Auswertung der Skorokhodmetrik

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Dipl.-Math. Oliver Falkenburg
aus Mannheim

Mannheim, 2016

Dekan:	Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:	Professor Dr. Jürgen Potthoff, Universität Mannheim
Korreferent:	Professor Dr. Andreas Neuenkirch, Universität Mannheim

Tag der mündlichen Prüfung: 27. September 2016

Abstract

The Skorokhod distance usually arises in the context of stochastic limits. A formula for evaluating the distance of real-valued step-functions defined on the unit interval is derived. Moreover, a polynomial time algorithm based on this formula is developed as well as an approximation scheme in order to approximate the Skorokhod distance of monotone càdlàg-functions to an arbitrary precision.

Zusammenfassung

Die Skorokhodmetrik steht üblicherweise im Zusammenhang mit stochastischen Grenzwerten. Eine Formel zur Auswertung der Skorokhodmetrik stückweise konstanter Pfade auf dem Einheitsintervall wird hergeleitet. Basierend auf dieser Formel wird ferner ein Algorithmus zur Auswertung in polynomieller Zeit sowie ein Approximationsschema für den Skorokhodabstand monotoner càdlàg-Funktionen mit beliebiger Genauigkeit entwickelt.

Danksagungen

Ich möchte mich bei Jürgen Potthoff für die Unterstützung und die zahlreichen fruchtbaren und inspirierenden Diskussionen bedanken.

Auch meiner Familie bin ich zutiefst dankbar für die vielen aufmunternden Gespräche.

Vor allem möchte ich mich bei meiner Frau, Jadwiga Falkenburg, bedanken, ohne deren Liebe, stete Unterstützung und enorme Geduld diese Arbeit nicht hätte abgeschlossen werden können.

Inhaltsverzeichnis

Darstellungsverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einführung

In den letzten Jahren haben in der Wahrscheinlichkeitstheorie sogenannte Lévyprozesse immer mehr an Bedeutung gewonnen. Dies sind stochastische Prozesse mit unabhängigen, stationären Zuwächsen. Im Zusammenhang mit der Simulation der Pfade solcher Prozesse kam die Frage auf, wie ähnlich sich zwei solche Pfade sind. Pfade von Lévyprozessen weisen typischerweise Sprünge auf. Die Frage kann man auch losgelöst von jeglichen wahrscheinlichkeitstheoretischen Überlegungen stellen: Wie ist die Ähnlichkeit zweier deterministischer Funktionen quantifizierbar? Ein bekannter Abstandsbegriff zwischen stetigen Funktionen ist die von der Supremumsnorm induzierte Metrik. Für Funktionen mit Unstetigkeitsstellen ist diese Metrik jedoch völlig ungeeignet. In diesem Zusammenhang hat A. Skorokhod mit dem Fokus auf stochastische Grenzwertsätze in seinem Artikel „Limit theorems for stochastic processes“ (siehe [1]) im Jahr 1956 mehrere Metriken vorgeschlagen. Der von ihm ursprünglich als $J1$-Metrik bezeichnete Abstandsbegriff hat sich im Laufe der Jahrzehnte durchgesetzt und ist mittlerweile als Skorokhodmetrik bekannt.

Leider konnte auch nach intensiver Recherche mit Ausnahme der Masterarbeit von L. Contreras (siehe [2]) keine Literatur über die Auswertung der Skorokhodmetrik gefunden werden.

Kapitel 2

Skorokhodmetrik

2.1 càdlàg-Funktionen

Wir beginnen mit der Definition einer weitreichenden Klasse von Funktionen. Diese sogenannten càdlàg-Funktionen („continue à droite, pourvue de limites à gauche“) spielen in der Wahrscheinlichkeitstheorie bei der Analyse von Pfaden stochastischer Prozesse eine zentrale Rolle. Die Abschnitte 2.1–2.2 dieser Arbeit basieren auf Kapitel 3, Abschnitt 12 des Buches von P. Billingsley (siehe [3]).

Definition 2.1.1. *Eine reellwertige Funktion f auf $[0,1]$ heißt* càdlàg*, falls für jedes $t \in (0,1]$ der Grenzwert*

$$\lim_{s \uparrow t} f(s) \tag{2.1.1}$$

existiert (und endlich ist) sowie für jedes $t \in [0,1)$

$$f(t) = \lim_{s \downarrow t} f(s) \tag{2.1.2}$$

gilt.

Wie wir nun sehen werden, reichen (2.1.1) und (2.1.2) bereits aus, um viele nützliche Eigenschaften abzuleiten. Der linksseitige Grenzwert in (2.1.1) wird fortan mit $f(t-)$ und der rechtsseitige in (2.1.2) mit $f(t+)$ abgekürzt. Wir bezeichnen $\Delta f(t) = f(t) - f(t-)$ für $t \in (0,1)$ als die Sprunghöhe von f an der Stelle t. Offensichtlich hat f genau dann eine Unstetigkeitsstelle in t, wenn $\Delta f(t) \neq 0$ gilt. Weiter sei $\mathbb{D}$ der Raum der reellwertigen càdlàg-Funktionen auf $[0,1]$. Das folgende Lemma von P. Billingsley ist von zentraler Bedeutung zur Herleitung wesentlicher Eigenschaften der Funktionen aus $\mathbb{D}$.

Lemma 2.1.2. *Für jedes $f \in \mathbb{D}$ und alle $\varepsilon > 0$ existieren endlich viele Punkte $0 = t_0 < t_1 < \ldots < t_N = 1, N \in \mathbb{N}$ mit*

$$\sup_{u,v \in [t_{i-1}, t_i)} |f(u) - f(v)| < \varepsilon, \qquad i = 1, \ldots, N. \tag{2.1.3}$$

Sei $f \in \mathbb{D}$ und wir nehmen an, es gäbe unendlich viele Unstetigkeitsstellen, an denen die absolute Sprunghöhe von f über einem vorgelegten $\varepsilon > 0$ liegt. Dann existiert eine Folge $(s_n, n \in \mathbb{N})$ in $[0,1]$ mit $s_i \neq s_j$ für alle i, j und $|\Delta f(s_n)| \geqslant \varepsilon$ für alle n. Nach Lemma 2.1.2 muss jedoch eine endliche Zerlegung des Einheitsintervalls mit (2.1.3) existieren. Also muss ein s_m mit $s_m \neq t_i$ für alle i in einem der von den t_i aufgespannten Intervalle liegen. Fällt das Folgenglied in das j-te Intervall, d.h. $s_m \in [t_{j-1}, t_j)$, so folgt

$$\sup_{u,v \in [t_{j-1}, t_j)} |f(u) - f(v)| \geqslant |\Delta f(s_m)| \geqslant \varepsilon,$$

was im direkten Widerspruch zu Lemma 2.1.2 steht. Folglich gilt

$$|\{t \mid |\Delta f(t)| > \varepsilon\}| < \infty$$

für alle $\varepsilon > 0$ und wegen $\{t \mid \Delta f(t) \neq 0\} = \bigcup_{m \in \mathbb{N}} \{t \mid |\Delta f(t)| > 1/m\}$ ist die Menge $\{t \mid \Delta f(t) \neq 0\}$ höchstens abzählbar.

Eine Funktion f bezeichnen wir als stückweise konstant, wenn es endlich viele Stellen t_i mit $0 = t_0 < t_1 < \cdots < t_N = 1$ derart gibt, dass f auf den halboffenen Intervallen $[t_{i-1}, t_i), 1 \leqslant i \leqslant N$ konstant ist. Von nun an sei $\mathbb{D}_c$ die Teilmenge der stückweise konstanten Funktionen in $\mathbb{D}$. Eine weitere unmittelbare Konsequenz aus Lemma 2.1.2 ist, dass $\mathbb{D}_c$ bezüglich der Supremumsnorm dicht in $\mathbb{D}$ liegt. Diese Eigenschaften der càdlàg-Funktionen fassen wir zu einem Korollar zusammen:

Korollar 2.1.3. *Für alle $f \in \mathbb{D}$ gilt:*

(i) *f ist beschränkt,*

(ii) *f besitzt höchstens abzählbar viele Sprungstellen,*

(iii) *Es gibt nur endlich viele Sprungstellen t_i, an denen die Sprunghöhe $|\Delta f(t_i)|$ ein vorgelegtes $\varepsilon > 0$ überschreitet,*

(iv) *f ist gleichmäßig durch stückweise konstante Funktionen approximierbar.*

2.2 Einführung

Wir beginnen mit der Definition der Skorokhodmetrik.
Definition 2.2.1. *Λ sei die Menge aller streng monoton wachsenden, stetigen Funktionen auf $[0,1]$, welche surjektiv nach $[0,1]$ abbilden. Dann ist die* Skorokhodmetrik *definiert als*

$$d_S(f,g) = \inf_{\lambda\in\Lambda}\left\{\|\lambda - Id\|_\infty \vee \|f - g\circ\lambda\|_\infty\right\} \tag{2.2.1}$$

für alle $f, g \in \mathbb{D}$.

Λ ist gerade die Menge aller wachsenden Homöomorphismen $\lambda : [0,1] \to [0,1]$. Für jedes $\lambda \in \Lambda$ ist $\lambda(0) = 0$ und $\lambda(1) = 1$. d_S ist das Infimum aller $\varepsilon > 0$ für die ein $\lambda \in \Lambda$ mit

$$\sup_{t\in[0,1]} |\lambda(t) - t| < \varepsilon$$

und

$$\sup_{t\in[0,1]} |f(t) - g\circ\lambda(t)| < \varepsilon$$

existiert. Unmittelbar aus (2.2.1) ersichtlich sind

$$d_S(f,g) \leqslant \|f - g\|_\infty \tag{2.2.2}$$

und

$$d_S(f,g) = d_S(-f,-g). \tag{2.2.3}$$

Für alle $\lambda \in \Lambda$ gilt

$$\|f - g\circ\lambda\|_\infty \geqslant |f(0) - g(0)| \vee |f(1) - g(1)|$$

und somit ist $|f(0) - g(0)| \vee |f(1) - g(1)|$ eine untere Schranke für $d_S(f,g)$.
Satz 2.2.2. *Die Skorokhodmetrik ist eine Metrik auf $\mathbb{D}$.*

Beweis. Die Positivität von d_S ist offensichtlich. Die Endlichkeit ist an der Ungleichung (2.2.2) erkennbar, da die Identitätsfunktion in Λ liegt und die Differenz zweier càdlàg-Funktion wieder càdlàg und als solche nach Korollar 2.1.3(i) beschränkt ist. $d_S(f,g) = 0$ impliziert die Existenz einer Folge $(\lambda_n, n \in \mathbb{N})$ in Λ derart, dass λ_n gleichmäßig gegen die Identitätsfunktion und $g\circ\lambda_n$ gleichmäßig gegen f konvergiert. An den Stetigkeitsstellen $t \in (0,1)$ von g stimmen f und g wegen $g(t) = g(\lim_n \lambda_n(t)) = \lim_n g\circ\lambda_n(t) = f(t)$ überein. Ist g jedoch nicht stetig in t, muss die reelle Folge $(\lambda_n(t), n \in \mathbb{N})$ entweder von unten oder von oben gegen t konvergieren. Um dies einzusehen,

nehme man zunächst an, dass die Folge unendlich oft zwischen $[0,t)$ und $[t,1]$ oszilliert. Dann existieren für alle $n \in \mathbb{N}$ ganze Zahlen $n_1, n_2 \in \mathbb{N}$ mit $n_i \geqslant n$ und $\lambda_{n_1}(t) < t$ und $\lambda_{n_2}(t) \geqslant t$. Da $\lambda_n(t) \to_n t$, gibt es (ebenfalls gegen t) konvergierende Teilfolgen $(\lambda_{n_u}(t), u \in \mathbb{N})$, $(\lambda_{n_v}(t), v \in \mathbb{N})$, welche respektive in $[0,t)$ bzw. $[t,1]$ liegen. Folglich gilt $g \circ \lambda_{n_u}(t) \to_u g(t-)$ und $g \circ \lambda_{n_v}(t) \to_v g(t)$. Wegen $g(t-) \neq g(t)$ steht dies im Widerspruch zu $g \circ \lambda_n(t) \to_n f(t)$. Für jede Unstetigkeitsstelle t von g muss daher entweder $f(t) = g(t-)$ oder $f(t) = g(t)$ gelten. Zusammen mit der geforderten Rechtsstetigkeit von f müssen die beiden Funktionen auf $[0,1]$ übereinstimmen. Für alle $f, g, h \in \mathbb{D}, \lambda_1, \lambda_2 \in \Lambda$ ist

$$\begin{aligned} \|f - g \circ \lambda_1 \circ \lambda_2\|_\infty &= \left\|f \circ \lambda_2^{-1} - g \circ \lambda_1\right\|_\infty \\ &\leqslant \left\|f \circ \lambda_2^{-1} - h\right\|_\infty + \|h - g \circ \lambda_1\|_\infty \\ &\leqslant \|f - h \circ \lambda_2\|_\infty + \|h - g \circ \lambda_1\|_\infty \end{aligned} \tag{2.2.4}$$

sowie

$$\|\lambda_1 \circ \lambda_2 - \mathrm{Id}\|_\infty \leqslant \|\lambda_1 - \mathrm{Id}\|_\infty + \|\lambda_2 - \mathrm{Id}\|_\infty .$$

Da die Komposition von λ_1 und λ_2 wieder in Λ liegt, folgt

$$\begin{aligned} d_S(f,g) &\leqslant \inf_{\lambda \in \Lambda} \{\|\lambda - \mathrm{Id}\|_\infty + \|f - g \circ \lambda\|_\infty\} \\ &= \inf_{\lambda_1, \lambda_2 \in \Lambda} \{\|\lambda_1 \circ \lambda_2 - \mathrm{Id}\|_\infty + \|f - g \circ \lambda_1 \circ \lambda_2\|_\infty\} \\ &\leqslant d_S(h,g) + d_S(f,h). \end{aligned}$$

Wegen (2.2.4) und $\|\lambda - \mathrm{Id}\|_\infty = \|\lambda^{-1} - \mathrm{Id}\|_\infty$ folgt die Symmetrie von d_S aus der Beobachtung, dass mit jedem $\lambda \in \Lambda$ auch deren Umkehrfunktion wieder in Λ liegt. □

Lemma 2.2.3. *Eine Folge von Funktionen $(f_n, n \in \mathbb{N})$ in $\mathbb{D}$ konvergiert genau dann gegen $f \in \mathbb{D}$ im Sinne der Skorokhodmetrik, wenn in Λ eine Folge $(\lambda_n, n \in \mathbb{N})$ derart existiert, dass $(f_n \circ \lambda_n, n \in \mathbb{N})$ gleichmäßig gegen f und $(\lambda_n, n \in \mathbb{N})$ gleichmäßig gegen Id konvergieren.*

Geht man über zur Folge der Umkehrfunktionen $(\lambda_n^{-1}, n \in \mathbb{N}) \subset \Lambda$, so wird klar, dass die Aussage in obigem Lemma äquivalent zur Existenz einer weiteren Folge $(\gamma_n, n \in \mathbb{N})$ in Λ ist, für die $\|f_n - f \circ \gamma_n\|_\infty \to_n 0$ und $\|\gamma_n - \mathrm{Id}\|_\infty \to_n 0$ gilt. Wegen

$$|f_n(t) - f(t)| \leqslant |f_n(t) - f \circ \lambda_n(t)| + |f \circ \lambda_n(t) - f(t)|$$

impliziert die Konvergenz im Sinne der Skorokhodmetrik die punktweise Konvergenz an allen Stetigkeitsstellen der Grenzfunktion sowie an den Rändern des Intervalls $[0,1]$.

Satz 2.2.4. *Die Skorokhod-Metrik ist nicht vollständig.*

Beweis. Zum Beweis der Behauptung müssen wir eine Folge im metrischen Raum $(\mathbb{D}, d_S)$ konstruieren, welche Cauchy ist, aber nicht in $\mathbb{D}$ konvergiert. Für $n, m \in \mathbb{N}$ definieren wir die Funktion $f_n : [0,1] \to \mathbb{R}$ durch $f_n = \mathbb{1}_{[0,1/2^n)}$ sowie eine weitere Funktion $\lambda_n \in \Lambda$, welche auf den Intervallen $[0, 1/2^n]$ und $[1/2^n, 1]$ affin linear ist und $\lambda_n(1/2^n) = 1/2^m$ erfüllt. Für jedes $n, m \in \mathbb{N}$ gilt dann $f_n = f_m \circ \lambda_n$ und

$$\begin{aligned} d_S(f_n, f_m) &\leqslant \|f_n - f_m \circ \lambda_n\|_\infty \vee \|\lambda_n - \mathrm{Id}\|_\infty \\ &= \left| \frac{1}{2^m} - \frac{1}{2^n} \right| \\ &< 2^{-(n \wedge m)}. \end{aligned}$$

Folglich existiert für jedes $\varepsilon > 0$ ein $n_0 \in \mathbb{N}$ derart, dass $d_S(f_n, f_m) < 2^{-(n \wedge m)} \leqslant 2^{-n_0} \leqslant \varepsilon$ für alle $n, m \geqslant n_0$ erfüllt ist. Andererseits konvergiert die Folge von Funktionen (f_n) auf dem halboffenen Intervall $(0, 1]$ punktweise gegen 0. Falls also eine Grenzfunktion $f \in \mathbb{D}$ existiert, gegen die (f_n) im Sinne der Skorokhodmetrik konvergiert, so muss f wegen der angestrebten Rechtsstetigkeit gleich der Nullfunktion sein. Aber für jedes $n \in \mathbb{N}$ ist $d_S(f_n, 0) \geqslant \|f_n\|_\infty \geqslant 1$. □

Es existiert eine Metrik, welche dieselbe Topologie erzeugt, aber vollständig ist (siehe [3] S. 125). Da diese jedoch deutlich umständlicher auszuwerten ist, konzentrieren wir uns auf d_S aus Definition 2.2.1.

Schließlich wollen wir noch ein einfaches Beispiel dafür angeben, dass die Skorokhodmetrik tatsächlich strikt kleiner als die von $\|\cdot\|_\infty$ induzierte Metrik sein kann. Sei hierzu $t_0 \in (0, 1/3)$ und die Funktionen $f, g \in \mathbb{D}$ definiert durch

$$f(t) = \begin{cases} 0, & t < t_0, \\ -1 + t/t_0, & t_0 \leqslant t < 2t_0, \\ 1, & 2t_0 \geqslant t \end{cases}$$

und

$$g(t) = \begin{cases} 0, & t < 2t_0, \\ -2 + t/t_0, & 2t_0 \leqslant t < 3t_0, \\ 1, & 3t_0 \geqslant t. \end{cases}$$

Damit ist $\|f-g\|_\infty = f(2t_0) - g(2t_0) = 1$. Weiter sei $\lambda \in \Lambda$ gegeben durch

$$\lambda(t) = \begin{cases} 2t, & t < t_0, \\ t_0 + t, & t_0 \leqslant t < 2t_0, \\ t_0/(1-2t_0) + (1-3t_0)/(1-2t_0), & 2t_0 \geqslant t. \end{cases}$$

Da sowohl f als auch $g \circ \lambda$ auf den Intervallen $[0, t_0)$ und $[2t_0, 1]$ konstant sind, hat die Wahl von $\lambda \in \Lambda$ auf diesen Intervallen keinerlei Auswirkungen auf $\|f - g \circ \lambda\|_\infty$. Wählt man der Einfachheit halber die lineare Interpolation, so gilt einerseits

$$\|\lambda - \mathrm{Id}\|_\infty = \sup_{t\in[0,1]} |\lambda(t) - t| = \max_{i=1,2}(\lambda(it_0) - it_0) = t_0,$$

andererseits verschwindet $\|f - g \circ \lambda\|_\infty$, da f auf $[t_0, 2t_0)$ mit $g \circ \lambda$ übereinstimmt und wir erhalten

$$d_S(f,g) \leqslant \|\lambda - \mathrm{Id}\|_\infty \vee \|f - g \circ \lambda\|_\infty = t_0 < 1.$$

2.3 Skorokhodmetrik für stückweise konstante Funktionen

Im folgenden Abschnitt wollen wir eine Formel für den Skorokhodabstand zweier Funktionen aus einer dichten Teilmenge von $\mathbb{D}$ herleiten. Für eine Zahl $a \in \mathbb{R}$ definieren wir $a^+ = \max(a, 0)$ und $a^- = -\min(a, 0)$. a^+ und a^- heißen *Positivteil* und *Negativteil* von a. Für die folgenden Berechnungen treffen wir die Konvention, dass das Maximum über eine leere Menge gleich 0 ist. Es sei daran erinnert, dass $\mathbb{D}_c$ aller stückweise konstanten, rechtsstetigen Funktionen auf $[0, 1]$ enthält. Der folgende Satz liefert eine Formel zur Auswertung der Skorokhodmetrik in $\mathbb{D}_c$.

Satz 2.3.1. *Seien $f, g \in \mathbb{D}_c$ mit den Unstetigkeitsstellen $0 < s_1^f < s_2^f < \cdots < s_{N_f}^f < 1$, $N_f \geqslant 1$, bzw. $0 < s_1^g < s_2^g < \cdots < s_{N_g}^g < 1$, $N_g \geqslant 1$, mit den Konventionen $s_0^f = s_0^g = 0$ und $s_{N_f+1}^f = s_{N_g+1}^g = 1$. Dann ist der Abstand der beiden Funktionen im Sinne der Skorokhodmetrik aus Definition 2.2.1*

$$d_S(f,g) = \min_{k\in\mathcal{K}} \Bigg[\max_{k_{N_f} \leqslant j \leqslant N_g} d_{N_f,j} \vee \max_{i\in\complement Q_k} \left(\Delta s^-_{i,k_i} \vee \max_{k_{i-1} \leqslant j < k_i} d_{i-1,j} \right) \vee \min_{\substack{R\in\mathcal{P}(\{1,\ldots,N_f\}),\\ R\supset Q_k}} \left\{ \max_{i\in R} \left(d_{i-1,k_i} \vee \Delta s^+_{i,k_i+1} \right) \vee \max_{i\in\complement R} \Delta s^+_{i,k_i} \right\} \Bigg]. \tag{2.3.1}$$

Hierbei sei

$$\mathcal{K} = \{k \in \{0, \dots, N_g\}^{N_f} \mid k_1 \leqslant k_2 \leqslant \dots \leqslant k_{N_f}\} \tag{2.3.2}$$

mit den Konventionen $k_0 = 0$ *und* $k_{N_f+1} = N_g + 1$ *und für* $k \in \mathcal{K}$ *sei die Menge* Q_k *gegeben durch*

$$Q_k = \{1 \leqslant i \leqslant N_f \mid k_{i-1} = k_i\}. \tag{2.3.3}$$

Des Weiteren seien für $i \in \{0, \dots, N_f\}, j \in \{0, \dots, N_g\}$

$$\Delta s_{i,j} = s_i^f - s_j^g \tag{2.3.4}$$

sowie

$$d_{i,j} = \left|f(s_i^f) - g(s_j^g)\right|. \tag{2.3.5}$$

Diesen Satz wollen wir nun beweisen.

Lemma 2.3.2. *Seien* f, g *wie in Satz 2.3.1. Dann gilt für die in* (2.2.1) *definierte Skorokhodmetrik*

$$d_S(f,g) = \inf_{x \in C}\left\{\left\|x - s^f\right\|_\infty \vee \max_{\substack{0 \leqslant i \leqslant N_f,\\ 0 \leqslant j \leqslant N_g}} 1_{\{s_j^g < x_{i+1}, s_{j+1}^g > x_i\}} |f(s_i^f) - g(s_j^g)|\right\}$$

mit $C = \{x \in (0,1)^{N_f} \mid x_1 < \dots < x_{N_f}\}$ *und den Konventionen* $x_0 = 0$ *und* $x_{N_f+1} = 1$ *für* $x \in C$.

Beweis. Die Intervalle $\lambda^{-1}([s_j^g, s_{j+1}^g)), j = 0, \dots, N_g$, bilden eine disjunkte Zerlegung von $[0,1)$, mit dessen Hilfe sich auch die Intervalle

$$\begin{aligned}
[s_i^f, s_{i+1}^f) &= [s_i^f, s_{i+1}^f) \cap [0,1) \\
&= [s_i^f, s_{i+1}^f) \cap \biguplus_{j=0}^{N_g} \lambda^{-1}([s_j^g, s_{j+1}^g)) \\
&= \biguplus_{j=0}^{N_g} ([s_i^f, s_{i+1}^f) \cap \lambda^{-1}([s_j^g, s_{j+1}^g)))
\end{aligned}$$

für $i = 0, \dots, N_f$ weiter zerlegen lassen. Wir haben nun hinreichend kleine, disjunkte Intervalle gefunden, auf denen sowohl f als auch $g \circ \lambda$ konstant sind. Der Schnitt von $[s_i^f, s_{i+1}^f)$ und $\lambda^{-1}([s_j^g, s_{j+1}^g))$ ist genau dann nichtleer,

wenn sowohl $s_j^g < \lambda(s_{i+1}^f)$ als auch $s_{j+1}^g > \lambda(s_i^f)$ ist. Somit ist für $\lambda \in \Lambda$

$$\begin{aligned}
\|f - g \circ \lambda\|_\infty &= \sup_{t\in[0,1)} |f(t) - g(\lambda(t))| \\
&= \max_{0\leqslant i\leqslant N_f} \sup_{t\in[s_i^f, s_{i+1}^f)} |f(t) - g(\lambda(t))| \\
&= \max_{0\leqslant i\leqslant N_f} \max_{0\leqslant j\leqslant N_g} 1_{\{[s_i^f, s_{i+1}^f)\cap\lambda^{-1}([s_j^g, s_{j+1}^g))\neq\varnothing\}} \\
&\qquad \sup_{t\in[s_i^f, s_{i+1}^f)\cap\lambda^{-1}([s_j^g, s_{j+1}^g))} |f(t) - g(\lambda(t))| \\
&= \max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s_j^g < \lambda(s_{i+1}^f), s_{j+1}^g > \lambda(s_i^f)\}} |f(s_i^f) - g(s_j^g)|, \qquad (2.3.6)
\end{aligned}$$

da für jedes j die Komposition $g \circ \lambda$ auf dem Intervall $\lambda^{-1}([s_j^g, s_{j+1}^g))$ den Wert $g(s_j^g)$ hat. Sei nun Λ_l die Menge aller stetigen und auf den Intervallen $[s_i^f, s_{i+1}^f), i = 0, \ldots, N_f$ affin linearen Funktionen $h : [0,1] \to [0,1]$ mit $h(0) = 0$ und $h(1) = 1$. Für vorgegebenes $\lambda \in \Lambda$ kann man leicht ein $\widetilde{\lambda} \in \Lambda_l$ konstruieren, welches

$$\widetilde{\lambda}(s_i^f) = \lambda(s_i^f), \qquad i = 0, \ldots, N_f + 1 \qquad (2.3.7)$$

erfüllt. In diesem Sinne können wir $\widetilde{\lambda}$ als lineare Approximation von λ auffassen. Jede solche Approximation aus Λ_l ist durch (2.3.7) eindeutig bestimmt. Aus Gleichung (2.3.6) ist ersichtlich, dass für die Norm $\|f - g \circ \lambda\|_\infty$ die Werte von λ außerhalb der Sprungstellen von f keine Rolle spielen. Die Norm ist also invariant gegenüber Veränderungen von λ auf den offenen Intervallen (s_i^f, s_{i+1}^f). Wir werden $\widetilde{\lambda}$ daher auf diesen Intervallen linear interpolieren. Darüber hinaus wird mit dieser linearen Interpolation der Abstand zur Identitätsfunktion minimiert, wie folgende Rechnung zeigt:

$$\begin{aligned}
\left\|\widetilde{\lambda} - \mathrm{Id}\right\|_\infty &= \sup_{t\in[0,1]} |\widetilde{\lambda}(t) - t| \\
&= \max_{1\leqslant i\leqslant N_f} |\widetilde{\lambda}(s_i^f) - s_i^f| \\
&= \max_{1\leqslant i\leqslant N_f} |\lambda(s_i^f) - s_i^f| \\
&\leqslant \sup_{t\in[0,1]} |\lambda(t) - t| \\
&= \|\lambda - \mathrm{Id}\|_\infty .
\end{aligned}$$

Nun ist klar, dass wir uns bei der Auswertung der Skorokhodmetrik für Funktionen aus $\mathbb{D}_c$ auf die Menge $\Lambda_l \subset \Lambda$ zurückziehen können und erhalten mit

Gleichung (2.3.6)

$$\begin{aligned} d_S(f,g) &= \inf_{\lambda\in\Lambda}\Big\{\|\lambda-\mathrm{Id}\|_\infty \vee \|f-g\circ\lambda\|_\infty\Big\} \\ &= \inf_{\lambda\in\Lambda_l}\Big\{\|\lambda-\mathrm{Id}\|_\infty \vee \|f-g\circ\lambda\|_\infty\Big\} \\ &= \inf_{\lambda\in\Lambda_l}\Big\{\max_{1\leqslant i\leqslant N_f}|\lambda(s_i^f)-s_i^f|\vee \\ &\qquad \max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s_j^g<\lambda(s_{i+1}^f),s_{j+1}^g>\lambda(s_i^f)\}}|f(s_i^f)-g(s_j^g)|\Big\} \\ &= \inf_{x\in C}\Big\{\max_{1\leqslant i\leqslant N_f}|x_i-s_i^f| \vee \max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s_j^g<x_{i+1},s_{j+1}^g>x_i\}}|f(s_i^f)-g(s_j^g)|\Big\}, \end{aligned} \tag{2.3.8}$$

wobei die Menge C definiert ist durch

$$C=\{x\in(0,1)^{N_f}\mid x_1<\ldots<x_{N_f}\}$$

mit den Konventionen $x_0=\lambda(s_0^f)=\lambda(0)=0$ und $x_{N_f+1}=\lambda(s_{N_f+1}^f)=\lambda(1)=1$. □

Lemma 2.3.2 besagt, dass man das Infimum nicht mehr über einen Funktionenraum, sondern nur noch über eine Menge von Vektoren bilden muss, die allesamt im offenen N_f-dimensionalen Einheitsquader liegen. Für $k\in\{0,\ldots,N_g\}^{N_f}$ sei B_k gegeben durch die Menge

$$B_k=\bigtimes_{i=1}^{N_f}[s_{k_i}^g,s_{k_i+1}^g).$$

Für die weiteren Rechnungen werden wir für $k\in\{0,\ldots,N_g\}^{N_f}$ die Konventionen $k_0=0$ und $k_{N_f+1}=N_g+1$ vereinbaren.

Lemma 2.3.3. *Seien $x\in B_k$, $k\in\{0,\ldots,N_g\}^{N_f}$. Dann gilt für alle $i\in\{0,\ldots,N_f\}$, $j\in\{0,\ldots,N_g\}$:*

(i) $x_i<s_{j+1}^g$ genau dann, wenn $j\geqslant k_i$;

(ii) $x_{i+1}>s_j^g$ genau dann, wenn $j<k_{i+1}\vee(j=k_{i+1}\wedge x_{i+1}\neq s_{k_{i+1}}^g)$.

Beweis. Für $i=0$ sind beide Seiten der Äquivalenz in Lemma 2.3.3(i) trivialerweise erfüllt. Für strikt positive i ist nach Voraussetzung $x_i<s_{k_i+1}^g\leqslant s_{j+1}^g$, falls $j\geqslant k_i$ und $x_i\geqslant s_{k_i}^g\geqslant s_{j+1}^g$ andernfalls. Für $i=N_f$ sind beide Seiten der Äquivalenz in Lemma 2.3.3(ii) trivialerweise erfüllt. Für $i<N_f$ ist nach

Voraussetzung $x_{i+1} < s^g_{k_{i+1}+1} \leqslant s^g_j$, falls $j > k_{i+1}$ und $x_{i+1} \geqslant s^g_{k_{i+1}} > s^g_j$, falls $j < k_{i+1}$. Ist $i = k_{i+1}$, so folgt $x_{i+1} \geqslant s^g_{k_{i+1}} = s^g_j$. □

Mit Lemma 2.3.3 folgt sofort, dass $(s^g_j < x_{i+1}) \wedge (s^g_{j+1} > x_i)$ zu

$$(k_i < k_{i+1} \wedge k_i \leqslant j < k_{i+1}) \vee (j = k_{i+1} \wedge x_{i+1} \neq s^g_{k_{i+1}})$$

äquivalent ist und wir erhalten

$$\max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s^g_j<x_{i+1},s^g_{j+1}>x_i\}} d_{i,j} = \max_{\substack{1\leqslant i\leqslant N_f+1,\ k_{i-1}\leqslant j<k_i\\ k_{i-1}<k_i}} d_{i-1,j} \vee \max_{\substack{1\leqslant i\leqslant N_f,\\ x_i\neq s^g_{k_i}}} d_{i-1,k_i}, \quad (2.3.9)$$

wobei $d_{i,j}$ wie in (2.3.5) definiert ist. Man erkennt leicht, dass für ein vorgegebenes $k \in \{0, \ldots, N_g\}^{N_f}$ die Menge $B_k \cap C$ genau dann nichtleer ist, wenn $k_1 \leqslant k_2 \leqslant \ldots \leqslant k_{N_f}$ gilt. Sei daher $\mathcal{K}$ definiert wie in (2.3.2). Dann ist wegen der Gleichungen (2.3.8) und (2.3.9)

$$\begin{aligned}
&d_S(f,g)\\
&= \inf_{x\in C}\Big\{\big\|x - s^f\big\|_\infty \vee \max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s^g_j<x_{i+1},s^g_{j+1}>x_i\}} d_{i,j}\Big\}\\
&= \min_{k\in\mathcal{K}} \inf_{x\in B_k\cap C}\Big\{\big\|x - s^f\big\|_\infty \vee \max_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}} 1_{\{s^g_j<x_{i+1},s^g_{j+1}>x_i\}} d_{i,j}\Big\}\\
&= \min_{k\in\mathcal{K}}\Bigg(\inf_{x\in B_k\cap C}\Big\{\big\|x - s^f\big\|_\infty \vee \max_{\substack{1\leqslant i\leqslant N_f,\\ x_i\neq s^g_{k_i}}} d_{i-1,k_i}\Big\} \vee \max_{\substack{1\leqslant i\leqslant N_f+1,\ k_{i-1}\leqslant j<k_i\\ k_{i-1}<k_i}} d_{i-1,j}\Bigg).
\end{aligned} \quad (2.3.10)$$

Man sieht, dass die Zerlegung von C in $B_k \cap C$ noch nicht hinreichend fein ist. Um diese weiter zu verfeinern, definieren wir für $R \in \mathcal{P}(\{1, \ldots, N_f\})$

$$G^R_k = \bigtimes_{i=1}^{N_f} J^R_i \quad (2.3.11)$$

mit

$$J^R_i = \begin{cases} (s^g_{k_i}, s^g_{k_i+1}), & i \in R,\\ \{s^g_{k_i}\}, & \text{sonst,}\end{cases}$$

$i = 1, \ldots, N_f$.

Lemma 2.3.4. *Sei $k \in \mathcal{K}, R \in \mathcal{P}(\{1, \ldots, N_f\}), Q_k$ wie in* (2.3.3) *und G^R_k wie in* (2.3.11)*, dann ist*

$$G^R_k \cap \bigcap_{i=1}^{N_f}\{x_{i-1} < x_i\} = \begin{cases} G^R_k \cap \bigcap\limits_{i\in Q_k\cap(R+1)} \{x_{i-1} < x_i\}, & R \supset Q_k,\\ \varnothing, & \textit{sonst,}\end{cases}$$

mit der Konvention, dass der Schnitt über die leere Menge gleich der Grundmenge C sei.

Beweis. Falls R nicht Obermenge von Q_k ist, muss ein $i \in Q_k$ derart existieren, dass i nicht in R liegt. Somit ist $x_{i-1} \geqslant s^g_{k_{i-1}} = s^g_{k_i} = x_i$ für alle $x \in G^R_k$. Wenden wir uns nun dem Fall $R \supset Q_k$ zu und geben uns ein $i \in \{1, \dots, N_f\}$ vor. $i \notin Q_k$ zieht wegen $x_{i-1} < s^g_{k_{i-1}+1} \leqslant s^g_{k_i} \leqslant x_i$ für alle $x \in G^R_k$ sofort $G^R_k \subset \{x_{i-1} < x_i\}$ nach sich. Falls $i \in Q_k \backslash (R+1)$, haben wir $x_{i-1} = s^g_{k_{i-1}} = s^g_{k_i} < x_i$ für alle $x \in G^R_k$, was ebenfalls $G^R_k \subset \{x_{i-1} < x_i\}$ impliziert. □

Da $x_{N_f} < s^g_{k_{N_f}+1} \leqslant s^g_{N_g+1} = 1$ für jedes $x \in B_k$ gilt, folgt mit Lemma 2.3.4

$$\begin{aligned} B_k \cap C &= B_k \cap \bigcap_{i=1}^{N_f} \{x_{i-1} < x_i\} \\ &= \biguplus_{R \in \mathcal{P}(\{1,\dots,N_f\})} \left(G^R_k \cap \bigcap_{i=1}^{N_f} \{x_{i-1} < x_i\} \right) \\ &= \biguplus_{\substack{R \in \mathcal{P}(\{1,\dots,N_f\}),\\ R \supset Q_k}} \left(G^R_k \cap \bigcap_{i \in Q_k \cap (R+1)} \{x_{i-1} < x_i\} \right). \end{aligned}$$

Nun definieren wir die Menge

$$U = G^R_k \cap \bigcap_{i \in Q_k \cap (R+1)} \{x_{i-1} < x_i\} \tag{2.3.12}$$

und schreiben für den Vektor $(s^f_1, s^f_2, \dots, s^f_{N_f})$ kurz s^f. Das folgende Lemma zeigt, dass z^R_k die Funktion $v \mapsto \left\| v - s^f \right\|_\infty$ auf U minimiert.

Lemma 2.3.5. *Sei $k \in \mathcal{K}$, Q_k wie in (2.3.3), $R \in \mathcal{P}(\{1, \dots, N_f\})$ mit $R \supset Q_k$, G^R_k wie in (2.3.11) und U wie in (2.3.12). Dann ist*

$$\inf_{x \in U} \left\| x - s^f \right\|_\infty = \left\| z^R_k - s^f \right\|_\infty,$$

wobei z^R_k der Vektor sei, dessen i-te Komponente durch

$$\left(z^R_k\right)_i = \begin{cases} s^g_{k_i}, & i \notin R, \\ s^g_{k_i}, & i \in R, s^f_i < s^g_{k_i}, \\ s^f_i, & i \in R, s^f_i \in [s^g_{k_i}, s^g_{k_i+1}], \\ s^g_{k_i+1}, & i \in R, s^f_i > s^g_{k_i+1}, \end{cases}$$

$i = 1, \dots, N_f$, gegeben ist.

Beweis. Es ist leicht nachzuweisen, dass für alle $i \in \{1, \dots, N_f\}, y \in J_i^R$ die Ungleichung

$$\left|(z_k^R)_i - s_i^f\right| \leqslant \left|y - s_i^f\right|$$

erfüllt ist. Somit folgt

$$\left\|z_k^R - s^f\right\|_\infty \leqslant \left\|x - s^f\right\|_\infty$$

für alle $x \in G_k^R$. Wegen der Stetigkeit der Norm müssen wir lediglich zeigen, dass z_k^R im Abschluss von U liegt. Offensichtlich gilt $z_k^R \in \overline{G_k^R} = \times_{i=1}^{N_f} \overline{J_i^R}$ sowie

$$\overline{U} = \overline{G_k^R} \cap \bigcap_{i \in Q_k \cap (R+1)} \{x_{i-1} \leqslant x_i\}.$$

Falls $Q_k \cap (R+1)$ nichtleer ist, so sieht man mittels Fallunterscheidung, dass wegen $R \supset Q_k$ die Ungleichung $(z_k^R)_i \leqslant (z_k^R)_{i+1}$ für alle $i \in Q_k \cap (R+1)$ erfüllt ist. Die Behauptung folgt mit der Stetigkeit der Norm. □

Mit (2.3.4) erhalten wir für alle $i \in \{1, \dots, N_f\}$

$$\begin{aligned}\left|(z_k^R)_i - s_i^f\right| &= \begin{cases} \left|s_{k_i}^g - s_i^f\right|, & i \notin R, \\ s_{k_i}^g - s_i^f, & i \in R, s_i^f < s_{k_i}^g, \\ s_i^f - s_{k_i+1}^g, & i \in R, s_i^f > s_{k_i+1}^g, \\ 0, & sonst, \end{cases} \\ &= \begin{cases} \left|\Delta s_{i,k_i}\right|, & i \notin R, \\ \Delta s_{i,k_i}^- \vee \Delta s_{i,k_i+1}^+, & i \in R \end{cases}\end{aligned}$$

und somit

$$\begin{aligned}\left\|z_k^R - s^f\right\|_\infty &= \max_{i \in \complement R} |\Delta s_{i,k_i}| \vee \max_{i \in R} \left(\Delta s_{i,k_i}^- \vee \Delta s_{i,k_i+1}^+\right) \\ &= \max_{i \in \complement R} \Delta s_{i,k_i}^+ \vee \max_{i \complement Q_k} \Delta s_{i,k_i}^- \vee \max_{i \in R} \Delta s_{i,k_i+1}^+.\end{aligned}$$

Es bleibt

$$\max_{1 \leqslant i \leqslant N_f} \Delta s_{i,k_i}^- = \max_{i \in \complement Q_k} \Delta s_{i,k_i}^- \tag{2.3.13}$$

zu zeigen. Wir nehmen an, es existiere ein $m \in Q_k$ mit $\Delta s_{m,k_m}^- > 0$. Andernfalls ist (2.3.13) trivialerweise erfüllt. Dann muss aber $k_m > k_0 = 0$ sein und somit ein $r \in \complement Q_k$ mit $r < m$ existieren. Wenn wir

$$l = \max\{r \in \complement Q_k \mid r < m\}$$

setzen, gilt $k_l = k_m$ und wir erhalten $\Delta s^-_{m,k_m} < \Delta s^-_{l,k_l}$, womit (2.3.13) nachgewiesen ist.

Da $x_i = s^g_{k_i}$ für alle $x \in G^R_k, i \in \complement R$ gilt, folgt aus Gleichung (2.3.10) zusammen mit den Lemmata 2.3.4 sowie 2.3.5

$$\begin{aligned}
&\inf_{x\in B_k\cap C}\left\{\left\|x-s^f\right\|_\infty \vee \max_{\substack{1\leqslant i\leqslant N_f,\\ x_i\neq s^g_{k_i}}} d_{i-1,k_i}\right\}\\
&= \min_{\substack{R\in\mathcal{P}(\{1,\dots,N_f\}),\\ R\supset Q_k}} \inf_{x\in U}\left\{\left\|x-s^f\right\|_\infty \vee \max_{\substack{1\leqslant i\leqslant N_f,\\ x_i\neq s^g_{k_i}}} d_{i-1,k_i}\right\}\\
&= \min_{\substack{R\in\mathcal{P}(\{1,\dots,N_f\}),\\ R\supset Q_k}} \left\{\max_{i\in R} d_{i-1,k_i} \vee \inf_{x\in U}\left\|x-s^f\right\|_\infty\right\}\\
&= \max_{i\in\complement Q_k}\Delta s^-_{i,k_i} \vee \min_{\substack{R\in\mathcal{P}(\{1,\dots,N_f\}),\\ R\supset Q_k}} \left\{\max_{i\in R} d_{i-1,k_i} \vee \max_{i\in\complement R}\Delta s^+_{i,k_i} \vee \max_{i\in R}\Delta s^+_{i,k_i+1}\right\}\\
&= \max_{i\in\complement Q_k}\Delta s^-_{i,k_i} \vee \min_{\substack{R\in\mathcal{P}(\{1,\dots,N_f\}),\\ R\supset Q_k}} \left\{\max_{i\in R}\left(d_{i-1,k_i}\vee\Delta s^+_{i,k_i+1}\right) \vee \max_{i\in\complement R}\Delta s^+_{i,k_i}\right\}.
\end{aligned}$$

Damit ist der Abstand von f und g im Sinne der Skorokhodmetrik gleich

$$\begin{aligned}
&d_S(f,g)\\
&= \min_{k\in\mathcal{K}}\left(\inf_{x\in B_k\cap C}\left\{\left\|x-s^f\right\|_\infty \vee \max_{\substack{1\leqslant i\leqslant N_f,\\ x_i\neq s^g_{k_i}}} d_{i-1,k_i}\right\} \vee \max_{\substack{1\leqslant i\leqslant N_f+1,\ k_{i-1}\leqslant j<k_i\\ k_{i-1}<k_i}} d_{i-1,j}\right)\\
&= \min_{k\in\mathcal{K}}\Bigg[\max_{k_{N_f}\leqslant j\leqslant N_g} d_{N_f,j} \vee \max_{i\in\complement Q_k}\left(\Delta s^-_{i,k_i} \vee \max_{k_{i-1}\leqslant j<k_i} d_{i-1,j}\right)\\
&\qquad\vee \min_{\substack{R\in\mathcal{P}(\{1,\dots,N_f\}),\\ R\supset Q_k}}\left\{\max_{i\in R}\left(d_{i-1,k_i}\vee\Delta s^+_{i,k_i+1}\right) \vee \max_{i\in\complement R}\Delta s^+_{i,k_i}\right\}\Bigg],
\end{aligned}$$

womit Satz 2.3.1 bewiesen ist.

Aus der Formel in (2.3.1) können wir zum einen

$$d_S(f,g) \geqslant d_{0,0} \vee d_{N_f,N_g} \tag{2.3.14}$$

ableiten, zum anderen wissen wir, dass für $f, g \in \mathbb{D}_c$ der Wert $d_S(f,g)$ in $\mathcal{E}'$ mit

$$\mathcal{E}' = \bigcup_{\substack{0\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}}\{d_{i,j}\} \cup \bigcup_{\substack{1\leqslant i\leqslant N_f,\\ 1\leqslant j\leqslant N_g}}\{\Delta s^-_{i,j}\} \cup \bigcup_{\substack{1\leqslant i\leqslant N_f,\\ 0\leqslant j\leqslant N_g}}\{\Delta s^+_{i,j}\}$$

liegen muss. Hierbei ist zu berücksichtigen, dass $\Delta s^-_{i,0} = \Delta s^+_{i,N_g+1} = 0$ für alle $1 \leqslant i \leqslant N_f$ gilt und $d_S(f,g) = 0$ wegen (2.3.14) nur dann erfüllt ist, wenn auch $d_{0,0} = d_{N_f,N_g} = 0$ gilt. Die Menge aller möglichen Ergebnisse $\mathcal{E}'$ lässt sich jedoch wegen (2.3.14) noch weiter auf

$$\mathcal{E} = \left\{ r \in \mathcal{E}' \;\middle|\; r \geqslant d_{0,0} \vee d_{N_f,N_g} \right\} \tag{2.3.15}$$

einschränken.

Im Fall $N_f = N_g = 1$ hat (2.3.1) die Gestalt

$$\begin{aligned} d_S(f,g) = \; & \big(d_{0,0} \vee d_{1,0} \vee d_{1,1} \vee \Delta s^+_{1,1}\big) \\ & \wedge \big(d_{0,0} \vee d_{1,1} \vee |\Delta s_{1,1}|\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{1,1} \vee \Delta s^-_{1,1}\big), \end{aligned}$$

für $N_f = N_g = 2$ jedoch bereits

$$\begin{aligned} d_S(f,g) = \; & \big(d_{0,0} \vee d_{1,0} \vee d_{2,0} \vee d_{2,1} \vee d_{2,2} \vee \Delta s^+_{1,1} \vee \Delta s^+_{2,1}\big) \\ & \wedge \big(d_{0,0} \vee d_{1,0} \vee d_{2,1} \vee d_{2,2} \vee \Delta s^+_{1,1} \vee |\Delta s_{2,1}|\big) \\ & \wedge \big(d_{0,0} \vee d_{1,0} \vee d_{1,1} \vee d_{2,1} \vee d_{2,2} \vee \Delta s^+_{1,1} \vee \Delta s^-_{2,1} \vee \Delta s^+_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{1,0} \vee d_{1,1} \vee d_{2,2} \vee \Delta s^+_{1,1} \vee |\Delta s_{2,2}|\big) \\ & \wedge \big(d_{0,0} \vee d_{1,0} \vee d_{1,1} \vee d_{1,2} \vee d_{2,2} \vee \Delta s^+_{1,1} \vee \Delta s^-_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{1,1} \vee d_{2,1} \vee d_{2,2} \vee |\Delta s_{1,1}| \vee \Delta s^-_{2,1} \vee \Delta s^+_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{1,1} \vee d_{2,1} \vee d_{2,2} \vee \Delta s^-_{1,1} \vee \Delta s^+_{1,2} \vee \Delta s^-_{2,1} \vee \Delta s^+_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{1,1} \vee d_{2,2} \vee |\Delta s_{1,1}| \vee |\Delta s_{2,2}|\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{1,1} \vee d_{2,2} \vee \Delta s^-_{1,1} \vee \Delta s^+_{1,2} \vee |\Delta s_{2,2}|\big) \\ & \wedge \big(d_{0,0} \vee d_{1,1} \vee d_{1,2} \vee d_{2,2} \vee |\Delta s_{1,1}| \vee \Delta s^-_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{1,1} \vee d_{1,2} \vee d_{2,2} \vee \Delta s^-_{1,1} \vee \Delta s^+_{1,2} \vee \Delta s^-_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{1,2} \vee d_{2,2} \vee |\Delta s_{1,2}| \vee \Delta s^-_{2,2}\big) \\ & \wedge \big(d_{0,0} \vee d_{0,1} \vee d_{0,2} \vee d_{1,2} \vee d_{2,2} \vee \Delta s^-_{1,2} \vee \Delta s^-_{2,2}\big). \end{aligned}$$

Hierbei ist zu beachten, dass $|\Delta s_{i,j}| = \Delta s^+_{i,j} \vee \Delta s^-_{i,j}$ gilt.

Kapitel 3

Auswertung der Skorokhodmetrik

3.1 Ein einfacher Algorithmus

Als Vorbetrachtung zur Herleitung eines Algorithmus zur Auswertung der Skorokhodmetrik stellen wir (2.3.1) um. Wir definieren für $M, N \in \mathbb{N}$ die Menge

$$\mathcal{K}_N^M = \{k \in \{0, \ldots, N\}^M \mid k_1 \leqslant k_2 \leqslant \ldots \leqslant k_M\}$$

mit den Konventionen $k_0 = 0$ und $k_{M+1} = N+1$ und für $k \in \mathcal{K}_N^M$ die Mengen

$$Q_k^M = \{1 \leqslant i \leqslant M \mid k_{i-1} = k_i\}$$

sowie

$$\mathcal{Z}_N^M = \left\{(k, R) \in \mathcal{K}_N^M \times \mathcal{P}_M \mid R \supset Q_k^M\right\}.$$

In Einklang mit der bisherigen Notation schreiben wir $\mathcal{K}$ für $\mathcal{K}_{N_g}^{N_f}$ und Q_k für $Q_k^{N_f}$. Des Weiteren kürzen wir $\mathcal{Z}_{N_g}^{N_f}$ mit $\mathcal{Z}$ ab und bezeichnen mit $\mathcal{P}_N$ die Potenzmenge der Zahlen von 1 bis N, $N \in \mathbb{N}$.

Seien $\mathcal{E}$ wie in (2.3.15) und die Funktion $\phi\colon \mathcal{Z} \to \mathcal{E}$ definiert durch

$$\begin{aligned}(k, R) \mapsto & \max_{k_{N_f} \leqslant j \leqslant N_g} d_{N_f, j} \vee \max_{i \in \complement Q_k} \left(\Delta s_{i,k_i}^- \vee \max_{k_{i-1} \leqslant j < k_i} d_{i-1,j}\right) \\ & \vee \max_{i \in R} \left(d_{i-1,k_i} \vee \Delta s_{i,k_i+1}^+\right) \vee \max_{i \in \complement R} \Delta s_{i,k_i}^+, \end{aligned} \tag{3.1.1}$$

dann hat (2.3.1) die Gestalt

$$d_S(f, g) = \min_{(k,R) \in \mathcal{Z}} \phi(k, R). \tag{3.1.2}$$

Da $|\mathcal{Z}|$ mit N_f und N_g exponentiell wächst, ist zu erwarten, dass man einen Auswertungsalgorithmus exponentieller Komplexität erhält.

Zur Berechnung des Skorokhodabstands in (2.3.1) müssen wir k über alle Elemente von $\mathcal{K}_N^M$ laufen lassen und für jedes k die Menge $R \in \mathcal{P}_N$ mit $R \supset Q_k^M$ variieren. Für eine effiziente Auswertung muss sichergestellt werden, dass kein Element mehrmals besucht wird. Wir wollen außerdem in der Lage sein, für jedes Element aus $\mathcal{K}_N^M$ bzw. $\mathcal{P}_N$ ein eindeutiges „nächstes“ Element zu berechnen. Somit benötigen wir auf den Mengen $\mathcal{K}_N^M$ und $\mathcal{P}_N$ sogenannte Totalordnungen. Eine Einführung in die Theorie der geordneten Mengen findet der Leser in Anhang A. Wir wollen nun versuchen, im Rahmen dieser Theorie einen Algorithmus herzuleiten, welcher dieses eindeutige nächste Element bezüglich einer vorher festgelegten Ordnung berechnet.

Seien $(I, <)$ eine endliche, totalgeordnete Menge, $(V_i, \leqslant_i)_{i\in I}$ eine Familie von abzählbaren, totalgeordneten Mengen und $V = \times_{i\in I} V_i$. Weiter nehmen wir an, dass $a, b, u \in V$ existieren mit $u < a$ und $u < b$. Dann ist $\{i \in I \mid u_i \neq a_i\} \neq \varnothing$ und wir bezeichnen das Minimum dieser Menge mit r. Falls $r < \min\{i \in I \mid u_i \neq b_i\}$, so muss $a_i = b_i = r$ für alle $i < r$ sowie $a_r > b_r = u_r$ gelten. Dann ist $r = \min\{i \in I \mid a_i \neq b_i\}$ und daher nach Definition A.2.2 $a > b$.

Aus dieser Beobachtung können wir umgehend ein Konstruktionsprinzip ableiten: Sei $b \in V$ vorgelegt. Falls ein $u \in V$ existiert mit $u > b$, so ist

$$a = \min\{u \in V \mid u > b\}$$

genau dann, wenn $a_q = \min\{j \in V_q \mid u > b\}$ und $a_r = b_r$ für alle $r < q$ wobei q das Maximum aller $i \in I$ bezüglich der Relation $\prec$ bezeichnet, für die ein $j \in V_i$ mit $j > b_i$ existiert. Damit haben wir gleichsam eine Anleitung für die explizite Konstruktion des nächstgrößeren Elements in einer totalgeordneten Menge.

Wir versehen $\mathcal{K}_N^M$ mit der lexikographischen Ordnung und erhalten zusammen mit der oben hergeleiteten Konstruktion umgehend einen Algorithmus zur Berechnung des nächstgrößeren Elements der Menge $\mathcal{K}_N^M$.

Lemma 3.1.1. *Sei $k \in \mathcal{K}_N^M$, $\widetilde{k}$ bezeichne den Vektor k vor der Ausführung von* `advance_k`(k, M, N). *Falls in $\mathcal{K}_N^M$ ein weiteres Element existiert, welches strikt größer als $\widetilde{k}$ ist, so gilt nach Ausführung von* `advance_k`(k, M, N)

$$k = \min\left\{l \in \mathcal{K}_N^M \middle| l > \widetilde{k}\right\}.$$

Andernfalls wird `stop` *auf wahr gesetzt.*

Algorithmus 3.1 Berechnung des nächstgrößeren Elementes von $\mathcal{K}_N^M$ im Sinne der lexikographischen Ordnung

```
1: procedure advance_k(k, M, N)
2:     q ← M
3:     while q > 0 and k_q = N do
4:         q ← q − 1
5:     end while
6:     if q > 0 then
7:         k_q ← k_q + 1
8:         k_i ← k_q,   i = q + 1, ..., M
9:     else
10:        stop ← true
11:    end if
12: end procedure
```

Beweis. Der Algorithmus setzt zunächst

$$q = \max\{i \in \{0, \ldots, N_f\} \mid k_i < N_g\}. \tag{3.1.3}$$

Wegen $0 = k_0 < N_g$ ist die Menge in (3.1.3) nichtleer und das Maximum existiert. q ist genau dann gleich 0, wenn $k = \max \mathcal{K}_N^M$ gilt. In diesem Fall wird die boolesche Variable `stop` auf wahr gesetzt. Andernfalls zählen wir k an dieser Position um 1 hoch und setzen alle weiteren Einträge auf die kleinstmöglichen Werte, so dass k wieder in $\mathcal{K}_N^M$ liegt. □

Durch leichte Modifikationen an Algorithmus 3.1 können wir eine (von der lexikographische Ordnung abgeleitete) Totalordnung auch auf einer endlichen Potenzmenge realisieren. Diese werden wir ebenfalls mit $<$ bezeichnen.

Sei $E = \{e_1, e_2, \ldots, e_M\} \subset \{1, \ldots, N\}, N \geqslant 1$ mit $e_1 < e_2 < \cdots < e_M, 0 \leqslant M \leqslant N$ die aktuelle Teilmenge. Der Algorithmus `advance_subset(`E, N`)` setzt E auf die nächste Teilmenge von $\{1, \ldots, N\}$, indem die Elemente von E ähnlich wie die Einträge der Vektoren aus $\mathcal{K}_N^M$ modifiziert werden.

Lemma 3.1.2. *Sei $E \in \mathcal{P}_N$, $\widetilde{E}$ bezeichne die Menge E vor der Ausführung des Algorithmus* `advance_subset`(E, N). *Falls in $\mathcal{P}_N$ ein weiteres Element existiert, welches strikt größer als $\widetilde{E}$ ist, so gilt nach Ausführung von* `advance_subset`(E, N)

$$E = \min\left\{W \in \mathcal{P}_N \middle| W > \widetilde{E}\right\}.$$

Andernfalls wird `stop` *auf wahr gesetzt.*

Algorithmus 3.2 Berechnung der nächsten Teilmenge von $\{1, \ldots, N\}$, $N \geqslant 1$

```
1:  procedure advance_subset(E, N)
2:      q ← |E|
3:      while q > 0 and e_q = N − |E| + q do
4:          q ← q − 1
5:      end while
6:      if q > 0 then
7:          e_q ← e_q + 1
8:          e_i ← e_{i−1} + 1,    i = q + 1, ..., |E|
9:      else
10:         if |E| < N then
11:             e_i ← i,    i = 1, ..., |E|
12:             M ← |E|
13:             E ← E ∪ {M + 1}
14:         else
15:             stop ← true
16:         end if
17:     end if
18: end procedure
```

Beweis. Zunächst bestimmen wir den größtmöglichen Index q mit $1 \leqslant q \leqslant M$, für den sich das Element e_q noch um mindestens 1 erhöhen lässt. Falls ein solcher Index existiert, gibt es eine weitere Teilmenge der Kardinalität M. e_q wird dann um 1 erhöht und allen Elementen mit höherem Index werden die kleinstmöglichen Werte zugewiesen. Falls kein solches e_q existiert, ist q gleich 0 und die vorliegende Teilmenge ist die letzte der Kardinalität M. Folglich starten wir für $M < N$ mit den Teilmengen der Größe $M + 1$ und setzen alle Elemente von E auf die kleinstmöglichen Werte. Ist jedoch $M = N$, so wurden alle Teilmengen der Kardinalität kleiner oder gleich N bereits durchlaufen und der Algorithmus bricht ab. □

Wollen wir nun unsere Herangehensweise auf die Potenzmenge einer beliebigen, endlichen Menge $A \subset \mathbb{N}$ ausdehnen, so können wir wie folgt vorgehen. Sei $A = \{a_1, a_2, \ldots, a_N\} \subset \mathbb{N}$ die Menge, von der alle Teilmengen gebildet werden sollen. Im Folgenden werden wir A als Grundmenge bezeichnen. Es ist unerheblich, ob die Darstellung von A durch ihre Elemente geordnet ist. Wir führen eine sogenannte Indexmenge $I = \{i_1, i_2, \ldots, i_M\} \subset \{1, \ldots, N\}$ mit $i_1 < i_2 < \cdots < i_M, 0 \leqslant M \leqslant N$ ein. Sie enthält die Indizes der Elemente der Grundmenge, welche sich in der aktuellen Teilmenge befinden. Statt der

Algorithmus 3.3 Einfacher Algorithmus zur Berechnung von $d_S(f,g)$ für $f,\, g \in \mathbb{D}_c$

```
 1: function distance(f, g)
 2:     stop_k ← false
 3:     stop_R ← false
 4:     u ← max E
 5:     k ← (0, ..., 0)
 6:     while ¬stop_k do
 7:         I ← ∅
 8:         C ← {1, ..., N_f}\Q_k
 9:         while ¬stop_R do
10:             R ← Q_k ⊎ {c_{i_1}, c_{i_2}, ...}
11:             if φ(k, R) < u then
12:                 u ← φ(k, R)
13:             end if
14:             advance_subset(I, |C|)
15:         end while
16:         advance_k(k, N_f, N_g)
17:     end while
18:     return u
19: end function
```

aktuellen Teilmenge E wird nun die Indexmenge hochgezählt. Die aktuelle Teilmenge ergibt sich dann als $E = \{a_{i_1}, a_{i_2}, \ldots, a_{i_M}\}$. Zum Berechnen der nächsten Teilmenge muss man `advance_subset`(I, N) aufrufen.

Eine weitere Verallgemeinerung ist an dieser Stelle möglich. Möchte man alle Teilmengen einer endlichen Menge $A \subset \mathbb{N}$ mit $A \supset B$ berechnen, so gehen wir vor wie im vorigen Abschnitt mit dem einzigen Unterschied, dass wir nun als Grundmenge $C = \{c_1, c_2, \ldots, c_L\} = A \backslash B$ mit $L = N - |B|$ statt A verwenden. Die Indexmenge $I = \{i_1, i_2, \ldots, i_M\} \subset \{1, \ldots, L\}$ mit $i_1 < i_2 < \cdots < i_M$, $0 \leqslant M \leqslant L$, bezieht sich nun nicht mehr auf die gesamte Grundmenge A, sondern lediglich auf C. Die aktuelle Teilmenge von A, welche B enthält, ist die disjunkte Vereinigung von B und der Menge $\{c_{i_1}, c_{i_2}, \ldots, c_{i_M}\}$. `advance_subset`$(I, L)$ setzt die Indexmenge auf das nächste Element.

Basierend auf den Algorithmen 3.1 und 3.2 erhalten wir einen einfachen Algorithmus zur Berechnung von $d_S(f,g)$ für $f,\, g \in \mathbb{D}_c$.

3.2 Ein Algorithmus mit polynomieller Laufzeit

Wir werden nun sehen, wie man die Auswertung der Skorokhodmetrik im Vergleich zum vorigen Abschnitt deutlich beschleunigen kann. Mit jeder Auswertung von ϕ gewinnen wir zusätzliche Informationen über das gesuchte Minimum, und zwar, dass es kleiner gleich dem aktuellen Minimum sein muss und welche Elemente künftig ausgeschlossen werden können.

Sei $\varnothing \neq \mathcal{Z}' \subset \mathcal{Z}$ beliebig, aber fest. Wir wollen

$$\min_{(k,R)\in\mathcal{Z}'} \phi(k,R) \tag{3.2.1}$$

auswerten. Seien ferner $(k',R') \in \mathcal{Z}', i \in \{0,\dots,N_f\}, j \in \{0,\dots,N_g\}$ vorgelegt und wir nehmen

$$\phi(k',R') \leqslant \Delta s^-_{i,j} \tag{3.2.2}$$

an. Dann gilt

$$\min_{(k,R)\in\mathcal{Z}'} \phi(k,R) \leqslant \phi(k',R') \leqslant \Delta s^-_{i,j} \tag{3.2.3}$$

und all jene $(k,R) \in \mathcal{Z}'$, für die $\Delta s^-_{i,j}$ in (3.1.1) vorkommt, erfüllen die Ungleichung $\phi(k,R) \geqslant \Delta s^-_{i,j}$. Solche Tupel leisten also keinen Beitrag zur Berechnung des Minimums in (3.2.1). Es liegt nun nahe, ϕ für solche Tupel gar nicht erst auszuwerten. Hierzu definieren wir für $m \in \{0,\dots,N_f\}, n \in \{0,\dots,N_g\}$ die Menge

$$\mathcal{G}^{\Delta s^-}_{m,n} = \{(k,R) \in \mathcal{Z} \mid m \in \complement Q_k \wedge n = k_m\}.$$

Wir können an (2.3.1) ablesen, dass $\Delta s^-_{m,n}$ genau dann in der Formel für $\phi(k,R)$ vorkommt, wenn (k,R) in $\mathcal{G}^{\Delta s^-}_{m,n}$ liegt. Man mache sich klar, dass dies im Allgemeinen nicht äquivalent zu $\phi(k,R) \geqslant \Delta s^-_{m,n}$ ist. Falls $\mathcal{Z}' \backslash \mathcal{G}^{\Delta s^-}_{i,j}$ nichtleer ist, haben wir

$$\min_{(k,R)\in\mathcal{Z}'} \phi(k,R) = \min_{(k,R)\in\mathcal{Z}'\backslash\mathcal{G}^{\Delta s^-}_{i,j}} \phi(k,R).$$

Andernfalls gibt es keine weiteren Tupel $(k,R) \in \mathcal{Z}'$, für die $\Delta s^-_{i,j}$ in (3.1.1) vorhanden ist und das Minimum in (3.2.1) ist gleich $\phi(k',R')$. Falls $\Delta s^-_{i,j} > 0$,

können wir wegen

$$\begin{aligned}\Delta s^-_{i,j} &= (s^f_i - s^g_j)^- \\ &= s^g_j - s^f_i \\ &\leqslant s^g_n - s^f_m \\ &= (s^f_m - s^g_n)^- \\ &= \Delta s^-_{m,n}\end{aligned}$$

für alle $m \leqslant i$ und $n \geqslant j$ sogar alle Tupel aus

$$\bigcup_{\substack{m\leqslant i,\\ n\geqslant j}} \mathcal{G}^{\Delta s^-}_{m,n} \tag{3.2.4}$$

ausschließen. Wir zeigen nun, dass die Elemente der Komplementärmenge von (3.2.4) und damit auch die Elemente der Menge selbst recht einfach charakterisierbar sind. Da $\Delta s^-_{i,j}$ strikt positiv ist, muss $j > 0$ gelten. Dann ist aber

$$\neg \bigvee_{\substack{m\leqslant i,\\ n\geqslant j}} (m \in \complement Q_k \wedge n = k_m)$$

äquivalent zu $k_1 < j \wedge \bigwedge_{m=2}^{i}(k_{m-1} = k_m \vee k_m < j)$, was sofort zu $k_i < j$ zusammenfällt und damit gilt

$$\mathcal{Z}' \setminus \bigcup_{\substack{m\leqslant i,\\ n\geqslant j}} \mathcal{G}^{\Delta s^-}_{m,n} = \{(k,R) \in \mathcal{Z}' \mid k_i < j\}.$$

Setzen wir $Y = \{(k,R) \in \mathcal{Z}' \mid k_i < j\}$, so erhalten wir

$$\min_{(k,R)\in\mathcal{Z}'} \phi(k,R) = \begin{cases} \phi(k',R'), & Y = \varnothing, \\ \min\limits_{(k,R)\in Y} \phi(k,R), & \text{sonst.} \end{cases} \tag{3.2.5}$$

Im Falle von $\Delta s^-_{i,j} = 0$ ist aufgrund von (3.2.3) auch der Ausdruck in (3.2.1) gleich 0. Dann gilt aber (3.2.5) erst recht.

Analog zu $\mathcal{G}^{\Delta s^-}_{m,n}$ definieren wir für $m \in \{0,\ldots,N_f\}, n \in \{0,\ldots,N_g\}$

$$\mathcal{G}^{\Delta s^+}_{m,n} = \{(k,R) \in \mathcal{Z} \mid (m \in R \wedge n = k_m + 1) \vee (m \notin R \wedge n = k_m)\}$$

sowie

$$\mathcal{G}^{d}_{m,n} = \{(k,R) \in \mathcal{Z} \mid (k_m \leqslant n < k_{m+1}) \vee (m+1 \in R \wedge n = k_m)\}.$$

Wir nehmen nun

$$\phi(k', R') \leqslant \Delta s^+_{i,j} \tag{3.2.6}$$

an. Falls $\Delta s^+_{i,j} > 0$, können wir uns wegen $\Delta s^+_{i,j} \leqslant \Delta s^+_{m,n}$ für alle $m \geqslant i$ und $n \leqslant j$ bei der Auswertung von (3.2.1) auf die Menge

$$\mathcal{Z}' \setminus \bigcup_{\substack{m \geqslant i,\\ n \leqslant j}} \mathcal{G}^{\Delta s^+}_{m,n}$$

zurückziehen. Die Aussage

$$\bigvee_{\substack{m \geqslant i,\\ n \leqslant j}} [(m \in R \wedge n = k_m + 1) \vee (m \notin R \wedge n = k_m)]$$

ist äquivalent zu

$$k_i < j \vee [k_i = j \wedge \bigvee_{m=i}^{N_f} (k_m = j \wedge m \notin R)]. \tag{3.2.7}$$

Setzen wir $q = \min\{r \mid k_r = j\}$, so muss wegen $k_i = j$ und $k \in \mathcal{K}$, die Gleichheit $k_m = j$ für alle $i \leqslant m \leqslant q$ gelten. Dann ist aber erst recht $\{i+1, \ldots, q\} \subset Q_k \subset R$. Somit ist der Ausdruck in der eckigen Klammer von (3.2.7) genau dann erfüllt, wenn $k_i = j \wedge i \notin R$ gilt und wir erhalten (3.2.5) mit $Y = \{(k, R) \in \mathcal{Z}' \mid (k_i \geqslant j) \wedge [(k_i = j) \Rightarrow (i \in R)]\}$.

Im Gegensatz zu (3.2.2) sowie (3.2.6) lassen sich aus der Kenntnis von

$$\phi(k', R') \leqslant d_{i,j}$$

lediglich diejenigen Tupel (k, R) ausschließen, in denen $d_{i,j}$ in der Formel für $\phi(k, R)$ vorkommt. Berücksichtigt man, dass $k_i = k_{i+1} = n$ sofort $i + 1 \in Q_k \subset R$ nach sich zieht, so kann man

$$\neg\left((k_i \leqslant j < k_{i+1}) \vee (i + 1 \in R \wedge j = k_i)\right)$$

zu

$$k_i \neq j \wedge (k_i < j \Rightarrow k_{i+1} \leqslant j) \wedge (k_{i+1} = j \Rightarrow i + 1 \notin R)$$

umformen. Das bisher Gezeigte fassen wir nun in folgendem Lemma zusammen.

Algorithmus 3.4 Erste Berechnung von $d_S(f,g)$ für $f,\, g \in \mathbb{D}_c$

```
1: Z' ← Z
2: v ← 1 + max E
3: while Z' ≠ ∅ do
4:     Setze (k, R) auf ein beliebiges Element in Z'
5:     u ← φ(k, R)
6:     for all (a, i, j) ∈ L with u ⩽ a_{i,j} < v do
7:         if a = Δs⁺ then
8:             Z' ← Z' ∩ ⋂_{m⩾i, n⩽j} ∁G^{Δs⁺}_{m,n}
9:         else if a = Δs⁻ then
10:            Z' ← Z' ∩ ⋂_{m⩽i, n⩾j} ∁G^{Δs⁻}_{m,n}
11:        else if a = d then
12:            Z' ← Z'\G^d_{m,n}
13:        end if
14:    end for
15:    v ← u
16: end while
17: return u
```

Lemma 3.2.1. *Seien* $f,\, g \in \mathbb{D}_c$, $(k', R') \in \mathcal{Z}' \subset \mathcal{Z}$, $i \in \{0, \ldots, N_f\}$, $j \in \{0, \ldots, N_g\}$.

(i) Falls $\phi(k', R') \leqslant \Delta s^-_{i,j}$, *gilt* (3.2.5) *mit*

$$Y = \{(k, R) \in \mathcal{Z}' \mid k_i < j\}.$$

(ii) Falls $\phi(k', R') \leqslant \Delta s^+_{i,j}$, *gilt* (3.2.5) *mit*

$$Y = \{(k, R) \in \mathcal{Z}' \mid k_i \geqslant j \wedge (k_i = j \Rightarrow i \in R)\}\,.$$

(iii) Falls $\phi(k', R') \leqslant d_{i,j}$, *gilt* (3.2.5) *mit*

$$Y = \{(k, R) \in \mathcal{Z}' \mid k_i \neq j \wedge (k_i < j \Rightarrow k_{i+1} \leqslant j) \\ \wedge (k_{i+1} = j \Rightarrow i + 1 \notin R)\}.$$

Die bisherigen Ausführungen zeigen, dass wir nicht nur den Wert, sondern auch die Bezeichner sowie die Indizes beachten müssen. Für einen ersten Algorithmus liegt es also nahe, die Elemente $\Delta s^-_{i,j}, \Delta s^+_{i,j}, d_{i,j}, i \in \{0, \ldots, N_f\}$,

$j \in \{0, \ldots, N_g\}$ zu abstrahieren, indem wir die symbolischen Bezeichner $\Delta s^-, \Delta s^+, d$ einführen. Damit können wir jedes der Elemente aus der Formel mit einem Tupel (a, i, j) aus $\{\Delta s^-, \Delta s^+, d\} \times \{0, \ldots, N_f\} \times \{0, \ldots, N_g\}$ identifizieren und bezeichnen mit $a_{i,j}$ dessen Wert. Hierzu definieren wir

$$\mathcal{L} = \left\{(a, i, j) \in \left\{\Delta s^-, \Delta s^+, d\right\} \times \{0, \ldots, N_f\} \times \{0, \ldots, N_g\} \middle|\ a_{i,j} \in \mathcal{E}\right\}$$

und erhalten einen ersten Algorithmus, welcher noch ohne Lemma 3.2.1 auskommt.

Lemma 3.2.2. *Seien $f, g \in \mathbb{D}_c$. Dann liefert Algorithmus 3.4 den Abstand der beiden Funktionen im Sinne der Skorokhodmetrik, wobei die innere Schleife insgesamt $\mathcal{O}(N_f N_g)$ mal durchlaufen wird.*

Beweis. Wir starten mit einem beliebigen Tupel $(k, R) \in \mathcal{Z}'$ und werten ϕ darauf aus. Nun können wir $\mathcal{Z}'$ gemäß der Ausführungen in diesem Abschnitt anpassen, indem wir alle Elemente herausnehmen, die zur Berechnung des Minimums keinen Beitrag leisten. In der inneren Schleife werden all jene Tupel (k', R') aus $\mathcal{Z}'$ entfernt, für die $\phi(k', R') \geqslant u$ gilt (und damit insbesondere das aktuelle Tupel (k, R)). Folglich befinden sich nach dem Verlassen dieser Schleife in $\mathcal{Z}'$ ausschließlich Tupel, für die ϕ einen Wert strikt unter dem laufenden Minimum u liefert. Damit ist $\mathcal{Z}'$ genau dann leer, wenn kein weiteres $(k', R') \in \mathcal{Z}$ mit $\phi(k', R') < u$ existiert. v bezeichnet den vorigen Wert des laufenden Minimums. Mit Hilfe dieser Variable stellen wir sicher, dass die Operationen in der inneren Schleife nur für jene Elemente durchgeführt werden, welche noch nicht abgearbeitet wurden. Somit bleibt die Gesamtanzahl der inneren Schleifendurchläufe auf $|\mathcal{L}| = \mathcal{O}(N_f N_g)$ beschränkt. Haben wir die innere Schleife verlassen, so können wir wieder von vorne beginnen, sofern $\mathcal{Z}'$ noch nicht leer ist. □

Gelingt es uns, in polynomieller Zeit ein Tupel $(k, R) \in \mathcal{Z}'$ zu finden, so können wir wegen Lemma 3.2.2 mit Hilfe von Algorithmus 3.4 den Abstand zweier stückweise konstanter Funktionen im Sinne der Skorokhodmetrik in polynomieller Zeit berechnen. Hierzu werden wir, anstatt die Menge der gültigen Tupel immer weiter zu reduzieren, die Bedingungen an die künftigen Tupel (k, R) schrittweise verschärfen. Um die Berechnung des nächsten Elementes zu vereinfachen, werden wir $\mathcal{Z}$ mit einer Ordnungsrelation versehen. Anhand dieser werden wir das nächste Tupel berechnen, welches die bisher akkumulierten Bedingungen allesamt erfüllt.

In Lemma 3.2.1 sind Bedingungen spezifiziert, welche ein Tupel $(k, R) \in \mathcal{Z}$ erfüllen muss, um in der dynamischen Menge $\mathcal{Z}'$ zu liegen. Diese Bedingungen

setzen sich aus Aussagen der Form

$$k_m \geqslant n \tag{3.2.8}$$
$$k_m \leqslant n \tag{3.2.9}$$
$$k_m \neq n \tag{3.2.10}$$
$$k_m < n \Rightarrow k_{m+1} \leqslant n \tag{3.2.11}$$
$$k_m = n \Rightarrow m \in R \tag{3.2.12}$$
$$k_m = n \Rightarrow m \notin R \tag{3.2.13}$$

für $m \in \{0, \ldots, N_f\}$, $n \in \{0, \ldots, N_g\}$ zusammen.

Um die Zeitkomplexität des gewünschten Algorithmus möglichst gering zu halten, werden wir für die Bedingungen (3.2.11)–(3.2.13) boolesche Matrizen statt Mengen verwenden. Wir definieren die booleschen $N_f \times N_g$-Matrizen G, H^1 und H^2 folgendermaßen. Soll ein Tupel $(k, R) \in \mathcal{Z}$ der Bedingung (3.2.11) für ein bestimmtes (m, n) mit $m \in \{1, \ldots, N_f - 1\}$, $n \in \{1, \ldots, N_g\}$ genügen, so wird $G_{m,n}$ auf „wahr“ und andernfalls auf „falsch“ gesetzt. Analog gehen wir für die Bedingungen (3.2.12) und (3.2.13) mit den Matrizen H^1 bzw. H^2 vor.

(3.2.12) ist genau dann für alle m, n mit $H^1_{m,n}$ wahr, wenn $\{m \mid H^1_{m,k_m}\} \subset R$ gilt. Analog ist (3.2.13) genau dann für alle m, n mit $H^2_{m,n}$ wahr, wenn $R \subset \{m \mid \neg H^2_{m,k_m}\}$ gilt.

Weiter definieren wir eine boolesche Funktion q_G auf $\mathcal{K}$ wie folgt. $q_G(k)$ soll genau dann wahr sein, wenn (3.2.11) für all diejenigen m, n erfüllt ist, für die $G_{m,n}$ wahr ist.

Für jedes $m \in \{1, \ldots, N_f\}$ sei V_m eine beliebige, nichtleere Teilmenge von $\{0, \ldots, N_g\}$ und V das kartesische Produkt $V = \times_{m=1}^{N_f} V_m$. V_m enthalte alle Werte n, welche k_m gemäß der Bedingungen (3.2.9)–(3.2.10) annehmen darf. Die Mengen V_m werden wir im Laufe der Auswertung immer weiter verkleinern. Somit gilt für $\mathcal{Z}'$ aus Algorithmus 3.4

$$\mathcal{Z}' = \Big\{(k, R) \in (\mathcal{K} \cap V) \times \mathcal{P}(\{1, \ldots, N_f\}) \mid q_G(k) \\ \wedge \big(Q_k \cup \{m \mid H^1_{m,k_m}\} \subset R \subset \{m \mid \neg H^2_{m,k_m}\}\big)\Big\}.$$

Für die kommenden Lemmata werden wir die Notierung etwas anpassen und schreiben $\mathcal{Y}(V, G, H^1, H^2)$ für $\mathcal{Z}'$.

Bei der Umsetzung der Bedingungen (3.2.8)–(3.2.10) reicht es aus, die Menge V_m anzupassen. Wir wollen genauer untersuchen, wann $\mathcal{Y}(V, G, H^1, H^2)$ leer

ist und definieren

$$\mathcal{X} = \{k \in \mathcal{K} \cap V \mid q_G(k)\}.$$

Falls $\mathcal{X}$ leer ist, gilt dies auch für $\mathcal{Y}(V, G, H^1, H^2)$. Dass unter gewissen Voraussetzungen auch die Umkehrung gilt, zeigt das folgende Lemma.

Lemma 3.2.3. *Für jedes $m \in \{1, \ldots, N_f\}$ gelte*

$$V_m \subset \{n \mid \neg(H^1_{m,n} \wedge H^2_{m,n})\} \tag{3.2.14}$$

sowie

$$V_{m-1} \subset \{n \mid \neg H^2_{m,n}\} \tag{3.2.15}$$

mit der Konvention $V_0 = \{0\}$. Dann existiert für jedes $k \in \mathcal{K} \cap V$ mit $q_G(k)$ ein $R \in \mathcal{P}(\{1, \ldots, N_f\})$ mit $(k, R) \in \mathcal{Y}(V, G, H^1, H^2)$.

Beweis. Für $k \in \mathcal{K} \cap V$ mit $q_G(k)$ setzen wir

$$R = Q_k \cup \{m \mid H^1_{m,k_m}\}.$$

(3.2.14) ist äquivalent zu

$$\{m \mid H^1_{m,k_m}\} \subset \{m \mid \neg H^2_{m,k_m}\}$$

für alle $k \in V$ und (3.2.15) zieht

$$k_{m-1} = k_m \Rightarrow \neg H^2_{m,k_m}$$

für alle $k \in V$ und alle m nach sich. Also liegt auch Q_k ganz in $\{m \mid \neg H^2_{m,k_m}\}$ und somit R. □

Mit Π_1 sei die Projektion auf die erste Komponente bezeichnet. Seien A, B, C Mengen mit $C \subset A \times B$, dann ist $\Pi_1(C)$ definiert als die Menge aller $a \in A$, für die ein $b \in B$ mit $(a, b) \in C$ existiert.

Gehen wir bei der dynamischen Anpassung von V geschickt vor, so reicht es zur Prüfung von $\mathcal{Y}(V, G, H^1, H^2) = \varnothing$ sogar aus, lediglich V zu betrachten. Aus Lemma 3.2.3 folgt außerdem sofort

$$\{k \in \mathcal{K} \cap V \mid q_G(k)\} = \Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right).$$

Korollar 3.2.4. *Die Voraussetzungen von Lemma 3.2.3 seien erfüllt und für $V \neq \varnothing$ existiere stets ein $k \in \mathcal{K} \cap V$ mit $q_G(k)$. Dann ist V genau dann leer, wenn $\mathcal{Y}(V, G, H^1, H^2)$ leer ist.*

Wir widmen uns nun der Umsetzung der Bedingungen (3.2.9)–(3.2.11) und betrachten hierzu folgende Fragestellung: Sei $V = \times_i V_i$ mit $V_i \subset \{0, \ldots, N_g\}$ für alle i vorgelegt und $\widetilde{V} = V$. Wie müssen wir unter den Voraussetzungen von Lemma 3.2.3 die Mengen V_i modifizieren, damit für fest vorgegebene $m \in \{1, \ldots, N_f\}$, $n \in \{0, \ldots, N_g\}$ die Gleichung

$$\left\{(k, R) \in \mathcal{Y}\left(\widetilde{V}, G, H^1, H^2\right) \mid k_m \geqslant n\right\} = \mathcal{Y}\left(V, G, H^1, H^2\right) \tag{3.2.16}$$

sowie

$$\Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right) = \varnothing \Rightarrow V = \varnothing \tag{3.2.17}$$

gelten? Da mit V ohnehin nur die erste Komponente modifiziert wird, reicht es aus, statt (3.2.16) lediglich

$$\left\{k \in \Pi_1\left(\mathcal{Y}\left(\widetilde{V}, G, H^1, H^2\right)\right) \mid k_m \geqslant n\right\} = \Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right) \tag{3.2.18}$$

zu zeigen.

Wie zu Beginn dieses Kapitels erwähnt, versehen wir V mit der lexikographischen Ordnung. Dies ermöglicht uns, Minima und Maxima beliebiger Teilmengen von V zu nutzen. Man mache sich klar, dass

$$\min V = (\min V_1, \min V_2, \ldots, \min V_{N_f})$$

gilt. (3.2.18) ist bereits erfüllt, wenn $V_m \subset \{j \in \widetilde{V}_m \mid j \geqslant n\}$ sichergestellt ist. Wir werden durch sukzessives Entfernen von Elementen der Mengen V_i das lexikographische Minimum von V so weit erhöhen bis entweder V leer ist oder

$$\min V \in \Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right) \tag{3.2.19}$$

gilt. Wir nehmen $V_m \neq \varnothing$ an. Handlungsbedarf besteht lediglich im Falle von $\min V_m < n$. Wir nehmen alle Elemente aus V_m heraus, welche echt kleiner als n sind. Damit erhöhen wir effektiv das Minimum von V_m. Diese Erhöhung kann jedoch $\min V_m > \min V_{m+1}$ und daher $\min V \notin \mathcal{K}$ nach sich ziehen. Folglich müssen wir nun V_{m+1} anheben, indem wir

$$V_{m+1} = \{j \in V_{m+1} \mid j \geqslant \min V_m\}$$

setzen. Es ist zu beachten, dass hiernach nicht zwingend $\min V_m = \min V_{m+1}$ gelten muss. Das ist beispielsweise der Fall, wenn $\min V_m$ nicht in V_{m+1} liegt. Eine Erhöhung an der Stelle $m+1$ kann wiederum zur Folge haben, dass ein j existiert mit $G_{m,j}$ und $\min V_m < j < \min V_{m+1}$ und somit die Bedingung $q_G(\min V)$ nicht mehr erfüllt ist. Für das kleinste Element $k' \in \mathcal{K} \cap V$, welches

Algorithmus 3.5 Umsetzung der Bedingung $k_m \geqslant n$

```
1: procedure set_lower_bound(m, n)
2:     if V ≠ ∅ and min V_m < n then
3:         V_m ← {j ∈ V_m | j ⩾ n}
4:         minV_changed ← true
5:         while V ≠ ∅ and (min V ∉ K or ¬q_G(min V)) do
6:             if min V ∉ K then
7:                 Wähle i mit min V_i > min V_{i+1}
8:                 V_{i+1} ← {j ∈ V_{i+1} | j ⩾ min V_i}
9:             end if
10:            if ¬q_G(min V) then
11:                Wähle i, j mit min V_i < j < min V_{i+1} and G_{i,j}
12:                r ← max{j | G_{i,j} ∧ j < min V_{i+1}}
13:                V_i ← {s ∈ V_i | s > r}
14:            end if
15:        end while
16:    end if
17: end procedure
```

$k'_m < j \Rightarrow k'_{m+1} \leqslant j$ erfüllt, muss daher $k'_m \geqslant j$ gelten. Sollten mehrere solche j existieren, so muss

$$k'_m \geqslant \max\{j \mid G_{m,j} \wedge j < \min V_{m+1}\} \tag{3.2.20}$$

gelten. Folglich müssen wir $\min V_m$ auf das Maximum in (3.2.20) anheben, was erneut $\min V_m > \min V_{m+1}$ implizieren kann. Wir wiederholen diese Schritte so lange, bis entweder V leer ist oder sowohl $\min V \in \mathcal{K}$ als auch $q_G(\min V)$ gelten. Damit ist (3.2.17) und im Falle von $V \neq \varnothing$ auch (3.2.19) erfüllt. Diese wechselseitige Beeinflussung führt nicht zu einer Endlosschleife, da bei jeder Anpassung mindestens ein Element aus einer der Mengen V_i herausgenommen wird. Damit bleibt die Anzahl der Schleifendurchläufe auf

$$1 + \sum_{i=1}^{N_f} \left(|\widetilde{V}_i| - 1 \right) \leqslant 1 + N_f \left(\max_{i=1,\dots,N_f} |\widetilde{V}_i| - 1 \right) \leqslant 1 + N_f N_g$$

begrenzt.

Lemma 3.2.5. *Seien $m \in \{1, \dots, N_f\}$, $n \in \{0, \dots, N_g\}$, $V_m \neq \varnothing$ und die Voraussetzungen von Lemma 3.2.3 seien erfüllt. Dann gilt nach der Ausführung von* `set_lower_bound`(m, n)

$$\left\{ k \in \Pi_1 \left(\mathcal{Y} \left(\widetilde{V}, G, H^1, H^2 \right) \right) \mid k_m \geqslant n \right\} = \Pi_1 \left(\mathcal{Y} \left(V, G, H^1, H^2 \right) \right),$$

wobei $\widetilde{V}$ die Menge V vor der Ausführung des Algorithmus bezeichnet. Ist V nichtleer, so ist (3.2.19) *erfüllt.*

Durch die Festlegung auf $\min V$ als dasjenige Element, welches wir später auswerten wollen, haben wir am oberen Ende von V etwas mehr Freiheiten. Hier müssen wir nicht sicherstellen, dass $\max V$ auch wirklich in $\mathcal{K}$ liegt. Der Bedingung (3.2.9) tragen wir dadurch Rechnung, dass wir V_m auf die Menge $V_m \cap \{j \mid j \leqslant n\}$ einschränken. Sofern nicht n strikt kleiner als $\min V_m$ ist, bleibt (3.2.17) von dieser Festlegung unberührt. Andernfalls sind V_m und damit auch V leer. Gehen wir davon aus, dass die Elemente der Mengen V_i geordnet vorliegen, ergibt sich eine Laufzeit von $\mathcal{O}(N_f N_g)$.

Lemma 3.2.6. *Seien $m \in \{1, \dots, N_f\}$, $n \in \{0, \dots, N_g\}$ und die Voraussetzungen von Lemma 3.2.3 seien erfüllt. Dann gilt nach der Ausführung von* `erase`(m, n)

$$\left\{k \in \Pi_1\left(\mathcal{Y}\left(\widetilde{V}, G, H^1, H^2\right)\right) \mid k_m \neq n\right\} = \Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right),$$

wobei $\widetilde{V}$ die Menge V vor der Ausführung des Algorithmus bezeichnet. Ist V nichtleer, so ist (3.2.19) *erfüllt.*

Algorithmus 3.6 Umsetzung der Bedingung $k_m \neq n$

```
1: procedure erase(m, n)
2:     if V ≠ ∅ then
3:         if n = min V_m then
4:             set_lower_bound(m, n + 1)
5:         else
6:             V_m ← V_m \ {n}
7:         end if
8:     end if
9: end procedure
```

Lemma 3.2.7. *Falls $0 = m < n$ oder*

$$(1 \leqslant m < N_f) \wedge (\max V_m < n)$$

gilt, ist (3.2.11) *äquivalent zu $k_{m+1} \leqslant n$ für alle $k \in V$. Falls $m = N_f$ oder*

$$(1 \leqslant m < N_f) \wedge (n < \min V_{m+1})$$

gilt, ist (3.2.11) *äquivalent zu $k_m \geqslant n$ für alle $k \in V$. Falls $m = n = 0$ oder*

$$(1 \leqslant m < N_f) \wedge ((\min V_{m+1} \leqslant n \leqslant \min V_m) \vee (\max V_{m+1} \leqslant n \leqslant \max V_m))$$

gilt, ist (3.2.11) *bereits erfüllt.*

Algorithmus 3.7 Umsetzung der Bedingung $k_m < n \Rightarrow k_{m+1} \leqslant n$

```
1:  procedure adj_cond(m, n)
2:      if V ≠ ∅ then
3:          if n < min V_{m+1} then
4:              set_lower_bound(m, n)
5:          else if max V_m < n then
6:              V_{m+1} ← {j ∈ V_{m+1} | j ⩽ n}
7:          else if min V_m < n < max V_{m+1} then
8:              G_{m,n} ← true
9:          end if
10:     end if
11: end procedure
```

Im Hinblick auf `set_lower_bound` ist es aus Gründen der Effizienz sinnvoll, $G_{i,j}$ möglichst nur dann zu setzen, wenn sich die Bedingung nicht weiter zerlegen lässt.

Lemma 3.2.8. *Seien $m \in \{1, \dots, N_f - 1\}$, $n \in \{0, \dots, N_g\}$, $V \neq \varnothing$ und die Voraussetzungen von Lemma 3.2.3 sowie* (3.2.19) *seien erfüllt. Dann gilt nach der Ausführung von* `adj_cond`(m, n)

$$\left\{k \in \Pi_1\left(\mathcal{Y}\left(\tilde{V}, \tilde{G}, H^1, H^2\right)\right) \mid k_m < n \Rightarrow k_{m+1} \leqslant n\right\} = \Pi_1\left(\mathcal{Y}\left(V, G, H^1, H^2\right)\right),$$

wobei $\tilde{V}$ die Menge V und $\tilde{G}$ die boolesche $N_f \times N_g$ Matrix G vor der Ausführung des Algorithmus bezeichnet. Ist V nichtleer, so ist (3.2.19) *erfüllt.*

Beweis. Falls $n < \min V_{m+1}$ gilt, so ist für alle $k \in V$ die Aussage (3.2.11) äquivalent zu $k_m \geqslant n$ und wir können `set_lower_bound`(m, n) nutzen. Die Behauptung folgt mit Lemma 3.2.5. Andernfalls ist

$$\min V_m < n \Rightarrow \min V_{m+1} \leqslant n$$

erfüllt und wegen $V \neq \varnothing$ und (3.2.19) nach Voraussetzung folgt mit Lemma 3.2.7 auch nach Ausführung des Algorithmus wieder (3.2.19). □

Lemma 3.2.9. *Seien $m \in \{1, \dots, N_f\}$, $n \in \{0, \dots, N_g\}$, $S \in \mathcal{P}(\{1, \dots, N_f\})$, $a \in \{1, 2\}$, $V \neq \varnothing$ und die Voraussetzungen von Lemma 3.2.3 sowie*

$$(\min V, S) \in \mathcal{Y}\left(V, G, H^1, H^2\right) \tag{3.2.21}$$

Algorithmus 3.8 Umsetzung der Bedingungen $k_m = n \Rightarrow m \in R$ und $k_m = n \Rightarrow m \notin R$

```
1: procedure restr_cond(a, m, n)
2:     if V ≠ ∅ then
3:         if H^{3-a}_{m,n} then
4:             erase(m, n)
5:         else
6:             H^a_{m,n} ← true
7:             if n = min V_m then
8:                 if a = 1 then
9:                     S ← S ∪ {m}
10:                else
11:                    S ← S\{m}
12:                end if
13:            end if
14:        end if
15:    end if
16: end procedure
```

seien erfüllt. Nach der Ausführung von `restr_cond`(a, m, n) *gilt für den Parameter* $a = 1$

$$\left\{(k, R) \in \mathcal{Y}\left(\widetilde{V}, G, \widetilde{H^1}, H^2\right) \mid k_m = n \Rightarrow m \in R\right\} = \mathcal{Y}\left(V, G, H^1, H^2\right),$$

und für den Parameter $a = 2$

$$\left\{(k, R) \in \mathcal{Y}\left(\widetilde{V}, G, H^1, \widetilde{H^2}\right) \mid k_m = n \Rightarrow m \notin R\right\} = \mathcal{Y}\left(V, G, H^1, H^2\right),$$

wobei $\widetilde{V}$ *die Menge* V *und* $\widetilde{H^1}$ *sowie* $\widetilde{H^2}$ *die booleschen* $N_f \times N_g$*-Matrizen* H^1 *und* H^2 *vor der Ausführung des Algorithmus bezeichnen. Die Voraussetzungen von Lemma 3.2.3 sind weiterhin erfüllt. Ist* V *nichtleer, so gilt* (3.2.21) *falls* $\min \widetilde{V} = \min V$ *und* (3.2.19) *falls* $\min \widetilde{V} \neq \min V$.

Beweis. Wir zeigen nur den Fall $a = 1$, der andere funktioniert analog. Gilt bereits $H^2_{m,n}$, so ist für alle $(k, R) \in \mathcal{Y}\left(\widetilde{V}, G, \widetilde{H^1}, H^2\right)$ die neue Bedingung (3.2.12) äquivalent zu (3.2.10) und die Behauptung folgt mit Lemma 3.2.6. Andernfalls bleibt die Menge V unverändert und wir stellen im Falle von $n = \min V$ durch die Modifikation von S sicher, dass (3.2.21) weiterhin gilt. □

Aus den bisher in diesem Abschnitt bewiesenen Lemmata und Algorithmen erhalten wir folgenden Satz.

Algorithmus 3.9 Berechnung von $d_S(f,g)$ für $f,\ g \in \mathbb{D}_c$ in polynomieller Zeit

```
1:  function distance_fast(f, g)
2:      V_i = {0, ..., N_g},   i = 1, ..., N_f
3:      G ← false
4:      H^1 ← false
5:      H^2 ← false
6:      v ← 1 + max E
7:      minV_changed ← true
8:      while V ≠ ∅ do
9:          if minV_changed then S ← Q_k ∪ {i | H^1_{i,min V_i}} end if
10:         u ← φ(min V, S)
11:         minV_changed ← false
12:         for all (a, i, j) ∈ L with u ⩽ a_{i,j} < v do
13:             if a = Δs^− then
14:                 V_i ← {s ∈ V_i | s < j}
15:             else if a = Δs^+ then
16:                 set_lower_bound(i, j)
17:                 restr_cond(1, i, j)
18:             else
19:                 if i = 0 then
20:                     if j = 0 then
21:                         V ← ∅
22:                     else
23:                         V_1 ← {s ∈ V_1 | s ⩽ j}
24:                         restr_cond(2, 1, j)
25:                     end if
26:                 else if i = N_f then
27:                     set_lower_bound(N_f, j + 1)
28:                 else
29:                     erase(i, j)
30:                     adj_cond(i, j)
31:                     restr_cond(2, i + 1, j)
32:                 end if
33:             end if
34:         end for
35:         v ← u
36:     end while
37:     return u
38: end function
```

Satz 3.2.10. *Algorithmus 3.9 berechnet den Abstand im Sinne der Skorokhodmetrik. Die Zeitkomplexität von* `distance_fast` *beträgt* $\mathcal{O}(N_f^2 N_g^2)$*, der Speicherplatzbedarf liegt bei* $\mathcal{O}(N_f N_g)$*.*

3.3 Ein randomisierter Algorithmus

Um die Effizienzsteigerung durch Algorithmus 3.3 in Relation zu Algorithmus 3.9 besser einschätzen zu können, ist es hilfreich, einen randomisierten Vergleichsalgorithmus zu haben. Einen solchen wollen wir nun schrittweise entwickeln.

Offensichtlich müssen wir mehrere Tupel $(k, R) \in \mathcal{Z}_N^M$ „zufällig“ ziehen und anschließend ϕ für jedes dieser Tupel auswerten. Das Minimum aller so erhaltenen Werte wäre dann der approximative Wert für den Abstand im Sinne der Skorokhodmetrik.

Sofern nichts Genaueres über die Struktur der Funktionen, deren Abstand man berechnen möchte, bekannt ist, erscheint eine Gleichverteilung angemessen. Die Wahl einer optimalen Verteilung könnte Gegenstand weiterer Untersuchungen sein. Doch wie erzeugt man Tupel, die auf $\mathcal{Z}_N^M$ gleichverteilt sind? Da es sich bei $\mathcal{Z}_N^M$ um eine Teilmenge von $\mathcal{K}_N^M \times \mathcal{P}_M$ handelt, werden wir nun Methoden entwickeln mit deren Hilfe man zufällige Elemente aus $\mathcal{K}_N^M$ und aus $\mathcal{P}_M$ ziehen kann.

3.3.1 Erzeugen von Samples aus $\mathcal{K}_N^M$

In diesem Abschnitt werden wir Methoden herleiten, mit deren Hilfe man Vektoren generieren kann, die einer beliebigen, vorgelegten Verteilung auf $\mathcal{K}_N^M$ folgen. Zur Motivation wollen wir mit der Gleichverteilung auf $\mathcal{K}_N^M$ beginnen. Eine erste Idee ist die sogenannte Acceptance-Rejection Methode. Sie ist besonders für Zufallsvektoren geeignet, welche in einer komplizierteren Teilmenge $A \subset \mathbb{R}^n$ liegen sollen. Man gibt sich eine einfachere Grundmenge $B \supset A$ vor, auf der gleichverteilte Samples erzeugt werden können. Ein solches Sample aus B wird verworfen, falls es nicht in der anvisierten Teilmenge liegt. Da es wünschenswert ist, den Ausschuss zu minimieren, sollte man die Grundmenge möglichst klein wählen. Algorithmus 3.10 greift diese Idee auf. Für kleine $M, N \in \mathbb{N}$ liefert dieser Algorithmus zufriedenstellende Ergebnisse. Für größere Werte ist diese Methode jedoch nicht mehr praktikabel. Der Grund dafür ist die Tatsache, dass die Teilmenge $\mathcal{K}_N^M \subset \{0, \ldots, N\}^M$

Algorithmus 3.10 Erzeugen eines Samples von $\mathcal{U}(\mathcal{K}_N^M)$ mit Hilfe von Acceptance-Rejection

```
1: function generate_k_AR
2:     v_i ← 0,   i = 1, ..., M
3:     while ¬(v_1 ⩽ v_2 ⩽ ··· ⩽ v_M) do
4:         Erzeuge v_i ~ U({0, ..., N}),   i = 1, ..., M
5:     end while
6:     return (v_1, v_2, ..., v_M)
7: end function
```

im Verhältnis zur Grundmenge sehr schnell klein wird. Dies treibt die Anzahl der Verwerfungen enorm in die Höhe, was wiederum den Algorithmus ineffizienter werden lässt.

Sei $X = (X_1, \dots, X_M)$ ein Zufallsvektor, wobei die X_i unabhängig und identisch verteilt sind (uiv) mit $X_i \sim \mathcal{U}(\{0, \dots, N\})$. Dann ist X auf $\{0, \dots, N\}^M$ gleichverteilt und es gilt

$$P(X \in \mathcal{K}_N^M) = \frac{|\mathcal{K}_N^M|}{(N+1)^M} = \frac{(M+N)!}{M!\,N!\,(N+1)^M}\,.$$

Mit der Stirlingformel erhalten wir für den Fall $N = M$ das asymptotische Verhalten

$$P(X \in \mathcal{K}_N^N) \approx \left(\frac{4}{N}\right)^N. \tag{3.3.1}$$

Bereits für $N = 10$ liegt die Wahrscheinlichkeit in (3.3.1) bei 7.1×10^{-6}. Dies bedeutet, dass man für eine Million erzeugte Zufallsvektoren in $\{0, \dots, N\}^M$ im Mittel 7 auf $\mathcal{K}_N^M$ gleichverteilte Vektoren erhält. Wir müssen also eine andere Herangehensweise wählen. Da bei den Elementen aus $\mathcal{K}_N^M$ jeder Eintrag direkt von dem vorherigen Eintrag abhängt, ist es naheliegend, mit bedingten Verteilungen zu arbeiten.

Die Verteilung eines Zufallsvektors auf $\mathcal{K}_N^M$ sei gegeben durch

$$\sum_{k \in \mathcal{K}_N^M} f(k)\,\epsilon_k, \tag{3.3.2}$$

mit der Dichte $f\colon \mathcal{K}_N^M \to [0,1]$, wobei ϵ_k das Diracmaß in $k \in \mathcal{K}_N^M$ bezeichnet.

Zur Anschauung betrachten wir kurz den Fall $M = 2$. Lässt man auch Elemente zu, welche außerhalb von $\mathcal{K}_N^M$ liegen und setzt $f(k) = 0$ für alle

$k \in \{0,\dots,N\}^M \backslash \mathcal{K}_N^M$, so wird f zu einer stochastischen $(N+1)\times(N+1)$-Matrix mit Dreiecksgestalt. Je nach Definition sind dann entweder die Zeilen- oder die Spaltensummen gleich 1.

Sei X ein Zufallsvektor mit Verteilung wie in (3.3.2). Dann gilt für alle $m \in \mathbb{N}$ mit $m < M$ und alle $k \in \mathcal{K}_N^m$

$$\begin{aligned} &P(X_1 = k_1,\dots,X_m = k_m) \\ &\quad = \sum_{r_1=k_m}^{N} \sum_{r_2=r_1}^{N} \cdots \sum_{r_{M-m}=r_{M-m-1}}^{N} f(k_1,\dots,k_m,r_1,\dots,r_{M-m}) \\ &\quad = \sum_{r_1=0}^{N-k_m} \sum_{r_2=r_1}^{N-k_m} \cdots \sum_{r_{M-m}=r_{M-m-1}}^{N-k_m} f(k_1,\dots,k_m,k_m+r_1,\dots,k_m+r_{M-m}) \\ &\quad = \sum_{r\in\mathcal{K}_{N-k_m}^{M-m}} f(k_1,\dots,k_m,k_m+r_1,\dots,k_m+r_{M-m}). \end{aligned} \tag{3.3.3}$$

Insbesondere ist

$$P(X_1 = n) = \sum_{r\in\mathcal{K}_{N-n}^{M-1}} f(n,n+r_1,\dots,n+r_{M-1}), \quad n = 0,\dots,N. \tag{3.3.4}$$

Mit der Definition der bedingten Wahrscheinlichkeit folgt aus (3.3.3) für alle $m \in \{2,\dots,M-1\}$ und alle $k \in \mathcal{K}_N^{m-1}$ mit $P(X_1 = k_1,\dots,X_{m-1} = k_{m-1}) > 0$,

$$\begin{aligned} &P(X_m = n \,|\, X_1 = k_1,\dots,X_{m-1} = k_{m-1}) \\ &\quad = \frac{\displaystyle\sum_{r\in\mathcal{K}_{N-n}^{M-m}} f(k_1,\dots,k_{m-1},n,n+r_1,\dots,n+r_{M-m})}{\displaystyle\sum_{s\in\mathcal{K}_{N-k_{m-1}}^{M-m+1}} f(k_1,\dots,k_{m-1},k_{m-1}+s_1,\dots,k_{m-1}+s_{M-m+1})}, \end{aligned} \tag{3.3.5}$$

$n = k_{m-1},\dots,N$, sowie für alle $k \in \mathcal{K}_N^{M-1}$ mit $P(X_1 = k_1,\dots,X_{M-1} = k_{M-1}) > 0$,

$$P(X_M = n \,|\, X_1 = k_1,\dots,X_{M-1} = k_{M-1}) = \frac{f(k_1,\dots,k_{M-1},n)}{\displaystyle\sum_{s=k_{M-1}}^{N} f(k_1,\dots,k_{M-1},s)}, \tag{3.3.6}$$

$n = k_{M-1},\dots,N$.

Wir kommen nun wieder zurück zur Gleichverteilung auf $\mathcal{K}_N^M$,

$$\sum_{k\in\mathcal{K}_N^M} \frac{1}{|K_N^M|}\,\epsilon_k,$$

welche wir – in Einklang mit der üblichen Notation – mit $\mathcal{U}(\mathcal{K}_N^M)$ bezeichnen. In diesem Fall liefert (3.3.3) für alle $m \in \mathbb{N}$ mit $m < M$ und alle $k \in \mathcal{K}_N^m$

$$P(X_1 = k_1, \ldots, X_m = k_m) = \frac{|\mathcal{K}_{N-k_m}^{M-m}|}{|\mathcal{K}_N^M|} > 0. \tag{3.3.7}$$

Folglich existieren die bedingten Wahrscheinlichkeiten in (3.3.5) und (3.3.6). Des Weiteren ist ersichtlich, dass der Ausdruck auf der rechten Seite von (3.3.7) nicht von $k_1, \ldots, k_{m-1}$ abhängt. Somit erhalten wir aus (3.3.5)–(3.3.6) für alle $m \in \{2, \ldots, M\}$ und alle $l \in \{0, \ldots, N\}$

$$\begin{aligned} P(X_m = n \,|\, X_{m-1} = l) &= \frac{|\mathcal{K}_{N-n}^{M-m}|}{|\mathcal{K}_{N-l}^{M-m+1}|} \\ &= \frac{M-m+1}{M-m+1+N-n} \prod_{r=l}^{n-1} \frac{N-r}{M-m+1+N-r}, \end{aligned} \tag{3.3.8}$$

$n = l, \ldots, N$. Da im Falle von $n - l > M - m$ in (3.3.8) einige Terme sowohl im Zähler als auch im Nenner auftreten, ist diese Wahrscheinlichkeit gleich

$$\frac{M-m+1}{M-m+1+N-l} \prod_{r=1}^{(n-l)\wedge(M-m)} \frac{N-n+r}{M-m+1+N-l-r}.$$

Auf die gleiche Weise erhalten wir aus Gleichung (3.3.4)

$$\begin{aligned} P(X_1 = n) &= \frac{|\mathcal{K}_{N-n}^{M-1}|}{|\mathcal{K}_N^M|} \\ &= \frac{M}{M+N-n} \prod_{r=0}^{n-1} \frac{N-r}{M+N-r} \\ &= \frac{M}{M+N} \prod_{r=1}^{n\wedge(M-1)} \frac{N-n+r}{M+N-r}, \quad n = 0, \ldots, N. \end{aligned} \tag{3.3.9}$$

Das bisher Gezeigte halten wir in einem Satz fest.

Satz 3.3.1. *Sei $X \sim \mathcal{U}(\mathcal{K}_N^M)$. Dann ist für jedes $m \in \{2, \ldots, M\}$ die bedingte Verteilung von X_m unter Vorkenntnis von $X_{m-1} = l$ gegeben durch die Familie von Verteilungen $(\mu_l^m, 0 \leqslant l \leqslant N)$ mit*

$$\mu_l^m = \frac{M-m+1}{M-m+1+N-l} \sum_{n=l}^{N} \prod_{r=1}^{(n-l)\wedge(M-m)} \frac{N-n+r}{M-m+1+N-l-r}\, \epsilon_n. \tag{3.3.10}$$

Die Verteilung von X_1 ist gegeben durch

$$\mu = \frac{M}{M+N} \sum_{n=0}^{N} \prod_{r=1}^{n \wedge (M-1)} \frac{N-n+r}{M+N-r} \, \epsilon_n.$$

Basierend auf Satz 3.3.1 können wir nun einen Zufallsvektor $X \sim \mathcal{U}(\mathcal{K}_N^M)$ erzeugen, indem wir zunächst $X_1 \sim \mu$ und dann sukzessiv alle weiteren Einträge gemäß der Verteilung in (3.3.10) generieren (siehe Algorithmus 3.11). Samples einer diskreten Verteilung erhält man unter anderem mit der inver-

Algorithmus 3.11 Erzeugen eines Samples $k \sim \mathcal{U}(\mathcal{K}_N^M)$

```
1: function gen_uni_k
2:     m ← 1
3:     Erzeuge k_1 ~ μ
4:     while m < M do
5:         m ← m + 1
6:         Erzeuge k_m ~ μ^m_{k_{m-1}}
7:     end while
8:     return (k_1, ..., k_M)
9: end function
```

sen Transformationsmethode. Nähere Informationen hierzu findet der Leser beispielsweise im Vorlesungsskript zur stochastischen Simulation von A. Lang und J. Potthoff (siehe [4]).

Für eine effiziente Implementation bietet es sich an, die Wahrscheinlichkeiten anhand von (3.3.8) und (3.3.9) rekursiv zu berechnen. Dies motiviert folgendes

Korollar 3.3.2. *Sei $X \sim \mathcal{U}(\mathcal{K}_N^M)$. Dann gilt für alle $m \in \{2, \dots, M\}$ und alle $l \in \{0, \dots, N-1\}$*

$$P(X_m = n+1 \,|\, X_{m-1} = l) = \frac{N-n}{N-n+M-m} \, P(X_m = n \,|\, X_{m-1} = l),$$

$n = l, \dots, N-1$, mit der Startwahrscheinlichkeit

$$P(X_m = l \,|\, X_{m-1} = l) = \frac{M-m+1}{M-m+1+N-l},$$

für alle $l \in \{0, \dots, N\}$. Für die Verteilung von X_1 gilt

$$P(X_1 = n+1) = \frac{N-n}{N-n+M-1} \, P(X_1 = n), \quad n = 0, \dots, N-1,$$

mit der Startwahrscheinlichkeit

$$P(X_1 = 0) = \frac{M}{M+N}\,.$$

Bevor wir uns mit einer weiteren Verteilung beschäftigen, führen wir die Funktionen

$$\begin{aligned} \zeta\colon \mathbb{N}_0^2 &\to \mathbb{N}, \\ (a,b) &\mapsto \sum_{v=0}^{a\wedge b} \binom{a}{v}\binom{b}{v} 2^v, \end{aligned} \tag{3.3.11}$$

und für $a \in \mathbb{N}_0$

$$\begin{aligned} \varphi_a^M\colon \mathbb{N}_0^M &\to \mathbb{N}_0, \\ k &\mapsto \mathbb{1}_{\{k_1 > a\}} + \sum_{i=1}^{M-1} \mathbb{1}_{\{k_i < k_{i+1}\}}, \end{aligned}$$

ein. Offensichtlich ist ζ symmetrisch, d.h. $\zeta(a,b) = \zeta(b,a)$, und für $b > 0$ gilt

$$\zeta(a,b) = |\mathcal{Z}_a^b|.$$

Unmittelbar aus der Definition von φ_a^M leiten sich die folgenden beiden Eigenschaften ab: Für alle $W \in \{1,\dots,M-1\}$ gilt

$$\varphi_a^M(k_1,\dots,k_M) = \varphi_a^W(k_1,\dots,k_W) + \varphi_{k_W}^{M-W}(k_{W+1},\dots,k_M), \tag{3.3.12}$$

und für alle $n \in \mathbb{N}_0$

$$\varphi_{n+a}^M(n+k_1,\dots,n+k_M) = \varphi_a^M(k_1,\dots,k_M). \tag{3.3.13}$$

Nun betrachten wir die Verteilung der Gestalt

$$\sum_{k\in\mathcal{K}_N^M} \frac{2^{\varphi_0^M(k)}}{\zeta(N,M)}\,\epsilon_k, \tag{3.3.14}$$

welche wir mit θ bezeichnen.

(3.3.3) liefert zusammen mit (3.3.12) und (3.3.13) für alle $m \in \mathbb{N}$ mit $m < M$ und alle $k \in \mathcal{K}_N^m$

$$P(X_1 = k_1,\dots,X_m = k_m) = \frac{2^{\varphi_0^m(k)}}{\zeta(N,M)} \sum_{r\in\mathcal{K}_{N-k_m}^{M-m}} 2^{\varphi_0^{M-m}(r)}. \tag{3.3.15}$$

Indem wir in (3.3.15) nicht mehr über die Elemente von $\mathcal{K}_{N-k_m}^{M-m}$ summieren, sondern über die Werte von φ_0^{M-m}, erhalten wir

$$\begin{aligned}\sum_{r\in\mathcal{K}_{N-k_m}^{M-m}} 2^{\varphi_0^{M-m}(r)} &= \sum_{v=0}^{M-m}\binom{M-m}{v}\binom{N-k_m}{v}2^v\\ &= \zeta(N-k_m, M-m).\end{aligned}$$

Damit ist

$$P(X_1 = n) = 2^{\mathbb{1}_{\{n>0\}}}\frac{\zeta(N-n, M-1)}{\zeta(N,M)}, \quad n = 0,\dots,N. \tag{3.3.16}$$

Da der Ausdruck auf der rechten Seite von (3.3.15) strikt positiv ist, können wir auch hier die bedingten Wahrscheinlichkeiten berechnen und erhalten aus (3.3.5)–(3.3.6) für alle $m \in \{2,\dots,M\}$ und alle $k \in \mathcal{K}_N^{m-1}$

$$\begin{aligned}&P(X_m = n \,|\, X_1 = k_1,\dots,X_{m-1} = k_{m-1})\\ &\qquad = 2^{\mathbb{1}_{\{k_{m-1}<n\}}}\frac{\zeta(N-n, M-m)}{\zeta(N-k_{m-1}, M-m+1)},\end{aligned} \tag{3.3.17}$$

$n = k_{m-1},\dots,N$. Offensichtlich hat die Vorkenntnis von $X_1,\dots,X_{m-2}$ keine Auswirkung auf die bedingte Wahrscheinlichkeit in (3.3.17). Dies formulieren wir in folgendem

Satz 3.3.3. *Sei ζ definiert wie in* (3.3.11), *θ wie in* (3.3.14), *$X \sim \theta$. Dann ist für jedes $m \in \{2,\dots,M\}$ die bedingte Verteilung von X_m unter Vorkenntnis von $X_{m-1} = l$ gegeben durch die Familie von Verteilungen $(\nu_l^m, 0 \leqslant l \leqslant N)$ mit*

$$\nu_l^m = \sum_{n=l}^{N} 2^{\mathbb{1}_{\{l<n\}}}\frac{\zeta(N-n, M-m)}{\zeta(N-l, M-m+1)}\,\epsilon_n.$$

Die Verteilung von X_1 ist gegeben durch

$$\nu = \sum_{n=0}^{N} 2^{\mathbb{1}_{\{n>0\}}}\frac{\zeta(N-n, M-1)}{\zeta(N,M)}\,\epsilon_n.$$

Um eine Zufallsvariable nach der Verteilung ν_l^m zu erzeugen, ist es notwendig, die Wahrscheinlichkeiten in (3.3.16) und (3.3.17) auszuwerten. Dafür die vorliegende Form zu verwenden, ist nicht ratsam. Da $\zeta(n,n)$ asymptotisch mindestens wie 2^{2n} wächst, aber der Quotient in $[0,1]$ liegen muss, ist dieser Ausdruck für eine Auswertung ungeeignet. Wir benötigen also eine etwas robustere Form. Es ist möglich, wenn auch aufgrund der vielen Fallunterscheidungen ziemlich umständlich, einen robusten geschlossenen Ausdruck

für (3.3.17) herzuleiten. Wir werden stattdessen die Wahrscheinlichkeiten rekursiv berechnen. Hierzu definieren wir die Funktion δ durch

$$\begin{aligned}\delta\colon \mathbb{N}\times\mathbb{N}_0 &\to [0,1],\\ (a,b) &\mapsto \frac{\zeta(a-1,b)}{\zeta(a,b)}\,.\end{aligned}$$

Man beachte, dass – im Gegensatz zu ζ – die Funktion δ nicht symmetrisch ist. Damit ist für alle $m\in\{2,\dots,M\}$ und alle $l\in\{0,\dots,N-1\}$

$$\begin{aligned}&P(X_m=n+1\,|\,X_{m-1}=l)\\ &\qquad=2^{\mathbb{1}_{\{l=n\}}}\frac{\zeta(N-n-1,M-m)}{\zeta(N-n,M-m)}\,P(X_m=n\,|\,X_{m-1}=l)\\ &\qquad=2^{\mathbb{1}_{\{l=n\}}}\delta(N-n,M-m)\,P(X_m=n\,|\,X_{m-1}=l),\end{aligned}$$

$n=l,\dots,N-1$, und für die Startwahrscheinlichkeit gilt wegen der Symmetrie von ζ für alle $l\in\{0,\dots,N\}$

$$\begin{aligned}P(X_m=l\,|\,X_{m-1}=l)&=\frac{\zeta(N-l,M-m)}{\zeta(N-l,M-m+1)}\\ &=\frac{\zeta(M-m,N-l)}{\zeta(M-m+1,N-l)}\\ &=\delta(M-m+1,N-l).\end{aligned}$$

Das Auswerten der Wahrscheinlichkeiten in (3.3.16) und (3.3.17) wird durch die rekursive Variante auf die Berechnung von δ reduziert.

Korollar 3.3.4. *Sei $X\sim\theta$. Dann gilt für alle $m\in\{2,\dots,M\}$ und alle $l\in\{0,\dots,N-1\}$*

$$P(X_m=n+1\,|\,X_{m-1}=l)=2^{\mathbb{1}_{\{l=n\}}}\delta(N-n,M-m)\,P(X_m=n\,|\,X_{m-1}=l),$$

$n=l,\dots,N-1$, mit der Startwahrscheinlichkeit

$$P(X_m=l\,|\,X_{m-1}=l)=\delta(M-m+1,N-l)$$

für alle $l\in\{0,\dots,N\}$. Für die Verteilung von X_1 gilt

$$P(X_1=n+1)=2^{\mathbb{1}_{\{n=0\}}}\delta(N-n,M-1)\,P(X_1=n),\quad n=0,\dots,N-1,$$

mit der Startwahrscheinlichkeit

$$P(X_1=0)=\delta(M,N).$$

Wir müssen nun noch einen Weg finden, δ effizient auszuwerten. Für $a \in \mathbb{N}, b \in \mathbb{N}_0$ gilt

$$\begin{aligned}\delta(a,b) &= \sum_{v=0}^{(a-1)\wedge b} \binom{a-1}{v}\binom{b}{v} 2^v \left(\sum_{w=0}^{a\wedge b}\binom{a}{w}\binom{b}{w}2^w\right)^{-1} \\ &= \sum_{v=0}^{(a-1)\wedge b}\left(\sum_{w=0}^{a\wedge b}\gamma_a^b(v,w)\right)^{-1}, \end{aligned} \tag{3.3.18}$$

wobei die Funktion γ_a^b durch

$$\begin{aligned}\gamma_a^b\colon \{0,\dots,a-1\}\times\{0,\dots,b\} &\to (0,+\infty), \\ (v,w) &\mapsto 2^{w-v}\binom{a}{w}\binom{b}{w}\binom{a-1}{v}^{-1}\binom{b}{v}^{-1},\end{aligned}$$

definiert ist. Werten wir $\gamma_a^b(v,w)$ weiter aus, so erhalten wir

$$\begin{aligned}\delta(a,b) = \frac{1}{a}\sum_{v=0}^{(a-1)\wedge b}\Bigg\{(a-v)\,2^{-v}\Bigg[\left(\sum_{w=0}^{v}2^w\prod_{q=w}^{v-1}\frac{(q+1)^2}{(a-q)(b-q)}\right) \\ + \left(\sum_{w=v+1}^{a\wedge b}2^w\prod_{q=v}^{w-1}\frac{(a-q)(b-q)}{(q+1)^2}\right)\Bigg]^{-1}\Bigg\}.\end{aligned}$$

Auch hier erscheint eine rekursive Variante schneller. Zur effizienteren Auswertung von (3.3.18) verwenden wir eine „zweidimensionale Rekursion“, bei der zunächst die erste Komponente und dann basierend darauf die zweite jeweils rekursiv berechnet werden. Für alle $v, w \in \mathbb{N}_0$ mit $v \leqslant (a-1)\wedge b$ und $w < a \wedge b$ gilt

$$\gamma_a^b(v,w+1) = \frac{2\,(a-w)(b-w)}{(w+1)^2}\,\gamma_a^b(v,w), \tag{3.3.19}$$

mit dem Startpunkt $\gamma_a^b(v,0)$. Dies ist die Rekursion in der zweiten, inneren Komponente. Die Startwerte für diese innere Rekursion werden ebenfalls rekursiv berechnet. Für alle $v \in \mathbb{N}_0$ mit $v < (a-1)\wedge b$ gilt

$$\gamma_a^b(v+1,0) = \frac{(v+1)^2}{2\,(a-v-1)(b-v)}\,\gamma_a^b(v,0),$$

mit dem Startpunkt $\gamma_a^b(0,0) = 1$. Da der Koeffizient bei der inneren Rekursion in (3.3.19) nicht von v abhängt, ist es sinnvoll diesen einmal zu Beginn für

Algorithmus 3.12 Berechnung von $\delta(a, b)$ mittels Rekursion in γ_a^b

```
1:  function delta(a, b)
2:      for w ← 1 to a ∧ b do
3:          c_w ← 2 (a − w + 1)(b − w + 1)/w²
4:      end for
5:      σ ← 0
6:      x ← 1
7:      for v ← 0 to (a − 1) ∧ b do
8:          y ← x
9:          μ ← y
10:         for w ← 1 to a ∧ b do
11:             y ← y c_w
12:             μ ← μ + y
13:         end for
14:     end for
15:     σ ← σ + 1/μ
16:     x ← x (v + 1)²/(2 (a − v − 1)(b − v))
17:     return σ
18: end function
```

alle w auszurechnen und dann die berechneten Werte innerhalb der inneren Schleife wiederzuverwenden. Algorithmus 3.12 greift diese Herangehensweise auf. Der Umstand, dass die beiden inneren Summen für kleine v extrem klein und für große v extrem groß sind, erschwert eine schnelle Berechnung von δ. Dies eröffnet jedoch auch Spielräume für Approximationen. Allerdings gilt es dann auch zu untersuchen, inwiefern sich ein Approximationsfehler bei der Berechnung von δ auf die Verteilung der Samples auswirkt.

3.3.2 Erzeugen von Samples aus $\mathcal{P}_N$

Wie erzeugt man zufällige Mengen, die einer beliebigen, vorgelegten Verteilung auf einer endlichen Potenzmenge folgen? Zunächst präzisieren wir, was damit gemeint ist. Wir werden uns eine spezielle Konstruktion der Elemente einer endlichen Potenzmenge zunutze machen, um das Problem auf die Erzeugung von Zufallsvektoren in einer Teilmenge des $\mathbb{R}^d$ zu überführen. Man kann jede Menge mit einem geordneten Tupel assoziieren, indem man die Elemente der Menge in geordneter Weise in einen Vektor schreibt. Auf dieser Grundlage werden wir einen Algorithmus entwickeln, welcher uns solche Mengen liefert. Hierzu zerlegen wir die Potenzmenge in die einzelnen Komponenten

und berechnen deren $\mathbb{R}^d$-wertige Verteilungen. Anschließend zeigen wir, dass man basierend auf den erzeugten Vektoren dieser Verteilungen Samples der ursprünglich vorgelegten, mengenwertigen Verteilung konstruieren kann.

Wir führen nun den Begriff endlicher Zufallsmengen ein. Sei hierzu eine endliche Menge E vorgelegt. Da die Potenzmenge einer endlichen Menge ebenfalls endlich ist, können wir als σ-Algebra über $\mathcal{P}(E)$ einfach die Potenzmenge wählen. Sie wird vom Banach-Tarski-Paradoxon nicht in Frage gestellt. $(\mathcal{P}(E), \mathcal{P}(\mathcal{P}(E)))$ wird so zu einem messbaren Raum.

Definition 3.3.5. *Sei $(\Omega, \mathcal{A}, P)$ ein Wahrscheinlichkeitsraum, E eine endliche Menge. Eine Abbildung $X\colon \Omega \to \mathcal{P}(E)$ heißt* (endliche) Zufallsmenge, *falls*

$$X^{-1}(\{B\}) \in \mathcal{A}$$

für alle $B \subset E$ gilt.

Eine Verteilung auf $(\mathcal{P}(E), \mathcal{P}(\mathcal{P}(E)))$ ist durch eine Funktion $f\colon \mathcal{P}(E) \to [0,1]$ mit $\sum_{A \in \mathcal{P}(E)} f(A) = 1$ eindeutig festgelegt:

$$\sum_{A \in \mathcal{P}(E)} f(A)\, \epsilon_A.$$

Ein solches f bezeichnen wir als (diskrete) Dichte.

Im Folgenden werden wir zufällige Mengen mit Werten in $\mathcal{P}_N, N \in \mathbb{N}$ betrachten. Für $M \in \mathbb{N}_0$ mit $M \leqslant N$ definieren wir

$$\mathcal{P}_N^M = \{U \in \mathcal{P}_N \mid |U| = M\}.$$

Offensichtlich ist $\mathcal{P}_N$ gerade die disjunkte Vereinigung der $\mathcal{P}_N^M$:

$$\mathcal{P}_N = \biguplus_{M=0}^{N} \mathcal{P}_N^M.$$

Oftmals (vor allem im Zusammenhang mit Algorithmen, die auf diesen Mengen operieren sollen) ist es sogar wünschenswert die Elemente einer Menge in geordneter Version vorliegen zu haben, im ungünstigsten Fall ist es nicht relevant. Algorithmen, welche auf Mengen operieren, sind meist wesentlich schneller und effizienter, wenn die Elemente der Mengen geordnet sind. Durch die Überführung auf Vektoren sollte es gelingen, die geordnete Version einer Zufallsmenge direkt zu erzeugen.

Die Verteilung einer endlichen Zufallsmenge Y mit Werten in $\mathcal{P}_N$ sei gegeben durch

$$\sum_{A \in \mathcal{P}_N} f(A)\, \epsilon_A,$$

mit $f\colon \mathcal{P}_N \to [0,1]$ und $\sum_{A\in\mathcal{P}_N} f(A) = 1$. Dann ist die Verteilung von $|Y|$ gleich

$$\rho_N = \sum_{l=0}^{N} \left(\sum_{A\in\mathcal{P}_N^l} f(A) \right) \epsilon_A$$

und die bedingte Verteilung von Y unter Vorkenntnis von $|Y| = l$ ist gegeben durch die Familie von Verteilungen $(\tau_l, 0 \leqslant l \leqslant N)$ auf $(\mathcal{P}_N, \mathcal{P}(\mathcal{P}_N))$, definiert durch

$$\tau_l = \sum_{B\in\mathcal{P}_N^l} \frac{f(B)}{\sum_{A\in\mathcal{P}_N^l} f(A)} \epsilon_B. \tag{3.3.20}$$

Der folgende Satz zeigt, wie man eine Verteilung auf $\mathcal{P}_N$ simuliert.

Satz 3.3.6. *Sei α eine Verteilung auf $\mathcal{P}(\mathcal{P}_N)$, L eine reellwertige Zufallsvariable mit*

$$L \sim \sum_{l=0}^{N} \alpha(\mathcal{P}_N^l)\, \epsilon_l,$$

Y eine endliche Zufallsmenge mit Werten in $\mathcal{P}_N$ gegeben durch die bedingte Verteilung

$$P_{Y|L} = \sum_{A\in\mathcal{P}_N^L} \frac{\alpha(\{A\})}{\alpha(\mathcal{P}_N^L)} \epsilon_A.$$

Dann gilt $Y \sim \alpha$.

Beweis. Nach Konstruktion liegt $l \in \{0,\dots,N\}$ genau dann außerhalb des Trägers von P_L, wenn $\mathcal{P}_N^l$ eine α-Nullmenge ist. Somit gilt für beliebiges $B \subset \mathcal{P}_N$

$$\begin{aligned}
P_Y(B) &= \int_\Omega P_{Y|L}(\omega, B)\, dP(\omega) \\
&= \int_\Omega \frac{\alpha(B \cap \mathcal{P}_N^{L(\omega)})}{\alpha(\mathcal{P}_N^{L(\omega)})}\, dP(\omega) \\
&= \int_{\mathbb{R}} \frac{\alpha(B \cap \mathcal{P}_N^{l})}{\alpha(\mathcal{P}_N^{l})}\, dP_L(l) \\
&= \sum_{l\in\mathrm{supp}(P_L)} \alpha(B \cap \mathcal{P}_N^l) \\
&= \alpha(B).
\end{aligned}$$

□

Um eine Menge der vorgelegten Verteilung α zu simulieren, können wir zunächst dessen Länge L erzeugen und dann Y nach $P_{Y|L}$. Das so erhaltene Sample Z folgt dann der vorgelegten Verteilung.

Es bleibt noch zu zeigen, wie man eine Zufallsmenge mit fest vorgegebener Kardinalität nach einer vorgelegten Verteilung erzeugt. Wir definieren für $1 \leqslant M \leqslant N$ die Menge

$$\mathcal{V}_N^M = \{v \in \{1, \ldots, N\}^M \mid v_1 < v_2 < \cdots < v_M\} \subset \mathbb{R}^M.$$

Indem wir die Elemente einer jeden Menge $A \in \mathcal{P}_N^M$ in einen Vektor der Dimension M schreiben und seine Einträge aufsteigend sortieren, erhalten wir einen eindeutigen Vektor aus $\mathcal{V}_N^M$. Umgekehrt können wir jedem Vektor $v \in \mathcal{V}_N^M$ eine eindeutige Menge in $\mathcal{P}_N^M$ zuordnen, deren Elemente aus den Einträgen von v bestehen. Auf diese Weise erhalten wir eine Bijektion zwischen $\mathcal{P}_N^M$ und $\mathcal{V}_N^M$. Nun können wir jedes Element der Potenzmenge mit einem Vektor identifizieren. Für eine $\mathcal{P}_N^M$-wertige Zufallsmenge $Y = \{Y_1, Y_2, \ldots, Y_M\}$ bezeichne $(Y_{(1)}, Y_{(2)}, \ldots, Y_{(M)})$ den zugehörigen geordneten Zufallsvektor aus $\mathcal{V}_N^M$.

Wenden wir uns nun der Gleichverteilung auf $\mathcal{P}_N$ zu. Der Fall $Y \sim \mathcal{U}(\mathcal{P}_N)$ ergibt wegen

$$P(|Y| = l) = \frac{|\mathcal{P}_N^l|}{|\mathcal{P}_N|} = \frac{\binom{N}{l}}{2^N}, \quad l = 0, \ldots, N,$$

die Verteilung

$$\rho_N = 2^{-N} \sum_{j=0}^{N} \binom{N}{j} \epsilon_j$$

für die Kardinalität einer Zufallsmenge aus $\mathcal{U}(\mathcal{P}_N)$. Die bedingten Verteilungen aus (3.3.20) sind in diesem Fall geben durch $(\mathcal{U}(\mathcal{P}_N^l), 0 \leqslant l \leqslant N)$. Um die geordnete Version einer zufälligen Menge aus $\mathcal{U}(\mathcal{P}_N)$ zu simulieren, müssen wir nun ein Sample aus $\mathcal{U}(V_N^l)$ für $l \geqslant 1$ erzeugen.

Die Mengen $\mathcal{K}_N^M$ und $\mathcal{V}_N^M$ sind sich in ihrer Struktur so ähnlich, dass alles, was wir in Abschnitt 3.3.1 bis einschließlich der Gleichverteilung gezeigt haben, mit geringen Modifikationen ebenso für $\mathcal{V}_N^M$ funktioniert. Man hat nur zusätzlich zu beachten, dass aufgrund der Ungleichheit der Einträge eines beliebigen Vektors aus $\mathcal{V}_N^M$ der m-te Eintrag zwischen m und $N - M + m$ liegen muss. Der folgende Satz ist das Analogon von Satz 3.3.1 für die Menge $\mathcal{P}_N^M$.

Satz 3.3.7. *Seien $M, N \in \mathbb{N}$ mit $M \leqslant N$, $Y \sim \mathcal{U}(\mathcal{P}_N^M)$. Falls $M \geqslant 2$, ist für jedes $m \in \{2, \dots, M\}$ die bedingte Verteilung von $Y_{(m)}$ unter Vorkenntnis von $Y_{(m-1)} = l$ gegeben durch die Familie von Verteilungen $(\eta_l^m, m-1 \leqslant l \leqslant N-M+m-1)$ mit*

$$\eta_l^m = \frac{M-m+1}{N-l} \sum_{n=l+1}^{N-M+m} \prod_{r=1}^{(n-l-1)\wedge(M-m)} \frac{N-M+m-n+r}{N-l-r}\, \epsilon_n.$$

Die Verteilung von $Y_{(1)}$ ist gegeben durch

$$\eta = \frac{M}{N} \sum_{n=1}^{N-M+1} \prod_{r=2}^{n\wedge M} \frac{N-M-n+r}{N-r+1}\, \epsilon_n.$$

Algorithmus 3.13 Erzeugen eines Samples $v \sim \mathcal{U}(\mathcal{P}_N), N \geqslant 1$

```
1:  function gen_uni_set
2:      Erzeuge M ~ ρ
3:      if M = 0 then
4:          return ∅
5:      else
6:          m ← 1
7:          Erzeuge v_1 ~ η
8:          while m < M do
9:              m ← m + 1
10:             Erzeuge v_m ~ η^m_{v_{m-1}}
11:         end while
12:         return {v_1, ..., v_M}
13:     end if
14: end function
```

Korollar 3.3.8. *Seien $M, N \in \mathbb{N}$ mit $M \leqslant N$, $Y \sim \mathcal{U}(\mathcal{P}_N^M)$. Falls $M \geqslant 2$, gilt für alle $m \in \{2, \dots, M\}$ und alle $l \in \{m-1, \dots, N-M+m-1\}$*

$$P(Y_{(m)} = n+1 \,|\, Y_{(m-1)} = l) = \frac{N-n-M+m}{N-n}\, P(Y_{(m)} = n \,|\, Y_{(m-1)} = l),$$

$n = l+1, \dots, N-M+m-1$, mit der Startwahrscheinlichkeit

$$P(Y_{(m)} = l+1 \,|\, Y_{(m-1)} = l) = \frac{M-m+1}{N-l}\,.$$

Für die Verteilung von $Y_{(1)}$ gilt

$$P(Y_{(1)} = n+1) = \frac{N-n-M+1}{N-n}\, P(Y_{(1)} = n), \quad n = 1, \dots, N-M,$$

mit der Startwahrscheinlichkeit

$$P(Y_{(1)} = 1) = \frac{M}{N}\,.$$

3.3.3 Erzeugen von Samples aus $\mathcal{Z}_N^M$

In den Abschnitten 3.3.1–3.3.2 haben wir gezeigt, wie man Samples beliebiger Verteilungen auf $\mathcal{K}_N^M$ sowie auf $\mathcal{P}_M$ generiert. Somit können wir zufällige Elemente erzeugen, welche auf $\mathcal{K}_N^M \times \mathcal{P}_M$ gleichverteilt sind. Wir wollen eine Gleichverteilung auf $\mathcal{Z}_N^M$ nutzen, müssen aber die funktionelle Abhängigkeit der zweiten Komponente von der ersten in Gestalt von Q_k^M berücksichtigen. In diesem Abschnitt werden wir uns mit Zufallsvariablen mit Werten in $\mathcal{K}_N^M \times \mathcal{P}_M$ auseinandersetzen. Da diese Menge diskret ist, werden wir auch hier als σ-Algebra die Potenzmenge wählen. Die Verteilung eines Zufallstupels (X, Y) mit Werten in $\mathcal{K}_N^M \times \mathcal{P}_M$ sei gegeben durch

$$\sum_{(k,R)\in\mathcal{K}_N^M\times\mathcal{P}_M} f(k,R)\,\epsilon_{(k,R)},$$

mit $f\colon \mathcal{K}_N^M \times \mathcal{P}_M \mapsto [0,1]$ und $\sum_{(k,R)\in\mathcal{K}_N^M\times\mathcal{P}_M} f(k,R) = 1$. Dann ist

$$P(X = k) = P(X = k, Y \in \mathcal{P}_M) = \sum_{R\in\mathcal{P}_M} f(k,R)$$

sowie

$$P(Y = R \,|\, X = k) = \frac{f(k,R)}{\sum\limits_{S\in\mathcal{P}_M} f(k,S)}.$$

Satz 3.3.9. *Sei β eine Verteilung auf $(\mathcal{K}_N^M \times \mathcal{P}_M, \mathcal{P}(\mathcal{K}_N^M \times \mathcal{P}_M))$, X eine Zufallsvariable mit*

$$X \sim \sum_{k\in\mathcal{K}_N^M}^{N} \beta(\{k\} \times \mathcal{P}_M)\,\epsilon_k\,,$$

Y eine endliche Zufallsmenge mit Werten in $\mathcal{P}_N$ gegeben durch die bedingte Verteilung

$$P_{Y|X} = \sum_{A\in\mathcal{P}_M} \frac{\beta(\{(X,A)\})}{\beta(\{X\}\times\mathcal{P}_M)}\,\epsilon_A.$$

Dann gilt $(X, Y) \sim \beta$.

Beweis. Nach Konstruktion von X ist $\complement\mathrm{supp}(P_X) \times \mathcal{P}_M$ eine β-Nullmenge. Somit gilt für beliebige $A \subset \mathcal{K}_N^M$, $B \subset \mathcal{P}_N$

$$\begin{aligned} P_{X\otimes Y}(A \times B) &= P(X \in A, Y \in B) \\ &= \int_{\{X\in A\}} P_{Y|X}(\omega, B)\, dP(\omega) \\ &= \int_{\{X\in A\}} \frac{\beta(\{X(\omega)\} \times B)}{\beta(\{X(\omega)\} \times \mathcal{P}_M)}\, dP(\omega) \\ &= \int_A \frac{\beta(\{k\} \times B)}{\beta(\{k\} \times \mathcal{P}_M)}\, dP_X(k) \\ &= \beta((A \cap \mathrm{supp}(P_X)) \times B) \\ &= \beta(A \times B)\,. \end{aligned}$$

Man beachte, dass $\mathcal{P}(\mathcal{K}_N^M \times \mathcal{P}_M) = \mathcal{P}(\mathcal{K}_N^M) \otimes \mathcal{P}(\mathcal{P}_M)$ gilt. □

Sei nun (X, Y) ein Zufallstupel mit Werten in $\mathcal{Z}_N^M$. Dann hat die Verteilung die Form

$$\sum_{\substack{(k,R)\in\mathcal{K}_N^M\times\mathcal{P}_M,\\ R\supset Q_k^M}} f(k,R)\,\epsilon_{(k,R)}$$

mit der diskreten Dichte f. Somit ist

$$P(X=k) = P(X=k, Y \in \{R \in \mathcal{P}_M \,|\, R \supset Q_k^M\}) = \sum_{\substack{R\in\mathcal{P}_M,\\ R\supset Q_k^M}} f(k,R)$$

Algorithmus 3.14 Erzeugen eines Samples (k, R) von $\mathcal{U}(\mathcal{Z}_N^M)$

```
1: function gen_uni_Z
2:     Erzeuge k ~ θ
3:     R ← Q_k^M
4:     if |R| < M then
5:         C ← {1,...,M}\R
6:         Erzeuge I ~ U(P_|C|)
7:         R ← R ⊎ C_I
8:     end if
9:     return (k, R)
10: end function
```

sowie

$$P(Y = R \,|\, X = k) = \frac{f(k,R)}{\sum\limits_{\substack{S \in \mathcal{P}_M, \\ S \supset Q_k^M}} f(k,S)}.$$

Mit Hilfe der in den Abschnitten 3.3.1–3.3.2 erworbenen Methoden können wir nun beliebige Verteilungen auf $\mathcal{Z}_N^M$ generieren. Wir müssen zuerst $X \sim P_X$ und anschließend $Y \sim P_{Y|X}$ erzeugen. Für den Fall einer Gleichverteilung auf $\mathcal{Z}_N^M$, d.h.

$$f(k,R) = |\mathcal{Z}_N^M|^{-1} \quad \text{für alle } (k,R) \in \mathcal{Z}_N^M,$$

erhalten wir

$$P(X = k) = \frac{2^{M-|Q_k^M|}}{|\mathcal{Z}_N^M|} = \frac{2^{\varphi_0^M(k)}}{\zeta(N,M)}$$

und damit $P_X = \theta$ sowie $P_{Y|X} = \mathcal{U}(\{R \in \mathcal{P}_M \,|\, R \supset Q_X^M\})$.

3.3.4 Auswertungsalgorithmus

Basierend auf Algorithmus 3.14 aus dem vorigen Abschnitt können wir nun $d_S(f,g)$ für $f,\, g \in \mathbb{D}_c$ approximieren. Hierzu ist es erforderlich, $n \in \mathbb{N}$ unabhängige Samples $Z_1, \ldots, Z_n$ aus $\mathcal{U}(\mathcal{Z})$ zu generieren. Der approximative Wert für den Skorokhodabstand von f und g ist dann durch

$$\min_{i=1,\ldots,n} \phi(Z_i)$$

gegeben.

Algorithmus 3.15 Randomisierter Algorithmus zur approximativen Berechnung von $d_S(f,g)$ für $f,\, g \in \mathbb{D}_c$ mit Parameter $n \in \mathbb{N}$

```
1: function distance_rand(f, g, n)
2:     u ← max E
3:     for i ← 1 to n do
4:         (k, R) ← gen_uni_Z
5:         if φ(k, R) < u then
6:             u ← φ(k, R)
7:         end if
8:     end for
9:     return u
10: end function
```

3.4 Vergleich der Algorithmen

Für einen Vergleich der Algorithmen müssen wir Pfade aus $\mathbb{D}_c$ simulieren. Eine in der Wahrscheinlichkeitstheorie sehr wichtige Klasse von Prozessen mit Pfaden in $\mathbb{D}_c$ sind die sogenannten *zusammengesetzten Poissonprozesse*. Nähere Informationen zu solchen Prozessen und ihrer Bedeutung für die wesentlich größere Klasse der Lévyprozesse findet der Leser in [5] von K.-I. Sato. Wir werden uns einer bekannten Methode zur Simulation solcher Prozesse bedienen. Zunächst simuliert man die Anzahl der Sprungstellen gemäß der

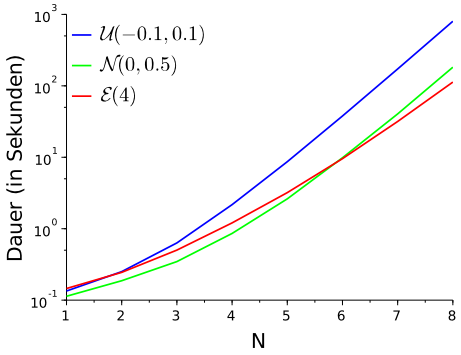
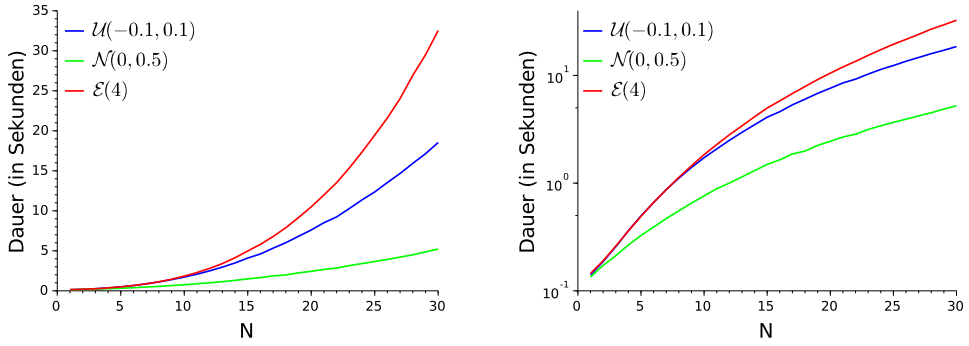


Abbildung 3.1: Laufzeiten von `distance` für 10 000 Auswertungen in Abhängigkeit der Anzahl der Unstetigkeitsstellen

Poissonverteilung und anschließend erzeugt man unabhängige, identisch verteilte Sprungstellen nach $\mathcal{U}((0,1))$ und sortiert diese. Zum Schluss werden die Sprunghöhen unabhängig von der Wahl der Sprungstellen uiv nach einer vorher festgelegten, sogenannten Sprungverteilung erzeugt (siehe Algorithmus 6.2 in R. Cont und P. Tankov [6]). Da wir die Anzahl der Unstetigkeitsstellen stets fest vorgeben, handelt es sich bei den in diesem Rahmen erzeugten Pfaden nicht um zusammengesetzte Poissonprozesse (siehe Implementation in Anhang B.1). In jedem Fall aber erhält man auf diesem Weg stückweise konstante Funktionen.

Es bezeichne $\mathcal{U}((a,b))$ die stetige Gleichverteilung auf dem Intervall $(a,b) \subset \mathbb{R}$, $\mathcal{N}(\mu,\sigma)$ die Normalverteilung mit Erwartungswert $\mu \in \mathbb{R}$ und Standardabweichung $\sigma > 0$ und $\mathcal{E}(\lambda), \lambda > 0$ die Exponentialverteilung mit Erwartungswert $1/\lambda$. Für die Analyse wurden die reellwertigen Verteilungen $\mathcal{U}((-0.1, 0.1))$, $\mathcal{N}(0, 0.5)$ und $\mathcal{E}(4)$ als Sprungverteilungen ausgewählt. Die Auswahl dieser Verteilungen basiert auf der Beobachtung, dass die Laufzeit der Algorithmen mit dem Anteil an kleinen Sprüngen wächst. Eine höhere



(a) lineare Skalierung (b) logarithmische Skalierung

Abbildung 3.2: Laufzeiten von `distance_fast` für 10 000 Auswertungen in Abhängigkeit der Anzahl der Unstetigkeitsstellen

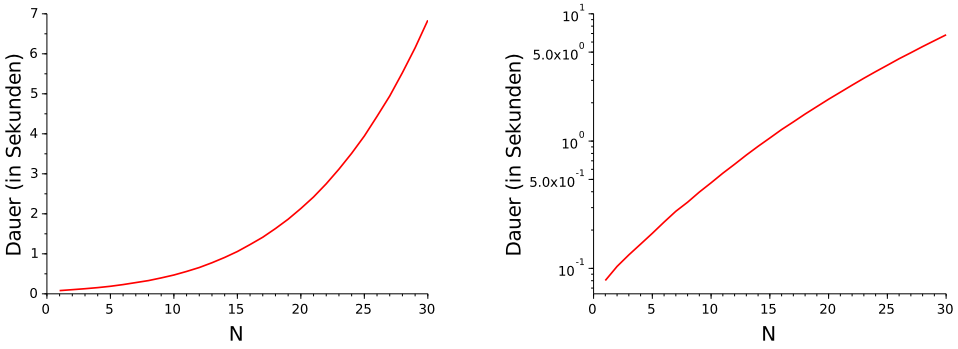
Anzahl kleiner Sprünge erlaubt tendenziell mehr Möglichkeiten, den Graphen einer Funktion so zu verschieben, dass der Abstand bezüglich $\|\cdot\|_\infty$ minimiert wird.

Sofern nichts anderes erwähnt wird, bezeichnen wir in diesem Abschnitt die Anzahl der Sprungstellen mit $N = N_f = N_g$. Die Funktionen $f, g \in \mathbb{D}_c$ wurden so erzeugt, dass $f(0) = g(0) = 0$ gilt. Die angegebenen Laufzeiten decken lediglich die Berechnung von $d_S(f, g)$ ab, die Erzeugung der Pfade wird hingegen nicht berücksichtigt. Die Tests wurden auf einem 64-bit Ubuntu System mit Intel Pentium D 3.20 GHz Dual Core Prozessor durchgeführt. Als Zufallszahlengenerator wurde „MT 19937“, eine Variante des Mersenne Twisters von M. Matsumoto und T. Nishimura (siehe [7]), verwendet.

3.4.1 `distance` vs. `distance_fast`

Beim Vergleich der Algorithmen 3.3 und 3.9 wurden für alle $N \in \{1, \dots, 7\}$ je 1 000 000 Mal die Funktionen f und g erzeugt und deren Abstand berechnet. In allen Fällen stimmten die Ergebnisse der beiden Algorithmen überein. Ergänzend wurden 1 000 000 Mal zunächst die Anzahl der Unstetigkeitsstellen N_f und N_g uiv aus $\mathcal{U}(\{2, \dots, 8\})$ simuliert, dann basierend darauf die Pfade f und g erzeugt und deren Abstand berechnet. Auch hier konnten keinerlei Abweichungen festgestellt werden.

Für eine Analyse der Laufzeiten wurden für jedes vorgelegte $N \in \{1, \dots, 8\}$ 10 000 Mal f und g erzeugt und deren Abstand mit `distance` berechnet.



(a) lineare Skalierung (b) logarithmische Skalierung

Abbildung 3.3: Laufzeiten von `distance_rand` mit Parameter 10 für 10 000 Auswertungen in Abhängigkeit der Anzahl der Unstetigkeitsstellen

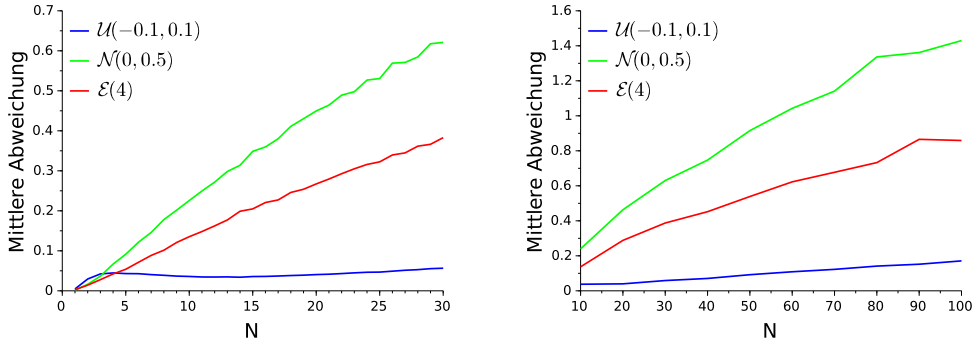
Die Gesamtdauer dieser 10 000 Auswertungen ist in Abbildung 3.1 mit logarithmischer Skalierung dargestellt. Das Gleiche wurde mit `distance_fast` durchgeführt. Aufgrund der deutlich kürzeren Laufzeit wurden die Auswertungen hier bis einschließlich $N = 30$ miteinbezogen. Der Test war so konzipiert, dass für $N \leqslant 8$ die Pfade für beide Algorithmen exakt die gleichen waren, um so eine bessere Vergleichbarkeit der Laufzeiten zu ermöglichen.

Die Abbildungen 3.1 und 3.2b bestätigen den in Kapitel 3 nachgewiesenen Sachverhalt, dass die Laufzeit von Algorithmus 3.3 exponentiell, die von Algorithmus 3.9 jedoch nur polynomiell mit N wächst. `distance` benötigte zum 10 000-maligen Auswerten von $d_S(f, g)$ für $N = 8$ Sprungstellen und der Sprungverteilung $\mathcal{E}(4)$ circa 112.87 Sekunden. Ein Lauf mit `distance_fast` für die gleichen Pfade hingegen nahm lediglich 1.13 Sekunden in Anspruch.

3.4.2 `distance_fast` vs. `distance_rand`

Die Laufzeit von `distance_rand` mit dem Parameter 10 für 1 000 Auswertungen in Abhängigkeit der Anzahl der Unstetigkeitsstellen ist in Abbildung 3.3 zu sehen. Die Auswahl der Tupel, auf denen ϕ ausgewertet wird, hängt in Algorithmus 3.15 weder von f noch von g ab. Daher reicht es aus, sich auf eine beispielhafte Sprungverteilung – in diesem Fall $\mathcal{U}((-0.1, 0.1))$ – festzulegen. Abbildung 3.3b zeigt, dass auch die Laufzeit von `distance_rand` exponentiell mit N wächst.

Schließlich bleibt noch die Güte der Approximation zu testen. Für $Z_1, \ldots, Z_n$



(a) 10 000 Auswertungen (b) 1 000 Auswertungen

Abbildung 3.4: Mittlerer Fehler der Approximation mittels `distance_rand` zum Parameter 10 in Abhängigkeit der Anzahl der Unstetigkeitsstellen

uiv mit $Z_i \sim \mathcal{U}(\mathcal{Z})$ ist der mittlere Fehler der Approximation durch die Funktion `distance_rand` gleich

$$\frac{1}{n}\left(\min_{i=1,\dots,n} \phi(Z_i) - \min_{(k,R)\in\mathcal{Z}} \phi((k,R))\right).$$

Für jedes $N \in \{1, \dots, 30\}$ wurden 10 000 Mal sowohl f als auch g erzeugt, `distance_rand` ausgewertet und der mittlere Fehler zu `distance_fast` berechnet. Diese mittleren Fehler sind in Abbildung 3.4a grafisch dargestellt. Weiterhin wurden für $N \in \{10, 20, \dots, 100\}$ aufgrund der langen Laufzeit lediglich 1 000 Auswertungen des randomisierten Algorithmus durchgeführt (siehe Abbildung 3.4b). Der mittlere Fehler scheint lediglich linear mit N zu steigen, obwohl der Anteil der Samples mit steigendem N exponentiell sinkt.

Kapitel 4

Approximation der Skorokhodmetrik

4.1 Vorbetrachtungen

Wir erinnern uns daran, dass nach Korollar 2.1.3(iv) jede càdlàg-Funktion gleichmäßig durch stückweise konstante, rechtsstetige Funktionen approximierbar ist. $\mathbb{D}_c$ liegt also bezüglich der Supremumsnorm dicht in $\mathbb{D}$. Es liegt nahe, dies zu nutzen, um den Abstand zweier càdlàg-Funktionen im Sinne der Skorokhodmetrik näherungsweise zu berechnen. Hierfür approximieren wir zunächst f und g durch stückweise konstante Funktionen $\widehat{f}, \widehat{g} \in \mathbb{D}_c$. Anschließend können wir mit Hilfe der Formel in (2.3.1) den Skorokhodabstand von $\widehat{f}$ und $\widehat{g}$ ausrechnen. Der Fehler, der sich aus dieser Approximation ergibt, ist dann gleich

$$|d_S(f,g) - d_S(\widehat{f},\widehat{g})|. \tag{4.1.1}$$

Das folgende Lemma gibt eine Abschätzung für den Fehler in (4.1.1) an.

Lemma 4.1.1. *Für alle $f, g \in \mathbb{D}, \widehat{f}, \widehat{g} \in \mathbb{D}_c$ gilt*

$$|d_S(f,g) - d_S(\widehat{f},\widehat{g})| \leqslant \|f - \widehat{f}\|_\infty + \|g - \widehat{g}\|_\infty.$$

Beweis. Da d_S nach Satz 2.2.2 eine Metrik auf $\mathbb{D}$ ist, erhalten wir durch mehrmaliges Anwenden der Dreiecksungleichung

$$\begin{aligned} d_S(\widehat{f},\widehat{g}) &\leqslant d_S(\widehat{f},f) + d_S(f,\widehat{g}) \\ &\leqslant d_S(\widehat{f},f) + d_S(f,g) + d_S(g,\widehat{g}). \end{aligned}$$

Andererseits gilt aber auch

$$\begin{aligned} d_S(f,g) &\leqslant d_S(f,\widehat{f}) + d_S(\widehat{f},g) \\ &\leqslant d_S(f,\widehat{f}) + d_S(\widehat{f},\widehat{g}) + d_S(\widehat{g},g). \end{aligned}$$

Zusammen mit (2.2.2) und der Symmetrie von d_S ergibt dies

$$\begin{aligned} |d_S(f,g) - d_S(\widehat{f},\widehat{g})| &\leqslant d_S(f,\widehat{f}) + d_S(g,\widehat{g}) \\ &\leqslant \|f-\widehat{f}\|_\infty + \|g-\widehat{g}\|_\infty. \end{aligned}$$

□

4.2 Monotone càdlàg-Funktionen

In diesem Abschnitt werden wir einen Algorithmus entwickeln, welcher den Skorokhodabstand zweier monotoner càdlàg-Funktionen zu einem vorgegebenen Fehler $\delta > 0$ approximiert. Im Hinblick auf die Effizienz bei der Auswertung von d_S sollte der Algorithmus mit einer möglichst geringen Anzahl von Stützstellen auskommen.

Satz 4.2.1. *Für $f \in \mathbb{D}$ monoton wachsend und $\varepsilon > 0$ liefert die Funktion* `approx_function`(f,ε) *ein $\widehat{f} \in \mathbb{D}_c$ mit $\|f-\widehat{f}\|_\infty < \varepsilon$.*

Algorithmus 4.1 Approximation von $f \in \mathbb{D}$ monoton wachsend durch $\widehat{f} \in \mathbb{D}_c$ zum Fehler $\varepsilon > 0$

```
1:  function approx_function(f, ε)
2:      Ξ ← {ξ ∈ (0,1) | f(ξ) − f(ξ−) ⩾ ε} ∪ {0,1} (geordnet)
3:      l ← 0
4:      while l ⩽ |Ξ| do
5:          while (f(ξ_{l+1}−) − f(ξ_l))/2 > ε do
6:              Füge (ξ_{l+1} − ξ_l)/2 so zu Ξ hinzu, dass Ξ geordnet bleibt
7:          end while
8:          f̂(t) ← (f(ξ_{l+1}−) + f(ξ_l))/2 für alle t ∈ [ξ_l, ξ_{l+1})
9:          l ← l + 1
10:     end while
11:     return f̂
12: end function
```

Beweis. Ξ enthalte zunächst neben den Randpunkten des Einheitsintervalls alle Stellen, an denen f einen Sprung von mindestens ε hat:

$$\Xi = \{\xi \in (0,1) \mid f(\xi) - f(\xi-) \geqslant \varepsilon\} \cup \{0,1\}.$$

Für den weiteren Verlauf gehen wir davon aus, dass diese Stützstellen geordnet vorliegen, d.h. $\Xi = \{\xi_0, \xi_1, \ldots, \xi_n\}$ mit $0 = \xi_0 < \xi_1 < \cdots < \xi_n = 1$. Wir wollen f auf den Intervallen $[\xi_l, \xi_{l+1})$ durch $f(\xi_{l+1}-) + f(\xi_l))/2$ approximieren. Der Fehler auf einem solchen Intervall ist dann wegen der Monotonie von f durch

$$\sup_{t\in[\xi_l,\xi_{l+1})} \left| f(t) - \frac{f(\xi_{l+1}-) + f(\xi_l)}{2} \right| = \frac{f(\xi_{l+1}-) - f(\xi_l)}{2}$$

gegeben. Solange dieser Fehler die vorgegebene Schwelle ε überschreitet, halbieren wir das betrachtete Intervall, indem wir die Stützstelle $(\xi_l + \xi_{l+1})/2$ in Ξ einfügen. Die neu hinzugefügte Stützstelle wird mit ξ_{l+1} bezeichnet. Sowohl n als auch die Indizes aller Stützstellen, die über dem neu eingefügten Wert liegen, erhöhen sich damit um 1. Da nach Definition auf den von Ξ aufgespannten Intervallen nur Sprünge auftreten, die strikt kleiner als ε sind, finden wir nach endlich vielen Schritten eine Stützstelle, welche die Schleifenbedingung erfüllt. Dann gehen wir zum nächsten Intervall über und wiederholen dieses Verfahren. □

Nun sind wir in der Lage, für vorgelegte, monotone Funktionen $f, g \in \mathbb{D}$ den Abstand im Sinne der Skorokhodmetrik näherungsweise zu berechnen. Lemma 4.1.1 und Satz 4.2.1 garantieren, dass der in (4.1.1) angegebene Fehler der Approximation durch `approx_metric`(f, g, δ) den vorgelegten Wert $\delta > 0$ nicht überschreitet.

Algorithmus 4.2 Approximation von $d_S(f, g)$ für monoton wachsende $f, g \in \mathbb{D}$ zum Fehler $\delta > 0$

```
1: function approx_metric(f, g, δ)
2:     f̂ ← approx_function(f, δ/2)
3:     ĝ ← approx_function(g, δ/2)
4:     return fast_distance_mon(f̂, ĝ)
5: end function
```

Algorithmus 4.1 ist sogar dann sinnvoll, wenn f bereits in $\mathbb{D}_c$ liegt. Besitzt f nämlich viele kleine Sprünge, so kann man durch die geschickte Wahl von ε die Anzahl der Sprungstellen mitunter deutlich reduzieren und damit Einfluss auf die Laufzeit der Auswertung der Skorokhodmetrik nehmen.

Mit geringem Aufwand können wir die für monoton wachsende Funktionen entwickelten Algorithmen zur Auswertung der Skorokhodmetrik auf monotone Funktionen in $\mathbb{D}$ erweitern.

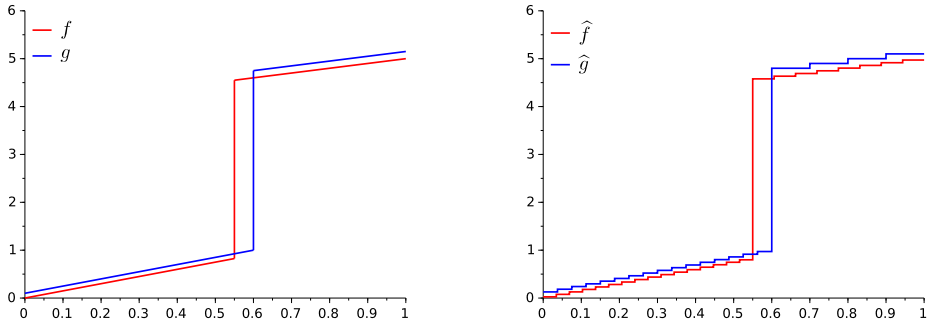


Abbildung 4.1: Die Beispielfunktionen f, g sowie deren Approximationen mittels `approx_function` mit $\varepsilon = 0.05$

Satz 4.2.2. *Seien $f, g \in \mathbb{D}$ monoton.*

(i) Falls sowohl f als auch g monoton steigen, berechnet der Algorithmus `approx_metric`(f, g, δ) *einen approximativen Wert für $d_S(f, g)$ mit dem maximalen Fehler $\delta > 0$.*

(ii) Falls sowohl f als auch g monoton fallen, berechnet der Algorithmus `approx_metric`$(-f, -g, \delta)$ *einen approximativen Wert für $d_S(f, g)$ mit dem maximalen Fehler $\delta > 0$.*

(iii) In allen anderen Fällen gilt

$$d_S(f, g) = |f(0) - g(0)| \vee |f(1) - g(1)|.$$

Beweis. Teil (i) haben wir bereits gezeigt, Teil (ii) folgt mit (2.2.3). Wegen der Symmetrie von d_S können wir o.B.d.A. annehmen, dass f monoton wachsend und g monoton fallend ist. Setzen wir

$$r = |f(0) - g(0)| \vee |f(1) - g(1)|,$$

so ist die Ungleichung

$$|f(t_1) - g(t_2)| \leqslant r$$

für alle $t_1, t_2 \in [0, 1]$ erfüllt und damit gilt $\|f - g \circ \lambda\|_\infty \leqslant r$ für alle $\lambda \in \Lambda$. Andererseits ist für alle $\lambda \in \Lambda$ auch $\|f - g \circ \lambda\|_\infty \geqslant r$ und es folgt (iii). □

Nun wollen wir uns einem Beispiel zuwenden. Die Funktionen f und g seien

	δ			
	1×10^{-1}	5×10^{-2}	2×10^{-2}	1×10^{-2}
$N_{\widehat{f}}$	24	48	96	192
$N_{\widehat{g}}$	20	40	96	192
$\|f - \widehat{f}\|_\infty$	2.8×10^{-2}	1.4×10^{-2}	7.0×10^{-3}	3.5×10^{-3}
$\|g - \widehat{g}\|_\infty$	5.0×10^{-2}	2.5×10^{-2}	6.2×10^{-3}	3.1×10^{-3}
$\|f - \widehat{f}\|_\infty + \|g - \widehat{g}\|_\infty$	7.8×10^{-2}	3.9×10^{-2}	1.3×10^{-2}	6.6×10^{-3}
$\lvert d_S(f,g) - d_S(\widehat{f},\widehat{g})\rvert$	2.2×10^{-2}	1.1×10^{-2}	7.8×10^{-4}	3.9×10^{-4}
	δ			
	5×10^{-3}	2×10^{-3}	1×10^{-3}	5×10^{-4}
$N_{\widehat{f}}$	384	768	1 536	3 072
$N_{\widehat{g}}$	384	768	1 536	3 072
$\|f - \widehat{f}\|_\infty$	1.8×10^{-3}	8.8×10^{-4}	4.4×10^{-4}	2.2×10^{-4}
$\|g - \widehat{g}\|_\infty$	1.6×10^{-3}	7.8×10^{-4}	3.9×10^{-4}	2.0×10^{-4}
$\|f - \widehat{f}\|_\infty + \|g - \widehat{g}\|_\infty$	3.3×10^{-3}	1.7×10^{-3}	8.3×10^{-4}	4.1×10^{-4}
$\lvert d_S(f,g) - d_S(\widehat{f},\widehat{g})\rvert$	2.0×10^{-4}	9.8×10^{-5}	4.9×10^{-5}	2.4×10^{-5}

Tabelle 4.2: Fehlertabelle zur Approximation des Skorokhodabstands von f und g durch `approx_metric`(f, g, δ)

gegeben durch

$$\begin{aligned} f\colon [0,1] &\to \mathbb{R} \\ t &\mapsto \begin{cases} 1.5t, & t < 0.55, \\ t + 4, & \text{sonst}, \end{cases} \end{aligned} \tag{4.2.1}$$

und

$$\begin{aligned} g\colon [0,1] &\to \mathbb{R} \\ t &\mapsto \begin{cases} 1.5t + 0.1, & t < 0.6, \\ t + 4.15, & \text{sonst}. \end{cases} \end{aligned} \tag{4.2.2}$$

L. Contreras hat in [2] nachgewiesen, dass der Abstand der in (4.2.1) und

(4.2.2) definierten Funktionen im Sinne der Skorokhodmetrik 0.2 beträgt. Die Graphen von f und g sowie deren Approximationen

$$\widehat{f} = \texttt{approx\_function}(f, 0.05)$$

und

$$\widehat{g} = \texttt{approx\_function}(g, 0.05)$$

sind in Abbildung 4.1 dargestellt. Die Skorokhodmetrik wurde für eine Reihe vorgegebener Werte für δ durch die Funktion $\texttt{approx\_metric}(f, g, \delta)$ approximiert und die Fehler wurden in Tabelle 4.2 notiert. $N_{\widehat{f}}$ bezeichnet die Anzahl der durch $\texttt{approx\_function}(f, 0.05)$ berechneten Stützstellen, analog ist die Bedeutung von $N_{\widehat{g}}$. In jeder Auswertung lag dieser Fehler auch tatsächlich unter der vorgegebenen Schwelle in Höhe von δ.

4.3 Nicht-monotone càdlàg-Funktionen

Für nicht-monotone càdlàg-Funktionen ist es nicht möglich, ein allgemeines Approximationsschema mit vorgegebenem maximalen Fehler wie in Abschnitt 4.2 zu entwickeln. Um einen solchen Fehler berechnen zu können, müsste man weitere Annahmen über die Regularität der zu approximierenden Funktionen treffen. Ohne weitere Einschränkungen an diese Funktionen ist es nicht sinnvoll einen Algorithmus zu entwickeln, da man keine Aussage zur Fehlerabschätzung machen kann. Wir geben uns an dieser Stelle damit zufrieden, anhand eines weiteren Beispiels aus [2] die Funktionstüchtigkeit der hergeleiteten Algorithmen zu testen. Wir werden nun ein denkbar simples Approximationsschema nutzen, bei dem wir jedoch i.A. keinen Fehler angeben können. Es sei eine Funktion $f \in \mathbb{D}$ vorgelegt. Wir unterteilen $[0, 1]$ in M Intervalle gleicher Länge und erhalten so die Stützpunkte. Wir geben uns ein $\varepsilon > 0$ vor und fügen den bisherigen Stützstellen diejenigen Stellen hinzu, an denen f um mindestens ε springt. Die Funktionswerte der approximierenden Funktionen sollen an diesen Stellen mit f übereinstimmen. Die Intervallstücke sind somit auf der linken Seite am Graphen von f fixiert. Wir definieren

$$\begin{aligned} f\colon [0,1] &\to \mathbb{R} \\ t &\mapsto \begin{cases} 1.25t, & t < 0.4, \\ -t + 1.9, & \text{sonst}, \end{cases} \end{aligned}$$

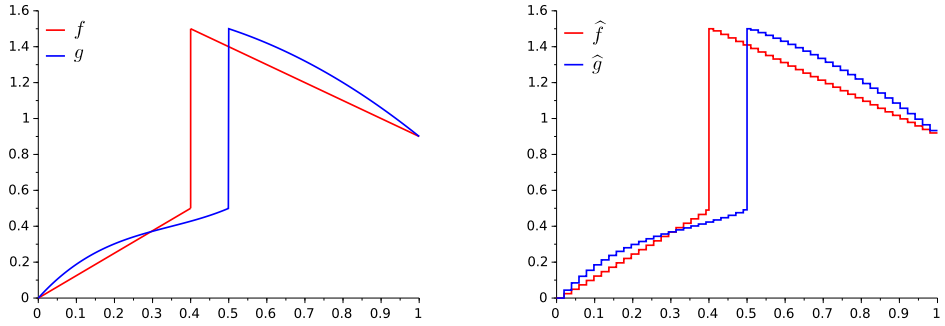


Abbildung 4.3: Die Beispielfunktionen f, g sowie deren Approximationen mit je 51 Stützstellen

sowie

$$g\colon [0,1] \to \mathbb{R}$$
$$t \mapsto \begin{cases} 5t^3 - 5.2t^2 + 2.35t, & t < 0.5, \\ -t^2 + 0.3t + 1.6, & \text{sonst.} \end{cases}$$

L. Contreras hat gezeigt, dass $d_S(f, g) = 0.1$ gilt. Die Graphen von f und g sowie deren Approximationen sind in Abbildung 4.3 dargestellt. $d_S(\hat{f}, \hat{g})$ wurde für mehrere N sowohl mit `distance_rand` als auch mit `distance_fast` berechnet. Abbildung 4.4 zeigt den Fehler in (4.1.1) in Abhängigkeit der Anzahl der Stützstellen. Für $N < 9$ stimmen die Werte von `distance_rand` und `distance_fast` überein, so dass im Schaubild in diesem Bereich lediglich der Graph von `distance_fast` sichtbar ist.

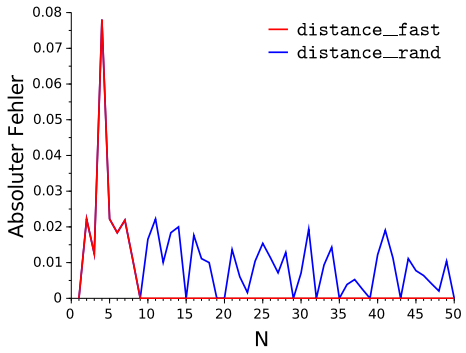


Abbildung 4.4: Absoluter Fehler in Abhängigkeit der Anzahl der Stützstellen

Kapitel 5

Fazit und Ausblick

Wir haben zunächst in Abschnitt 2.3 eine Formel zur Berechnung der Skorokhodmetrik für stückweise konstante Pfade hergeleitet. Anschließend wurde basierend auf dieser Formel in Kapitel 3 eine Reihe von Algorithmen zur Auswertung entwickelt.

Für die Algorithmen `distance` und `distance_fast` haben wir Totalordnungen eingeführt, um die Auswertungen so effizient wie möglich zu gestalten. Darüber hinaus konnte `distance_fast` sogar so weit optimiert werden, dass dieser Algorithmus lediglich polynomielle Laufzeit sowie Speicherkomplexität benötigt, um den Abstand zweier Funktionen $f, g \in \mathbb{D}_c$ im Sinne der Skorokhodmetrik exakt zu berechnen.

In Abschnitt 3.3 haben wir uns intensiv mit Verteilungen auf Teilmengen des $\mathbb{R}^d$ und Verteilungen auf Potenzmengen auseinandergesetzt. Hierbei wurden auch Algorithmen zur Generierung von Samples entwickelt. Mit diesen Ergebnissen war es nun möglich, einen randomisierten Algorithmus zur Berechnung von (2.3.1) herzuleiten.

Ausführliche Tests und Vergleiche in den Abschnitten 3.4 sowie 4.2–4.3 haben gezeigt, dass die vorgeschlagene Herangehensweise tatsächlich funktioniert und plausible Ergebnisse liefert. In diesem Rahmen konnte ein Approximationsschema für monotone càdlàg-Funktionen etabliert und getestet werden. Es wurde nachgewiesen, dass man solche Funktionen mit beliebiger Genauigkeit approximieren kann.

Offen geblieben ist die Frage, inwieweit die Wahl anderer Totalordnungen Einfluss auf die Effizienz der Algorithmen hat. Wenn es gelänge, die Formel in (2.3.1) auf die Klasse der stückweise affin linearen Funktionen zu verallgemeinern und diese auszuwerten, könnte die Approximationsgenauigkeit deutlich verbessert werden.

Anhang A

Geordnete Mengen

Die Sätze und Definitionen dieses Abschnitts sind den Kapiteln 1 und 4 des Buches von E. Harzheim [8] entnommen.

A.1 Einführung

Definition A.1.1. *Sei S ein Menge. Eine* Relation auf S *ist eine Teilmenge $\mathscr{R} \subseteq S \times S$. Eine weitere Relation $\mathscr{R}'$ heißt* Teilrelation von $\mathscr{R}$, *falls $\mathscr{R}' \subseteq \mathscr{R}$ gilt.*

Falls für $x, y \in S$ auch $(x, y) \in \mathscr{R}$ gilt, so schreiben wir $x\mathscr{R}y$. $\mathscr{R}'$ ist genau dann Teilrelation von $\mathscr{R}$, wenn

$$x\mathscr{R}'y \Longrightarrow x\mathscr{R}y$$

für alle $x, y \in S$ erfüllt ist.

Definition A.1.2. *Eine Relation $\mathscr{R}$ auf einer Menge S heißt*

(i) reflexiv, *falls $x\mathscr{R}x$ für alle $x \in S$ gilt,*

(ii) irreflexiv, *falls kein $x \in S$ mit $x\mathscr{R}x$ existiert,*

(iii) antisymmetrisch, *falls $(x\mathscr{R}y \wedge y\mathscr{R}x) \Longrightarrow x = y$ für alle $x, y \in S$ gilt,*

(iv) transitiv, *falls $(x\mathscr{R}y \wedge y\mathscr{R}z) \Longrightarrow x\mathscr{R}z$ für alle $x, y, z \in S$ gilt,*

(v) pseudototal, *falls entweder $x\mathscr{R}y$ oder $y\mathscr{R}x$ für alle $x, y \in S$ mit $x \neq y$ gilt.*

Definition A.1.3. *Eine Relation $\mathscr{R}$ auf einer Menge S heißt*

(i) Partialordnung, *wenn sie reflexiv, transitiv und antisymmetrisch ist,*

(ii) strenge Partialordnung, *wenn sie irreflexiv und transitiv ist,*

(iii) (strenge) Totalordnung, *wenn sie eine pseudototale (strenge) Partialordnung ist.*

Es gilt zu beachten, dass eine strenge Partialordnung automatisch antisymmetrisch ist. Aus einer strengen Partialordnung lässt sich leicht eine Partialordnung konstruieren. Gleiches gilt für Totalordnungen.

Satz A.1.4. *Sei $<$ eine strenge Partialordnung (strenge Totalordnung) auf einer Menge S. Dann ist die durch*

$$a \leqslant b \Longleftrightarrow (a = b) \vee (a < b), \qquad a, b \in S, \tag{A.1.1}$$

definierte Relation $\leqslant$ eine Partialordnung (Totalordnung) auf S.

Beweis. Sei $<$ eine strenge Partialordnung. Die Relation $\leqslant$ in A.1.1 ist per Definition reflexiv und es bleiben lediglich Antisymmetrie und Transitivität nachzuweisen. Seien $x, y \in S$ mit $x \leqslant y$ und $y \leqslant x$. Wegen der Transitivität und der Irreflexivität von $<$ können nicht sowohl $x < y$ als auch $y < x$ gelten und es folgt $x = y$. Seien $x, y, z \in S$ mit $x \leqslant y$ und $y \leqslant z$. Für $x < y$ und $y < z$ folgt mit der Transitivität von $<$ sofort $x < z$ und wir erhalten $x < z$ oder $x = z$. Somit ist $\leqslant$ eine Partialordnung. Falls $<$ sogar eine strenge Totalordnung ist, gilt es lediglich zu berücksichtigen, dass sich die Pseudototaliät der Relation $<$ mittels (A.1.1) direkt auf $\leqslant$ überträgt. □

Ebenso können wir umgekehrt aus einer Partialordnung eine strenge Partialordnung konstruieren. Auch dies gilt in gleicher Weise für Totalordnungen.

Satz A.1.5. *Sei $\leqslant$ eine Partialordnung (Totalordnung) auf einer Menge S. Dann ist die durch*

$$a < b \Longleftrightarrow (a \leqslant b) \wedge (a \neq b), \qquad a, b \in S, \tag{A.1.2}$$

definierte Relation $<$ eine strenge Partialordnung (strenge Totalordnung) auf S.

Wir bezeichnen das Tupel $(S, \mathscr{R})$ als streng partiell geordnete (resp. streng totalgeordnete, partiell geordnete, totalgeordnete) Menge, wenn $\mathscr{R}$ eine strenge Partialordnung (resp. strenge Totalordnung, Partialordnung, Totalordnung) auf S ist.

Definition A.1.6. *Sei $(V, \leqslant)$ eine partiell geordnete Menge mit $V \neq \varnothing$.*

(i) Falls V ein Element a besitzt, welches $a \leqslant v$ für alle $v \in V$ erfüllt, so heißt a Minimum *von V. Es wird mit* $\min V$ *bezeichnet.*

(ii) Falls V ein Element a besitzt, welches $a \geqslant v$ für alle $v \in V$ erfüllt, so heißt a Maximum *von V. Es wird mit* $\max V$ *bezeichnet.*

Minimum und Maximum einer partiell geordneten Menge sind im Falle ihrer Existenz eindeutig bestimmt. Seien $a, b \in V$ mit $a = \min V$ und $b = \min V$. Dann gilt insbesondere $a \leqslant b$ sowie $b \leqslant a$. Schließlich erzwingt die Antisymmetrie die Gleichheit von a und b. Es ist klar, dass jede endliche, totalgeordnete Menge sowohl ein Minimum als auch ein Maximum besitzt.

Definition A.1.7. *Eine Totalordnung heißt* Wohlordnung, *falls jede nichtleere Teilmenge ein Minimum besitzt.*

A.2 Ordnungen auf kartesischen Produkten

Im Folgenden werden wir Relationen auf kartesischen Produkten geordneter Mengen definieren.

Definition A.2.1. *Sei I eine Menge, $(V_i, \leqslant_i)_{i \in I}$ eine Familie von Partialordnungen. Die durch*

$$a \leqslant b \Longleftrightarrow \bigwedge_{i \in I} (a_i \leqslant_i b_i), \qquad a, b \in \bigtimes_{i \in I} V_i,$$

definierte Relation $\leqslant$ auf dem kartesischen Produkt $\times_{i \in I} V_i$ heißt komponentenweise Ordnung.

Es ist unmittelbar aus der Definition ersichtlich, dass die komponentenweise Ordnung eine Partialordnung ist. Falls es sich bei der Indexmenge I in Definition A.2.1 um eine Wohlordnung handelt, können wir mit folgender Konstruktion stets eine Ordnung auf dem kartesischen Produkt einführen.

Definition A.2.2. *Sei $(I, \prec)$ eine Wohlordnung, $(V_i, \leqslant_i)_{i \in I}$ eine Familie von Partialordnungen. Dann definieren wir auf dem kartesischen Produkt $V = \times_{i \in I} V_i$ eine Relation $\leqslant$ wie folgt:*
Für $a, b \in V$ mit $a \neq b$ sei $j = \min\{i \in I \mid a_i \neq b_i\}$ und wir setzen

$$a < b \Longleftrightarrow a_j <_j b_j.$$

Die durch

$$a \leqslant b \Longleftrightarrow (a = b) \vee (a < b), \qquad a, b \in V,$$

definierte Relation wird lexikographische Ordnung auf V *genannt.*

Satz A.2.3. *Die in Definition A.2.2 eingeführte Relation ist eine Partialordnung.*

Satz A.2.4. *Sei $(I, \prec)$ eine endliche, wohlgeordnete Menge, $(V_i, \leqslant_i)_{i \in I}$ eine Familie von wohlgeordneten Mengen. Dann ist die lexikographische Ordnung auf dem kartesischen Produkt $\times_{i \in I} V_i$ eine Wohlordnung.*

Satz A.2.5. *Sei $(I, \prec)$ eine Wohlordnung, $(V_i, \leqslant_i)_{i \in I}$ eine Familie von Partialordnungen, $V = \times_{i \in I} V_i$. Dann ist die komponentenweise Ordnung auf V eine Teilrelation der lexikographischen Ordnung auf V.*

Anhang B

Implementation in C++

Es werden in diesem Kapitel stets zuerst die Headerdatei und dann – sofern vorhanden – die .cpp-Datei aufgelistet.

B.1 Die Klasse `StepFunction`

Die Klasse `StepFunction` beschreibt eine stückweise konstante, rechtsstetige, reellwertige Funktion auf dem Intervall $[0, 1]$, welche in 1 linksstetig ist. Die Sprungstellen sowie deren Funktionswerte sind getrennt in `_xData` und `_yData` gespeichert. Beide müssen die gleiche Länge haben. Die Einträge in `_xData` müssen die Randpunkte 0 und 1 enthalten und geordnet sein. Der letzte Eintrag von `_yData` sollte (wegen der Linksstetigkeit in 1) gleich dem vorletzten Wert sein.

Listing B.1: step_function.h

```
1  #ifndef STEPFUNCTION
2  #define STEPFUNCTION
3
4  #include <iomanip> // setw, setfill
5  #include <iostream>
6  #include <random>
7  #include <vector>
8  #include <string>
9  #include <algorithm> // sort()
10
11
12 class StepFunction {
13 // represents a right-continuous piecewise constant function on [0,1]
```

```
14 // which is left-continuous at 1
15    private:
16        std::string _name;
17
18        std::vector<double> _xData;
19        std::vector<double> _yData;
20
21    public:
22        StepFunction(const std::string& name) : _name(name) {}
23        void set_xyData(const std::vector<double>& xData,
24          const std::vector<double>& yData);
25        inline const std::string get_name() const { return _name; }
26        inline const std::vector<double>& get_xData() const
27          { return _xData; }
28        inline const std::vector<double>& get_yData() const
29          { return _yData; }
30        inline const size_t size() const { return _xData.size(); }
31        // size = number of jumps + 2
32
33        void print() const;
34
35        template <typename distribution_type, typename generator_type>
36        void generate(size_t n, distribution_type& jump_size_distr,
37          generator_type& rng);
38 };
39
40
41 template <typename distribution_type, typename generator_type>
42 void StepFunction::generate(size_t n, distribution_type& jump_size_distr,
43   generator_type& rng) {
44
45    // n jumps plus initial value plus final value
46    _xData.resize(n+2);
47    _yData.resize(n+2);
48    // generate initial value
49    _xData[0] = 0.0;
50    _yData[0] = 0.0;
51
52    // distribution for jumping times: default uniform [0,1)
53    std::uniform_real_distribution<double> distr_uni;
54
55    for ( size_t i=1; i<=n; ++i ) {
56        _xData[i] = distr_uni(rng);
57        _yData[i] = _yData[i-1] + jump_size_distr(rng);
58    }
59    _xData[n+1] = 1.0;
60    std::sort(_xData.begin(), _xData.end());
61    _yData[n+1] = _yData[n];
62 }
```

```
63
64
65 #endif // STEPFUNCTION
```

Listing B.2: step_function.cpp

```
1  #include "step_function.h"
2
3
4  void StepFunction::set_xyData(const std::vector<double>& xData,
5    const std::vector<double>& yData) {
6      if ( xData.size() != yData.size() )
7          std::cout << "error setting data for function " << _name
8            << ": dimensions not matching!" << std::endl;
9      else {
10         _xData = xData;
11         _yData = yData;
12     }
13 }
14
15
16 void StepFunction::print() const {
17 // prints jumping times and the function values
18
19     std::cout.setf(std::ios::fixed, std::ios::floatfield);
20     std::cout.precision(4);
21
22     std::cout << "FUNCTION VALUES FOR " << _name << std::endl;
23     std::cout << "index time  value" << std::endl;
24     std::cout << std::setfill('=') << std::setw(22) << "=" << std::endl;
25
26     for ( size_t i=0; i<_xData.size(); ++i ) {
27         std::cout << std::setfill(' ') << std::setw(5) << i << " "
28              << _xData[i] << " " << std::setw(7) << _yData[i] << std::endl;
29     }
30 }
```

B.2 Die Klasse Kombos

Listing B.3: kombos.h

```
1  #ifndef KOMBOS_H
2  #define KOMBOS_H
3
4  #include <vector>
5
6
```

```
7  // ABSTRACT CLASS
8  class Kombos_basic {
9      protected:
10         int N_f;   // >= 1!
11         int N_g;   // >= 1!
12
13         std::vector<int> k;
14         // k has length N_f+2 with k[0]=0 and k[N_f+1]=N_g+1
15
16         bool _stop;
17
18     public:
19         virtual void init(const int N_f, const int N_g) = 0;
20         virtual void clear() = 0;
21
22         inline const std::vector<int>& get_k() const { return k; }
23         inline const bool stopped() const { return _stop; }
24 };
25
26
27 class Kombos_simple: public Kombos_basic {
28     public:
29         ~Kombos_simple() { this->clear(); }
30         Kombos_simple& operator++();
31
32         virtual void init(const int N_f, const int N_g);
33         virtual inline void clear() { k.clear(); }
34 };
35
36
37 class Kombos: public Kombos_basic {
38     private:
39         std::vector<int> k_max; // upper boundary for k
40         // k_max has length N_f+2 with k[0]=0 and k[N_f+1]=N_g+1
41         std::vector<bool> R;
42
43         bool _changed;
44
45         // dynamic 2-dimensional (N_f+1)x(N_g+1) arrays
46         int** _a;
47         int** _b;
48         bool** _adj_matrix;
49         int** _restr_matrix;
50         // each entry contains the info:
51         // -1:  => i \notin R
52         // 0:  i not in restriction list
53         // +1:  => i \in R
54
55         void invalidate_element(int i, int j);
```

```
56          void increase_kmin(int m, int n);
57
58      public:
59          ~Kombos() { this->clear(); }
60          virtual void init(int N_f, int N_g);
61          virtual void clear();
62
63          void adj_cond(int m, int n);
64          void set_lower_bound(int m, int n);
65          void set_upper_bound(int m, int n);
66          void erase(int m, int n);
67          void compute_R();
68          void restr_cond(int restr_type, int m, int n);
69
70          // GETTER METHODS
71          inline const std::vector<bool>& get_R() const { return R; }
72          inline bool changed() const { return _changed; }
73
74          // SETTER METHODS
75          void reset() { _changed = false; }
76          void stop() { _stop = true; }
77  };
78
79
80  #endif // KOMBOS_H
```

Listing B.4: kombos.cpp

```
1   #include "kombos.h"
2
3
4   void Kombos_simple::init(const int N_f, const int N_g) {
5       this->N_f = N_f;
6       this->N_g = N_g;
7
8       _stop = false;
9
10      // initialize k
11      k.resize(N_f+2);
12      k[0] = 0;
13      for ( std::size_t i=1; i<=N_f; ++i )
14          k[i] = 0;
15      k[N_f+1] = N_g+1;
16  }
17
18
19  Kombos_simple& Kombos_simple::operator++() {
20  // increments the current k
21      int pos = N_f;
```

```
22      while ( pos > 0 and k[pos] == N_g ) { --pos; }
23
24      if ( pos > 0 ) {
25          ++k[pos];
26          for ( int i=pos+1; i<=N_f; ++i )
27              k[i] = k[pos];
28      } else {
29          _stop = true;
30      }
31
32      return *this;
33  }
34
35
36  void Kombos::compute_R() {
37      for ( int i=0; i<N_f; ++i ) {
38          if ( _restr_matrix[i+1][k[i+1]] > 0
39            or ( _restr_matrix[i+1][k[i+1]] == 0
40            and k[i] == k[i+1] ) )
41              R[i] = true;
42          else
43              R[i] = false;
44      }
45  }
46
47
48  void Kombos::erase(int m, int n) {
49  // erase (m,n), adjust bounds if possible
50      if ( k[m] == n )
51          set_lower_bound(m, n + 1);
52      else if ( k_max[m] == n )
53          // invalid value touches upper bound
54          set_upper_bound(m, n - 1);
55      else if ( k[m] < n and n < k_max[m] )
56          invalidate_element(m, n);
57  }
58
59
60  void Kombos::restr_cond(int restr_type, int m, int n) {
61  // restr_type \in {-1,1}
62  // handles restriction on R:
63  // if restr_type = -1 k_m = n ==> m \in R
64  // if restr_type = 1 k_m = n ==> m \notin R
65      if ( _restr_matrix[m][n] == restr_type )
66          erase(m, n);
67      else {
68          _restr_matrix[m][n] = -restr_type;
69          if ( k[m] == n )
70              // restr_type -1 => true, restr_type 1 => false
```

```
71              R[m-1] = !!(restr_type-1);
72      }
73  }
74
75
76  void Kombos::adj_cond(int m, int n) {
77  // 1 <= m <= N_f-1
78  // 0 <= n <= N_g
79      if ( n < k[m+1] )
80          set_lower_bound(m, n);
81      else if ( k_max[m] < n )
82          set_upper_bound(m+1, n);
83      else if ( k[m] < n and n < k_max[m+1] )
84          _adj_matrix[m][n] = true;
85  }
86
87
88  void Kombos::invalidate_element(int i, int j) {
89  // 0 <= i <= N_f, 0 <= j <= N_g
90      // update _a
91      if ( j < N_g ) {
92          _a[i][j] = _a[i][j+1];
93      } else {
94          _a[i][j] = N_g+1;
95      }
96      // update entries pointing to _a[i][j]
97      int l = j-1;
98      while ( l >= 0 and _a[i][l] == j )
99          _a[i][l--] = _a[i][j+1];
100
101     // update b
102     if ( j > 0 ) {
103         _b[i][j] = _b[i][j-1];
104     } else
105         _b[i][j] = -1;
106
107     // update entries pointing to _b[i][j]
108     l = j+1;
109     while ( l <= N_g and _b[i][l] == j )
110         _b[i][l++] = _b[i][j-1];
111 }
112
113
114 void Kombos::increase_kmin(int m, int n) {
115     if ( n > k_max[m] )
116         _stop = true;
117     else
118         k[m] = n;
119 }
```

```
120
121
122 void Kombos::set_lower_bound(int m, int n) {
123 // IMPORTANT: 0 <= n <= N_g due to the matrix _a
124     if ( n > k_max[m] )
125         _stop = true;
126     else if ( k[m] < n ) {
127         // otherwise new condition is already fulfilled
128
129         _changed = true;
130         increase_kmin(m, _a[m][n]);
131         bool _changed = true;
132         int i = m;
133         int alpha = m;
134         int beta = m;
135         while ( _changed and !_stop ) {
136             _changed = false;
137
138             while ( i < N_f and !_stop
139               and (k[i+1] < _a[i+1][k[i]] or i < beta) ) {
140                 ++i;
141                 if ( k[i] < _a[i][k[i-1]] ) {
142                     increase_kmin(i, _a[i][k[i-1]]);
143                     _changed = true;
144                 }
145             }
146
147             if ( !_stop ) {
148                 if ( !_changed )
149                     i = alpha;
150                 else {
151                     beta = i;
152                     _changed = false;
153                 }
154                 int j = k[i]-1;
155                 while ( j > k[i-1] and !_adj_matrix[i-1][j] )
156                   { --j; }
157                 while ( i > 1 and !_stop
158                   and ( j > k[i-1] or alpha < i ) ) {
159                     --i;
160                     if ( j > k[i] ) {
161                         increase_kmin(i, _a[i][j+1]);
162                         _changed = true;
163                     }
164                     j = k[i]-1;
165                     while ( j > k[i-1] and !_adj_matrix[i-1][j] )
166                       { --j; }
167                 }
168                 alpha = i;
```

```
169             }
170         }
171     }
172 }
173 
174 
175 void Kombos::set_upper_bound(int m, int n) {
176     if ( n < k[m] )
177         _stop = true;
178     else if ( n < k_max[m] )
179         // otherwise new condition is already fulfilled
180         k_max[m] = _b[m][n];
181 }
182 
183 
184 void Kombos::init(int N_f, int N_g) {
185     this->N_f = N_f;
186     this->N_g = N_g;
187 
188     _stop = false;
189     _changed = true;
190 
191     // initialize k and k_max
192     k.resize(N_f+2);
193     k_max.resize(N_f+2);
194     k[0] = 0;
195     k_max[0] = 0;
196     for ( int i=1; i<=N_f; ++i ) {
197         k[i] = 0;
198         k_max[i] = N_g;
199     }
200     k[N_f+1] = N_g+1;
201     k_max[N_f+1] = N_g+1;
202 
203     R.resize(N_f);
204 
205     // create dynamic 2-dimensional (N_f+1)x(N_g+1) arrays
206     _a = new int *[N_f+1];
207     _b = new int *[N_f+1];
208     _restr_matrix = new int *[N_f+1];
209     _adj_matrix = new bool *[N_f+1];
210     for ( int i=0; i<=N_f; ++i ) {
211         _a[i] = new int[N_g+1];
212         _b[i] = new int[N_g+1];
213         _restr_matrix[i] = new int[N_g+1];
214         _adj_matrix[i] = new bool[N_g+1];
215 
216         for ( int j=0; j<=N_g; ++j ) {
217             _a[i][j] = j;
```

```
218             _b[i][j] = j;
219             _restr_matrix[i][j] = 0;
220             _adj_matrix[i][j] = false;
221         }
222     }
223 }
224
225
226 void Kombos::clear() {
227     k.clear();
228     k_max.clear();
229     R.clear();
230
231     // delete all dynamic 2-dimensional arrays
232     if ( _a != nullptr ) {
233         for ( int i=0; i<=N_f; ++i )
234             delete[] _a[i];
235         delete[] _a;
236         _a = nullptr;
237     }
238     if ( _b != nullptr ) {
239         for ( int i=0; i<=N_f; ++i )
240             delete[] _b[i];
241         delete[] _b;
242         _b = nullptr;
243     }
244     if ( _adj_matrix != nullptr ) {
245         for ( int i=0; i<=N_f; ++i )
246             delete[] _adj_matrix[i];
247         delete[] _adj_matrix;
248         _adj_matrix = nullptr;
249     }
250     if ( _restr_matrix != nullptr ) {
251         for ( int i=0; i<=N_f; ++i )
252             delete[] _restr_matrix[i];
253         delete[] _restr_matrix;
254         _restr_matrix = nullptr;
255     }
256 }
```

B.3 Die Klasse SkoroDistance

Listing B.5: skoro_distance.h

```
1 #ifndef SKORO_DISTANCE_H
2 #define SKORO_DISTANCE_H
3
4 #include <vector>
```

```
5  #include <cmath>      // abs()
6  #include <algorithm>  // min(), max()
7  #include "./step_function.h"
8  #include "./power_set.h"
9  #include "./kombos.h"
10 #include "./randomized.h"
11 #include <chrono>     // "random" seed
12 #include <random>     // mt_19937_64, uniform_int_distribution
13
14
15 enum id_type {s_plus, s_minus, d};
16
17 struct info_type {
18     id_type id;
19     int i;
20     int j;
21     double value;
22
23     //for sorting
24     //comparing two info_type elements means comparing their value
25     inline bool operator<(const info_type& min) const
26       { return value < min.value; }
27 };
28
29
30 class SkoroDistance {
31     private:
32         const StepFunction& _f;
33         const StepFunction& _g;
34
35         // dynamic 2-dimensional arrays
36         double **_d;   // d_{i,j}
37         double **_S;   // \delta s_{i,j}
38         std::vector<info_type> _itemlist;
39         // list of all possible results
40
41         double eta(const std::vector<int>& k) const;
42         // evaluates
43         // max_{i \in A} max_{k_i <= j < k_(i+1)} d_{i,j}
44         // \lor max_{i=1,..N_f} \Delta s_{i,k_i}^{-}
45         double kappa(const std::vector<int>& k,
46           const std::vector<bool>& R) const;
47         // evaluates
48         // (max_{i \in R} ( d_{i-1,k_i} \lor \Delta s_{i,k_i+1}^{+} )
49         // \lor (max_{i \in CR} \Delta s_{i,k_i}^{+}
50
51     public:
52
53         SkoroDistance(const StepFunction& f, const StepFunction& g)
```

```
54          : _f(f), _g(g) {}
55        // only establishes the connection between the step functions
56        // and the SkoroDistance class
57        ~SkoroDistance() { clear_data(); };
58        void init();
59        // computes the arrayys _d and _S and creates the itemlist
60
61        void clear_data();
62
63        double compute() const;      // Algorithm distance_fast
64        double compute_simple() const; // Algorithm distance
65
66        template <typename distribution, typename generator_type>
67        double randomized(const int sample_size,
68          generator_type& generator) const;
69        // Algorithm distance_rand
70 };
71
72
73 template <typename distribution, typename generator_type>
74 double SkoroDistance::randomized(const int sample_size,
75   generator_type& generator) const {
76     double cur_min = _itemlist[0].value;
77     double phi_val;
78     std::vector<int> k;
79     std::vector<bool> R;
80
81     int W = std::max(_f.size(),_g.size())-2;
82     GenSamples_Z_Uni<generator_type> gel(generator, W, W);
83     gel.init(_f.size()-2, _g.size()-2);
84
85     for ( int r=0; r<sample_size; ++r ) {
86         gel.gen_element(k, R);
87         phi_val = std::max(eta(k), kappa(k, R));
88         if ( phi_val < cur_min )
89             cur_min = phi_val;
90     }
91
92     return cur_min;
93 }
94
95
96 #endif // SKORO_DISTANCE_H
```

Listing B.6: skoro_distance.cpp

```
1 #include "skoro_distance.h"
2
3
```

```
4  void SkoroDistance::init() {
5      // max(d_{0,0},d_{N_f,N_g})
6      double dS_lower_bound
7        = std::max( std::abs(_f.get_yData()[0] - _g.get_yData()[0]),
8        std::abs(_f.get_yData()[_f.size()-2]
9        - _g.get_yData()[_g.size()-2]) );
10
11     // communicate maximum size of list
12     _itemlist.reserve( (_f.size()-2)*(_g.size()-2)
13       + (_f.size()-1)*(_g.size()-1) );
14
15     // create dynamic 2-dimensional (N_f+2)x(N_g+2) arrays _d and _S
16     // and fill the arrays with the approriate values
17     // d_{i,j} = |f(s_i^f) - g(s_j^g)|, i=0,...,N_f+1, j=0,...,N_g+1
18     // S_{i,j} = s_i^f - s_j^g, i=0,...,N_f+1, j=0,...,N_g+1
19     _d = new double *[_f.size()];
20     _S = new double *[_f.size()];
21     for ( int i=0; i<_f.size(); ++i ) {
22         _d[i] = new double[_g.size()];
23         _S[i] = new double[_g.size()];
24         for ( int j=0; j<_g.size(); ++j ) {
25             _d[i][j] = std::abs(_f.get_yData()[i] - _g.get_yData()[j]);
26             _S[i][j] = _f.get_xData()[i] - _g.get_xData()[j];
27         }
28     }
29
30     // populate itemlist
31     for ( int i=1; i<_f.size()-1; ++i ) {
32         for ( int j=1; j<_g.size()-1; ++j ) {
33             if ( std::abs(_S[i][j]) >= dS_lower_bound ) {
34                 if ( _S[i][j] < 0.0 )
35                     _itemlist.emplace_back( info_type{s_minus,
36                       i, j, -_S[i][j]} );
37                 else if ( _S[i][j] > 0.0 )
38                     _itemlist.emplace_back( info_type{s_plus,
39                       i, j, _S[i][j]} );
40             }
41         }
42     }
43     for ( int i=0; i<_f.size()-1; ++i ) {
44         for ( int j=0; j<_g.size()-1; ++j ) {
45             if ( _d[i][j] >= dS_lower_bound )
46                 _itemlist.emplace_back( info_type{d, i, j, _d[i][j]} );
47         }
48     }
49
50     // sort itemlist in decreasing order
51     sort(_itemlist.begin(), _itemlist.end());
52     reverse(_itemlist.begin(), _itemlist.end());
```

```
53 }
54
55
56 double SkoroDistance::eta(const std::vector<int>& k) const {
57     double cur_max = 0.0;
58
59     for ( int i=0; i<_f.size()-2; ++i ) {
60         if ( k[i] < k[i+1] ) {
61             // \Delta s_{i,k_i}^{-} = max(0,-S_{i,k_i})
62             // note that the condition below implies S_{i,k_i}<0,
63             // since max_info.value has been initialized to 0
64             if ( -_S[i+1][k[i+1]] > cur_max )
65                 cur_max = -_S[i+1][k[i+1]];
66
67             for ( int j=k[i]; j<k[i+1]; ++j ) {
68                 if ( _d[i][j] > cur_max )
69                     cur_max = _d[i][j];
70             }
71         }
72     }
73
74     for ( int j=k[_f.size()-2]; j<=_g.size()-2; ++j ) {
75         if ( _d[_f.size()-2][j] > cur_max )
76             cur_max = _d[_f.size()-2][j];
77     }
78
79     return cur_max;
80 }
81
82
83 double SkoroDistance::kappa(const std::vector<int>& k,
84   const std::vector<bool>& R) const {
85
86     double cur_max = 0.0;
87
88     for (int i=0; i<R.size(); ++i) {
89         if ( R[i] ) {
90             // "i+1 \in R"
91             // d_{i,k_{i+1}}
92             if ( _d[i][k[i+1]] > cur_max )
93                 cur_max = _d[i][k[i+1]];
94             // \Delta s_{i+1,k_{i+1}+1}^{+} = max(0,S_{i+1,k_{i+1}+1})
95             if ( _S[i+1][k[i+1]+1] > cur_max )
96                 cur_max = _S[i+1][k[i+1]+1];
97         } else {
98             // "i+1 \in CR"
99             // \Delta s_{i+1,k_{i+1}}^{+} = max(0,S_{i+1,k_{i+1}})
100            if ( _S[i+1][k[i+1]] > cur_max )
101                cur_max = _S[i+1][k[i+1]];
```

```
102         }
103     }
104
105     return cur_max;
106 }
107
108
109 void SkoroDistance::clear_data() {
110     if ( _d != nullptr ) {
111         for ( int i=0; i<_f.size()-1; ++i )
112             delete[] _d[i];
113         delete[] _d;
114         _d = nullptr;
115     }
116     if ( _S != nullptr ) {
117         for ( int i=0; i<_f.size()-1; ++i )
118             delete[] _S[i];
119         delete[] _S;
120         _S = nullptr;
121     }
122
123     _itemlist.clear();
124 }
125
126
127 double SkoroDistance::compute() const {
128     int pos = 0;
129     // contains the position of current min in itemlist,
130     // 0 means the largest entry
131     double cur_min;
132     double eta_val;
133
134     Kombos kombi;
135     kombi.init(_f.size()-2, _g.size()-2);
136     while ( !kombi.stopped() ) {
137         if ( kombi.changed() ) {
138             eta_val = eta(kombi.get_k());
139             kombi.compute_R();
140
141             // set kmin_changed to false
142             kombi.reset();
143         }
144
145         cur_min = std::max(eta_val, kappa(kombi.get_k(), kombi.get_R()));
146
147         while ( pos < _itemlist.size() and !kombi.stopped()
148           and _itemlist[pos].value >= cur_min ) {
149             if ( _itemlist[pos].id == s_minus ) {
150                 kombi.set_upper_bound(_itemlist[pos].i,
```

```
151                 _itemlist[pos].j-1);
152             } else if ( _itemlist[pos].id == s_plus ) {
153                 kombi.set_lower_bound(_itemlist[pos].i,
154                   _itemlist[pos].j);
155                 kombi.restr_cond(-1, _itemlist[pos].i,
156                   _itemlist[pos].j);
157             } else {
158                 if ( _itemlist[pos].i == 0 ) {
159                     if ( _itemlist[pos].j == 0 )
160                         kombi.stop();
161                     else {
162                         kombi.set_upper_bound(1, _itemlist[pos].j);
163                         kombi.restr_cond(1, 1, _itemlist[pos].j);
164                     }
165                 } else if ( _itemlist[pos].i == _f.size()-2 ) {
166                     // i = N_f
167                     kombi.set_lower_bound(_itemlist[pos].i,
168                       _itemlist[pos].j + 1);
169                 } else {
170                     kombi.adj_cond(_itemlist[pos].i, _itemlist[pos].j);
171                     kombi.erase(_itemlist[pos].i, _itemlist[pos].j);
172                     kombi.restr_cond(1, _itemlist[pos].i + 1,
173                       _itemlist[pos].j);
174                 }
175             }
176             ++pos;
177         }
178     }
179
180     return cur_min;
181 }
182
183
184 double SkoroDistance::compute_simple() const {
185     PowSetCounter powset;
186
187     double cur_min;  // current minimum
188     cur_min = _itemlist[0].value;
189
190     Kombos_simple kombi;
191     kombi.init(_f.size()-2, _g.size()-2);
192     while ( !kombi.stopped() ) {
193         double eta_val = eta(kombi.get_k());
194
195         if ( eta_val < cur_min ) {
196             std::vector<int> min;
197             std::vector<int> cmin;
198             for ( int l=0; l<_f.size()-2; ++l ) {
199                 if ( kombi.get_k()[l] == kombi.get_k()[l+1] ) {
```

```
200                 min.push_back(l+1);
201             } else {
202                 cmin.push_back(l+1);
203             }
204         }
205         powset.init(_f.size()-2, min, cmin);
206 
207         while ( !powset.stopped() ) {
208             double kappa_val = kappa(kombi.get_k(),
209               powset.get_total_vec());
210             if ( kappa_val < cur_min )
211                 cur_min = std::max(kappa_val, eta_val);
212             ++powset;
213         }
214         powset.clear();
215     }
216     ++kombi;
217 }
218 
219 return cur_min;
220 }
```

B.4 Die GenSamples-Klassen

Listing B.7: randomized.h

```
1  #ifndef RANDOMIZED_H
2  #define RANDOMIZED_H
3  
4  
5  #include <iostream>
6  #include <cstdlib>    // exit()
7  #include <random>
8  #include <vector>
9  #include <iterator>
10 #include <list>
11 
12 
13 
14 struct Z_element_type {
15     std::vector<int> k;
16     std::vector<bool> R;
17 };
18 
19 
20 // ABSTRACT CLASS
21 template <typename generator_type, typename element_type>
22 class GenSamples {
```

```
23      protected:
24          generator_type& _generator;
25          std::list<std::vector<element_type>> _samples;
26
27      public:
28          GenSamples(generator_type& generator): _generator(generator) {}
29          virtual void generate(const size_t n) = 0;
30          // generates n independent samples
31          inline const std::list<std::vector<element_type>>& get_samples()
32            const { return _samples; }
33  };
34
35
36  // ABSTRACT CLASS
37  template <typename generator_type>
38  class GenSamples_Kombo: public GenSamples<generator_type, int> {
39      protected:
40          int _M;
41          int _N;
42
43      public:
44          GenSamples_Kombo(generator_type& generator)
45            : GenSamples<generator_type, int>(generator) {}
46          inline int get_M() { return _M; }
47          inline int get_N() { return _N; }
48          virtual void gen_element(std::vector<int>& element) const = 0;
49          // generates a single sample
50          void generate(const size_t n);
51          // generates n independent samples
52  };
53
54
55  template <typename generator_type>
56  class GenSamples_Kombo_Theta: public GenSamples_Kombo<generator_type> {
57      private:
58          double delta(const int a, const int b) const;
59          std::vector<std::vector<double>> _delta_matrix;
60          // CAUTION: _delta_matrix[i][j] contains
61          // delta(i+1,j), i=0,...,M-1, j=0,...,N
62      public:
63          GenSamples_Kombo_Theta(generator_type& generator,
64            const int M_MAX, const int N_MAX);
65          // computes the delta matrix with size M_MAX x N_MAX
66          void init(const int M, const int N);
67          void gen_element(std::vector<int>& k) const;
68          // generates a single sample
69  };
70
71
```

```
72  // ABSTRACT CLASS
73  template <typename generator_type>
74  class GenSamples_Powset: public GenSamples<generator_type, int> {
75      protected:
76          int _N;
77
78      public:
79          GenSamples_Powset(generator_type& generator)
80            : GenSamples<generator_type, int>(generator) {}
81          inline int get_N() { return _N; }
82          virtual void gen_element(std::vector<int>& v) const = 0;
83          // generates a single sample
84          void generate(const size_t n);
85          // generates n independent samples
86  };
87
88
89  template <typename generator_type>
90  class GenSamples_Powset_Uni: public GenSamples_Powset<generator_type> {
91      private:
92          std::vector<double> _cum_pmf_rho;
93          std::vector< std::vector<double> > _cum_pmf_matrix_rho;
94
95      public:
96          GenSamples_Powset_Uni(generator_type& generator,
97            const int N_MAX);
98          void init(const int N);
99          void gen_element(std::vector<int>& v) const;
100         // generates a single sample
101 };
102
103
104 // generates uniformly distributed samples in \cZ_N^M
105 template <typename generator_type>
106 class GenSamples_Z_Uni {
107     protected:
108         int _N;
109         int _M;
110         generator_type& _generator;
111         std::list<Z_element_type> _samples;
112         GenSamples_Kombo_Theta<generator_type> _gen_kombo;
113         GenSamples_Powset_Uni<generator_type> _gen_powset;
114
115     public:
116         GenSamples_Z_Uni(generator_type& generator,
117           const int M_MAX, const int N_MAX)
118           : _generator(generator), _gen_kombo(generator, M_MAX, N_MAX),
119           _gen_powset(generator, M_MAX) {}
120
```

```
121         void init(const int M, const int N);
122         void gen_element(std::vector<int>& k, std::vector<bool>& R);
123         // generates a single sample
124         void generate(const size_t n);
125         // generates n independent samples
126         inline const std::list<Z_element_type>& get_samples()
127           const { return _samples; }
128         inline int get_M() { return _M; }
129         inline int get_N() { return _N; }
130 };
131
132
133
134 /*** CLASS GenSamples_Kombo ***/
135 template <typename generator_type>
136 void GenSamples_Kombo<generator_type>::generate(const size_t n) {
137     std::vector<int> k(_M+2);
138     k[0] = 0;
139     k[_M+1] = _N+1;
140     for (size_t j=0; j<n; ++j) {
141         gen_element(k);
142         this->_samples.push_front(k);
143     }
144 }
145
146
147 /*** CLASS GenSamples_Kombo_Theta ***/
148 template <typename generator_type>
149 double GenSamples_Kombo_Theta<generator_type>
150   ::delta(const int a, const int b) const {
151 // a>=1, b>=0
152
153     std::vector<long double> coeff(std::min(a,b)+1);
154     // first compute coefficients for inner recursion
155     for (int w=0; w<std::min(a,b); ++w) {
156         coeff[w] = 2.0 * static_cast<long double>( (a-w) * (b-w) )
157           / static_cast<long double>(pow(w+1, 2));
158     }
159
160     long double x = 1.0;
161     long double outer_sum = 0.0;
162
163     for (int v=0; v<=std::min(a-1,b); ++v) {
164         long double y = x;
165         long double inner_sum = y;
166         for (int w=1; w<=std::min(a,b); ++w) {
167             // compute y for next round
168             y *= coeff[w-1];
169             inner_sum += y;
```

```
170         }
171         outer_sum += 1.0 / inner_sum;
172         x *= 0.5 * static_cast<long double>(pow(v+1, 2))
173           / static_cast<long double>( (a-v-1) * (b-v) );
174     }
175
176     return outer_sum;
177 }
178
179
180 template <typename generator_type>
181 GenSamples_Kombo_Theta<generator_type>::GenSamples_Kombo_Theta(
182   generator_type& generator, const int M_MAX, const int N_MAX)
183   : GenSamples_Kombo<generator_type>(generator) {
184 // computes the delta matrix with size M_MAX x (N_MAX+1)
185
186     _delta_matrix.resize(M_MAX);
187     for (int i=0; i<M_MAX; ++i) {
188         _delta_matrix[i].resize(N_MAX+1);
189         for (int j=0; j<=N_MAX; ++j)
190             _delta_matrix[i][j] = delta(i+1,j);
191     }
192 }
193
194
195 template <typename generator_type>
196 void GenSamples_Kombo_Theta<generator_type>
197   ::init(const int M, const int N) {
198     if ( not ( M > 0 and M <= _delta_matrix.size()
199       and N > 0 and N <= _delta_matrix[0].size() ) ) {
200         std::cout << "Error in function GenSamples_Kombo_Theta::init: "
201           << "invalid parameters!" << std::endl;
202         std::exit(EXIT_FAILURE);
203     }
204     this->_M=M;
205     this->_N=N;
206     this->_samples.clear();
207 }
208
209
210 template <typename generator_type>
211 void GenSamples_Kombo_Theta<generator_type>
212   ::gen_element(std::vector<int>& k) const {
213 // ensure that vector k has length _M+1 and k[0]=0,
214 // otherwise behavior is undefined
215
216     // uniform [0,1)
217     static std::uniform_real_distribution<double> uni_real(0.0, 1.0);
218
```

```
219     for (int m=1; m<=this->_M; ++m) {
220         double u = uni_real(this->_generator);
221         double a = this->_M-m;
222         double b = this->_N-k[m-1];
223         // probability of first value
224         double p = _delta_matrix[a][b];
225         double q = p;
226         p *= 2;
227         int l = k[m-1];
228         while ( q < u ) {
229             b = this->_N-l;
230             // compute probability of next value
231             p *= _delta_matrix[b-1][a];
232             q += p;
233             ++l;
234         }
235         k[m] = l;
236     }
237 }
238 
239 
240 /*** CLASS GenSamples_Powset ***/
241 template <typename generator_type>
242 void GenSamples_Powset<generator_type>::generate(const size_t n) {
243     std::vector<int> v;
244     // max size of v is N
245     v.reserve(_N);
246     for (size_t j=0; j<n; ++j) {
247         gen_element(v);
248         this->_samples.push_front(v);
249     }
250 }
251 
252 
253 /*** CLASS GenSamples_Powset_Uni ***/
254 template <typename generator_type>
255 GenSamples_Powset_Uni<generator_type>::GenSamples_Powset_Uni(
256   generator_type& generator, const int N_MAX)
257   : GenSamples_Powset<generator_type>(generator) {
258 // N_MAX >= 1
259 
260     // recursively compute the cumulative probability mass function
261     // for the distribution rho
262     _cum_pmf_matrix_rho.resize(N_MAX);
263     for (int n=1; n<=N_MAX; ++n) {
264         _cum_pmf_matrix_rho[n-1].resize(n+1);
265         long double p = pow(2.0,-static_cast<long double>(n));
266         long double q = p;
267         _cum_pmf_matrix_rho[n-1][0] = q;
```

```
268         for (int l=1; l<=n; ++l) {
269             p *= static_cast<long double>(n-l+1)
270               / static_cast<long double>(l);
271             q += p;
272             _cum_pmf_matrix_rho[n-1][l] = q;
273         }
274     }
275 }
276 
277 
278 template <typename generator_type>
279 void GenSamples_Powset_Uni<generator_type>::init(const int N) {
280     this->_N = N;
281     this->_samples.clear();
282 }
283 
284 
285 template <typename generator_type>
286 void GenSamples_Powset_Uni<generator_type>
287   ::gen_element(std::vector<int>& v) const {
288 // generates a sample from V_N and puts it in v
289 
290     // uniform [0,1)
291     static std::uniform_real_distribution<double> uni_real(0.0, 1.0);
292 
293     // first generate length
294     double u = uni_real(this->_generator);
295     int l = 0;
296     while ( _cum_pmf_matrix_rho[this->_N-1][l++] < u ) {}
297     int M = l-1;
298 
299     v.resize(M);
300     if ( M > 0 ) {
301         // treat case m=1 separately
302         // unlike with k \in \cK, here v[0] is not equal to 0 by convention
303         u = uni_real(this->_generator);
304         int a = M-1;
305         int b = this->_N;
306         double p = static_cast<double>(a+1) / static_cast<double>(b);
307         double q = p;
308         while ( q < u ) {
309             --b;
310             p *= static_cast<double>(b-a) / static_cast<double>(b);
311             q += p;
312         }
313         v[0] = this->_N-b+1;
314 
315         for (int m=2; m<=M; ++m) {
316             u = uni_real(this->_generator);
```

```
317             int a = M-m;
318             int b = this->_N-v[m-2];
319             double p = static_cast<double>(a+1) / static_cast<double>(b);
320             double q = p;
321             while ( q < u ) {
322                 --b;
323                 p *= static_cast<double>(b-a) / static_cast<double>(b);
324                 q += p;
325             }
326             v[m-1] = this->_N-b+1;
327         }
328     }
329 }
330
331
332 /*** CLASS GenSamples_Z_Uni ***/
333 template <typename generator_type>
334 void GenSamples_Z_Uni<generator_type>::init(const int M, const int N) {
335     _M=M;
336     _N=N;
337     _samples.clear();
338     _gen_kombo.init(_M, _N);
339 }
340
341
342 template <typename generator_type>
343 void GenSamples_Z_Uni<generator_type>::gen_element(std::vector<int>& k,
344   std::vector<bool>& R) {
345     k.resize(_M+2);
346     k[0] = 0;
347     k[_M+1] = _N+1;
348
349     _gen_kombo.gen_element(k);
350     std::vector<int> CQk;
351
352     R.resize(_M);
353     CQk.reserve(_M);
354     for (int i=0; i<_M; ++i) {
355         if ( k[i] == k[i+1] )
356             R[i] = true;
357         else {
358             R[i] = false;
359             CQk.push_back(i+1);
360         }
361     }
362
363     if ( CQk.size() > 0 ) {
364         std::vector<int> I;
365         _gen_powset.init(CQk.size());
```

```
366         _gen_powset.gen_element(I);
367
368         for (int i=0; i<I.size(); ++i)
369             R[CQk[I[i]-1]-1] = true;
370     }
371 }
372
373
374 template <typename generator_type>
375 void GenSamples_Z_Uni<generator_type>::generate(const size_t n) {
376     for (size_t j=0; j<n; ++j) {
377         _samples.push_front(Z_element_type());
378         gen_element(_samples.front().k, _samples.front().R);
379     }
380 }
381
382
383
384 #endif // RANDOMIZED_H
```

B.5 Die Klasse PowSetCounter

Listing B.8: power_set.h

```
1  #ifndef POWER_SET_H
2  #define POWER_SET_H
3
4  #include <vector>
5
6
7  class PowSetCounter {
8  // cycles through all the subsets of the set {1,..,_N}
9  // containing the set min
10     private:
11         int _N;
12         std::vector<bool> _total_vec;
13
14         int _min_size;
15         std::vector<int> _cmin;
16         std::vector<int> _cmin_indices;
17         bool _stop;
18
19     public:
20         void init(int N, const std::vector<int>& min,
21           const std::vector<int>& cmin);
22         PowSetCounter& operator++();
23         // computes the next subset
24         inline const std::vector<bool>& get_total_vec() const
```

```
25          { return _total_vec; }
26        // returns current subset
27        inline bool stopped() const { return _stop; }
28        void clear();
29  };
30
31
32  #endif // POWER_SET_H
```

Listing B.9: power_set.cpp

```
1   #include "power_set.h"
2
3
4   void PowSetCounter::init(int N, const std::vector<int>& min,
5     const std::vector<int>& cmin) {
6   // requirements:
7   // - neither min nor cmin should have a value which occurrs more than once
8   // - min and cmin have to be disjoint
9   // - N = max(min \union cmin)
10      _N = N;
11      _total_vec = std::vector<bool>(N, false);
12      for ( int l=0; l<min.size(); ++l )
13          _total_vec[min[l]-1] = true;
14
15      _stop = false;
16      _min_size = min.size();
17      _cmin = cmin;
18      _cmin_indices.reserve(cmin.size());
19  }
20
21
22  PowSetCounter& PowSetCounter::operator++() {
23      int pos = _cmin_indices.size();
24
25      while ( pos and _cmin_indices[pos-1]
26        == (int)_cmin.size()-(int)_cmin_indices.size()+pos-1 )
27          --pos;
28
29      if ( pos ) {
30          _total_vec[_cmin[_cmin_indices[pos-1]++]-1] = false;
31          _total_vec[_cmin[_cmin_indices[pos-1]]-1] = true;
32
33          for ( int i=pos; i<_cmin_indices.size(); ++i ) {
34              _total_vec[_cmin[_cmin_indices[i]]-1] = false;
35              _cmin_indices[i] = _cmin_indices[i-1]+1;
36              _total_vec[_cmin[_cmin_indices[i]]-1] = true;
37          }
38      } else {
```

```
39          if ( _cmin_indices.size() < _cmin.size() ) {
40              // now start with _cmin_indices.size()+1 elements
41              _cmin_indices.resize(_cmin_indices.size()+1);
42              for ( int i=0; i<_cmin_indices.size(); ++i )
43                  _cmin_indices[i] = i;
44              for ( int l=1; l<=std::min(_cmin.size()-_cmin_indices.size()+1,
45                _cmin_indices.size()); ++l )
46                  _total_vec[_cmin[l-1]-1] = true;
47              for ( int l=std::max(_cmin.size()-_cmin_indices.size()+1,
48                _cmin_indices.size())+1; l<=_cmin.size(); ++l )
49                  _total_vec[_cmin[l-1]-1] = false;
50          } else
51              _stop = true;
52      }
53
54      return *this;
55  }
56
57
58  void PowSetCounter::clear() {
59      _total_vec.clear();
60      _cmin.clear();
61      _cmin_indices.clear();
62  }
```

Literaturverzeichnis

[1] Anatoliy Skorokhod, *Limit theorems for stochastic processes*, Theory of Probability and Its Applications **1** (1956), Nr. 3, 261–290.

[2] Larisa Contreras, *Skorokhod metric*, Master's thesis, California State University – Channel Islands, 2009.

[3] Patrick Billingsley, *Convergence of probability measures (second edition)*, John Wiley & Sons, 1999.

[4] Annika Lang und Jürgen Potthoff, *Stochastic simulation*, Vorlesungsskript Universität Mannheim, 2014.

[5] Ken-Iti Sato, *Lévy processes and infinitely divisible distributions*, Cambridge University Press, 1999.

[6] Rama Cont und Peter Tankov, *Financial modelling with jump processes*, Chapman & Hall/CRC, 2004.

[7] Makoto Matsumoto und Takuji Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation **8** (1998), Nr. 1, 3–30.

[8] Egbert Harzheim, *Ordered sets*, Springer, 2005.