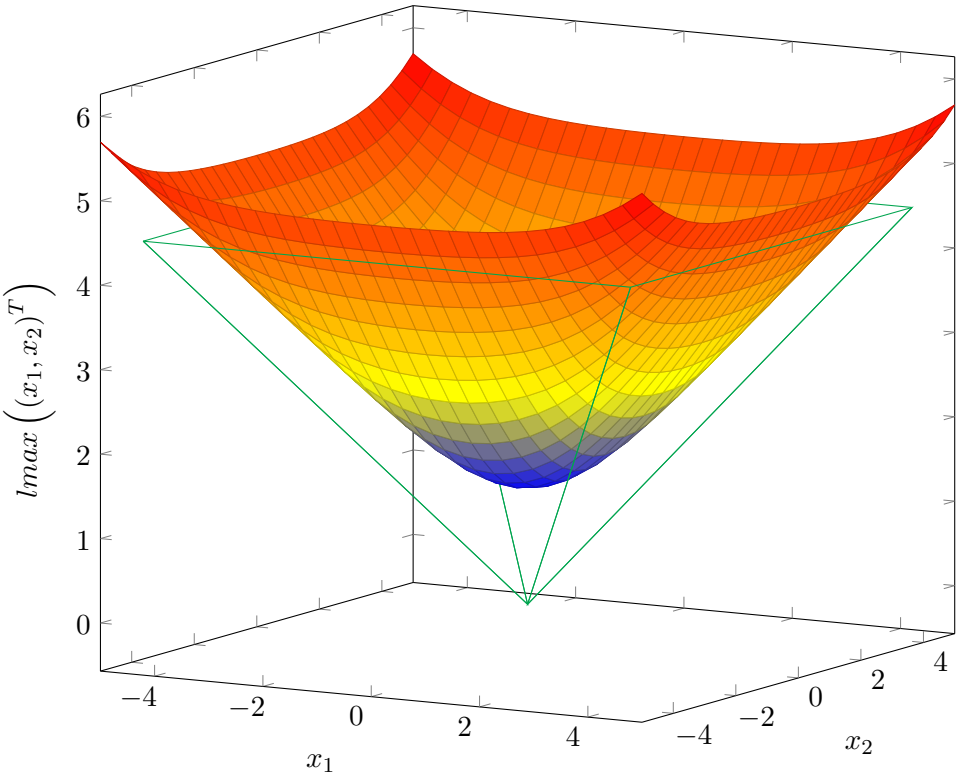


Figure 1 A plot of the $lmax\,(\vec{x})$ function for a two-dimensional input vector $\vec{x} = (x_1, x_2)^T$. $\max(x_1, x_2)$ has a downward pyramid shape, as also hinted in this figure.

This gives us the partial derivative of $lmax(\vec{x})$ for the variable x_j:

$$\begin{aligned}
\frac{\partial}{\partial x_j} lmax(\vec{x}) &\overset{1}{=} \frac{\partial}{\partial x_j} \ln\left(\sum_{i=1}^{n} e^{x_i} + e^{-x_i}\right) && \text{(1) by Definition 1}\\
&\overset{2}{=} \frac{1}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}} \cdot \frac{\partial}{\partial x_j}\left(\sum_{i=1}^{n} e^{x_i} + e^{-x_i}\right) && \text{(2) by Derivative of } ln(x)\text{, Chain rule}\\
&\overset{3}{=} \frac{\sum_{i=1}^{n}\left(\frac{\partial}{\partial x_j} e^{x_i}\right) + \left(\frac{\partial}{\partial x_j} e^{-x_i}\right)}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}} && \text{(3) by Sum rule}\\
&\overset{4}{=} \frac{\sum_{i=1}^{n}\left(e^{x_i} \cdot \frac{\partial x_i}{\partial x_j}\right) - \left(e^{-x_i} \cdot \frac{\partial x_i}{\partial x_j}\right)}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}} && \text{(4) by Derivative of } e^x\text{, Chain rule}\\
&\overset{5}{=} \frac{e^{x_j} - e^{-x_j}}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}} && \text{(5) by Derivative of variables}
\end{aligned}$$

Thus, the gradient of $lmax(\vec{x})$ is a vector with its j-th entry evaluating to:

$$\left(\nabla lmax(\vec{x})\right)_j = \frac{e^{x_j} - e^{-x_j}}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}} \qquad \text{(Equation 2)}$$

3.1.3. Second Derivative

For $i \neq j$:

$$\begin{aligned}
\frac{\partial^2}{\partial x_i \partial x_j} lmax(\vec{x}) &= \frac{\partial}{\partial x_i} \left(\nabla lmax(\vec{x})\right)_j \\
&= \frac{\partial}{\partial x_i} \frac{e^{x_j} - e^{-x_j}}{\sum_{k=1}^{n} e^{x_k} + e^{-x_k}} \\
&= \left(e^{x_i} - e^{-x_i}\right) \cdot \frac{-1}{\left(\sum_{k=1}^{n} e^{x_k} + e^{-x_k}\right)^2} \cdot \left(e^{x_j} - e^{-x_j}\right) \\
&= -\left(\nabla lmax(\vec{x})\right)_i \cdot \left(\nabla lmax(\vec{x})\right)_j \\
&= -\left(\nabla lmax(\vec{x}) \cdot \nabla lmax(\vec{x})^T\right)_{ij}
\end{aligned}$$

For $i = j$:

$$\begin{aligned}
\frac{\partial^2}{\partial x_i \partial x_j} lmax(\vec{x}) &= \frac{\partial}{\partial x_i} \left(\nabla lmax(\vec{x})\right)_i \\
&= \frac{\partial}{\partial x_i} \frac{e^{x_i} - e^{-x_i}}{\sum_{k=1}^{n} e^{x_k} + e^{-x_k}} \\
&= \frac{\left(e^{x_i} + e^{-x_i}\right)\left(\sum_{k=1}^{n} e^{x_k} + e^{-x_k}\right) - \left(e^{x_i} - e^{-x_i}\right)\left(e^{x_i} - e^{-x_i}\right)}{\left(\sum_{k=1}^{n} e^{x_k} + e^{-x_k}\right)^2} \\
&= \frac{\left(e^{x_i} + e^{-x_i}\right)}{\sum_{k=1}^{n} e^{x_k} + e^{-x_k}} - \left(\nabla lmax(\vec{x}) \cdot \nabla lmax(\vec{x})^T\right)_{ii}
\end{aligned}$$

We can conclude that

$$\nabla^2 lmax(\vec{x}) = \mathsf{diag}_{j=1,\ldots,n}\left(\frac{e^{x_j} + e^{-x_j}}{\sum_{k=1}^{n} e^{x_k} + e^{-x_k}}\right) - \nabla lmax(\vec{x}) \cdot \nabla lmax(\vec{x})^T. \qquad \text{(Equation 3)}$$

3.1.4. Properties

We use the l_1- and l_∞-norms $\|\cdot\|_1$ and $\|\cdot\|_\infty$, defined as follows:

$$\|\vec{x}\|_1 \coloneqq \sum_{i=1}^{n} |x_i| \qquad \text{(Definition 4)}$$

$$\|\vec{x}\|_\infty \coloneqq \max_{i=1,\ldots,n} |x_i| \qquad \text{(Definition 5)}$$

We will now prove the following properties of the $lmax(\vec{x})$ function, as stated in [She13], where d is the dimension of the vector (i.e., $d = n$):

$$\begin{aligned}
\|\nabla lmax(\vec{x})\|_1 &\leq 1 && \text{(Property 6)} \\
\nabla lmax(\vec{x})^T \cdot \vec{x} &\geq lmax(\vec{x}) - \ln(2d) && \text{(Property 7)} \\
\|\nabla lmax(\vec{x}) - \nabla lmax(\vec{y})\|_1 &\leq \|\vec{x} - \vec{y}\|_\infty && \text{(Property 8)} \\
lmax(\vec{x}) &> \|\vec{x}\|_\infty && \text{(Property 9)}
\end{aligned}$$

Proof of Property 6:

$$
\begin{aligned}
\|\nabla lmax(\vec{x})\|_1 &\overset{1}{=} \sum_{i=1}^{n} |(\nabla lmax(\vec{x}))_i| && \text{(1) by Definition 4}\\
&\overset{2}{=} \sum_{i=1}^{n} \left| \frac{e^{x_i} - e^{-x_i}}{\sum_{j=1}^{n} e^{x_j} + e^{-x_j}} \right| && \text{(2) by Definition 1}\\
&\overset{3}{=} \frac{1}{\left| \sum_{j=1}^{n} e^{x_j} + e^{-x_j} \right|} \cdot \sum_{i=1}^{n} \left| e^{x_i} - e^{-x_i} \right| && \text{(3) by Distributivity}\\
&\overset{4}{\leq} \frac{1}{\left| \sum_{j=1}^{n} e^{x_j} + e^{-x_j} \right|} \cdot \sum_{i=1}^{n} \left| e^{x_i} + e^{-x_i} \right| && \text{(4) by Rev. Tri. Ineq., Pos. of } e^x\\
&\overset{5}{=} \frac{1}{\sum_{j=1}^{n} e^{x_j} + e^{-x_j}} \cdot \sum_{i=1}^{n} e^{x_i} + e^{-x_i} && \text{(5) by Pos. of } e^x\\
&= 1. \square
\end{aligned}
$$

For (4), The reverse triangle inequality states that $\|\vec{x}\| - \|\vec{y}\| \leq \|\vec{x} - \vec{y}\|$, so for $-1 \cdot \vec{y}$, it states $\|\vec{x}\| - \|\vec{y}\| \leq \|\vec{x} + \vec{y}\|$. Positivity of e^x implies $e^x = |e^x|$.
Proof of Property 9:

$$
\begin{aligned}
lmax(\vec{x}) &\overset{1}{=} \ln\left(\sum_{i=1}^{n} e^{x_i} + e^{-x_i} \right)\\
&> \ln\left(\sum_{i=1}^{n} e^{|x_i|} \right)\\
&\geq \ln\left(\max_{i=1,\ldots,n} \{e^{|x_i|}\} \right)\\
&\overset{2}{=} \ln\left(e^{\max_{i=1,\ldots,n}\{|x_i|\}} \right)\\
&\overset{3}{=} \ln\left(e^{\|\vec{x}\|_\infty} \right)\\
&= \|\vec{x}\|_\infty
\end{aligned}
$$

(1) by Definition 1, (2) by strictly increasing monotonicity of e^x, (3) by Definition 5.

3.2. Norms

A norm over $\mathbb{R}^n$ is a real-valued function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R} : x \mapsto \|x\|$ that satisfies the following properties:

1. Subadditivity: $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{R}^n$.
2. Absolute homogeneity: $\|s \cdot x\| = |s| \cdot \|x\|$ for all $x \in \mathbb{R}^n$, $s \in \mathbb{R}$.
3. Positive definiteness: if $\|x\| = 0$, then $x = \vec{0}$ follows.

The l_1 and l_∞ "norms" from Definition 4 and Definition 5 are thus actual norms.

3.3. Dual Norms

3.3.1. Definition

Suppose $\|\cdot\|$ to be a norm on $\mathbb{R}^n$. The *dual norm* of $\|\cdot\|$, $\|\cdot\|_*$, is defined for each $x \in \mathbb{R}^n$ as follows:

$$\|x\|_* \coloneqq \sup_{y \in \mathbb{R}^n} \{x^T y : \|y\| \leq 1\} \qquad \text{(Definition 10)}$$

3.3.2. Examples

l_1 and l_∞ are dual norms of each other: The supremum of $x^T y$ for $\|y\|_1 \leq 1$ has $y_i = \pm 1$ for i as the index with maximal value of all $\pm x_i$, and $y_j = 0$ for all other $j \neq i$. This gives a supremum of $\sup\{x^T y\} = \max_{i=1,\ldots,n}\{|x_i|\}$, which is the l_∞ norm. On the other hand, when restricting $\|y\|_\infty \leq 1$ instead, we get the supremum by setting all values y_i to ± 1 according to the sign of x_i, resulting in the value $\sum_{i=1}^n |x_i|$, which is the l_1 norm.
Another example is the l_2 norm: it is self-dual. In geometric terms, $\|y\|_2 \leq 1$ restricts y to the surface of the unit hypersphere, and $x^T y$ gets maximal when choosing y as $\lambda \cdot x$ with $\lambda = \frac{1}{\|x\|_2}$. This supremum evaluates to $x^T \frac{x}{\|x\|_2} = \frac{\|x\|_2^2}{\|x\|_2} = \|x\|_2$.

3.3.3. Properties

For $\|\cdot\|$ and its dual norm $\|\cdot\|_*$, it holds that

$$x^T y \leq \|x\|_* \cdot \|y\|. \qquad \text{(Equation 11)}$$

A short proof: If $\|y\| = 0$, $y = 0$ and the equality is trivial. Else,

$$\begin{aligned}
x^T y &= \|y\| \cdot \left(x^T \frac{y}{\|y\|} \right) \\
&\leq \|y\| \cdot \sup_{z \in \mathbb{R}^n} \left\{ x^T \frac{z}{\|z\|} \right\} \\
&= \|y\| \cdot \sup_{z \in \mathbb{R}^n} \left\{ x^T z : \|z\| = 1 \right\} \\
&\leq \|y\| \cdot \sup_{z \in \mathbb{R}^n} \left\{ x^T z : \|z\| \leq 1 \right\} \\
&= \|y\| \cdot \|x\|_*.
\end{aligned}$$

3.4. Algorithmic Mathematics

We postpone the definition of some symbols to later sections, where they are also elaborated. For an overview, those definitions that are relevant and other useful equations for this section are:

$$\begin{aligned}\phi(f) &\coloneqq lmax\left(C^{-1}f\right) + lmax\left(2\alpha \cdot R \cdot (b - Bf)\right)\\ \nabla\phi(f) &= \left(C^{-1}\right)^T \cdot \nabla lmax\left(C^{-1}\right) - 2\alpha \cdot B^T R^T \cdot \nabla lmax\left(2\alpha \cdot R \cdot (b - Bf)\right)\\ \Phi(f) &\coloneqq \left\|C^{-1}(f)\right\|_\infty + 2\alpha \left\|R(b - B(f))\right\|_\infty\end{aligned}$$

3.4.1. Almost Route

The algorithm `AlmostRoute` will be discussed later in Section 5.1. The mathematical proofs are moved to this section.

3.4.1.1 Flow Optimality

In this section, we can use

$$\begin{aligned}&\phi(f) \geq 16\varepsilon^{-1}\ln(n),\\ &\delta \coloneqq \|C\nabla\phi(f)\|_1,\\ &v \coloneqq R^T(\nabla lmax(2\alpha R(b - Bf)) \text{ and}\\ &\delta < \varepsilon/4.\end{aligned}$$

[She13] states the Lemma

$$VertC^{-1}f\|_\infty + 2\alpha\|R(b - Bf)\|_\infty \leq (1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1} \text{with } v \quad = R^T \nabla lmax(2\alpha R(b - Bf))$$

and provides a proof that proceeds as follows:

$$\begin{aligned}2\alpha\|CB^T v\|_1 &= \|\nabla lmax(C^{-1}f)\|_1 - \|\nabla lmax(C^{-1}f)\|_1 + 2\alpha\|CB^T v\|_1\\ &\overset{1}{\leq} \|\nabla lmax(C^{-1}f)\|_1 + \|\nabla lmax(C^{-1}f) - 2\alpha CB^T v\|_1\\ &= \|\nabla lmax(C^{-1}f)\|_1 + \|C\nabla\phi(f)\|_1\\ &\overset{2}{=} \|\nabla lmax(C^{-1}f)\|_1 + \delta\\ &\overset{3}{\leq} 1 + \delta\end{aligned}$$

(1) follows from the reverse triangle inequality together with $\|x\|_1 = \| - x\|_1$, (2) follows from the definition of δ and (3) follows from Property 6. We continue the proof from [She13]:

$$
\begin{aligned}
\delta\phi(f) &\overset{1}{=} \|C\nabla\phi(f)\|_1 \cdot \phi(f) \\
&\overset{2}{\geq} \|C\nabla\phi(f)\|_1 \|C^{-1}f\|_\infty \\
&\overset{3}{\geq} (C\nabla\phi(f))^T (C^{-1}f) \\
&= \nabla\phi(f)^T f \\
&\overset{4}{=} (C^{-1}\nabla lmax(C^{-1}f) - 2\alpha B^T R^T \nabla lmax(2\alpha R(b - Bf))^T f \\
&\overset{5}{=} (\nabla lmax(C^{-1}f))^T C^{-1} f - 2\alpha(\nabla lmax(2\alpha R(b - Bf))^T RBf \\
&= (\nabla lmax(C^{-1}f))^T C^{-1} f + 2\alpha(\nabla lmax(2\alpha R(b - Bf))^T R(b - Bf) - \\
&\quad 2\alpha(\nabla lmax(2\alpha R(b - Bf))^T Rb \\
&= (\nabla lmax(C^{-1}f))^T C^{-1} f + 2\alpha(\nabla lmax(2\alpha R(b - Bf))^T R(b - Bf) - \\
&\quad 2\alpha b^T R^T (\nabla lmax(2\alpha R(b - Bf)) \\
&\overset{6}{=} (\nabla lmax(C^{-1}f))^T C^{-1} f + 2\alpha(\nabla lmax(2\alpha R(b - Bf))^T R(b - Bf) - 2\alpha b^T v \\
&\overset{7}{\geq} lmax(C^{-1}f) - \ln(2m) + lmax(2\alpha R(b - Bf)) - \ln(2 \cdot \mathsf{rows}(R)) - 2\alpha b^T v \\
&\overset{8}{=} \phi(f) - ln(4m \cdot \mathsf{rows}(R)) - 2\alpha b^T v \\
&\overset{9}{\geq} \phi(f) - 4\ln(n) - 2\alpha b^T v \\
&\overset{10}{\geq} \phi(f)(1 - \varepsilon/4) - 2\alpha b^T v
\end{aligned}
$$

(1) by definition of δ.
(2) by definition of $\phi(f)$, positivity of $lmax$ and Property 9.
(3) by Equation 11.
(4) by Equation 12 and symmetry of C^{-1}.
(5) by basic matrix and vector calculation rules.
(6) by definition of v.
(7) by Property 7.
(8) by Def. of $lmax$-Potential and basic rules for logarithms.
(9) by using the facts $m \leq n^2/2$ and $\mathsf{rows}(R) \leq n^2/2$.
(10) by $\phi(f) \geq 16\varepsilon^{-1}\ln(n)$ and thus $\phi(f) \cdot (\varepsilon/4) \geq 4\ln(n)$.
This gives us the inequality $2\alpha b^T v \geq \phi(f)(1 - \varepsilon/4 - \delta)$.

As in [She13], we can use the two last proofs for another inequality:

$$\begin{aligned}\frac{b^T v}{\|CB^T v\|_1} &= \frac{2\alpha b^T v}{2\alpha\|CB^T v\|_1} \\ &\overset{1}{\geq} \frac{\phi(f)(1-\varepsilon/4-\delta)}{1+\delta} \\ &\overset{2}{>} \frac{\phi(f)(1-\varepsilon/2)}{1+\varepsilon/4} \\ &\overset{3}{\geq} \frac{\phi(f)}{1+\varepsilon} \\ &\overset{4}{>} \frac{\Phi(f)}{1+\varepsilon}\end{aligned}$$

(1) by the two last proofs.
(2) by $\delta < \varepsilon/4$.
(3) by $a_1(x) \geq a_2(x)$ with $a_1(x) = \frac{1-x/2}{1+x/4}$ and $a_2(x) = \frac{1}{1+x}$ for $x \in [0, 1/2]$. For $x \in \{0, 1/2\}$, $a_1(x) = a_2(x)$. The highest value for $|a_1(x) - a_2(x)|$ in this interval is approx. 0.0239322565748 at $x \approx 0.2174822697048$, the highest value for $\frac{a_1(x)}{a_2(x)}$ in this interval is approx. 1.0294372515229 at $x \approx 0.2426406987879$.
(4) by Property 9.

3.4.1.2 Upper Limit of Iterations

Only for this section, we will use n as the size of the flow vectors.
We want to prove $\phi(f+h) \leq \phi(f) - \Omega(\varepsilon^2\alpha^{-2})$, where $h_e = -\frac{\delta}{1+4\alpha^2}\text{sgn}(\nabla_f \phi(f)_e)c_e$ and $\delta \coloneqq \|C\nabla\phi(f)\|_1$.
First, approximating $\phi(f+h)$ with the Taylor series gives us $\phi(f+h) \leq \phi(f) + \nabla\phi(f)^T h + \frac{1}{2}h^T \cdot \left(\nabla^2\phi(f)\right) \cdot h$. We first prove $h^T \cdot \left(\nabla^2\phi(f)\right) \cdot h \leq (1+4\alpha^2) \cdot \|C^{-1}h\|_\infty^2$:

$$\begin{aligned}h^T\left(\nabla^2\phi(f)\right)h &= h^T \cdot \left(\nabla^2 lmax(C^{-1}f) + 2\alpha B^T R^T\left(\nabla^2 lmax(2\alpha R(b-Bf))\right)\cdot 2\alpha RB\right)\cdot h \\ &= h^T\cdot\left(\nabla^2 lmax(C^{-1}f)\right)\cdot h + h^T\cdot\left(2\alpha B^T R^T\left(\nabla^2 lmax(2\alpha R(b-Bf))\right)\cdot 2\alpha RB\right)\cdot h \\ &\overset{1}{\leq} \|C^{-1}h\|_\infty^2 + \|2\alpha RBh\|_\infty^2 \\ &= \|C^{-1}h\|_\infty^2 + \|2\alpha RBC\cdot C^{-1}h\|_\infty^2 \\ &\overset{2}{=} \|C^{-1}h\|_\infty^2 + 4\alpha^2\cdot\|RBC\cdot C^{-1}h\|_\infty^2 \\ &\overset{3}{\leq} \|C^{-1}h\|_\infty^2 + 4\alpha^2\cdot\|C^{-1}h\|_\infty^2 \\ &\overset{4}{=} (1+4\alpha^2)\cdot\|C^{-1}h\|_\infty^2\end{aligned}$$

(1) by $x^T(\nabla^2 lmax(y))x \leq \|x\|_\infty^2$ for all $x, y \in \mathbb{R}^n$, without proof.
(2) and (4) by properties of norms.
(3) by using $\|RBC\|_{\infty\to\infty} \leq 1$ for congestion approximators R. This means $\sup_{x\in\mathbb{R}^n}\frac{\|RBCx\|_\infty}{\|x\|_\infty} \leq 1$, thus $\frac{\|RBCx\|_\infty}{\|x\|_\infty} \leq 1$ for all $x \in \mathbb{R}^n$, implying $\|RBCx\|_\infty \leq \|x\|_\infty$ for all $x \in \mathbb{R}^n$.

We combine those two inequalities and continue:

$$
\begin{aligned}
\phi(f+h) &\overset{1}{\leq} \phi(f) + \nabla\phi(f)^T h + \frac{1}{2}\cdot h^T\left(\nabla^2\phi(f)\right)h \\
&\overset{2}{\leq} \phi(f) + \nabla\phi(f)^T h + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\|C^{-1}h\|_\infty^2 \\
&= \phi(f) + \left(\sum_{j=1}^{n}\left(\nabla\phi(f)\right)_j h_j\right) + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\{|c_i^{-1}h_i|\}\right)^2 \\
&\overset{3}{=} \phi(f) + \left(\sum_{j=1}^{n}\left(\nabla\phi(f)\right)_j\cdot\left(-\frac{\delta}{1+4\alpha^2}\mathsf{sgn}(\nabla_f\phi(f)_j)c_j\right)\right) \\
&\quad + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\left\{c_i^{-1}\left|-\frac{\delta}{1+4\alpha^2}\mathsf{sgn}(\nabla_f\phi(f)_i)c_i\right|\right\}\right)^2 \\
&\overset{4}{=} \phi(f) - \frac{\delta}{1+4\alpha^2}\cdot\left(\sum_{j=1}^{n}|(\nabla\phi(f))_j|\cdot c_j\right) \\
&\quad + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\left\{c_i^{-1}\left|-\frac{\delta}{1+4\alpha^2}\mathsf{sgn}(\nabla_f\phi(f)_i)c_i\right|\right\}\right)^2 \\
&= \phi(f) - \frac{\delta}{1+4\alpha^2}\cdot\|C\nabla\phi(f)\|_1 \\
&\quad + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\left\{c_i^{-1}\left|-\frac{\delta}{1+4\alpha^2}\mathsf{sgn}(\nabla_f\phi(f)_i)c_i\right|\right\}\right)^2 \\
&\overset{5}{=} \phi(f) - \frac{\delta^2}{1+4\alpha^2} + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\left\{c_i^{-1}\left|-\frac{\delta}{1+4\alpha^2}\mathsf{sgn}(\nabla_f\phi(f)_i)c_i\right|\right\}\right)^2 \\
&= \phi(f) - \frac{\delta^2}{1+4\alpha^2} + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\left(\max_{i=1,\dots,n}\left\{\left|\frac{\delta}{1+4\alpha^2}\right|\right\}\right)^2 \\
&= \phi(f) - \frac{\delta^2}{1+4\alpha^2} + \frac{1}{2}\cdot(1+4\alpha^2)\cdot\frac{\delta^2}{(1+4\alpha^2)^2} \\
&= \phi(f) - \frac{\delta^2}{2+8\alpha^2} \\
&\overset{6}{\leq} \phi(f) - \frac{\varepsilon^2/16}{2+8\alpha^2} \\
&= \phi(f) - \Omega(\varepsilon^2\alpha^{-2}).
\end{aligned}
$$

(1) and (2) by Taylor approximation and $h^T(\nabla^2\phi(f))h \leq (1+4\alpha^2)\cdot\|C^{-1}h\|_\infty^2$.

(3) by definition of h.

(4) by $x\cdot\mathsf{sgn}(x) = |x|$ for all $x\in\mathbb{R}^n$.

(5) by definition of δ.

(6) by $\delta \geq \varepsilon/4$: This is valid for each iteration that does a step, by design of the algorithm `AlmostRoute`.

For a complete proof, only $x^T(\nabla^2 lmax(y))x \leq \|x\|_\infty^2$ still needs to be proven, but we keep it as an assumption in this thesis due to temporal reasons.

Inserting a step size factor s and changing the step to $f + s\cdot h$ yields the inequality $\phi(f+s\cdot h) \leq \phi(f) - \frac{\delta^2}{2+8\alpha^2}\cdot(s^2-2s)$. Since this is just an upper limit, the actual potential could still be decreasing, even with $s<0$ or $s>2$. But modelling with this equation yields the value $s=1$ as choice for the

step size factor to optimize this inequality's prediction. As we will see in the results, calculating a dynamic s still improves the number of iterations of `AlmostRoute` for our unit-capacity multidimensional grid graphs.

3.4.2. Complete Route

We follow the proof from [She13] to show that `CompleteRoute` outputs a flow f with $Bf = b$ and $\Phi(f) \leq (1+\varepsilon)\frac{b_{S_0}}{c_{S_0}}$, with S_0 as the potential threshold cut of the first call of `AlmostRoute`.
If `AlmostRoute` is implemented *with* return of the potential threshold cut S, we can write $(f, S) =$ `AlmostRoute`(b, ε) for the returned objects.
We have following definitions:

$$\begin{aligned} T &= \log_2(2m), \\ b_0 &= b, \\ (f_0, S_0) &= \texttt{AlmostRoute}(b_0, \varepsilon), \\ b_i &= b_{i-1} - Bf_{i-1}, \\ (f_i, S_i) &= \texttt{AlmostRoute}(b_i, 1/2) \text{ for all } i = 1, ..., T, \\ b_{T+1} &= b_T - Bf_T \text{ and} \\ f_{T+1} &= \texttt{GridMST.route}(b_{T+1}). \end{aligned}$$

Also, we assume that $\varepsilon \in [0, 1/2]$ and `AlmostRoute`(b, ε) has $\mathcal{O}(\alpha\varepsilon^{-2}\ln(n))$ iterations and the returned f and S satisfy $\Phi(f) \leq (1+\varepsilon)\frac{b_S}{c_S}$. For $i > 0$, we also assume $\Phi(f_i) \leq (1+\varepsilon)\text{opt}(b_i)$, as shown in Lemma 2.2 in [She13].
Together with $(b_i - Bf_i) = b_{i+1}$, this gives us following equations:

$$\begin{aligned} \Phi(f_0) &= \|C^{-1}f_0\|_\infty + 2\alpha\|Rb_1\|_\infty \leq (1+\varepsilon)\frac{b_{S_0}}{c_{S_0}} \\ \Phi(f_i) &= \|C^{-1}f_i\|_\infty + 2\alpha\|Rb_{i+1}\|_\infty \leq (3/2)\text{opt}(b_i) \overset{1}{\leq} (3/2)\alpha\|Rb_i\|_\infty \end{aligned}$$

(1) by 4.1.
The second inequality can be rearranged to $2\alpha\|Rb_i\|_\infty \geq \|C^{-1}f_i\|_\infty + 2\|Rb_{i+1}\|_\infty + (1/2)\|Rb_i\|_\infty$. The inequality $2\alpha\|Rb_i\|_\infty \geq \|C^{-1}f_i\|_\infty + 2\|Rb_{i+1}\|_\infty$ follows.

Iterative application then yields:

$$
\begin{aligned}
(1+\varepsilon)\frac{b_{S_0}}{c_{S_0}} &\geq \|C^{-1}f_0\|_\infty + 2\alpha\|Rb_1\|_\infty \\
&\geq \|C^{-1}f_0\|_\infty + \|C^{-1}f_1\|_\infty + 2\|Rb_2\|_\infty + (1/2)\|Rb_1\|_\infty \\
&\geq \|C^{-1}f_0\|_\infty + \|C^{-1}f_1\|_\infty + \|C^{-1}f_2\|_\infty + 2\|Rb_3\|_\infty + (1/2)\|Rb_2\|_\infty + (1/2)\|Rb_1\|_\infty \\
&\vdots \\
&\geq (1/2)\alpha\|Rb_1\|_\infty + ... + (1/2)\alpha\|Rb_T\|_\infty + \|C^{-1}f_0\|_\infty + ... + \|C^{-1}f_T\|_\infty + 2\alpha\|Rb_{T+1}\|_\infty \\
&\geq (1/2)\alpha\|Rb_1\|_\infty + \|C^{-1}f_0\|_\infty + ... + \|C^{-1}f_T\|_\infty \\
&\overset{1}{\geq} \|C^{-1}f_{T+1}\|_\infty + \|C^{-1}f_0\|_\infty + ... + \|C^{-1}f_T\|_\infty \\
&\overset{2}{\geq} \|C^{-1}(f_0 + ... + f_{T+1})\|_\infty
\end{aligned}
$$

(1) is proven by $\|C^{-1}f_{T+1}\|_\infty \leq (1/2)\alpha\|Rb_1\|_\infty$ in [She13].
(2) by subadditivity of $\|\cdot\|_\infty$.
This means that $f = \sum_{i=0}^{T+1} f_i$ satisfies $\|C^{-1}f\|_\infty \leq (1+\varepsilon)\frac{b_{S_0}}{c_{S_0}}$.
Finally, observe that f_{T+1} routes b_{T+1} exactly (i.e. $b_{T+1} = Bf_{T+1}$), and thus, $B\left(\left(\sum_{i=0}^{T} f_i\right) + f_{T+1}\right) = \left(\sum_{i=0}^{T} Bf_i\right) + Bf_{T+1} = \sum_{i=0}^{T}(b_i - b_{i+1}) + b_{T+1} = b_0 = b$. This means that $f = \sum_{i=0}^{T+1} f_i$ satisfies $Bf = b$.

4. Grid Graphs

Since constructing good congestion approximators is not always straight-forward, we examine the algorithm for simple multidimensional grid graphs with undirected unit capacity edges as an example. In this thesis, most examples shown in figures will use a 2-dimensional 4×4 example grid.

4.1. Graph & Data Structures

4.1.1. Grid Graph

The vertices of a grid graph can be embedded and then indexed in a multidimensional integer grid. With integer tuple representations for the vertices, they can be written as the set $V = \bigtimes_{i=0}^{d-1} [n_i]_{index}$, where d is the dimension of the grid graph, n_i the number of vertices in the i-th dimension and $[n]_{index} \coloneqq \{0, ..., n-1\}$. For example, a grid graph with $d = 3$ and $(n_0, n_1, n_2) = (2, 7, 4)$ has vertices from $(0, 0, 0)$ to $(1, 6, 3)$.

The edges of a grid graph are exactly between direct neighbours, or more precise: exactly between each two vertices whose tuple representations are the same for $d-1$ entries and differ in the remaining dimension by 1. Formally, $E = \{\{v_1, v_2\} \in \binom{V}{2} \mid \|(v_1 - v_2)\|_1 = 1\}$, where $\binom{V}{2}$ denotes the set of all subsets of V containing 2 elements.

Thus, a grid graph can be fully characterized by a dimensional vector $(n_1, ..., n_d)$ containing the information of nodes per dimension, and the total dimension as its length. More about the implementation will be discussed in Chapter 6.

An examplary grid graph can be seen in Figure 2. An enumeration of the nodes will be convenient for further investigation; more details will also be part of the implementation chapter.

4.1.2. Grid Flow

A *flow with divergence* $d : V \to \mathbb{R}$ is a function $f : V^2 \to \mathbb{R}_\infty$ that satisfies the following constraints:

- Skew symmetry: $f(u, v) = -f(v, u)$ for all $u, v \in V$.
- Capacity constraints: $f(u, v) \leq c(u, v)$, where $c : V^2 \to \mathbb{R}_\infty$ is the capacity function (in our case, if $\{u, v\}$ is an edge of the grid graph, then $c(u, v) = 1$, else $c(u, v) = 0$).
- Flow conservation with divergence d: The net flow at u, $f(u) \coloneqq \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$, has to equal the divergence at u: for all u, $f(u) = d(u)$.

An example can be seen in Figure 3.

4.1.3. Grid Demand & Divergence

The grid demand is part of the input and specifies the demanded excess at each vertex. The excess at vertex u is defined as the net flow at u, meaning “incoming flow minus outgoing flow”, or formally: $f(u) \coloneqq \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v)$ (same as in Section 4.1.2).

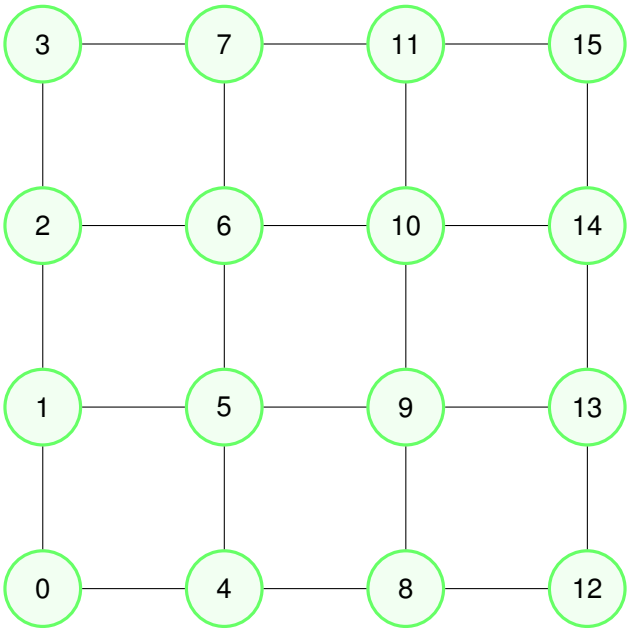


Figure 2 Two-dimensional 4×4 Grid Graph. All edges are undirected and have unit capacity 1. The numbers in the nodes correspond to their index, which is the output of the bijective enumeration function $\texttt{index} : \mathbb{N}_0^d \to \mathbb{N}_0$.

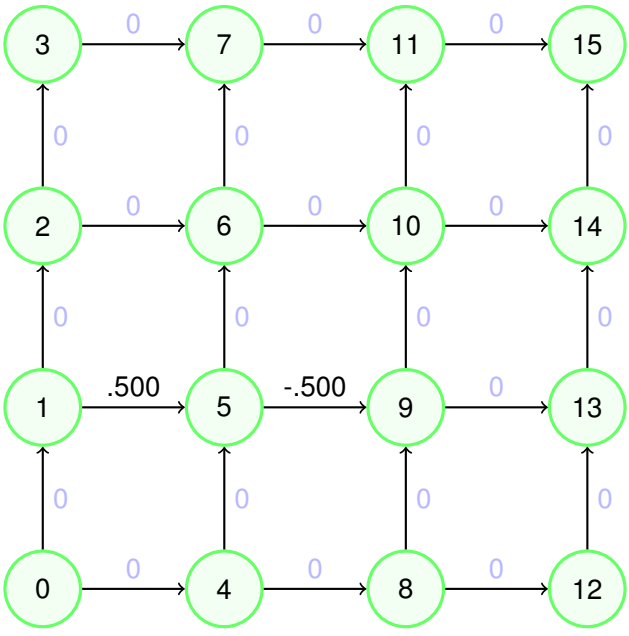


Figure 3 Example flow with divergence for our example grid graph. Edges are directed from lower to higher index. Flow from higher index to lower index node is represented with negative flow along the reversed edge. This chart represents a flow from 1 to 5 with value 0.5 and a flow from 9 to 5 with value 0.5. The divergence is $d(u) = -0.5$ for vertices 1 and 9, $d(5) = +1$ and 0 for all other vertices.

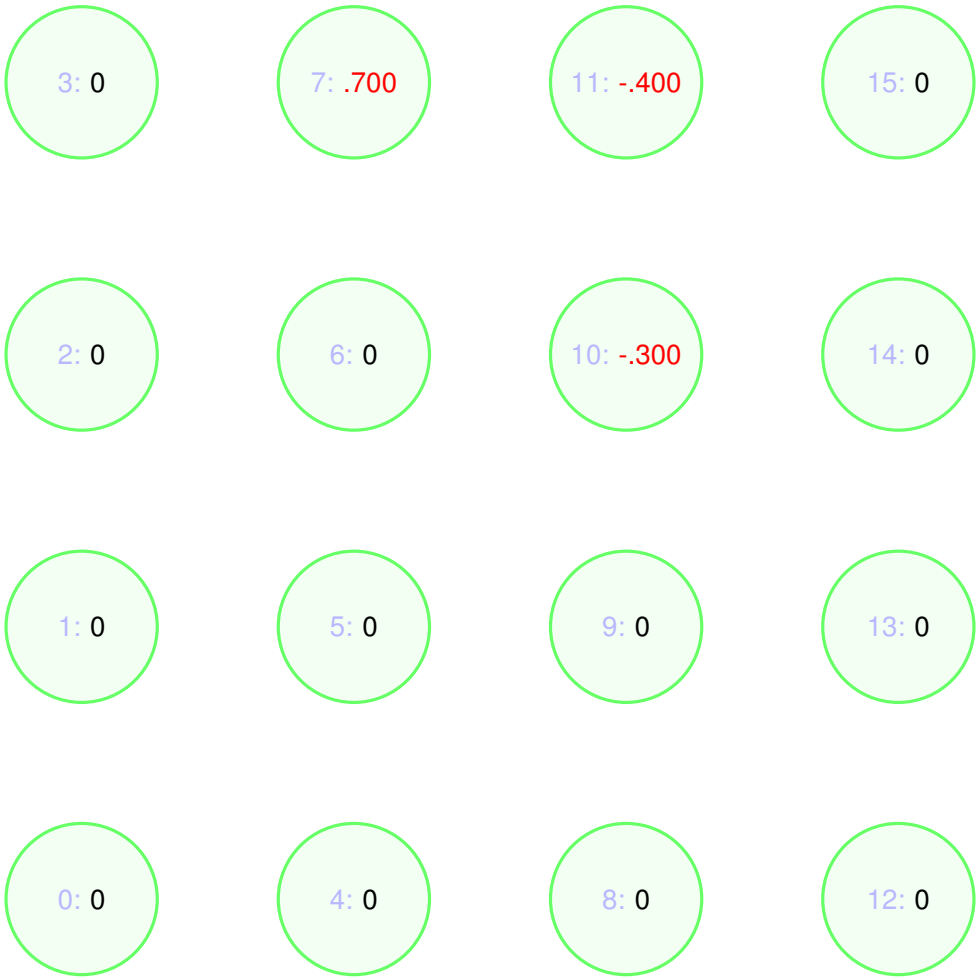


Figure 4 Example grid demand for our example grid graph. Since "demand" refers to the demanded divergence, the actual divergence and the demand share this representation. This figure shows a visualization of the implemented data structure (actually code-generated TikZ-input), which does not store information about neighbourhood of vertices; hence, the edges aren't visualized here.

The *divergence* encapsulates the excesses at each vertex into one mathematical object. When viewing the vertices of our graph abstractly, it is convenient to define it as a function with mapping $V \to \mathbb{R}$. The implementation will use indices instead, so in this context, the divergence can be written as vector or array in $\mathbb{R}^{|V|}$; more details in Chapter 6.
An exemplary grid demand can be seen in Figure 4 - this divergence could be achieved by sending 0.4 from 11 to 7, 0.3 from 10 to 6 and 0.3 from 6 to 7, though it is not a minimally congested flow.

4.1.4. Grid Approximator

When trying to achieve a flow with divergence d, even for our small example exist infinitely many solutions. The problem of finding a solution of this infinite set, which is optimal in a certain property, can be too complex. An idea to improve the complexity is the use of an approximator that estimates the property that is subject to optimization, where this approximator makes the calculation more efficient.
The property we try to optimize is the maximum congestion: the aim is to find a flow that has minimal maximum congestion, where maximum congestion of a flow is defined as $\max_{e \in E} \left| \frac{f_e}{c_e} \right|$ with f_e as

flow along edge e and c_e as capacity of edge e. Formally, as in [She13], but with our notation: Our optimization problem for demand b is

$$\min_f \left(\max_{e \in E} \left| \frac{f_e}{c_e} \right| \right) \quad s.t.\ B(f) = b \qquad \text{(Optimization Problem)}$$

where B maps a flow to its divergence.
A useful property is the propagation of additivity from flows to divergences: If f is a flow with divergence d and g is a flow with divergence e, the flow $f + g$ has divergence $d + e$. Proof:

$$\begin{aligned} f(u) + g(u) &= \sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) + \sum_{v \in V} g(v,u) - \sum_{v \in V} g(u,v) \\ &= \sum_{v \in V} (f(v,u) + g(v,u)) - \sum_{v \in V} (f(u,v) + g(u,v)) \\ &= \sum_{v \in V} (f+g)(v,u) - \sum_{v \in V} (f+g)(u,v) \\ &= (f+g)(u) \text{ for all } u \in V. \end{aligned}$$

Hence, $d + e$ equals the divergence of $f + g$ in each $u \in V$. □
For finding a flow satisfying a specific divergence/demand d_x, it is thus possible to use a flow f with divergence d_f that doesn't satisfy the demand yet, and then adding a flow g with divergence d_g equal to the residual demand $d_x - d_f$ s.t. $d_f + d_g = d_x$, meaning $f + g$ is the final flow satisfying d_x. We can use this property to design algorithms for finding an optimal flow by using iteration: induction directly shows that an arbitrary sum of flows has divergence equal to the sum of the individual flow divergences. We can start with an initial flow f_0 (default: $f_0 = 0$) and then use some strategy to find some flow f_1 based on the residual demand such that hopefully, the residual demand will vanish at some point when using this strategy multiple times. An idea is to introduce a metric and choose a strategy with which this metric monotonically increases/decreases and where some extreme point of this metric represents the optimal solution.
As proposed in [She13], our metric is built around an approximator: the maximum congestion is metricized by the potential function

$$\Phi(f) := \left\| C^{-1}(f) \right\|_\infty + 2\alpha \left\| R(b - B(f)) \right\|_\infty, \qquad \text{(Def. of Potential)}$$

where C^{-1} maps a flow to another function g with $g(u,v) = \frac{f(u,v)}{c(u,v)}$ where $c(u,v)$ is the capacity of edge $\{u,v\}$, α is a parameter describing some property of the approximator in regard of its quality, b the initial demand, f the input flow to be metricized, B mapping a flow to its divergence, R the approximator (mapping a divergence to some collection of approximative metrics, i.e. this metric can be a vector) and $\|\cdot\|_\infty$ the conventional l_∞-norm applied to the respective vector or to the image (for maps). At this point, we introduce the approximator definition from [She13], converted into abstract terms for graphs:
An α-*congestion-approximator* is a map $R : \mathbb{R}^V \to \mathbb{R}^{n_R}$ that satisfies for all demands b:

$$\|R(b)\|_\infty \leq \text{opt}(b) \leq \alpha \|R(b)\|_\infty \qquad (4.1)$$

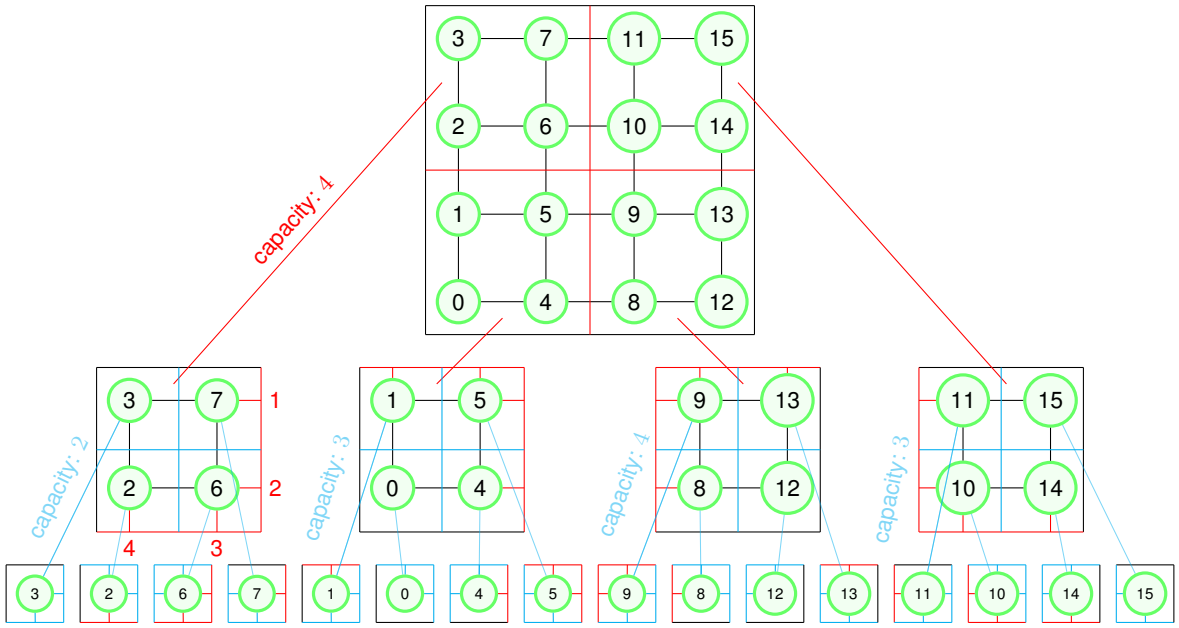


Figure 5 Schematic view of the approximator. Each node of the approximator tree represents a subdivision: the root represents the graph itself, the leaves represent a single vertex in the original grid graph. The capacity assigned to a tree edge is the capacity of the cut of the child tree node's represented subdivision.

where n_R is the number of approximation metrics of R (for a 4×4 grid graph, we will use an approximator with $n_R = 20$) and opt(b) is the optimal solution of the Optimization Problem. We don't yet know the smallest valid α for our approximator and will try to find it in an experimental evaluation. Now, we finally describe the approximator that we will use. A *grid approximator*, an approximator for grid graphs, is constructed as a tree structure. The tree nodes represent subdivisions of the main graph, starting with the whole graph at the root tree node, and the vertices of this subdivision always fill a multidimensional box. If the subdivision represented by a tree node only contains one graph node, it is a leaf. Else, it is further divided, where the tree nodes representing the new subdivisions are the children to be assigned. The box is thereby divided into half in each dimension, except those dimensions where this box has only length 1 (length of a dimension: number of vertices along this dimension when taking a single line of the multidimensional box). An example of this subdivision can be seen in Figure 5.

The approximation is now done tree-node-wise by interpreting the subdivisions - except the root node - as cut and calculating their excess flow and capacity. For convenience, this will be labelled to the edge from the tree node representing this cut to their parent tree node - by this labelling, the maximum congestion of the tree equals the l_∞-norm of the approximator output for a divergence, $\|R(b)\|_\infty$. Some example capacities can also be seen in Figure 5.

Given a demand b, it is easy to calculate the tree: the leaves correspond to a single graph node v, so the excess flow of this cut is the same as the excess flow for this graph node in b, $b(v)$. This gives a flow value that we can "already write on top of the fraction $\frac{f}{c}$" for the edges to the leaf level. Now we use another useful property regarding the addition of flows: With $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ disjoint subgraphs of $G = (V, E)$ and given net flows $f(V_1), f(V_2)$ for those subgraphs, the subgraph $G_3 = (V_3, E_3)$ of G with $V_3 = V_1 \cup V_2$ and $E_3 = E_1 \cup E_2 \cup \{\{v_1, v_2\} \in E : v_1 \in V_1, v_2 \in V_2\}$ has

net flow $f(V_1) + f(V_2)$. Proof:

$$
\begin{aligned}
f(V_1) + f(V_2) &= \sum_{u \notin V_1, v \in V_1} f(u,v) + \sum_{u \notin V_2, v \in V_2} f(u,v) \\
&= \textstyle\sum_{u \in V_2, v \in V_1} f(u,v) + \sum_{u \notin (V_1 \cup V_2), v \in V_1} f(u,v) + \sum_{u \in V_1, v \in V_2} f(u,v) + \sum_{u \notin (V_2 \cup V_1), v \in V_2} f(u,v) \\
&= \sum_{u \in V_2, v \in V_1} f(u,v) + \sum_{u \notin (V_1 \cup V_2), v \in (V_1 \cup V_2)} f(u,v) + \sum_{u \in V_2, v \in V_1} -f(u,v) \\
&= \sum_{u \notin (V_1 \cup V_2), v \in (V_1 \cup V_2)} f(u,v) \\
&= f(V_1 \cup V_2). \square
\end{aligned}
$$

This means that we can find the excess flow for cuts represented by higher-level tree nodes by simply adding the excess flows of the cuts represented by its children.

For multidimensional grid graphs with unit-capacity undirected edges, there is also a way to calculate the capacity of a cut, given its boundaries inside the graph. As an example, consider a 4-dimensional graph with corners $(0,0,0,0)$ and $(4,5,6,7)$ and a multidimensional box cut with corners $(0,0,1,1)$ and $(4,2,1,7)$. There are four dimensions along which an edge can exist: from (a,b,c,d) to $(a \pm 1, b, c, d)$, $(a, b \pm 1, c, d)$, $(a, b, c \pm 1, d)$ and $(a, b, c, d \pm 1)$. Each vertex $(0,b,c,d)$ of our cut does not have an edge to vertex $(-1,b,c,d)$, as this would be outside of the graph - so there, no edge exists in this direction along this dimension. Also, we are not interested in the edges to $(1,b,c,d)$, as they are not part of the cut: we "remain" in the box. The vertices $(4,b,c,d)$ of our box also cannot have an edge to $(5,b,c,d)$, as this is also no vertex of our graph. So this dimension adds nothing to our cut capacity.
In the next dimension, the same holds for the lower index boundary. But for the higher index boundary, there exist edges between a vertex of our cut $(a,2,c,d)$ and $(a,3,c,d)$. This applies to all a, c, d for which $(a,2,c,d)$ is in our box, and there are 5 possible values for a, 1 for c and 8 for d. Thus, there are $5 \cdot 1 \cdot 8 = 40$ edges contributing to the cut in this dimension.
With the multidimensional size vector $\vec{n} = (n_1, ..., n_d)$ for our box cut, where n_i is the amount of vertices along dimension i in this cut, we can generalize that along dimension i, there are either 0, $\prod_{j \neq i} n_j$ or $2 \cdot \prod_{j \neq i} n_j$ edges that contribute to our cut, depending on whether 0, 1 or 2 boundaries are shared with the graph boundary. When we will have calculated the total amount of contributing edges, the result will also be the capacity of the cut, since we have unit-capacity edges. We will now construct a generalized formula for this capacity.

Let G be a multidimensional grid graph of dimension d with unit-capacity undirected edges. Let $(n_1, ..., n_d)$ be the vertex counts along each dimension. W.l.o.g., the grid graph has vertices with integer coordinates in the box $(0, ..., 0)$ to $(n_1 - 1, ..., n_d - 1)$. Let C be a cut, including all vertices in the multidimensional box from $(a_1, ..., a_d)$ to $(b_1, ..., b_d)$ with $0 \leq a_i \leq b_i \leq n_i - 1$ for all $i = 1, ..., d$.

Then, the capacity of C in G is

$$\sum_{i=1}^{d} \left((\overline{\delta}_{a_i,0} + \overline{\delta}_{b_i,n_i-1}) \cdot \prod_{j=1,j\neq i}^{d} (b_j - a_j + 1) \right), \tag{4.2}$$

where $\overline{\delta}_{x,y} = 1 - \delta_{x,y}$ with δ as Kronecker-Delta, as already defined in Derivative of variables. We can make this formula a bit more handy by precalculating the total amount of nodes in C.

$$n_C \coloneqq \prod_{j=1}^{d} (b_j - a_j + 1)$$

$$\prod_{j=1,j\neq i}^{d} (b_j - a_j + 1) = \frac{\prod_{j=1}^{d}(b_j - a_j + 1)}{(b_i - a_i + 1)} = \frac{n_C}{(b_i - a_i + 1)}$$

$$\text{cap}(C) = \sum_{i=1}^{d} \left((\overline{\delta}_{a_i,0} + \overline{\delta}_{b_i,n_i-1}) \cdot \frac{n_C}{(b_i - a_i + 1)} \right)$$

Now, we will have another look on the approximator with matrix and vector notation. Since a vector provides distinction of the different value informations by ordering, we need an enumeration of the vertices and edges of the grid graph as well as of the edges of the approximmator tree structure. These enumerations can be arbitrary, but have to be consistent within operations (including matrix multiplication). Our enumeration schemes are defined in the implementation, i.e. by the function `index`, so we will only work with an example here and move the details to the implementation chapter (Chapter 6).
We use the 4×4 example grid again. Let us choose the enumeration as in Figure 2 and an example demand as in Figure 4. In vector notation, we can now write:

$$b = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0.7 \\ 0 \\ 0 \\ -0.3 \\ -0.4 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The congestion approximator can now be written as a 20×16 matrix, where the i-th row corresponds to the i-th edge of the tree graph in one of its enumerations; we choose the enumeration as order of visit during a depth-first search that prioritizes from left to right. Then, the congestion approximator

would look like this:

$$R = \begin{pmatrix}
\frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 \\
0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & 0 & 0 & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}
\end{pmatrix}$$

Multiplying R with b gives us the congestion of the approximator tree edges. As an example of an approximation for the maximum congestion, we get $\|R \cdot b\|_\infty = \frac{0.7}{3}$ for those R and b. A graphical representation of R in the multiplication $R \cdot b$ can be seen in Figure 6.

4.2. Potential

As already defined in Def. of Potential according to [She13], our potential function is $\Phi(f) = \left\|C^{-1}(f)\right\|_\infty + 2\alpha \left\|R(b - B(f))\right\|_\infty$. This potential can be slightly modified to transform it into an optimization problem that is solvable with a gradient descent algorithm: the $\|\cdot\|_\infty$ norm can be replaced by the symmetric softmax function $lmax(\cdot)$, as defined in Definition 1. [She13] thus states a new differentiable potential function:

$$\phi(f) \coloneqq lmax\left(C^{-1}(f)\right) + lmax\left(2\alpha \cdot R(b - B(f))\right) \qquad \text{(Def. of } lmax\text{-Potential)}$$

Or in vector notation: $\phi(f) \coloneqq lmax\left(C^{-1}f\right) + lmax\left(2\alpha \cdot R \cdot (b - Bf)\right)$. For our unit-capacity grid graphs, the formula simplifies to $\phi(f) = lmax\left(f\right) + lmax\left(2\alpha \cdot R \cdot (b - Bf)\right)$.

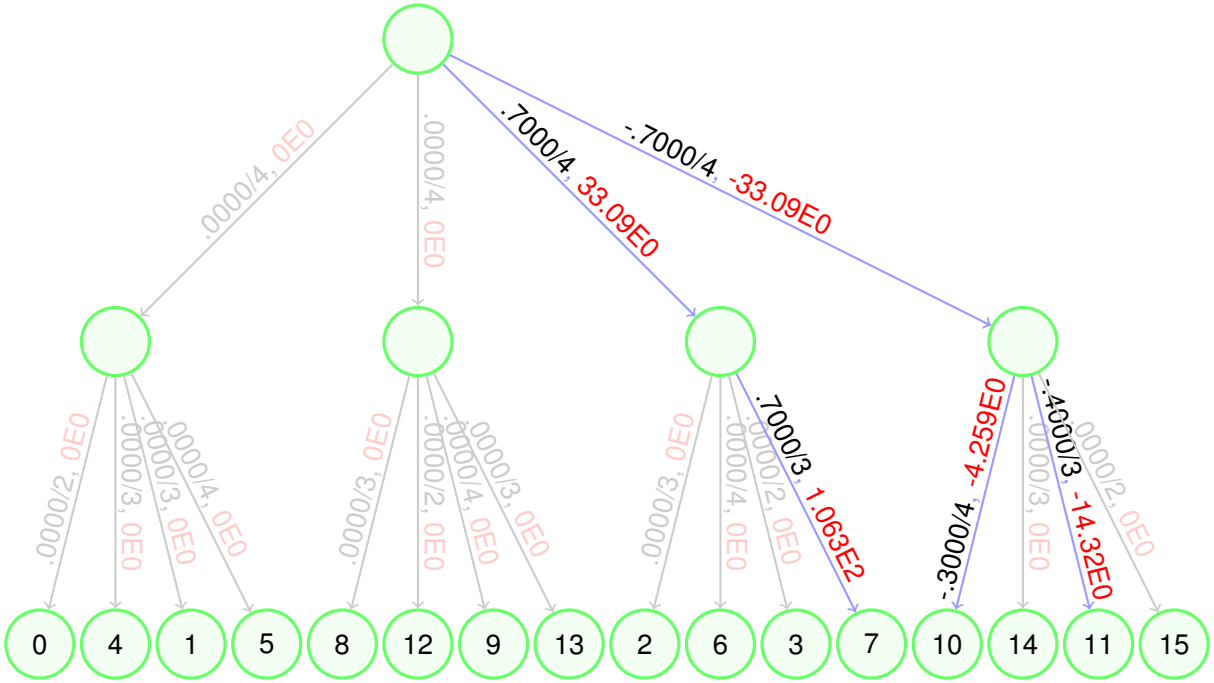


Figure 6 The congestion approximator R, together with edge labels representing $R \cdot b$ and the gradient $\nabla lmax(2\alpha \cdot R \cdot b)$ for the example demand vector b from figure 4. The scaling in $\nabla lmax$ by $1/\left(\sum_i e^{x_i} + e^{-x_i}\right)$ is postponed to the calculation of $-2\alpha \cdot B^T \cdot R^T \cdot \nabla lmax$ for more runtime efficiency, as the linear operations allow us to do so.

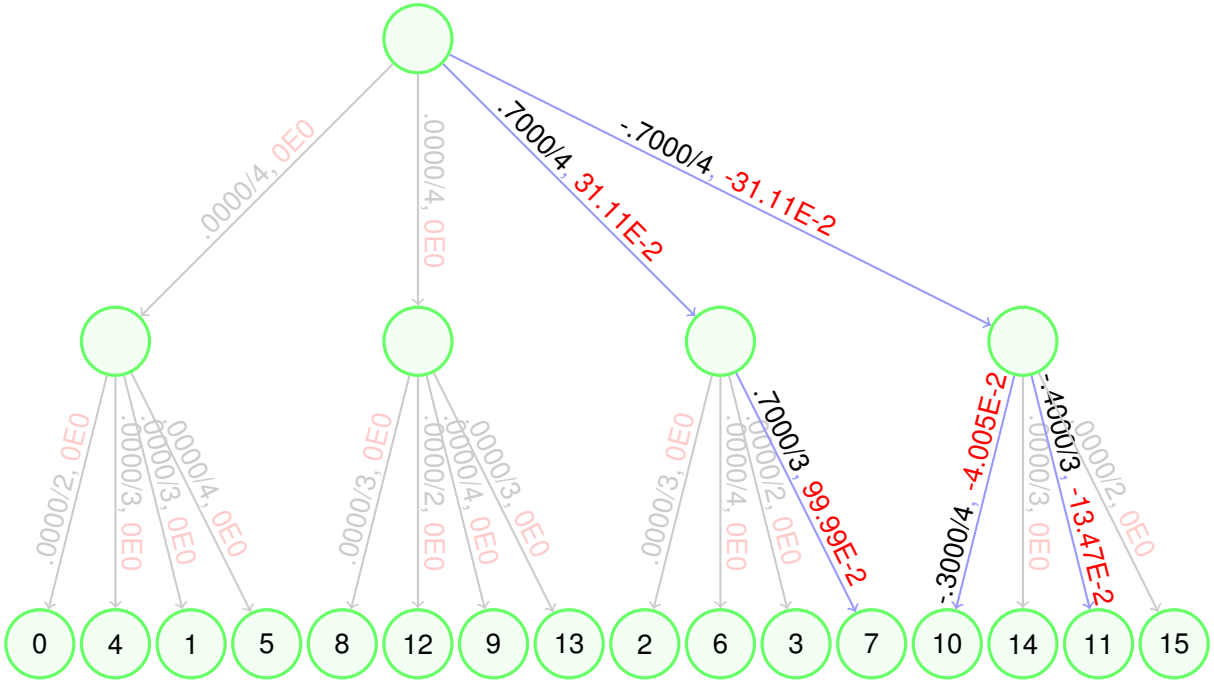


Figure 7 The congestion approximator R, together with edge labels representing $R \cdot b$ and the *shifted* gradient $\nabla lmax(2\alpha \cdot R \cdot b - s \cdot \mathbb{1})$ for the example demand vector b from figure 4, where s is the maximum of the absolutes of the entries in $2\alpha \cdot R \cdot b$ and $\mathbb{1}$ is a vector with 1 at each entry, shifting all entries uniformly. The shifting prevents overflows resulting from latter exponentiation. With $\alpha = 8$, the shift here is $2\alpha \cdot 0.7/3 = 3.7\overline{3}$ (from edge to the leaf 7; the bigger the difference of s and $-s$, the more approximate will this edge be to ± 1). Again, the scaling in $\nabla lmax$ is postponed.

4.3. Gradients

Let $f : \mathbb{R}^m \to \mathbb{R}$ be a totally differentiable function with variables $x_1, ..., x_m$. Then, the *gradient of* f, ∇f, is defined as

$$\nabla f \coloneqq \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_m} \end{bmatrix}.$$

$\frac{\partial f}{\partial x_i}$ is the partial derivative of f, which is defined as a limit, where this limit is in the same function space as f, if it exists, namely $\mathbb{R}^m \to \mathbb{R}$. This means that we can input a vector v in each partial derivative and get a mapping in $\mathbb{R}^m \to \mathbb{R}^m$:

$$\nabla f : \mathbb{R}^m \to \mathbb{R}^m : v \mapsto \begin{pmatrix} \frac{\partial f}{\partial x_1}(v) \\ \vdots \\ \frac{\partial f}{\partial x_m}(v) \end{pmatrix}.$$

If we implement the mapping, we can thus maintain the data structure of the input vector, as its function space is the same as the function space of the output vector, namely $\mathbb{R}^m$. Hence, no additional data structure is introduced for the gradients; they are just implemented as a program function and their evaluation at a point is then stored in a data object that is structurally equivalent to the input vector.

4.3.1. $lmax$-Potential

As in Equation 2, the gradient of $lmax(\cdot)$ is given as $(\nabla lmax(\vec{x}))_j = \frac{e^{x_j} - e^{-x_j}}{\sum_{i=1}^{n} e^{x_i} + e^{-x_i}}$. This, together with the chain rule $\nabla f(g(x)) = \nabla_g f(g) \cdot \nabla g(x)$ and the fact that $\nabla_x(Ax - b) = A^T$, gives us the following gradient for ϕ:[1]

$$\nabla\phi(f) = \left(C^{-1}\right)^T \cdot \nabla lmax\left(C^{-1}f\right) - 2\alpha \cdot B^T R^T \cdot \nabla lmax\left(2\alpha \cdot R \cdot (b - Bf)\right). \quad \text{(Equation 12)}$$

With C as unity matrix, we can instead use the gradient formula

$$\nabla\phi(f) = \nabla lmax\left(f\right) - 2\alpha \cdot B^T R^T \cdot \nabla lmax\left(2\alpha \cdot R \cdot (b - Bf)\right).$$

4.4. Maximal Spanning Tree

In a final phase of the routing algorithm described in [She13], a maximal spanning tree is used to route the residual demand. The choice is justified by both the efficient running time of $\mathcal{O}(n)$ and

[1] Since C is a diagonal matrix, transposing C^{-1} has no effect; it can be omitted, as in [She13].

the efficient congestion approximation, relative to its simplicity, with a possible approximator that satisfies $\alpha = m$. For a grid graph with multidimensional size vector $(n_1, ..., n_d)$, the edge count is:

$$m = \left(\sum_{i=1}^{d} \frac{n_i - 1}{n_i} \right) \cdot n \qquad \text{(Equation 13)}$$

where $n = \prod_{i=1}^{d} n_i$ is the node count.

4.4.1. Routing

A demand can be efficiently routed in a spanning tree: Each leaf l directly defines the flow of its edge $\{v, l\}$, and if v eventually becomes a leaf itself after consecutive removal of leaves (and their edge), the difference of current net flow at v and the demand for v define the residual demand for v, which then also defines the flow of its last edge that hasn't been removed yet. This strategy can be iterated until only one node is remaining. If the input demand was valid (i.e. its entries sum up to 0), the residual demand for this last node will be zero and the procedure can terminate. A short proof: Let $r(v)$ be the residual demand at vertex v. At the start, we have yet to route all of the input demand (whose entries we assume to sum up to 0). When removing a leaf l and its edge $\{v, l\}$, we send $r(l)$ flow along this edge, so the residual demand of l is then $r_{new}(l) = r(l) - r(l) = 0$ (l is satisfied) and the residual demand of v is then $r_{new}(v) = r(v) + r(l)$. Observe that $r_{new}(v) + r_{new}(l) = r(v) + r(l)$. This means the new residual demand still sums up to 0 and is thus valid. At termination, all vertices are thus satisfied and the residual demand is $\vec{0}$.

4.4.2. Construction

Since the edges of the grid graph have unit capacity, every spanning tree is also a maximal spanning tree. This gives us the freedom to design a spanning tree arbitrarily.
Consider a grid graph of dimension d. We choose a naive approach: for each vertex, choose all edges along one fixed dimension. After this, we have lines along this dimension that contain no cycle, so we choose one representative per line and choose a strategy for connecting them. If we always use the node with index 0 at the dimension in which the line is drawn, the representatives form the vertices of a grid graph with dimension $d - 1$, so we can repeat the strategy used for the first dimension. This gives us a spanning tree as in Figure 8.

Since we can always construct the same tree as maximal spanning tree, we don't have to store it explicitly. An implementation will be provided in chapter 6.

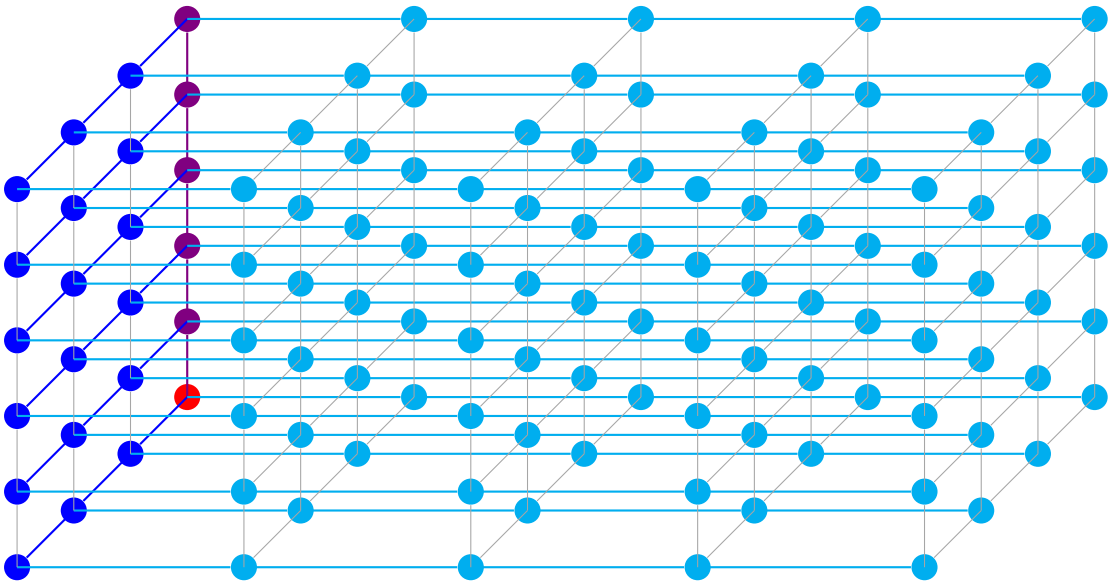


Figure 8 A spanning tree T for a 3-dimensional grid graph G with dimensions $(5, 4, 6)$. Each vertex can be seen as "representative" of itself. We can then choose the first dimension and connect vertices $(x, y, z), (x + 1, y, z)$ in T, as those connections are also edges of G. Then, we choose point $(0, y, z)$ on each line additionally as representative for its line and connect all $(0, y, z), (0, y + 1, z)$ for the second dimension. The third dimension then chooses the vertices $(0, 0, z)$ as representatives for those planes and connects vertices $(0, 0, z), (0, 0, z + 1)$. The point $(0, 0, 0)$ could be seen as representative for the whole cuboid. This strategy can be extended to arbitrary dimensions. Edges of G that are not part of T are greyed out.

5. Algorithm

In this thesis, we implement an algorithm for routing a demand b through a unit-capacity multidimensional grid graph G, using the algorithm provided in [She13] and adding some adjustments, with the approximator discussed in Section 4.1.4.

5.1. Almost Route

The algorithm `AlmostRoute`(b, ε) from [She13] is shown in Algorithm 1. The second return value here is $\nabla\phi(f)$ in opposition to [She13], but can be converted to a cut by taking the potentials induced from it and constructing a threshold cut. In this thesis, this cut is not needed, so we just use $\nabla\phi(f)$ as return value, which still facilitates possible future implementation of the potential threshold cut. The idea of `AlmostRoute` is to route only part of b into G, and later executing another call

Algorithm 1 `AlmostRoute` algorithm from [She13].

```
1: procedure ALMOSTROUTE(b, ε)
2:     repeat
3:         while φ(f) < 16ε^{-1} log(n) do
4:             f ← (17/16) · f
5:             b ← (17/16) · b
6:         end while
7:         δ ← ‖C∇φ(f)‖_1
8:         if δ ≥ ε/4 then
9:             f_e ← f_e − δ/(1+4α^2) sgn(∇φ(f)_e) c_e
10:        end if
11:    until δ < ε/4
12:    return f, ∇φ(f)
13: end procedure
```

on it with the residual demand. [She13] provides a lemma that for the induced potentials after termination of `AlmostRoute`, it holds that $\|C^{-1}f\|_\infty + 2\alpha\|R(b - Bf)\|_\infty \leq (1 + \varepsilon)\frac{b^T v}{\|CB^T v\|_1}$ with $v = R^T \nabla lmax(2\alpha R(b - Bf))$. For a proof, see Section 3.4.1.1.

[She13] also provides a proof for $\phi(f + h) \leq \phi(f) - \Omega(\varepsilon^2\alpha^{-2})$, which is shown in Section 3.4.1.2 in more detail. Since $\phi(f)$ was raised at most by $\varepsilon^{-1}\ln(n)$, there are at most $\mathcal{O}(\alpha^2\varepsilon^{-3}\ln(n))$ iterations in `AlmostRoute`.

Altogether, we get that `AlmostRoute`(b, ε) can be called with $b \in \mathbb{R}^n$ and $\varepsilon \in [0, 1/2]$ needs at most $\mathcal{O}(\alpha^2\varepsilon^{-3}\ln(n))$ iterations to return a flow f and a potential threshold cut S that satisfies $\Phi(f) \leq (1+\varepsilon)\cdot\frac{b_S}{c_S}$. [She13] mentions that the number of iterations can be reduced to $\mathcal{O}(\alpha\varepsilon^{-2}\ln(n))$ by using Nesterov's accelerated gradient method.

5.2. Complete Route

The algorithm from [She13] that routes the complete demand b with relative error ε is shown as pseudocode in Algorithm 2.

Algorithm 2 Algorithm for routing a complete demand from [She13].

procedure COMPLETEROUTE(b, ε)
    $r \leftarrow b$
    $(f_0, S_0) \leftarrow$ `AlmostRoute`(r, ε)
    for $i \leftarrow 1, \log_2(2m)$ **do**
        $r \leftarrow r - Bf_{i-1}$
        $(f_i, S_i) \leftarrow$ `AlmostRoute`$(r, 1/2)$
    end for
    $r \leftarrow r - Bf_i$ ▷ Assuming $i = log(2m)$ after the loop.
    $f_{i+1} \leftarrow$ `GridMST.route`(r)
    return $\sum_{j=0}^{i+1} f_j$
end procedure

The original `CompleteRoute` algorithm additionaly outputs the potential threshold cut S_0 from the first call of `AlmostRoute`.

The proof from [She13] showing that `CompleteRoute` terminates after $\mathcal{O}(\alpha\varepsilon^{-2}\ln^2(n))$ iterations and the returned f (and $S = S_0$) satisfy $Bf = b$ and $\|C^{-1}f\|_\infty \leq (1+\varepsilon)b_S/c_S$ is shown in Section 3.4.2. (Note that $\ln(n) = \frac{\log_2(n)}{\log_2(e)}$ and hence, the base of the logarithm is irrelevant for our asymptotic analysis.)

6. Implementation

Our implementation is written in *Java*. The code is available open-source at `https://github.com/js97/approximate-flows`, together with generated datasets. In this chapter, the most important algorithms will be provided in pseudocode, and some technical implementation details will be discussed.

6.1. Overview

The provided code features basic functionality classes and data structures, test classes and also some classes for usage and data generation.
The classes Grid Flow, Grid Demand and Grid Approximator Tree represent the main mathematical objects on which we perform operations. Most of their functions are implementations of mathematical operations.
The Grid Graph and Grid MST classes are also mathematical objects, but they have more structural and algorithmic importance.
The Grid Approximation class implements the main algorithmic functionality of the algorithms `AlmostRoute` and `ComopleteRoute` from [She13] and provide additional parametrization and adjustments.

6.1.1. Grid Graph

As discussed in Section 4.1.1, a multidimensional grid graph with unit-capacity edges is fully characterized by a dimensional vector $(n_1, ..., n_d)$, where d is the dimension. It is thus the only field of our Grid Graph class. A UML Chart for this class is provided in Figure 9.

Details for each function are provided in the documentation, which can be found in the repository. We will only discuss some important function implementations.

The function `getM()` calculates the number of edges of the grid graph according to Equation 13. The number of nodes is calculated via $n = \prod_{i=1}^{d} n_i$ in `getN()`. `different(int[] a, int[] b)` is a helper function that counts the number of dimensions along which `a` and `b` differ, which determines how many splits an approximator will need for a hypercube from `a` to `b`: if d_{diff} is the output, this means there have to be $2^{d_{\text{diff}}}$ splits - the return value of `countSplits`.
`capHyperBox(int[], int[])` calculates the capacity of the hypercube cut, given its corner coordinates, according to 4.2. Other algorithms that take a hypercube - given by its corners - as input, are `volume`, `indicesOfBoxNodes` and `split`. Further details of `split`, `toIndex` and `toVector` will be discussed in Sections 6.1.1.1 and 6.1.1.2.

GridGraph
nodesPerDim : `int[]`
GridGraph(`int...` nodesPerDim) capHyperBox(`int[]` lowerBounds, `int[]` higherBounds) : `int` countSplits(`int[]` lowerBounds, `int[]` higherBounds) : `int` ^{H}different(`int[]` a, `int[]` b) : `int` getDim() : `int` getM() : `int` getN() : `int` getNeighbours(`int[]` at) : `int[][]` inbound(`int[][]` bounds) : `boolean` indicesOfBoxNodes(`int[]` lowerBounds, `int[]` higherBounds) : `int[]` split(`int[]` lowerBounds, `int[]` higherBounds) : `int[][][]` ^{D}tikz2D() : `String` toIndex(`int[]` position) : `int` toPosition(`int` index) : `int[]` volume(`int[]` lowerIndices, `int[]` higherIndices) : `int`

Figure 9 UML Chart for the Grid Graph class. ^{H}: Helper function, ^{D}: Debug function. Constructors are underlined.

6.1.1.1 Vertex Enumeration

The vertices are enumerated, s.t. an efficient implementation with arrays for node properties (i.e., the demand) are possible. Our Grid Graph class provides functions `toIndex(int[]) : int` and `toVector(int) : int[]`[1] for converting between grid coordinates and the index from the enumeration scheme. For implementations, see Algorithms 3 and 4.

Algorithm 3 Conversion from grid coordinate vector to index. Defines the vertex enumeration.

```
1: procedure toIndex(v⃗)
2:     index ← 0
3:     factor ← 1
4:     for i ← d, 1 do
5:         index ← index + v⃗_i · factor
6:         factor ← factor · n⃗_i
7:     end for
8:     return index
9: end procedure
```

6.1.1.2 Hypercube Splitting

The `split` function returns a set of hypercubes that the given hypercube is split into by our approximator for grid graphs, represented by a three-dimensional integer array, where the first dimension is the index of the split, the second one the index of the corner (0: lower, 1: upper), and the third one is just the array for the coordinate vector of the corner. An implementation is given in Algorithm 5; note that it uses indices starting at 1 again, as it is kept in pseudocode with mathematical notations.

[1] In our implementation, the function `toVector` is called `toPosition`.

Algorithm 4 Conversion from index to grid coordinate vector. Inverse function to Algorithm 3.

procedure TOVECTOR(*index*)
  for $i \leftarrow d, 1$ **do**
    $\vec{v}_i \leftarrow$ *index* $- \left\lfloor \frac{\textit{index}}{\vec{n}_i} \right\rfloor \cdot \vec{n}_i$ ▷ Alternative: $\vec{v}_i \leftarrow$ *index* **mod** $\vec{n}_i$
    index $\leftarrow \frac{\textit{index} - \vec{v}_i}{\vec{n}_i}$
  end for
  return $\vec{v}$
end procedure

Algorithm 5 Splitting scheme of the approximator for hypercubes in grid graphs.

procedure SPLIT(*lowerBounds*, *higherBounds*)
  for $i \leftarrow 1, \texttt{countSplits}($*lowerBounds*, *higherBounds*$)$ **do**
    for $k \leftarrow 1, d$ **do**
      if *lowerBounds*$_k <$ *higherBounds*$_k$ **then**
        $m \leftarrow ($*lowerBounds*$_k +$ *higherBounds*$_k)/2$
        if $\texttt{isLowerSplit}(i, k,$ *lowerBounds*, *higherBounds*$)$ **then**
          ▷ The actual algorithm doesn't use a subroutine, but the bit-encoding of i and the index of k along the generating dimensions to determine the sub-interval for this split.
          $S_{i,1,k} \leftarrow$ *lowerBounds*$_k$
          $S_{i,2,k} \leftarrow m$
        else
          $S_{i,1,k} \leftarrow m + 1$
          $S_{i,2,k} \leftarrow$ *higherBounds*$_k$
        end if
      else
        $S_{i,1,k} \leftarrow$ *lowerBounds*$_k$
        $S_{i,2,k} \leftarrow$ *lowerBounds*$_k$
      end if
    end for
  end for
  return S
end procedure

GridFlow
entries : HashMap<Integer, Double> ^{H}g : GridGraph ^{H}m : int
GridFlow(GridGraph g) ^{H}GridFlow(GridFlow f, boolean clone) add(int from, int to, double value) add(GridFlow a, GridFlow b) : GridFlow calculateExcessFlows() : GridDemand calculateM() : int get(int from, int to) : double ^{H}getDefaultShift() : double getM() : int getN() : int gradientLmax() : GridFlow ^{H}gradientLmaxShifted(double shift = getDefaultShift()) : GridFlow hashEntry(Tuple<Integer,Integer> t) : int hashToTuple(int hash) : Tuple<Integer, Integer> l1() : double linf() : double lmax() : double ^{H}lmaxShifted(double shift = getDefaultShift()) : double ^{H}lmaxExp() : double ^{H}lmaxExpShifted(double shift = getDefaultShift()) : double scale(double d) : GridFlow ^{H}scaleInplace(double d) : GridFlow set(int from, int to, double value) ^{D}tikz2D() : String ^{D}toString() : String

Figure 10 UML Chart for the Grid Flow class. ^{H}: Helper function, ^{D}: Debug function. Constructors are underlined.

6.1.2. Grid Flow

An overview of the Grid Flow class is given with the UML Chart in Figure 10.

6.1.2.1 Data Storage

An enumeration scheme can also be used for the edges, enabling the use of an array for the values. But for now, our implementation uses a simpler identifier for the edges, that is the concatenation of the vertex identifiers. Currently, for an edge $(\vec{v_1}, \vec{v_2})$, the indices of v_1 and v_2 along the enumeration, $\texttt{toIndex}(\vec{v_1}) =: i_1$ and $\texttt{toIndex}(\vec{v_1}) =: i_1$, are first compared with $i_1 < i_2$. If false, the flow is handled as negative flow in the opposite direction of the input edge. Then, the last 16 bits of i_1 and i_2 are concatenated to a new 32-bit integer, which is our current identifier for the edge. When only observing the range in which those identifiers are, interpreting this range as key set would be too large to store. Hence, a HashMap is used to store the flow values for each edge.
The current scheme would fail for input graphs with more than $2^{16} = 65536$ nodes. There were only

lesser examples throughout this thesis, but for compatibility with bigger examples, future updates of the repository may change the edge identifier to 64 bit and maintain all 32 bit of each vertex, or even switch to an array, using an enumeration for the edges.

6.1.2.2 Mathematical Functions

Many functions of our Grid Flow class implement a mathematical function. The following table shows the correspondences:

Code function	Mathematical function
`f.add(from, to, value)`	$f_{(from,to)} \leftarrow f_{(from,to)} + value$
`add(fa, fb)`	$f_a +_{flow} f_b$
`f.calculateExcessFlows()`	$B(f)$ or $B \cdot f$
`get(from, to)`	$f_{(from,to)}$
`f.gradientLmax()`	$\nabla lmax(f) = \left(\frac{e^{f_j} - e^{-f_j}}{\sum_{i=1}^{m} e^{f_i} + e^{-f_i}}\right)_j$
`f.gradientLmaxShifted(shift)`	$\nabla lmax(f) = \left(\frac{e^{f_j + shift} - e^{-f_j + shift}}{\sum_{i=1}^{m} e^{f_i + shift} + e^{-f_i + shift}}\right)_j$
`f.l1()`	$\|f\|_1$
`f.linf()`	$\|f\|_\infty$
`f.lmax()`	$lmax(f) = \ln\left(\sum_i e^{f_i} + e^{-f_i}\right)$
`f.lmaxShifted(shift)`	$lmax(f) = \ln\left(\sum_i e^{f_i + shift} + e^{-f_i + shift}\right) - shift$
`f.lmaxExp()`	$\sum_i e^{f_i} + e^{-f_i}$
`f.lmaxExpShifted(shift)`	$\sum_i e^{f_i + shift} + e^{-f_i + shift}$
`f.scale(s)`	$s \cdot f$
`f.set(from, to, value)`	$f_{(from,to)} \leftarrow value$

The functions using the shift serve for preventing numerical overflows: when using 64-bit `double` with standard 11-bit biased exponent and 52-bit mantissa, `exp(709.782712893384)` still evaluates to `1.7976931348622732E308`, which is below the maximal value of a `double`, that is `1.7976931348623157E308`. In contrast, `exp(709.7827128933841)` would be above this limit and hence evaluates to `Infinity`. Since the algorithm from [She13] scales the flow up with factor $\frac{17}{16}$ per scaling, it is possible to exceed

GridDemand
entries : double[] ^{H}entryIndices : HashSet<Integer> ^{H}g : GridGraph
GridDemand(GridGraph g) ^{H}GridDemand(GridFlow f) add(int at, double value) ^{H}clone() : GridDemand get(int at) : double ^{H}iterator() : Iterator<Tuple<Integer, Double» l1() : double linf() : double lmax() : double ^{H}lmaxExp() : double scalarProd(GridDemand a, GridDemand b) : double scale(double s) : GridDemand ^{H}scaleInplace(double s) : GridDemand set(int at, double value) ^{H}subtract(GridDemand a, GridDemand b) : GridDemand toPotentialDiffEdgesFlow() : GridFlow ^{D}toString() : String ^{D}tikz2D() : String

Figure 11 UML Chart for the Grid Demand class. ^{H}: Helper function, ^{D}: Debug function. Constructors are underlined.

numerical limits here. The shifted variants of `lmax` and `gradientLmax` calculate the same value, but prevent overflows. `getDefaultShift()` returns the default shift, which is $-exp_+$, where exp_+ is the maximum exponent. Since the exponents are f_i and $-f_i$, it holds that $exp_+ = \|f\|_\infty$.

6.1.3. Grid Demand

The Grid Demand class is used for both the demanded excess flows at the vertices and the actual divergence of the flow. A UML Chart can be seen in Figure 11.

6.1.3.1 Data Storage

For vertices, we already elaborated an enumeration scheme in Section 6.1.1.1. This scheme serves as the correspondence of a vertex v to its position in the array, that is used in this class to store the demanded excess flow at v.

The field `entryIndices` is used for better runtime on sparse vectors. It stores the indices of non-zero entries. Since the actual excess flows are mostly non-sparse, this feature may be removed or moved to a new class in future repository updates.

6.1.3.2 Mathematical Functions

The implemented mathematical functions of the Grid Demand class are shown in the following table:

Code function	Mathematical function
`b.add(v, value)`	$b_v \leftarrow b_v + \textit{value}$
`b.get(v)`	b_v
`b.l1()`	$\|b\|_1$
`b.linf()`	$\|b\|_\infty$
`b.lmax()`	$lmax(b) = \ln\left(\sum_i e^{b_i} + e^{-b_i}\right)$
`b.lmaxExp()`	$\sum_i e^{b_i} + e^{-b_i}$
`scalarProd(ba, bb)`	$\vec{b_a}^T \cdot \vec{b_b}$
`b.scale(s)`	$s \cdot b$
`b.set(v, value)`	$b_v \leftarrow \textit{value}$
`subtract(ba, bb)`	$\vec{b_a} - \vec{b_b}$
`b.toPotentialDiffEdgesFlow()`	$\left(f_{ij} = (b_j - b_i) \cdot \delta_{N(i,j)}\right)_{ij}$ or $B^T \cdot b$

The `lmax` and `lmaxExp` functions are not used in any algorithm of this thesis (except the unit test), they can currently be ignored.
Since potential differences are skew-symmetric ($b_i - b_j = -(b_j - b_i)$), the result of $B^T \cdot b$ can be stored in a Grid Flow object. $\delta_{N(i,j)}$ is 1 if i and j are neighbours, and 0 else.

6.1.4. Grid Approximator Tree

The grid approximator is stored as tree structure in the Grid Approximator Tree class. A UML Chart can be seen in Figure 12. A UML Chart of the inner class, Node, can be seen in Figure 13.

`alpha` currently has the default value of 10. It is not trivial to find the best choice of α. More details will be discussed in Section 6.3.

Figure 12 UML Chart for the Grid Approximator Tree class. ^{H}: Helper function, ^{D}: Debug function, ^{S}: Delegates to inner class. Constructors are underlined.

6.1.4.1 Mathematical Functions

An overview of the mathematical functions implemented in the Grid Approximator Tree class:

Code function	Mathematical function
`computeLC(b)`	$\frac{\vec{b}^T \cdot v}{\|B^T \cdot v\|_1}$ with $v = R^T \cdot \nabla lmax_S\left(2\alpha \cdot R\vec{b_G}\right)$
`computeLCUpper(LC, eps)`	$(1+\epsilon) \cdot LC$
`linf()`	$\left\|R\vec{b_T}\right\|_\infty$
`lmax2Alpha()`	$lmax\left(2\alpha \cdot R\vec{b_T}\right) = \ln\left(\sum_i e^{\left(2\alpha \cdot R\vec{b_T}\right)_i} + e^{-\left(2\alpha \cdot R\vec{b_T}\right)_i}\right)$
`lmaxShifted2Alpha(shift)`	$lmax\left(2\alpha \cdot R\vec{b_T}\right) = \ln\left(\sum_i e^{\left(2\alpha \cdot R\vec{b_T}\right)_i + shift} + e^{-\left(2\alpha \cdot R\vec{b_T}\right)_i + shift}\right) - shift$
`lmaxExp2Alpha()`	$\sum_i e^{\left(2\alpha \cdot R\vec{b_T}\right)_i} + e^{-\left(2\alpha \cdot R\vec{b_T}\right)_i}$
`lmaxExpShifted2Alpha(shift)`	$\sum_i e^{\left(2\alpha \cdot R\vec{b_T}\right)_i + shift} + e^{-\left(2\alpha \cdot R\vec{b_T}\right)_i + shift}$
`multRtGradient()`	$R^T \cdot \nabla lmax_S\left(2\alpha \cdot R\vec{b_G}\right)$
`setGradient2Alpha()`	$\left(\nabla lmax_S\left(2\alpha \cdot R\vec{b_G}\right)_j\right)_j \leftarrow e^{2\alpha \cdot \left(R\vec{b_T}\right)_j} - e^{-2\alpha \cdot \left(R\vec{b_T}\right)_j}$
`setGradientShifted2Alpha(shift)`	$\left(\nabla lmax_S\left(2\alpha \cdot R\vec{b_G}\right)_j\right)_j \leftarrow e^{2\alpha \cdot \left(R\vec{b_T}\right)_j + shift} - e^{-2\alpha \cdot \left(R\vec{b_T}\right)_j + shift}$
`updateExcessFlows(b)`	$R\vec{b_T} \leftarrow R\vec{b}$

The functions `computeLC` and `computeLCUpper` serve for debug purposes; they are used to check the validity of Lemma 2.4 of [She13]. Since the evaluation of $R\vec{b}$ is lazy (i.e. only relevant information of $\vec{b}$ is stored at first with `updateExcessFlows(b)` and `setGradient`-functions without calculating $R\vec{b}$), the table above introduced $\vec{b_T}$ as the current demand that will be used when getting the value of $R\vec{b}$, and $\vec{b_G}$ for the current demand that will be used when getting the value of $\nabla lmax_S\left(2\alpha \cdot R\vec{b_G}\right)$. Since the denominator of the actual $\nabla lmax\left(2\alpha \cdot R\vec{b}\right)_j$ function is the same for each j, that is $\sum_i e^{\left(2\alpha \cdot R\vec{b}\right)_i} + e^{-\left(2\alpha \cdot R\vec{b}\right)_i}$ or $\sum_i e^{\left(2\alpha \cdot R\vec{b}\right)_i + shift} + e^{-\left(2\alpha \cdot R\vec{b}\right)_i + shift}$ for the shifted variant, we use a scaled version of $\nabla lmax$ that only consists of the numerator, namely $\nabla lmax_S\left(2\alpha \cdot R\vec{b}\right)_j = e^{2\alpha \cdot \left(R\vec{b}\right)_j} - e^{-2\alpha \cdot \left(R\vec{b}\right)_j}$, or for the shifted variant: $\nabla lmax_S\left(2\alpha \cdot R\vec{b}\right)_j = e^{2\alpha \cdot \left(R\vec{b}\right)_j + shift} - e^{-2\alpha \cdot \left(R\vec{b}\right)_j + shift}$.
The metric for which we need the gradient is $\nabla\phi(f) = \nabla\phi_G(f) + \phi_T(f)$ with $\phi_G = \nabla lmax(f)$ as graph potential gradient and $\phi_T = -2\alpha \cdot B^T \cdot R^T \cdot \nabla lmax\left(2\alpha \cdot R(b - Bf)\right)$ as tree potential gradient. Since both scalar and matrix multiplications are linear operations, rescaling can be postponed to after all linear operations, i.e. the multiplications throughout $2\alpha \cdot B^T \cdot R^T$. The actual $\nabla\phi_T(f)$ can be obtained by first routing the residual demand $b_R = (b - Bf)$ in the tree (`br = GridDemand.subtract(b, f.calculateExcessFlows())` via `t.updateExcessFlows(br)`, then updating the gradients via `t.setGradient2Alpha()` or its variant with shift, and finally evaluating

`t.MultRtGradient().toPotentialDiffEdgesFlow().scale(-2*t.getAlpha()/t.lmaxExp2Alpha())` or with a scalar of `-2*t.getAlpha()/t.lmaxExpShifted2Alpha(shift)` according to the shift during the gradient update. The default shift again shifts the maximal exponent to 0 with *shift* $= -2\alpha \cdot \|R\vec{b}\|_\infty$, as implemented in `getDefaultShift()`.
Before calling `computeLC()`, the residual demand is routed into R. This means that $\vec{b_G} = b - Bf$, as needed. The use of $\nabla lmax_S$ instead of $\nabla lmax$ is irrelevant, as the scaling factor will be in both the numerator and denominator and hence cancel out.
Finally, it should be mentioned, that - in most cases - the residual demand $b - Bf$ is routed in R for getting $lmax(2\alpha \cdot (b - Bf))$ or $\nabla lmax(2\alpha R(b - Bf))$, so the implementation currently just traverses the tree and uses $\vec{b_G}$ (which mostly is $b - Bf$) when setting the gradient, which results in $\vec{b_T} = \vec{b_G}$ at the time the gradient is set.

6.1.4.2 Tree Node

Each Tree Node represents a cut in the original graph. The estimates from Rb are measuring $\frac{b_S}{c_S}$ for the represented cut. The only unused node is the root node, which will be omitted, as it always will have an excess flow of 0 as it represents the cut $(G, \emptyset)$. Since c_S will not be changed, it is calculated via `g.capHyperBox` during the Node constructor call. The leaves represent cuts of the single vertices, so with a leaf representing the cut of vertex v, its value b_S can be set to b_v. Nodes above can follow the recursive formula $b_S = \sum_s b_s$, where s are the cuts represented by their children. Our current implementation uses post-order traversal to update b to achieve running time linear in the number of Tree Nodes, where the return value of `updateExcessFlows` in the Node class is the current excess flow at the Node it is called on after updating it.
The splitting scheme is also applied during Node construction, starting with the root node. `lowerIndices` and `higherIndices` store the corners of the hypercube whose cut is represented by this Node.

6.1.5. Grid Approximation

The Grid Approximation maintains the high-level algorithmic functionalities of our code, i.e. `AlmostRoute` and `CompleteRoute`. An overview is shown in the UML Chart in Figure 14. The fields `cpg` and `cpt` store the values of the graph and tree potentials $\phi_G(f)$ and $\phi_T(f)$ that were calculated in the last call of `potential(f, b)`.
`currentScale` keeps track of the total scale factor that is applied to f and b.
`dynamicOptStepsize` and `-Precision` are used to enable the step size optimization described in Section 6.2 and set the precision of the minimum approximation's termination condition. `goldenSectionSearch` chooses whether the standard ternary search or golden section search are used for the minimum approximation.
The `stepSize` field is a flexible possibility to change the step size. It is of type `DoubleSequence`, a simple interface, that maps an `int` to a `double`. In iteration i, the step size is set to the `double` it is mapped to, relative to the standard value. The parametrized versions of `AlmostRoute` and `CompleteRoute` take this step size sequence as an argument. The value of α can also be given as additional parameter there.
`optimizeStepsize` implements the selection of the initial interval for minimum search, as described

GridApproximatorTree.Node
T : GridApproximatorTree capacityCut : `int` children : Node[] currentExcessFlow : `double` currentGradient : `double` higherIndices : `int[]` lowerIndices : `int[]` m : `int` parent : Node[]
Node(GridGraph g, `int[]` lowerIndices, `int[]` higherIndices, GridApproximatorTree T) height() : `int` isLeaf() : `boolean` isRoot() : `boolean` ^{D}leadingWhitespaceString(`int` whitespaces) : `String` ^{S}linf() : `double` ^{SH}lmaxExp2Alpha() : `double` ^{SH}lmaxExpShifted2Alpha(`double` shift) : `double` ^{S}multRtGradient(GridDemand d) ^{S}setGradient2Alpha() ^{SH}setGradientShifted2Alpha(`double` shift) ^{S}updateExcessFlows(GridDemand b) : `double` ^{SD}tikz2D(`double` xpos) : `String` ^{SD}toString() : `String`

Figure 13 UML Chart for the inner class Node of the Grid Approximator Tree. ^{H}: Helper function, ^{D}: Debug function, ^{S}: Implements outer class functionality. Constructors are underlined.

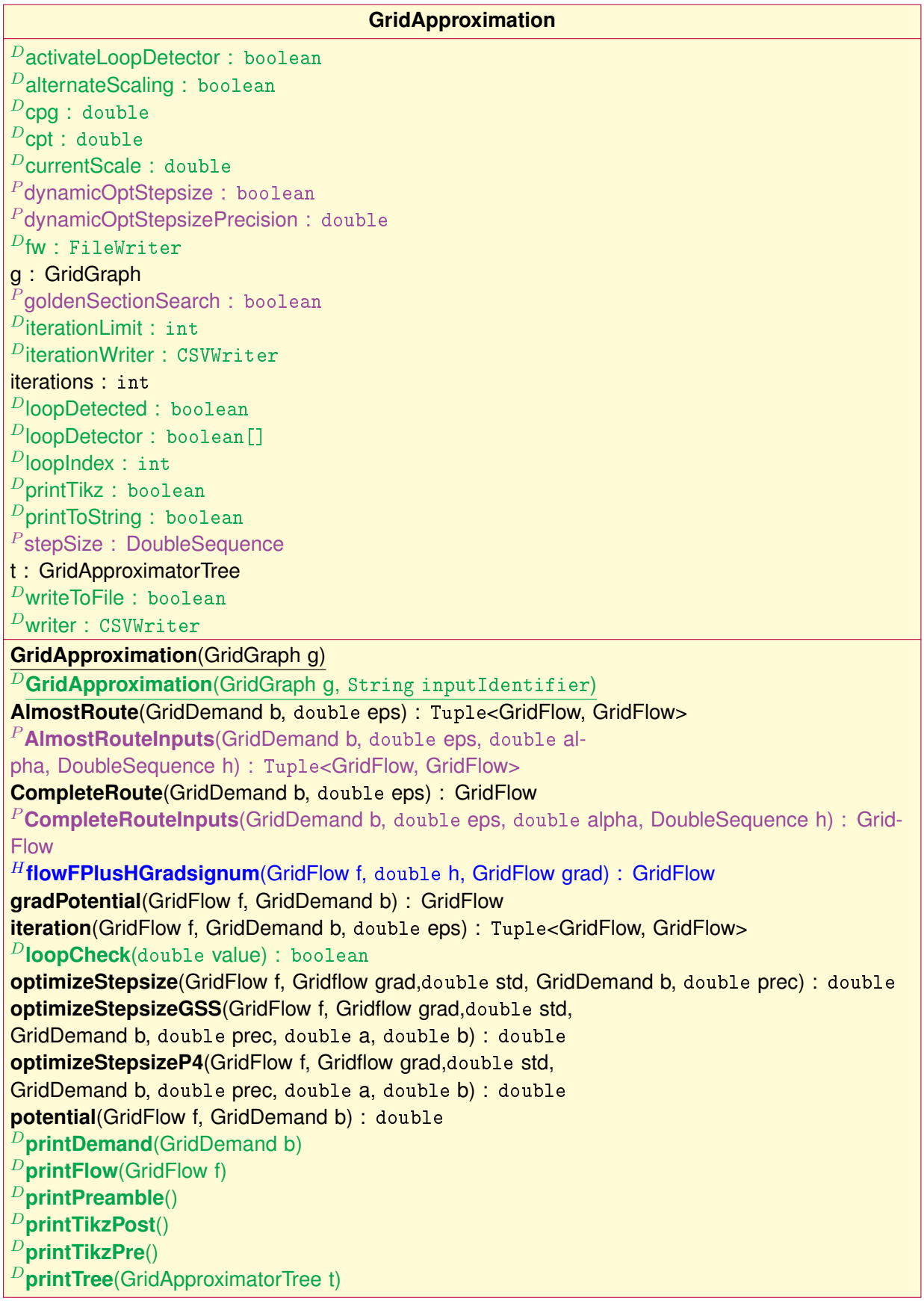


Figure 14 UML Chart for the Grid Approximation class. ^{H}: Helper function, ^{D}: Debug field/function, ^{P}: Parametrization field/parametrized version of another function. Constructors are underlined. For better readability, function names are in **bold font**.

GridMST
g : GridGraph
GridMST(GridGraph g) ^{H}iterNext(int[] current) : int[] route(GridDemand b) : GridFlow route2(GridDemand b) : GridFlow

Figure 15 UML Chart for the Grid Maximal Spanning Tree class. ^{H}: Helper function. Constructors are underlined.

in Section 6.2, and delegates the minimum search on the interval to `optimizeStepSizeP4` or `optimizeStepsizeGSS`, according to the `goldenSectionSearch` flag.

6.1.6. Maximal Spanning Tree

The Maximal Spanning Tree class, `GridMST`, implements the routing of a demand through a maximal spanning tree of G. It is used in the last stage of `CompleteRoute` to route the residual demand in G. The UML Chart is shown in Figure 15. The routing discussed in Section 4.4.2 is implemented based on vertex coordinate vectors in the `route` function. For the pseudocode, see Algorithm 6.

Taking a closer look at the iteration gives the opportunity of an optimization of this algorithm: let v^i_{start} and v^i_{end} denote v_{start} and v_{end} at iteration i, and observe that our enumeration (see Section 6.1.1.1) satisfies $\texttt{index}(v^1_{start}) \geq \texttt{index}(v^1_{end}) \geq \texttt{index}(v^2_{start}) \geq \texttt{index}(v^2_{end}) \geq \ldots \geq \texttt{index}(v^d_{start}) \geq \texttt{index}(v^d_{end})$, except $\texttt{index}(v^i_{start}) < \texttt{index}(v^i_{end})$ iff $n_i = 1$, where also the leaf elimination loop will be empty. Also observe that $\texttt{index}(v_{start}) \geq \texttt{index}(v) \geq \texttt{index}(v_{end})$ for all non-empty loops. Finally, observe that for each vertex pair v_1 and v_2 from vertices v of the leaf elimination loop, where $(v_1)_j = (v_2)_j$ for $j \neq i$ and $(v_1)_i > (v_2)_i$, $\texttt{index}(v_1) > \texttt{index}(v_2)$.
We can conclude that iterating vertices in reverse order of the `index`-function, starting at $n-1$ and ending at 1, is an adequate approach that simplifies Algorithm 6: the third observation shows validity of the inner iteration, and the first and second observation show that the start and end vertices as well as the iterated vertex set of the inner loop are the same. We can also conclude the information of the dimension i in which we have to pick u, based on v alone: i is exactly the first dimension j for which $v_j \neq 0$. This gives us Algorithm 7.

6.2. Optimizing Step Size

The algorithm from [She13] updates the flow according to $f_i \leftarrow f_i - s_i$, where $s_i = \frac{\|C\nabla\phi(f)\|_1}{1+4\alpha^2} sgn\left(\nabla\phi(f)_i\right) c_i$. We now try to optimize the "step size" by scaling the step s_i with h for all i. Our goal is to minimize the function $g_f(h) := \phi\left(f - h \cdot s\right)$.

Algorithm 6 Routing Demand b through a (Maximal) Spanning Tree of Grid Graph G

procedure ROUTEMST(GridGraph G, GridDemand b)
  $f \leftarrow 0$
  $r \leftarrow b$ ▷ Residual demand
  $(n_1, ..., n_d) \leftarrow G.nodesPerDimension$
  for $i \leftarrow 1, d$ **do**
    for $j \leftarrow 1, i-1$ **do** ▷ Set v_{start} and v_{end} properly
      $(v_{start})_j \leftarrow 0$
      $(v_{end})_j \leftarrow 0$
    end for
    $(v_{start})_i \leftarrow n_i - 1$
    $(v_{end})_i \leftarrow 1$
    for $j \leftarrow i+1, d$ **do**
      $(v_{start})_j \leftarrow n_j - 1$
      $(v_{end})_j \leftarrow 0$
    end for
    for $v \leftarrow v_{start}, v_{end}$ **do** ▷ Consecutively eliminate leaves along this dimension. Iteration is valid iff for each vertex pair v_1 and v_2 with $(v_1)_j = (v_2)_j$ for $j \neq i$ and $(v_1)_i > (v_2)_i$, v_1 will be eliminated before v_2.
      $u \leftarrow v$
      $u_i \leftarrow v_i - 1$
      $e \leftarrow (u, v)$
      $f_e \leftarrow r_v$ ▷ r_v: Residual demand of leaf v. Every iterated v is a leaf iff the iteration is valid.
      $r_u \leftarrow r_u + r_v$
      $r_v \leftarrow 0$
    end for
  end for
  return f
end procedure

Algorithm 7 Simplified Version of Algorithm 6

procedure ROUTEMST2(GridGraph G, GridDemand b)
  $f \leftarrow 0$
  $r \leftarrow b$ ▷ Residual demand
  $i \leftarrow 1$
  for $k \leftarrow n-1, 1$ **do**
    $\vec{v} \leftarrow \texttt{toVector}(k)$
    while $\vec{v}_i = 0$ **do**
      $i \leftarrow i + 1$
    end while
    $\vec{v}_i \leftarrow \vec{v}_i - 1$
    $u \leftarrow \texttt{toIndex}(\vec{v})$
    $e \leftarrow (u, k)$
    $f_e \leftarrow r_k$
    $r_u \leftarrow r_u + r_k$
    $r_k \leftarrow 0$
  end for
  return f
end procedure

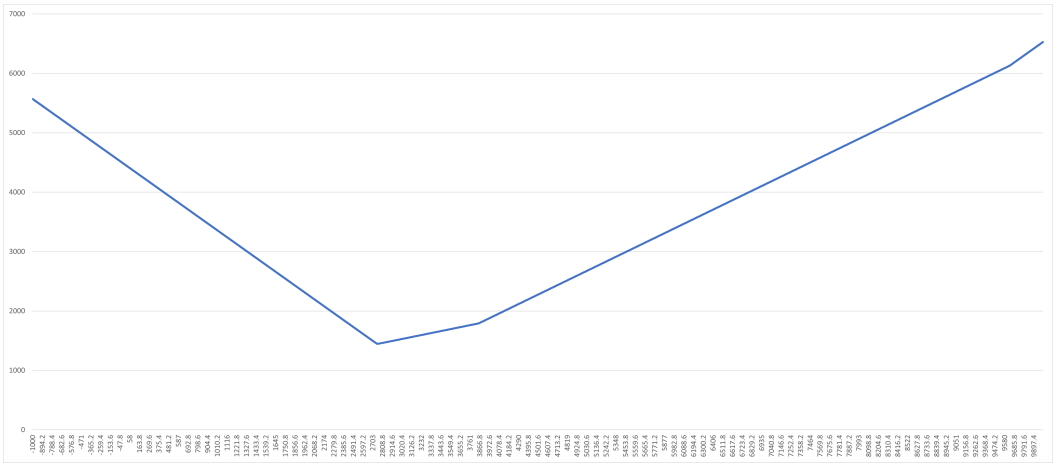
For this, we need to find a (local) minimum of g_f, so we aim to find h with $\frac{\partial}{\partial h} g_f(h) = 0$.

$$
\begin{aligned}
&\frac{\partial}{\partial h} g_f(h) \\
&= \frac{\partial}{\partial h} \phi(f - h \cdot s) \\
&= \frac{\partial}{\partial h} \left(lmax\left(C^{-1} \cdot (f - h \cdot s)\right) + lmax\left(2\alpha R(b - B \cdot (f - h \cdot s))\right)\right) \\
&= \left(\frac{\partial}{\partial h} lmax\left(C^{-1} \cdot (f - h \cdot s)\right)\right) + \left(\frac{\partial}{\partial h} lmax\left(2\alpha R(b - B \cdot (f - h \cdot s))\right)\right) \\
&= \left(-C^{-1}s\right)^T \cdot \nabla lmax\left(C^{-1} \cdot (f - h \cdot s)\right) + (2\alpha RBs)^T \cdot \nabla lmax\left(2\alpha R(b - B \cdot (f - h \cdot s))\right) \\
&=: g'_f(h)
\end{aligned}
$$

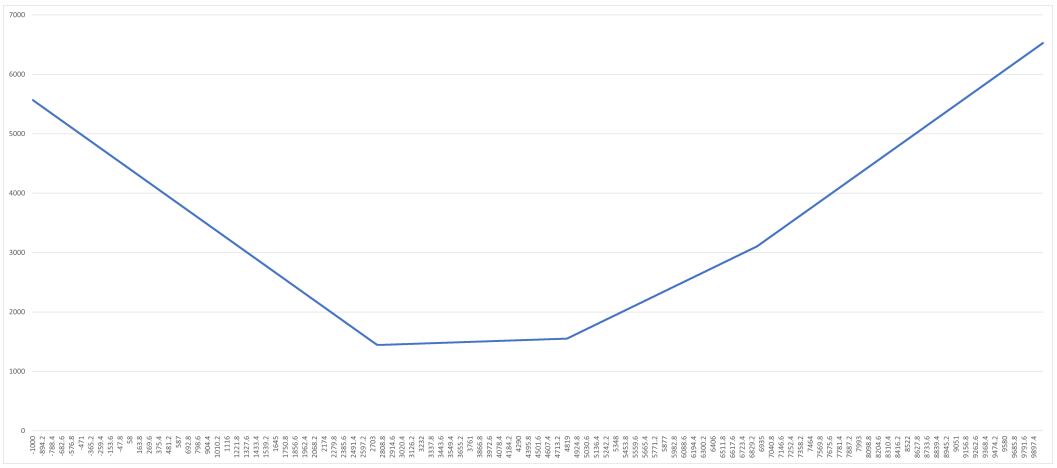
Looking at $\nabla lmax$ reveals that we can't find a direct algebraic formula for a solution h^* that satisfies $g'_f(h^*) = 0$. There are two convenient approaches at this point:

1. Use a root approximator (e.g. Newton's Method)
2. Use a (local) minimum approximator (e.g. Line Search)

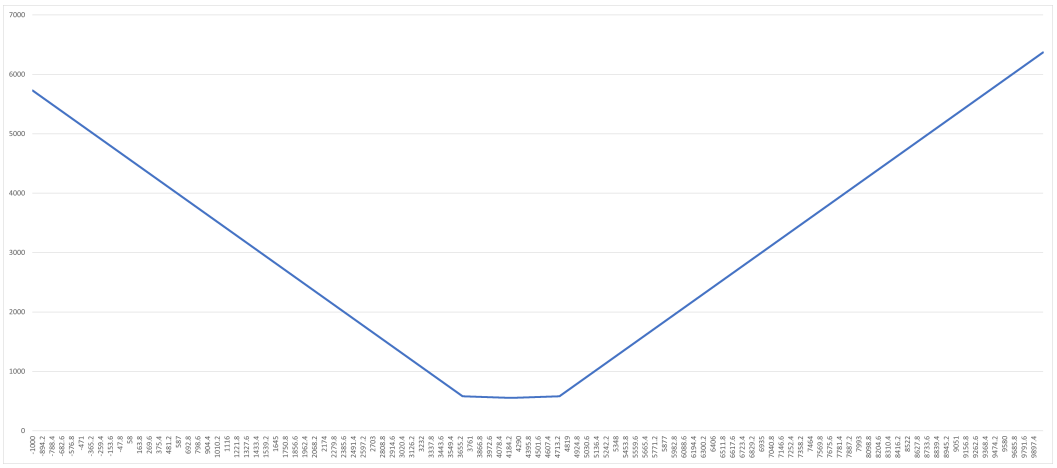
For the first approach, Newton's Method offers an algorithm with quadratic order of convergence for finding $\frac{\partial}{\partial h} g_f(h) = 0$. Both g'_f and g''_f need to be implemented in this method.
The second approach only needs an implementation of $g_f(h)$. As ϕ is already implemented, we are nearly done.
Comparing different approaches at this stage of the algorithm can be done in future work, but might not have much impact on the overall performance of this algorithm, so we will stick to the second approach for now.
Sampled plots of $g_f(h)$ can be seen in Figures 16, 17 and 18.



(a) $g_f(h)$ for Example 1.



(b) $g_f(h)$ for Example 2.



(c) $g_f(h)$ for Example 3.

Figure 16 Examples for $g_f(h)$.

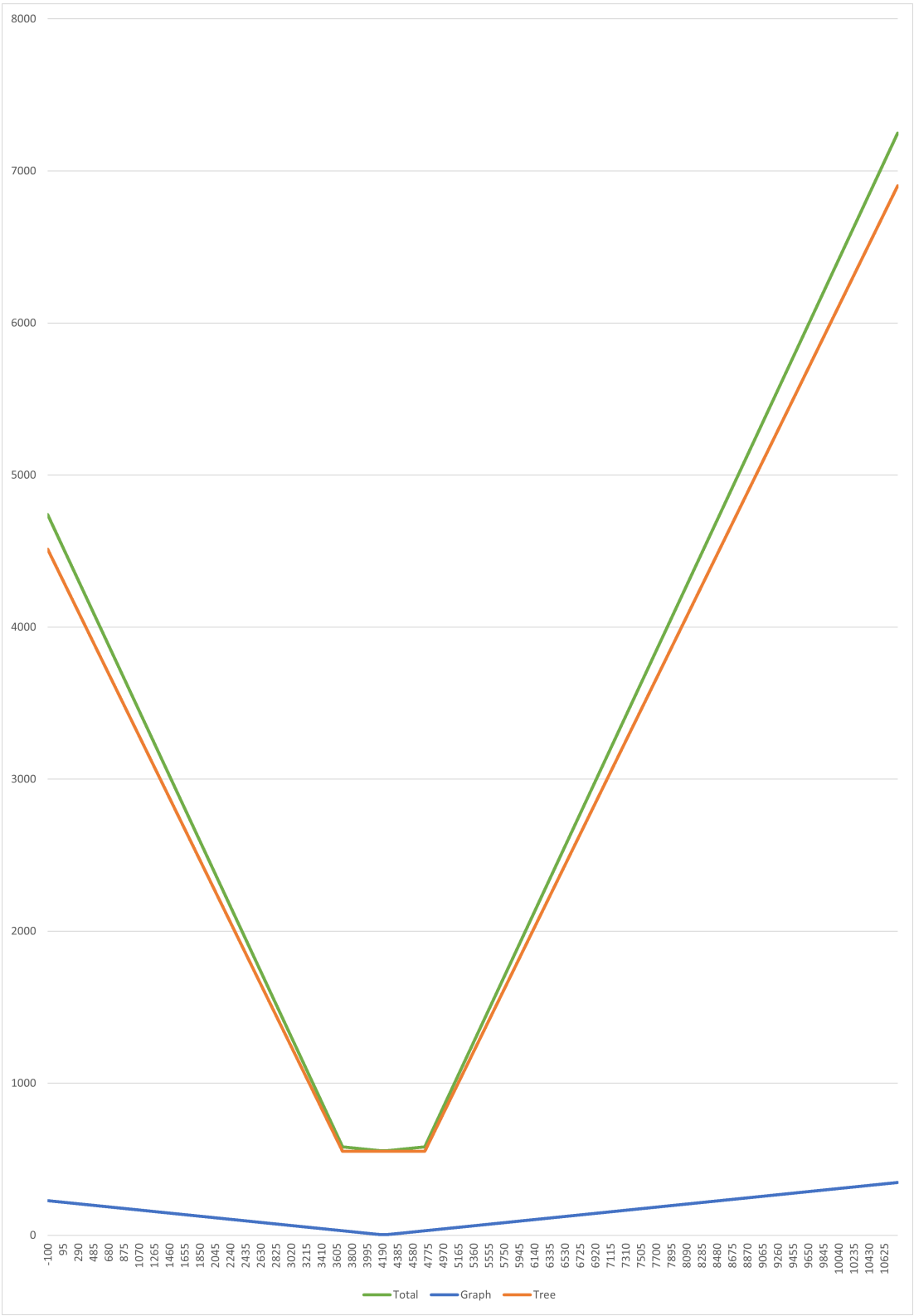
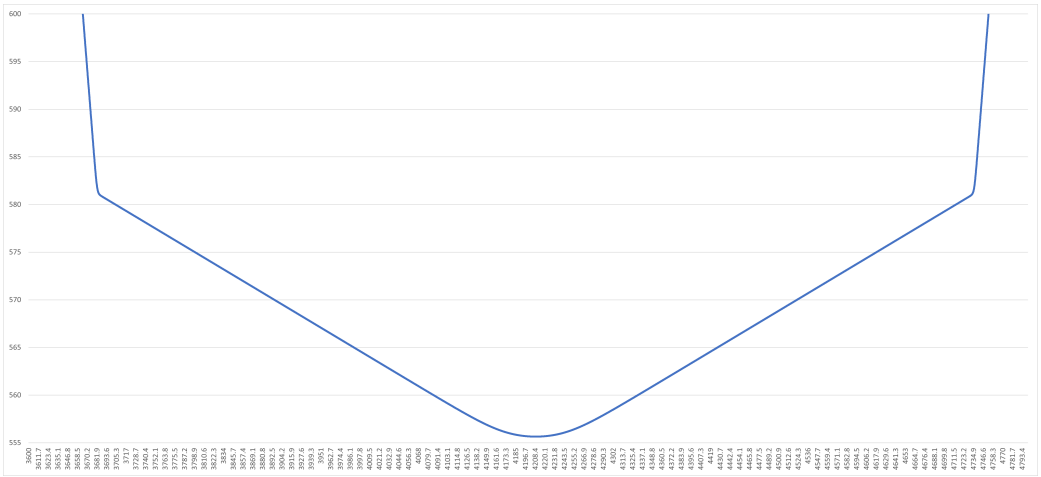
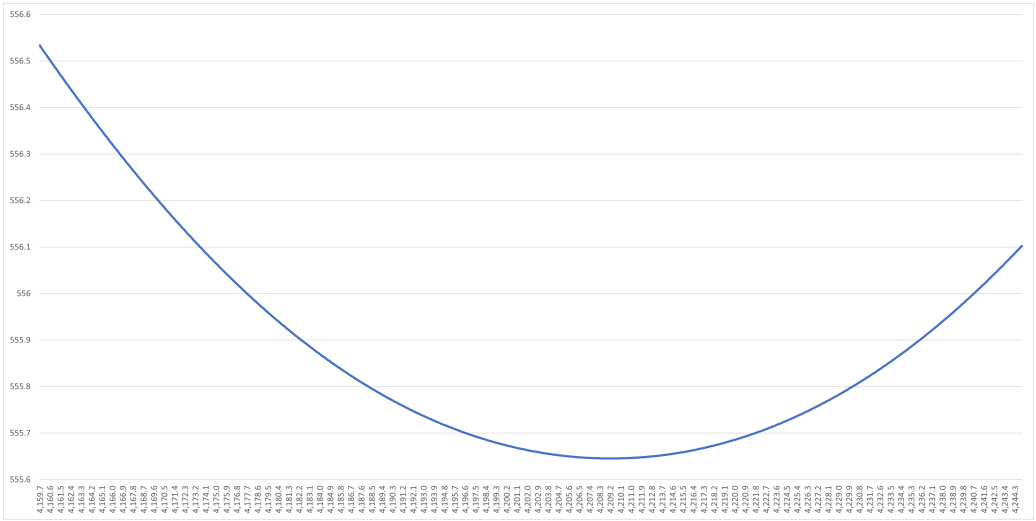


Figure 17 $g_f(h)$ for Example 3, split into $\phi = \phi_G + \phi_T$.



(a) $g_f(h)$ for Example 3, refined.



(b) $g_f(h)$ for Example 3, further refined.

Figure 18 Refinements of $g_f(h)$ for Example 3, showing more detailed regions around the minimum.

6.2.1. Approach

With h^* denoting the (local) minimum, we assume $g_f(h)$ to be strictly monotonically decreasing for $h < h^*$ and strictly monotonically increasing for $h > h^*$, and thus, h^* being the only local minimum of $g_f(h)$ and hence also the global minimum. With this assumption of strict monotonicity, we can search for the minimum with following scheme:

1. Search for a_0 and b_0 s.t. $a_0 \leq h^* \leq b_0$
2. Start with $i = 0$ and repeat:
   a. Refine the interval from $[a_i, b_i]$ to $[a_{i+1}, b_{i+1}]$ s.t. $a_{i+1} \geq a_i$, $b_{i+1} \leq b_i$, $b_{i+1} - a_{i+1} < b_i - a_i$ and $a_{i+1} \leq h^* \leq b_{i+1}$
   b. Increment i
   c. Terminate at some condition and output $\arg\min_{\{a_i, b_i\}} g_f$

The first step can be implemented by choosing an initial parameter h_0 and then, searching at higher parameters $h_1, h_2, ...$ until a parameter h_k is reached where $g_f(h_k) \geq g_f(h_{k-1})$. This would give us the information that h^* lies in $[h_{k-2}, h_k]$, if $k \geq 2$. If $k < 2$, h^* might be smaller than h_0. In this case, one could iterate a sequence with subsequently smaller parameters $h_{-1}, h_{-2}, ...$ until a parameter $h_{-\overline{k}}$ is reached where $g_f(h_{-\overline{k}}) \geq g_f(h_{-(\overline{k}-1)})$. Then, we could deduce that h^* lies in $[h_{-\overline{k}}, h_{-\overline{k}+2}]$. If $\overline{k} < 2$, h^* might be bigger than h^0. It can only be possible (based on our assumption of strict monotonicity) that h^* is in $[h_{-1}, h_1]$ if both directions instantly stop at $k = 1 = \overline{k}$.
A possible choice for those sequences is to define $h_i := \gamma \cdot h_{i-1}$ and $h_{-i} := \frac{1}{\gamma} \cdot h_{-(i-1)}$ for all $i \geq 1$ with γ as growth factor.
This is also the sequence we use, where we set $\gamma = 2$ and $h_0 = 1$. We will also postpone the search at $(0, h_{-2})$ and start with h_{-2}, h_{-1}, h_0, testing $g_f(h_0) < g_f(h_{-1})$, as our first idea is to accelerate the gradient-like descent and observe the impact - if the default parameter could be optimized by more than the precision of the minimum search with a decrease, we would still be able to decrease it downto $\frac{1}{4}$. This gives us algorithm 8 (note that algorithm 8 uses the indices i in the expression h_i dynamically). The results were generated using Algorithm 8, whereas inspecting results from the enhanced version, Algorithm 9, are postponed for temporal reasons.

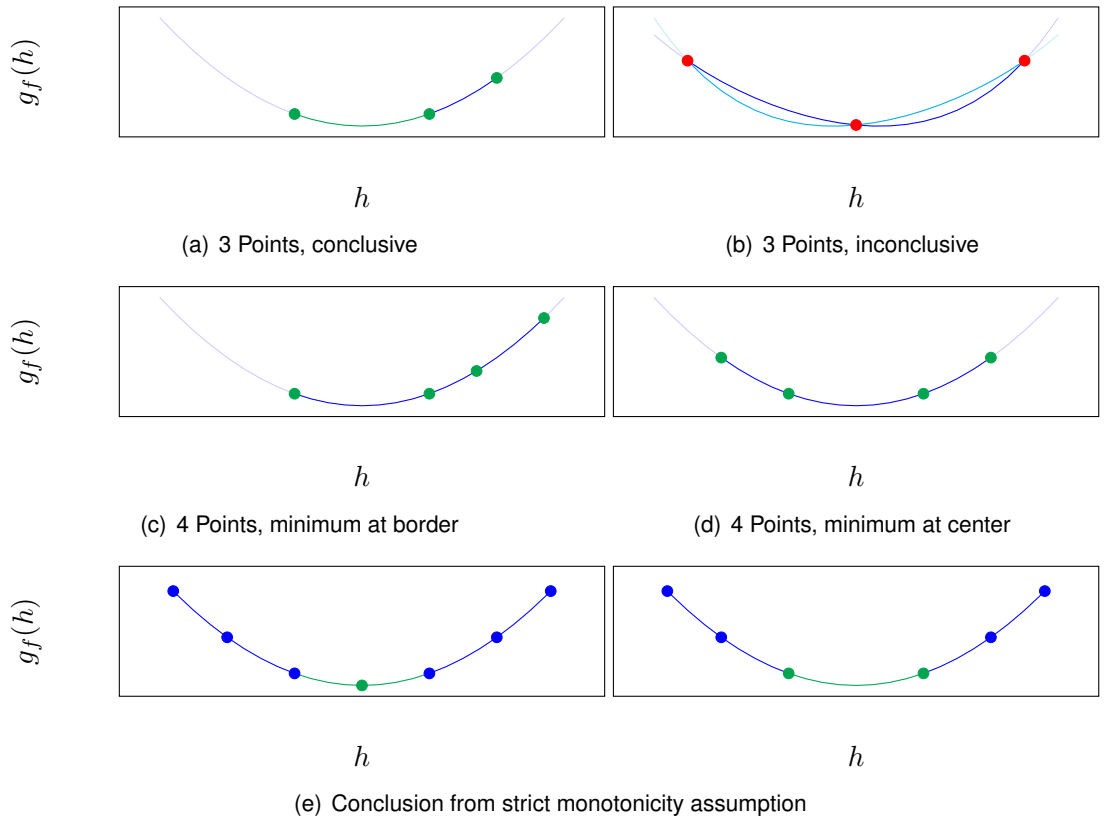
The interval refinement can be done via Ternary Search: Select two points l_i, r_i inside $[a_i, b_i]$ ("Ternary Search" uses $l_i = a_i + \frac{1}{3} \cdot (b_i - a_i)$ and $r_i = a_i + \frac{2}{3} \cdot (b_i - a_i)$) and compare $g_f(l_i)$ with $g_f(r_i)$. As can be seen in Figure 19(e), we now can deduce from the four points we have that h^* is either in one or two out of our three intervals. In all cases, we can be sure that at least one interval does not contain the minimum, based on our assumptions, namely $[a_i, l_i)$, if $g_f(l_i) \geq g_f(r_i)$, or $(r_i, b_i]$, if $g_f(l_i) \leq g_f(r_i)$. This interval is discarded.
Since the function evaluation is *very* expensive relatively to the rest of the iteration, we use the Golden-Section Search, which works just like the Ternary Search, but has less function evaluations and a better interval shrinking factor.
Before elaborating further, we will shortly discuss the termination condition. Usually, one is inter-

Algorithm 8 Calculation of Initial Search Interval for Line Search

```
1: procedure INITIALIZESEARCHINTERVAL(f)
2:     γ ← 2
3:     h_2 ← 1
4:     h_1 ← h_2/γ
5:     h_0 ← h_1/γ
6:     (g_0, g_1, g_2) ← (g_f(h_0), g_f(h_1), g_f(h_2))
7:     while g_1 > g_2 do
8:         h_0 ← h_1
9:         h_1 ← h_2
10:        h_2 ← γ · h_2
11:        g_0 ← g_1
12:        g_1 ← g_2
13:        g_2 ← g_f(h_2)
14:    end while
15:    return (h_0, h_2)
16: end procedure
```

Algorithm 9 Enhanced Version of Algorithm 8

```
1: procedure INITIALIZESEARCHINTERVAL(f, γ, h_0)
2:     h_1 ← γ · h_0
3:     (g_0, g_1) ← (g_f(h_0), g_f(h_1))
4:     i ← 1
5:     while g_{i−1} > g_i do
6:         i ← i + 1
7:         h_i ← γ · h_{i−1}
8:         g_i ← g_f(h_i)
9:     end while
10:    if i = 1 then
11:        h_{−1} ← h_0/γ
12:        g_{−1} ← g_f(h_{−1})
13:        j ← −1
14:        while g_j < g_{j+1} do
15:            j ← j − 1
16:            h_j ← h_{j+1}/γ
17:            g_j ← g_f(h_j)
18:        end while
19:        return (h_j, h_{j+2})
20:    else
21:        return (h_{i−2}, h_i)
22:    end if
23: end procedure
```



(a) 3 Points, conclusive

(b) 3 Points, inconclusive

(c) 4 Points, minimum at border

(d) 4 Points, minimum at center

(e) Conclusion from strict monotonicity assumption

Figure 19 Visualization of the cases during line search. The function is assumed to have a local minimum between the two border points, for which the function strictly monotonically decreases before and strictly monotonically increases after this minimum (strict monotonicity assumption).

ested in having the function *parameter*, h in our case, as precisely as possible, so a fitting termination condition would be that $|a_i - b_i|$ is below some threshold. This is not the case for us, as we are only interested in a function *value* that is as small as possible, so a termination condition concerning the precision of the function value would be more fitting. This is a bit more tricky, as we don't have a precision estimate, since, for the parameter with $h^* \in [a_i, b_i]$, we can only use $g_f(h^*) \leq \min\{g_f(a_i), g_f(b_i)\}$. With the current assumptions, it would be possible for h^* to be arbitrarily small. But it is *very likely* that $g_f(h)$ is also convex, so linear extrapolation at $[a_i \leftrightarrow l_i \rightarrow]$ for $(l_i, b_i]$ or at $[\leftarrow r_i \leftrightarrow b_i]$ for $[a_i, r_i)$ could be used to estimate the maximum error of the function *value* of $g_f(h^*)$, based on the new assumption.

This will be implemented in the future, but for now, we will use $|g_f(l_i) - g_f(r_i)| < \rho$ as termination condition for some precision parameter ρ. Note that this is bad design, but an easy condition to start with, as it is a necessary, but not sufficient indicator that our approximation is precise. The results shown later use this "bad" termination condition.

6.2.2. Golden-Section Search

The Golden-Section Search uses intermediate points $l_i = b_i - \frac{b_i - a_i}{\varphi}$ and $r_i = a_i + \frac{b_i - a_i}{\varphi}$, where $\varphi := \frac{1+\sqrt{5}}{2}$ is called "golden section". Directly matching the expression to the quadratic formula shows the fact $\varphi^2 - \varphi - 1 = 0$, from which we can also derive $\varphi(\varphi - 1) = 1$, $\varphi^2 = \varphi + 1$ and $\frac{1}{\varphi} = \varphi - 1$.

When we sort out the interval $(r_i, b_i]$, the new interval would be $[a_{i+1}, b_{i+1}]$ with $a_{i+1} = a_i$, $b_{i+1} = r_i$, size $b_{i+1} - a_{i+1} = \frac{b_i - a_i}{\varphi}$, $l_{i+1} = b_{i+1} - \frac{b_{i+1} - a_{i+1}}{\varphi}$ and $r_{i+1} = a_{i+1} + \frac{b_{i+1} - a_{i+1}}{\varphi}$. We can now prove that $r_{i+1} = l_i$:

$$\begin{aligned}
r_{i+1} &= a_{i+1} + \frac{b_{i+1} - a_{i+1}}{\varphi} \\
&= a_i + \frac{b_i - a_i}{\varphi^2} \\
&= a_i + (b_i - a_i) \cdot (\varphi - 1)^2 \\
&= a_i + (b_i - a_i) \cdot (\varphi^2 - 2\varphi + 1) \\
&= a_i + (b_i - a_i) \cdot (2 - \varphi) \\
&= (b_i - a_i) + a_i + (b_i - a_i) \cdot (1 - \varphi) \\
&= b_i - (b_i - a_i) \cdot (\varphi - 1) \\
&= b_i - \frac{b_i - a_i}{\varphi} \\
&= l_i \ . \ \square
\end{aligned}$$

This proof can be done symmetrically for $l_{i+1} = r_i$ for the other case of sorting out $[a_i, l_i)$. This fact implies that we only need to evaluate the function at one intermediate point, while the other intermediate point is already evaluated, as it is conserved from iteration i.

6.2.3. Possible Enhancements

As already mentioned, there are many possible enhancements or alternatives at this stage, but they are postponed for temporal reasons, for example:

- Extend initial phase to both directions (see Algorithm 9)
- Optimized growth factor in initial phase
- Extrapolation-based precision estimate for $\frac{\min\{g_f(a_i), g_f(b_i)\}}{g_f(h^*)}$ and better extrapolation techniques as described earlier
- Implementation of g_f', enabling Binary Search or a better extrapolation
- Case-distinction between 4-point-minimum at border or at intermediate points
  - If 4-point-minimum at border point, two intervals can be ruled out, shrinking the interval to size $\frac{b-a}{\varphi^2}$ in one instead of two iterations (1 iteration skipped)
  - Would require two new function evaluations, resulting in the same ratio of function evaluations per total shrink factor
  - "Lazy" function evaluation, i.e. only evaluating a second intermediate point if its interval wasn't ruled out yet (first border lower than first intermediate point), could skip some function evaluations
  - For lazy evaluation: choose the first of both intermediate points in the direction of the 2-point

minimum of the border points (s.t. the other one could lie in a then already ruled-out interval)

- Usage of a root approximator on g'_f

6.3. Empiric Search for α

In Section 4.1.4, α was defined within the definition of the grid approximator. When R and b are given, we can calculate $\|R(b)\|_\infty$ as well as opt(b). We can deduce the (in-)equation $1 \leq \frac{\text{opt}(b)}{\|R(b)\|_\infty} \leq \alpha$. Note that the first inequality is always satisfied for our approximator: While our formula for opt(b) uses *solutions for f that satisfy b*, the approximator R only calculates *restrictions* about the optimality that a solution for f *could* have *at most*. Each value of $R(b)$ is related to exactly one cut, and represents the total flow that has to be routed into this cut, divided by its capacity. When looking at $\|R(b)\|_\infty$, it is clear that no solution for f could achieve a lower maximum congestion than this value, as both the nodes inside the cut can't be satisfied with lower flow into this cut, and the capacity of the cut can't be exceeded, as this would mean that some edge capacity would be exceeded as well. As the algorithm's runtime scales in quadratic order of α, it is beneficial to find the minimal α s.t. R is an α-approximator. A choice with perfect precision would be $\alpha = \inf_b \left\{ \frac{\text{opt}(b)}{\|R(b)\|_\infty} \right\}$. However, it is not trivial to find the value of this infimum, even for one-dimensional unit-capacity grid graphs ("chain graph"). It is even possible to construct several demands b that all satisfy $\|R(b)\|_\infty = \text{opt}(b)$, e.g. by forcing a cut that is represented in R to send/receive flow equal to its capacity. When experimenting with algorithms, it is thus possible by chance to only iterate with demands $\vec{b}_1, ..., \vec{b}_I$ that satisfy opt$(\vec{b}_k) \leq \alpha_I \cdot \|R(\vec{b}_k)\|_\infty$ with $\alpha_I < \alpha$ for all $k = 1, ..., I$, even with $\alpha_I = 1$ (though very unlikely).
We will now discuss some properties of unit-capacity grid graphs and provide some algorithms for an empiric search for α.

6.3.1. Random Sampling for $d = 1$

One-dimensional grid graphs are trees. Hence, a valid demand $b = (b_1, ..., b_n)^T$ with $\sum_{i=1}^n b_i = 0$ has a unique solution for a flow, which can be calculated by leaf elimination. When we look at (v_{i+1}, v_i), representing the edge $\{v_i, v_{i+1}\}$, we see that it has to route exactly the value of the cut $(\{v_1, ..., v_i\}, \{v_{i+1}, ..., v_n\})$, which is $\sum_{j=1}^i b_j$. Since we have unit-capacity edges, this gives us the formula opt$(b) = \max_{i=1,...,n-1} \left| \sum_{j=1}^i b_j \right|$.
For this dimension, we will make a random sampling. When constructing a random demand b, it is important to consider two things in order for a solution f to fulfil the capacity constraints: firstly, each cut can only have total demand less or equal to its capacity, and secondly, the total demand of the graph has to equal 0. One idea is to just generate random numbers between -1 and 1 for the edges, and then calculate the excess flow at each vertex. With the formula for the cut capacity, we can also work in the opposite direction: if we already chose the demands b_1 to b_i, we know that the edge (v_{i+1}, v_i) has value $\sum_{j=1}^i b_j$, so if the next edge also has a random value between -1 and 1, this is just the same as stating that the excess flow at v_{i+1}, b_{i+1}, has a random value between $-1 - \sum_{j=1}^i b_j$ and $1 - \sum_{j=1}^i b_j$. This has the advantage of generating b directly. This gives an iterative approach, where the last demanded excess b_n can be calculated via the restriction of total neutrality, i.e. $b_n = -\sum_{j=1}^{n-1} b_j$. An algorithm for generating a sample is given with Algorithm 10,

containing one further adjustment that will be discussed in the next paragraph.

There is no "bad" sample in our case, but there are "good" ones: the question is, how high the ratio

Algorithm 10 Algorithm for generating a Random Sample for a 1-dimensional grid graph with n nodes, with its optimal value limited to ς. Returns both the random demand b and the actual value of opt(b).

```
1: procedure GENERATESAMPLE(n, ς)
2:     bsum ← 0.0
3:     opt ← 0.0
4:     for j ← 1, n − 1 do
5:         val ← Random.Uniform(−ς, +ς) − bsum
6:         b_j ← val
7:         bsum ← bsum + val
8:         if opt < |bsum| then
9:             opt ← |bsum|
10:        end if
11:    end for
12:    b_n ← −bsum
13:    return b, opt
14: end procedure
```

$\frac{\text{opt}(b)}{\|R(b)\|_\infty}$ can be, where a "good" sample has this ratio as high as possible, and all samples that are "worse" will just be omitted for this estimate. Yet, there are ways to avoid generating "good" samples: for example, if we just randomly pick the flow value at edges, it is highly probable that an edge will have flow value near to its capacity. There is also a high chance that this will occur in some cut represented by the approximator metrics. This would mean that the estimate for α at this sample will be near its limit of 1, so we would get no good estimate. It is therefore reasonable to introduce classes with lower values of opt(b) in the sampling. A simple approach would be to define a class with *maximal* optimal value ς, as we don't want the randomness to pick the optimal value itself, but to limit it. This can be done by modifying the random choice of b_{i+1} to a random value between $-\varsigma - \sum_{j=1}^{i} b_j$ and $\varsigma - \sum_{j=1}^{i} b_j$.

Note that for $n = 4$, $\alpha = 1$, as each edge also represents a cut in the 1-dimensional case, and all the cuts are also represented in the approximator tree for $n = 4$. For $n \in \{5, 6\}$, this can also be the case. Consider the tree structures $S_1^5 = [[[L, L], L], [L, L]]$, $S_2^5 = [[L, [L, L]], [L, L]]$, $S_3^5 = [[L, L], [[L, L], L]]$ and $S_4^5 = [[L, L], [L, [L, L]]]$. S_1^5 and S_4^5 also represent an $(\alpha = 1)$-approximator. Consider $S_1^6 = [[[L, L], L], [[L, L], L]]$, $S_2^6 = [[[L, L], L], [L, [L, L]]]$, $S_3^6 = [[L, [L, L]], [[L, L], L]]$ and $S_4^6 = [[L, [L, L]], [L, [L, L]]]$. S_2^6 also would be an $(\alpha = 1)$-approximator. The reason behind this is the symmetry of the cuts $(S, V \setminus S)$ and $(V \setminus S, S)$ together with the possibility of structuring a three-node subtree such that there exist three nodes that, when interpreted as new subtree, have consecutively all three possible "heads" of the leaf series $((l_1), (l_1, l_2), (l_1, l_2, l_3))$. Thus, all three sums that are in the form of the explicit formula for the optimal value will be an entry in $R(b)$. Together with the symmetry, the other subtree can also be arranged to (for $n = 6$) or already does (for $n = 5$) contain the remaining formulae for the edge flow values. This means that the optimal value is then contained in $R(b)$. (Note: the splitting algorithm in our program would result in choosing the structures S_1^5 and S_1^6.)

For $n \geq 7$, the aforementioned strategy for designing $(\alpha = 1)$-approximators won't work further, as

$7/2 > 3$ and hence, there will be a subtree with more than 3 leaves.

6.3.1.1 Extension to Higher Dimensions

We can reuse the strategy from $d = 1$ to some extent in higher dimensions. With $d \geq 2$ and $\vec{b}^1$ as one-dimensional reference demand, we pick a dimension d_l and set all demands to $(\vec{b}^1)_i$ where the position in grid coordinates $(v_1, ..., v_d)$ satisfies $v_{d_l} = i$. This is our higher-dimensional demand $\vec{b}^d$, which we will handle as tensor in this section, where $(\vec{b}^d)_{v_1,...,v_d}$ denotes the demand at vertex $(v_1, ..., v_d)$ in grid coordinates.
For each i, the vertices can be seen as hyperplane, each one corresponding to exactly one entry of the one-dimensional reference demand. With N_H as count of vertices of any hyperplane, the net flow of this hyperplane has to equal $N_H \cdot (\vec{b}^1)_i$. Observing the cuts $(H_1 \cup ... \cup H_i, H_{i+1} \cup ... \cup H_n)$ (with H_i being the i-th hyperplane) reveals that these cuts have to route exactly $N_H \cdot \sum_{j=1}^{i} b_j$ from $H_{i+1} \cup ... \cup H_n$ to $H_1 \cup ... \cup H_i$. With $\max_{i=1,...,n-1} \left| N_H \cdot \sum_{j=1}^{i} b_j \right| = N_H \cdot \max_{i=1,...,n-1} \left| \sum_{j=1}^{i} b_j \right|$ as biggest absolute of those cut values, we have a lower limit for the maximum congestion, that is $\frac{max_{i=1,...,n-1} \left| N_H \cdot \sum_{j=1}^{i} b_j \right|}{N_H} = \max_{i=1,...,n-1} \left| \sum_{j=1}^{i} b_j \right| = \text{opt}(\vec{b}^1)$. As we can construct a routing by sending $\sum_{j=1}^{i} b_j$ from each $(v_1, ..., v_{d_l}+1 = i+1, ..., v_d)$ to $(v_1, ..., v_{d_l} = i, ..., v_d)$ (for $i = 1, ..., n-1$) that has exactly maximum congestion $\max_{i=1,...,n-1} | \sum_{j=1}^{i} b_j | = \text{opt}(\vec{b}^1)$, we can finally conclude that $\text{opt}(\vec{b}^d) = \text{opt}(\vec{b}^1)$. This gives us the opportunity to extend the sampling to further dimensions, where the result may not approximate α directly, but a lower limit for α.

To some extent, additional logical conclusions are at reach. We start with observing aforementioned strategy for $d = 2$. The splitting strategy leads to cuts S that are rectangle-like subgraphs with dimensions $m_S \times n_S$. As the calculation is the same for each dimension, the "width" m_S will *always* observe the same "horizontal" interval as the approximator structure for the reference demand. Hence, we can conclude that $s_2 = n_S \cdot s_1$, where s_1 and s_2 are the sums of the demanded excess flows inside this cut, for the one-dimensional reference cut and S respectively. This means that the approximator will have $\frac{s_2}{(1+\delta_v)m_S+(1+\delta_h)n_S}$ as one metric, where δ_v and δ_h are 1 if S has neighbours to both sides in horizontal (δ_h)/vertical (δ_v) direction, and 0 else. With $\beta = \frac{m_S}{n_S}$ and using the previous fact, we can conclude that this one metric will be $\frac{s_1}{(1+\delta_v)\beta+(1+\delta_h)}$. In the one-dimensional reference, the corresponding metric was $\frac{s_1}{1+\delta_h}$. Taking the maximal value among all cuts represented by the approximator structure, we get a relatively precise prediction of what the output of the higher-dimensional sampling will be for the demands we constructed, when we already have the output of the reference demand estimation. With $\beta \in (0, \infty)$, we can state that the lower limit for α obtained from the sampling in two dimensions is less or equal to the computed α of one dimension. Since we can always take the dimension with smallest node count for comparison with a reference, we can even restrict β to $(0, 1]$ when sampling accordingly.

6.3.2. Randomized Cut with Maximum Congestion

To obtain a randomized cut with maximum congestion, a relatively easy strategy would be the following, where we construct a flow with optimal maximum congestion:

1. Choose an arbitrary (e.g. random) cut S.
2. Calculate the capacity cap$(S) = c_S$.
3. Choose a maximum congestion ς.
4. Set the flow value f_e for all edges e that connect S and $V \setminus S$ to ς, where the direction is uniformly out of S or uniformly into S. This way, $\varsigma \cdot c_S$ flow have to be routed into (or out of) S in total.
5. Set all other flow values to an arbitrary (e.g. random) value between $-\varsigma$ and ς.
6. Calculate the excess flows via $b_R = B(f)$.
7. The demand b_R satisfies opt$(b_R) = \varsigma$.

This approach could also be seen as observing the "missing metrics" of our grid approximator, as the optimal value is opt$(b_R) = \varsigma = \frac{\varsigma \cdot c_S}{c_S} = \frac{b_S}{c_S}$. If the random cut is also one of the cuts represented by the approximator nodes, our approximator will just return $\|R(b)\|_\infty = \varsigma =$ opt(b_R). We thus get no information about α when choosing the cuts represented by the approximator nodes, so it makes sense to skip them for a random sampling.

6.3.3. Pre-Calculating Optimal Maximum Congestion

Another idea is to pre-calculate the optimal maximum congestion. We could either choose another algorithm with probably much worse running time to calculate the optimal solution, or we over-estimate α (i.e., $\alpha = 10$ has been a huge over-estimate for all tests done for this thesis) and choose the algorithm from [She13] with or without our adjustments to calculate an approximate solution. An approximate solution is enough for an empiric search, as α will be estimated by the upper limit of the empiric values of $\frac{\text{opt}(b)}{\|R(b)\|_\infty}$, so an inaccuracy of ϵ in the optimal congestion approximation will also lead to an inaccuracy of ϵ in the α estimate.
This approach may have a large running time to get good results, as for each sampled b, an approximation would be needed. But it also has the advantage of being able to handle all demands b and getting a relatively precise estimate for α with increasing runtime.

7. Results

All datasets are included in the repository, together with a class `ApproximationRunner` that facilitates the data generation by providing some example test runs.

For evaluation, we use the demands b_1, b_2, b_3 on the 4×4 2-dimensional grid graph, and b_4 on the 8×8 2-dimensional grid graph. All evaluations in this chapter either concern `AlmostRoute` or the empiric search for α.
In vector coordinates (starting at 0), b_1 has demand $+1$ for $(0, j)$, demand -1 for $(3, j)$ and 0 at the rest.
b_2 has demand $+1$ at $(0, 0)$, -1 at $(3, 3)$ and 0 at the rest.
b_3 has demand $+0.2$ at $(1, 1)$, -0.4 at $(1, 2)$, $+0.5$ at $(2, 1)$, -0.3 at $(2, 2)$ and 0 at the rest.
b_4 has demand $+1$ at $(0, 0)$, -1 at $(7, 7)$ and 0 at the rest.
It is easy to see that opt$(b_1) = 1$ and opt$(b_2) =$ opt$(b_4) = 0.5$. For b_3, the optimal congestion is 0.175 (at least 0.7 has to be routed over the 4 edges $(1, j) \to (2, j)$, restricting opt$(b_3) \geq 0.7/3 = 0.175$, and it is also relatively easy to construct a flow with this maximum congestion; we skip giving an explicit optimal flow, since this is rather tedious than helpful).
For the number of iterations and the potential threshold cut optimality, the generated data for b_1, b_2, b_3, b_4 for each $\varepsilon = 0.01, 0.02, ..., 0.1$ are found in the Results/Iterations-Basic folder, each file containing the results from runs with several values for α (0.15, 0.7515, 1.353, ...) and s, where s is a "static" factorization of the step size by s in each step.
Results from an evaluation of the step size optimization are found in the Results/Stepsize-Optimization folder.
Note that there was an iteration limit of $500,000$. Runs with more iterations were aborted, meaning that their result is most likely not correct and can be discarded.
We won't show all generated data in this chapter, but some relevant examples with visualization. The other data can be accessed via the repository, where also model files with visualizations are provided. The data is formatted in CSV, the model files are formatted in Microsoft Excel Workbooks (.xlsx).

7.1. Iterations

We measure the number of iterations, depending on the parametrization of α and the choice of s. The experimental evaluations confirm the quadratic order of α for the runtime of `AlmostRoute` when using naive steepest descent. Results for the number of iterations for b_1 can be seen in Figure 20 for $\varepsilon = 0.01$, and in Figure 21 for $\varepsilon = 0.1$. The result for b_2 and $\varepsilon = 0.1$ can be seen in Figure 22.
The optimal choices for α for minimal iterations were between 1.353 and 2.556, out of the observed parameters. s was optimal for higher values of s, but the step size optimization yielded results that had even less iterations.

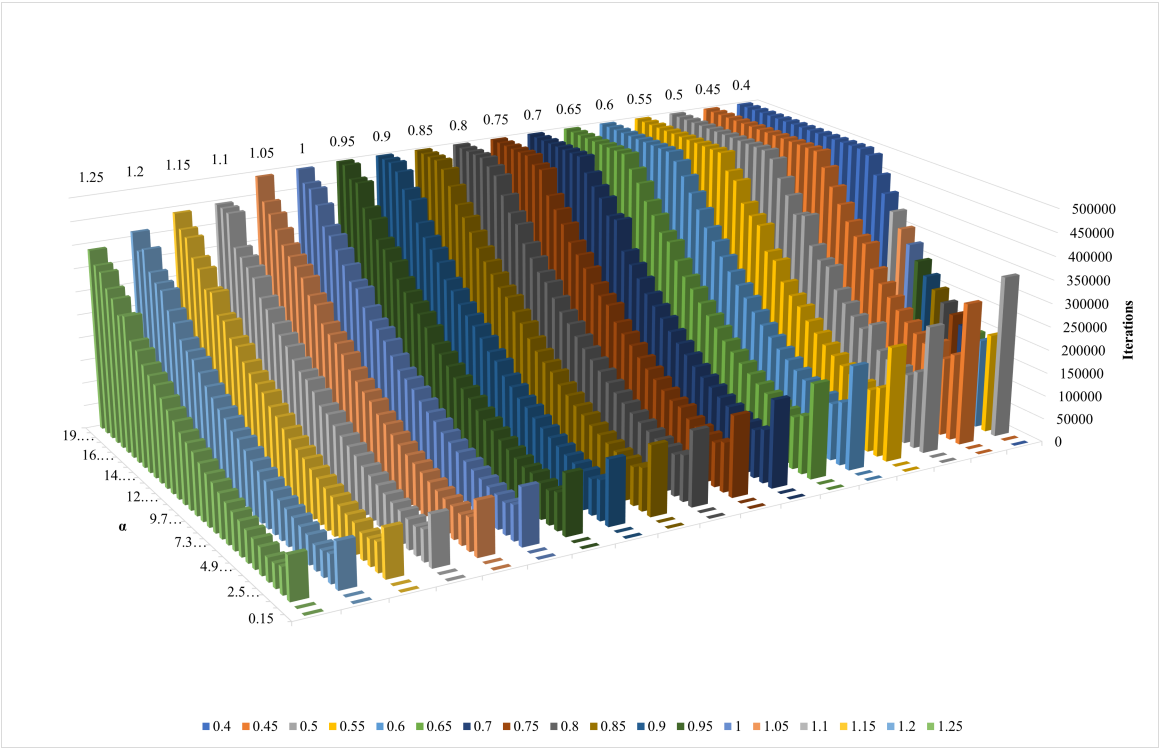


Figure 20 Number of iterations for b_1 and $\varepsilon = 0.01$, w.r.t. α and s.

Note that for $\alpha < 1$, the potential is changed s.t. the penalty ϕ_T nearly vanishes, and `AlmostRoute` will output a flow that is close to 0, which obviously is not what we want.

7.2. Potential Threshold Cut Optimality

The optimality of the cut that is induced by the potentials of `AlmostRoute` according to $\Phi(f) \leq (1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1}$ with $v = R^T \nabla lmax(2\alpha R(b - Bf))$ was valid for all test runs with $\alpha > 1.353$. Results for this congestion metric are shown in Figures 23 and 24 for b_1 with $\varepsilon = 0.01$ and $\varepsilon = 0.1$.

7.3. Step Size Optimization

The step size optimization showed significant iteration and runtime improvements, even for our simple implementations of the ternary search and the golden section search. The runtime and iterations were slightly better with golden section search compared to ternary search for most runs, and sometimes significantly better.
Comparisons between an unoptimized run and runs with golden section search are shown in Figures 25, 27 and 26.

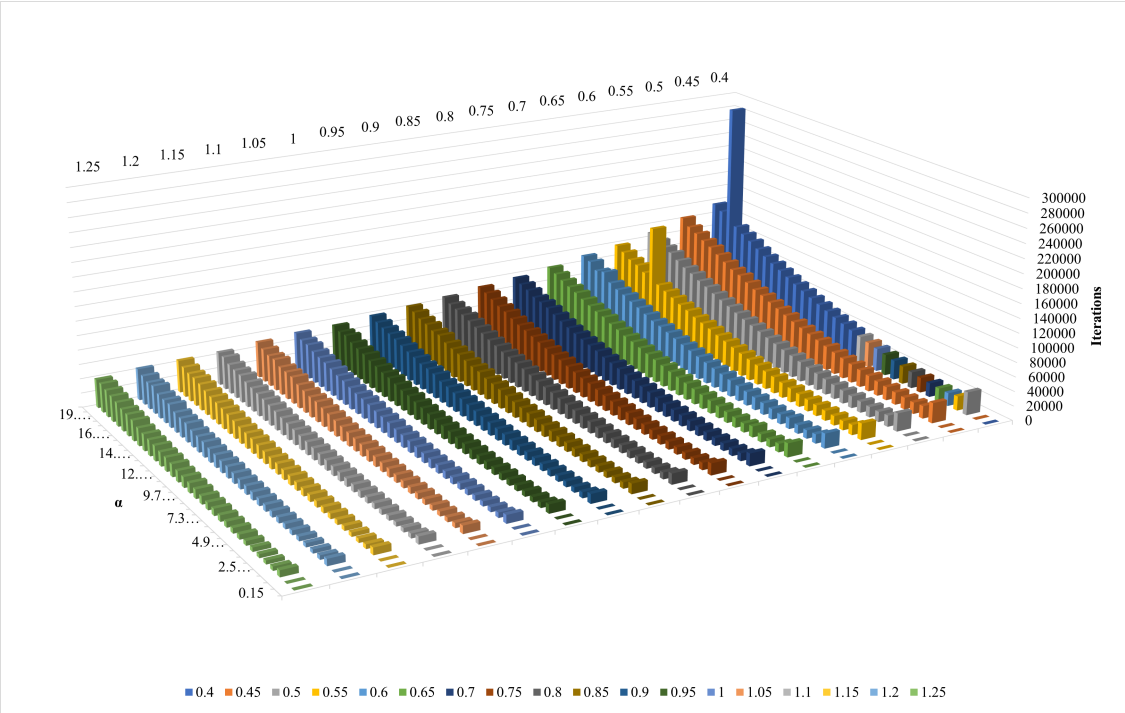


Figure 21 Number of iterations for b_1 and $\varepsilon = 0.1$, w.r.t. α and s.

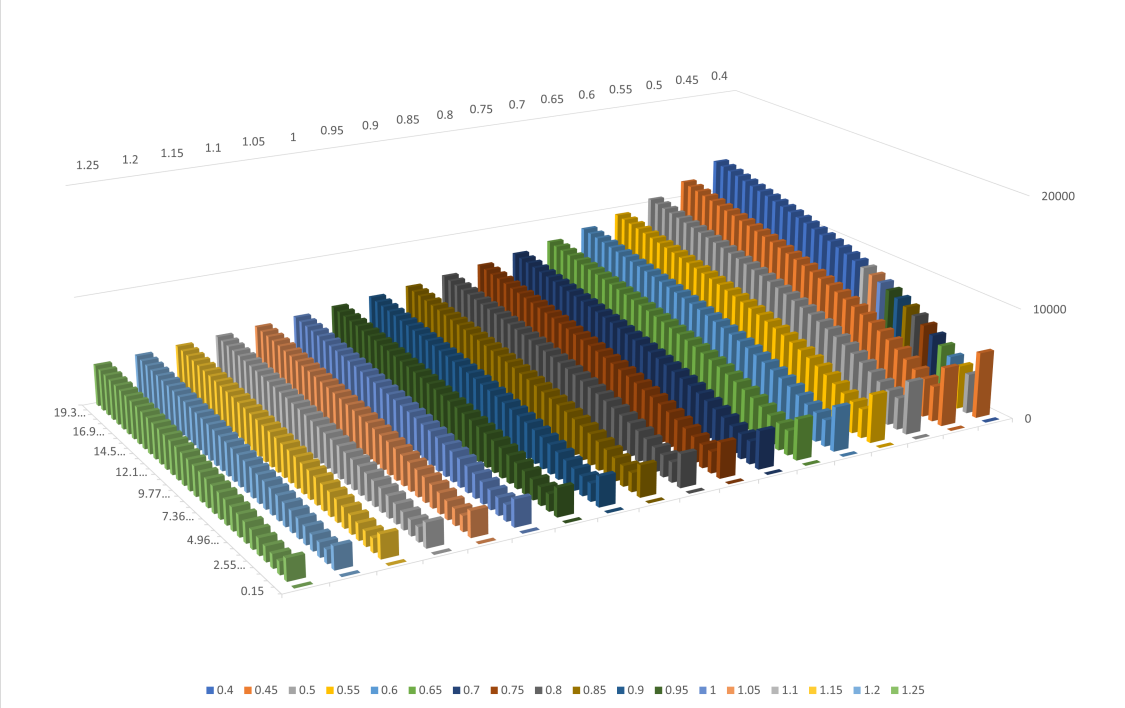
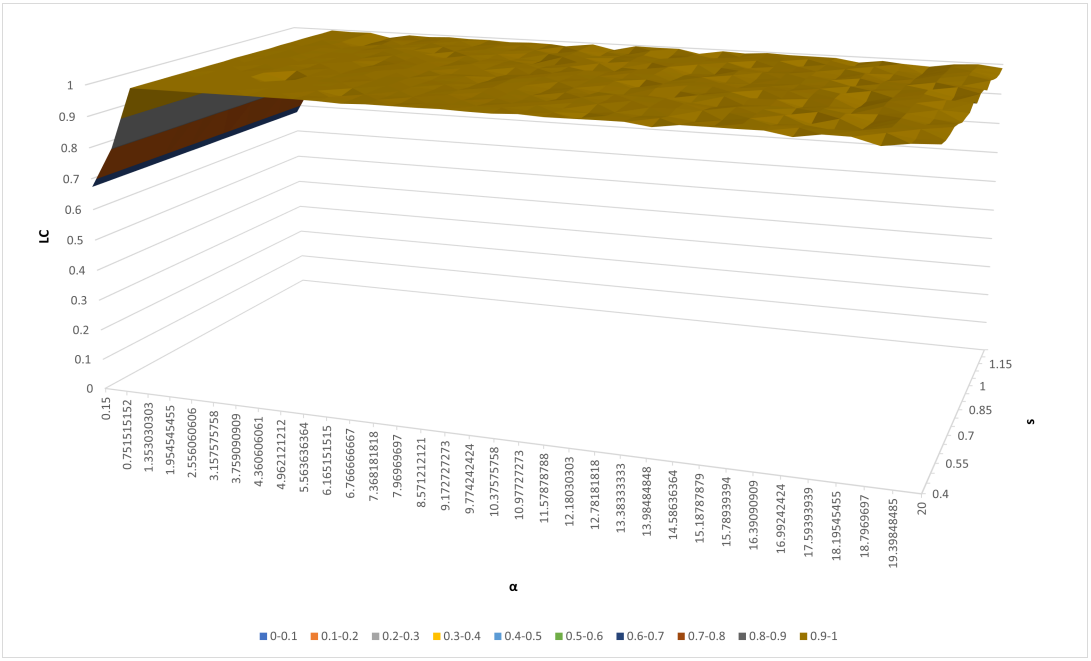
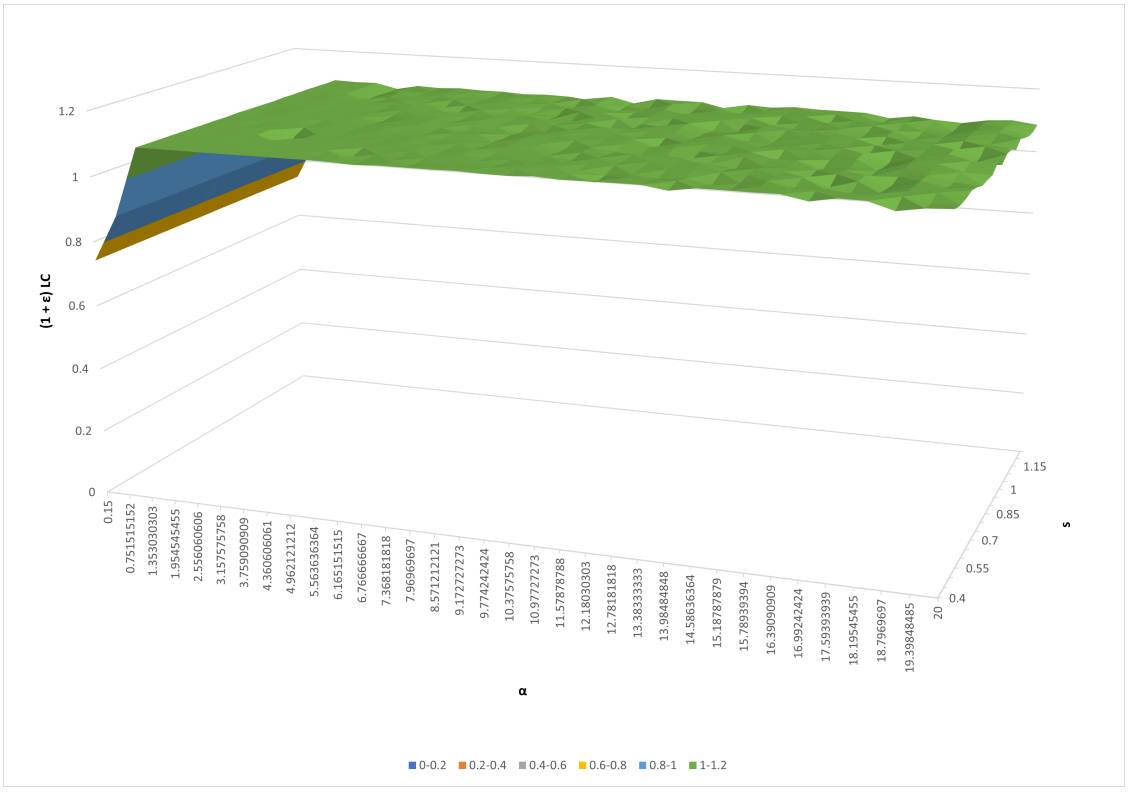


Figure 22 Number of iterations for b_2 and $\varepsilon = 0.1$, w.r.t. α and s.

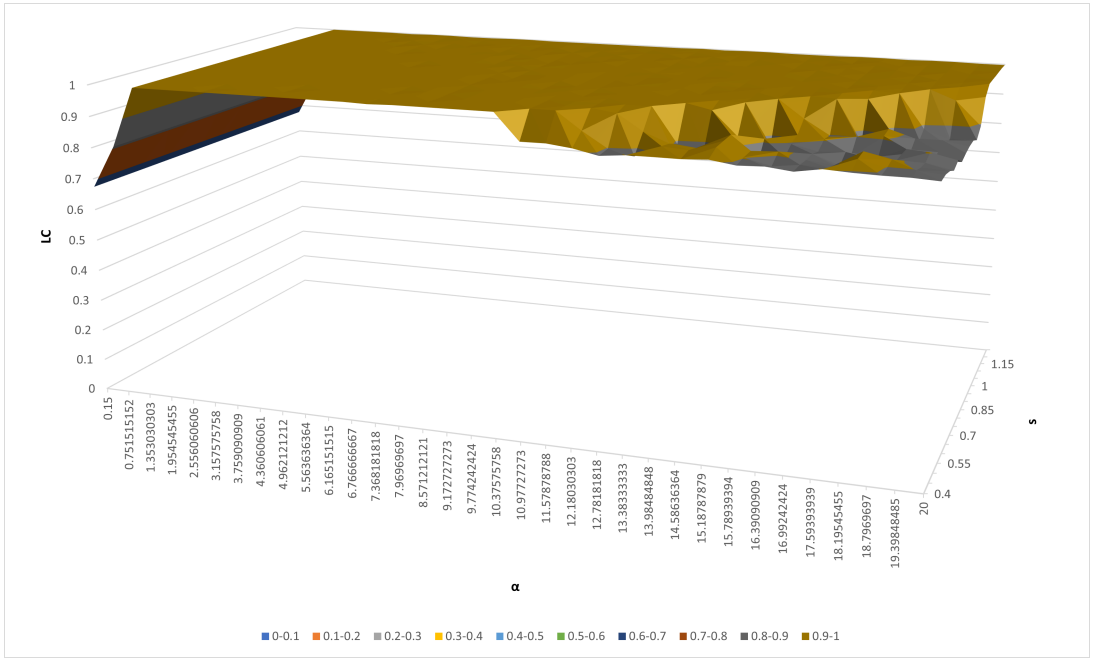


(a) $\frac{b^T v}{\|CB^T v\|_1}$.

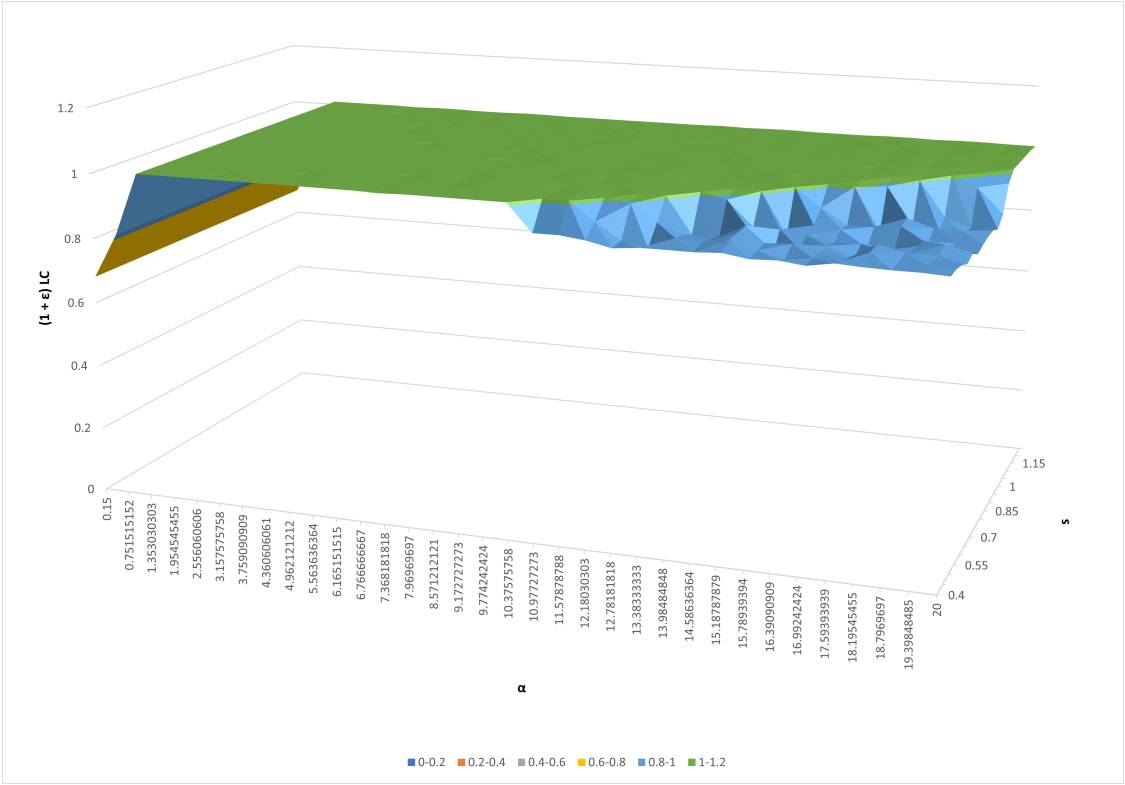


(b) $(1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1}$.

Figure 23 $\frac{b^T v}{\|CB^T v\|_1}$ and $(1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1}$ for potentials v after termination of `AlmostRoute`$(b_1, 0.1)$. The optimal solution is 1 and has to lie between both values.

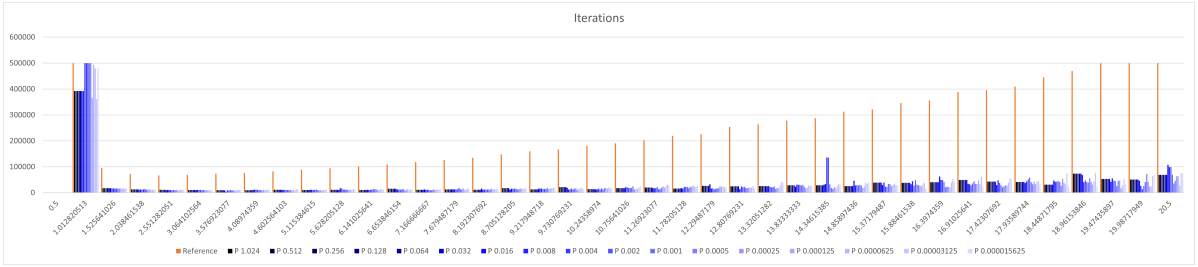


(a) $\frac{b^T v}{\|CB^T v\|_1}$.

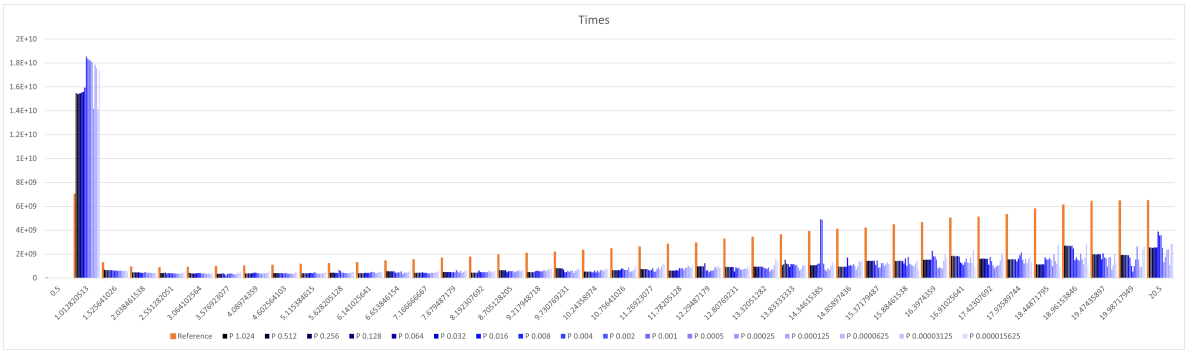


(b) $(1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1}$.

Figure 24 $\frac{b^T v}{\|CB^T v\|_1}$ and $(1+\varepsilon)\frac{b^T v}{\|CB^T v\|_1}$ for potentials v after termination of `AlmostRoute`$(b_1, 0.01)$. The optimal solution is 1 and has to lie between both values. Note that the results from runs that took $500,000$ iterations should be ignored here, as the iteration was prematurely terminated. See Figure 20 for iterations.

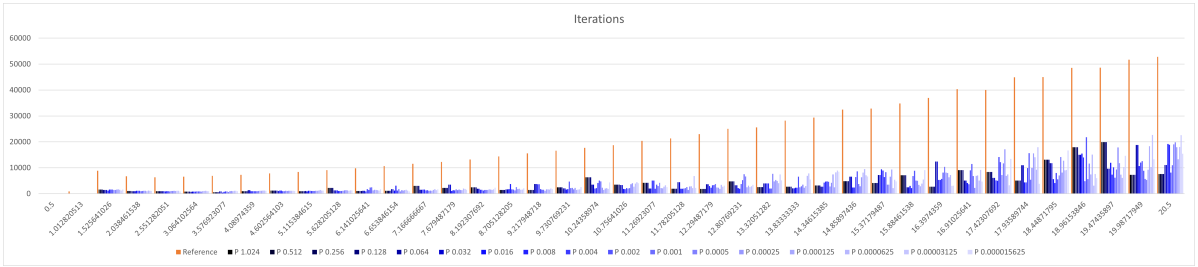


(a) Comparison of iterations.

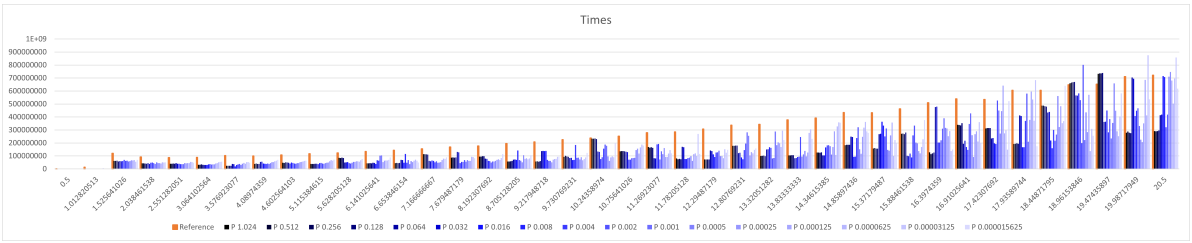


(b) Comparisons of runtimes.

Figure 25 Comparison of iterations and runtimes between naive steepest descent and golden section search optimization for b_1 and $\varepsilon = 0.01$.

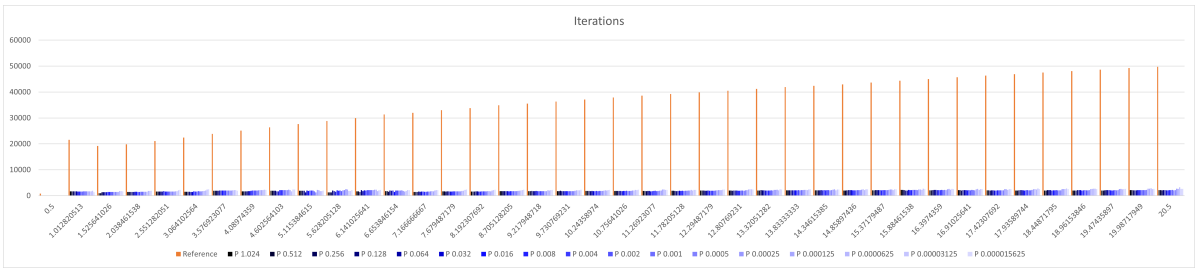


(a) Comparison of iterations.

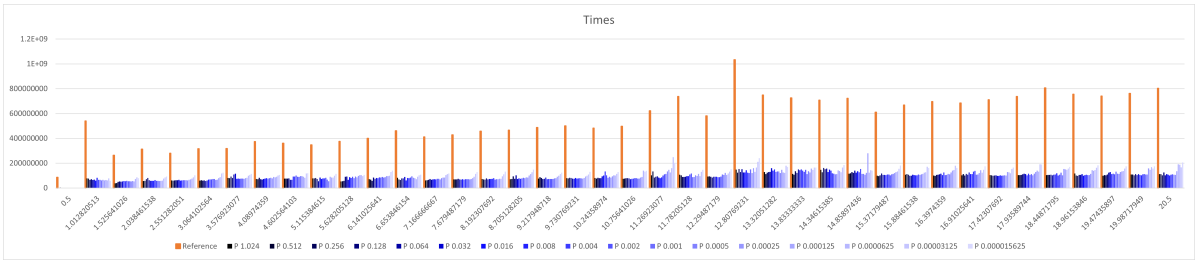


(b) Comparisons of runtimes.

Figure 26 Comparison of iterations and runtimes between naive steepest descent and golden section search optimization for b_1 and $\varepsilon = 0.1$.



(a) Comparison of iterations.



(b) Comparisons of runtimes.

Figure 27 Comparison of iterations and runtimes between naive steepest descent and golden section search optimization for b_2 and $\varepsilon = 0.01$.

7.4. Empiric Search for α

For $d = 1$, an extensive random sampling was performed. For millions of samples, the highest value of the sampled lower limits for α was outputted, condensed into a single data point. For the results, see Figure 28. As expected, $n = 4, 5$ have an approximator with $\alpha = 1$.
Some smaller samplings with $\varsigma = 0.02 \cdot i$ and 1000 samples for each $i \in [50]$ were also performed, outputting the value of the α estimate $\frac{\mathsf{opt}(b)}{\|Rb\|_\infty}$ for each sample b. This gives a better overview on the distribution of this ratio. The results for $n \in \{4, 8, 16, 32, 64\}$ are shown in Figures 29, 30, 31, 32 and 33. Taking a look at the evaluations of α, we can conclude that $\alpha = 3$ can be used for our approximator at $d = 1$. Values nearly at 3, like 2.99758364093494, were reached in most samplings for small n, but no estimate was found that would suggest $\alpha < 1$ or $\alpha > 3$. (Note that due to numerical errors, sometimes the estimate is $0.99999999...$ instead of 1.)
For bigger n at $d = 1$, the estimates of α decreased overall. Since an increase in samples results in higher estimates, but only slightly increasing relative to the number of samples, it might be necessary to choose another strategy for sampling in order to get better estimates in acceptable time.

It seems that $\alpha = 3$ should be a good choice for α, but since the approximator will have even better congestion approximations for *most* demands, it can be chosen even smaller. This evaluation suggests choosing $\alpha = 3$ as parameter when looking for a correct α, or $\alpha \approx 2 \pm 0.5$ as value leading to less iterations on average.

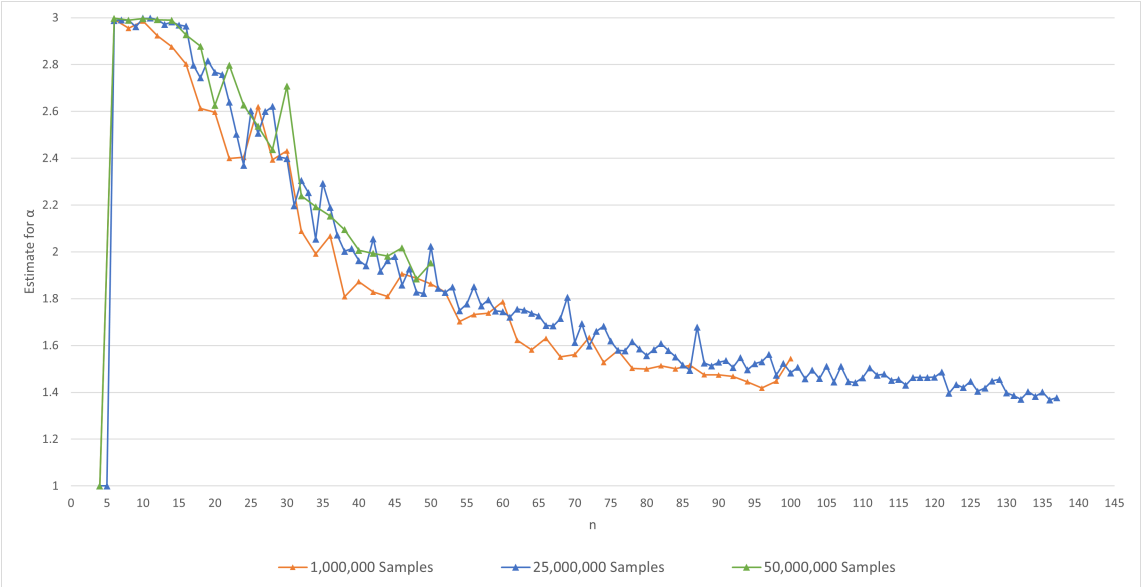


Figure 28 Results of Random Sampling for $d = 1$. Each data point corresponds to the result of 1 / 25 / 50 million samples.

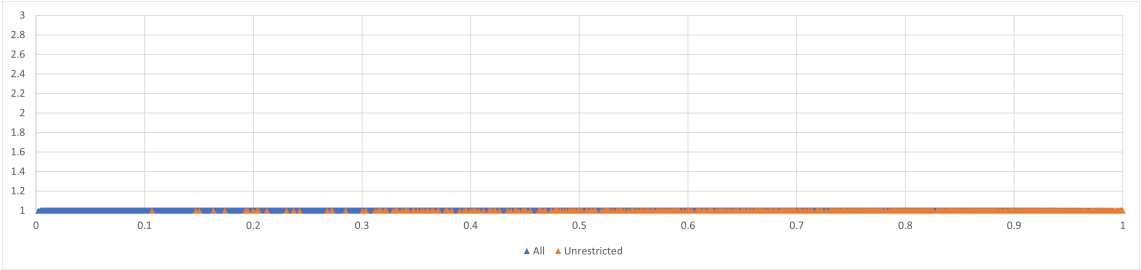


Figure 29 Results of Random Sampling for $d = 1$ and $n = 4$ with $50 \cdot 1000$ samples. Each data point corresponds to the result of $\frac{\text{opt}(b)}{\|Rb\|_\infty}$ for one sampled b. The results of the unrestrictedly generated samples with $\varsigma = 1$ are marked.

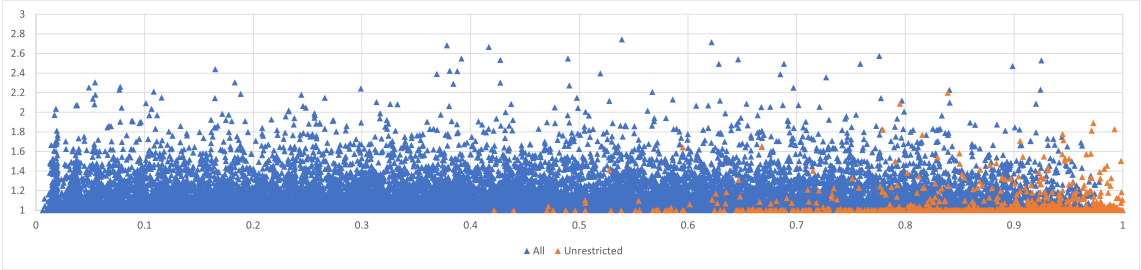


Figure 30 Results of Random Sampling for $d = 1$ and $n = 8$ with $50 \cdot 1000$ samples. Each data point corresponds to the result of $\frac{\text{opt}(b)}{\|Rb\|_\infty}$ for one sampled b. The results of the unrestrictedly generated samples with $\varsigma = 1$ are marked.

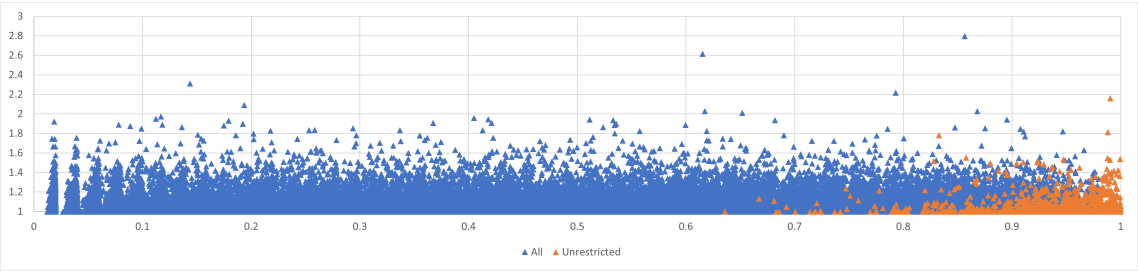


Figure 31 Results of Random Sampling for $d = 1$ and $n = 16$ with $50 \cdot 1000$ samples. Each data point corresponds to the result of $\frac{\text{opt}(b)}{\|Rb\|_\infty}$ for one sampled b. The results of the unrestrictedly generated samples with $\varsigma = 1$ are marked.

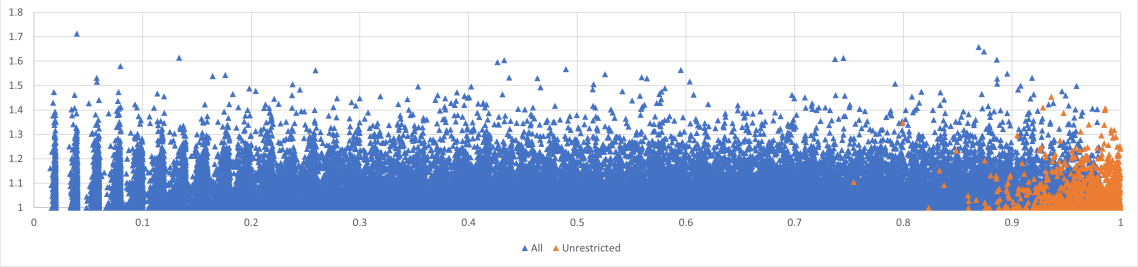


Figure 32 Results of Random Sampling for $d = 1$ and $n = 32$ with $50 \cdot 1000$ samples. Each data point corresponds to the result of $\frac{\text{opt}(b)}{\|Rb\|_\infty}$ for one sampled b. The results of the unrestrictedly generated samples with $\varsigma = 1$ are marked.

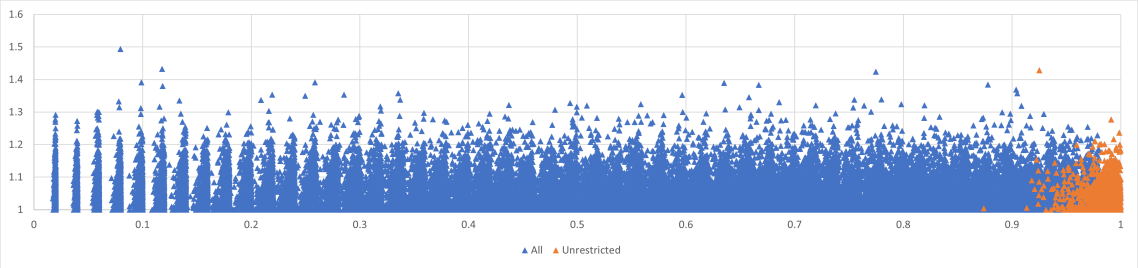


Figure 33 Results of Random Sampling for $d = 1$ and $n = 64$ with $50 \cdot 1000$ samples. Each data point corresponds to the result of $\frac{\text{opt}(b)}{\|Rb\|_\infty}$ for one sampled b. The results of the unrestrictedly generated samples with $\varsigma = 1$ are marked.

8. Final Remarks

There are many possible improvements that lie at hand, but that were omitted in this thesis due to temporal reasons. Those include the enhancements and alternatives discussed for the golden section search, some steps of proofs that could be shown in more detail, a comparison with Nesterov's accelerated gradient descent method instead of the naive steepest descent at the step size optimization, more samplings for α (for example, the suggestions from the implementation chapter, namely randomized cut with maximum congestion or a pre-calculation of the approximate optimal flow value) and optimizations of the data structures.
Also, `AlmostRoute` could be extended to also return the potential threshold cuts, and an evaluation of `CompleteRoute` can be done.
The repository is public - if desired, feel free to contact me: `js97.sw@gmail.com`.
I finally want to thank Prof. Dr. Harald Räcke for supervising and advising this thesis with interesting ideas, helpful inputs and all the time to answer my many questions, and Jonah Sherman for his great publication.

List of Figures

Bibliography

[She13] Jonah Sherman. “Nearly Maximum Flows in Nearly Linear Time”. In: *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. 2013, pp. 263–269. DOI: 10.1109/FOCS.2013.36.