

Distanzerhaltende Approximation von Kantenzügen

Nikolas Klug

Sommersemester 2018
Universität Augsburg
Seminar Algorithmen und Datenstrukturen

Zusammenfassung

Diese Seminararbeit basiert auf „Distance-preserving approximations of polygonal paths“ von J. Gudmundsson et al. ([4]). Sei $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug und $t \geq 1$. Ein Kantenzug Q approximiert P , falls er nur aus Punkten von P besteht und falls dessen Kanten jeweils nur um t vom P abweichen. Diese Arbeit stellt exakte und approximative Algorithmen dar, um Q so zu berechnen, dass Q entweder die minimale Zahl von Knoten besitzt, oder für eine feste Knotenzahl minimales t hat.

1 Einführung und Definitionen

Ein (*polygonaler*) *Kantenzug* $P = (p_1, p_2, \dots, p_n)$ ist eine Aneinanderreihung von Geradensegmenten, die für $1 \leq i < n$ jeweils p_i und p_{i+1} verbinden. Bei den p_i handelt es sich dabei um Punkte aus \mathbb{R}^d (für $d \in \mathbb{N}$). Aufgrund der starken Ähnlichkeit zu Pfaden in (gerichteten) Graphen, werden wir für Kantenzüge auch häufig Begriffe aus der Graphentheorie verwenden. Beispielsweise heißen die Punkte des Kantenzuges auch Knoten, die verbindenden Geraden auch (gerichtete) Kanten etc. In anderen Vorlesungen und Fachbüchern werden und wurden die Begriffe Kantenzug und Pfad teilweise synonym verwendet, wir wollen diese hier aber - der besseren Verständlichkeit wegen - klar unterscheiden.

In der Realität kann ein solcher Kantenzug mitunter eine hohe Zahl von Knoten besitzen, von denen die meisten für die Anwendung uninteressant sind. Hier ist es vorteilhaft, diesen Kantenzug durch einen ähnlichen Kantenzug mit deutlich weniger Kanten (und somit auch Knoten) zu approximieren, wobei sich wichtige Parameter allerdings nicht stark ändern. In der Fachliteratur werden dabei zahlreiche verschiedene Parameter aufgeführt, die bei der Approximation erhalten werden sollen, einige davon sind die Fläche [1], die der Pfad einschließt und die Distanz bzw. die Länge des Pfades [4].

Ein Anwendungsszenario der Kantenzugapproximation liegt zum Beispiel bei der Erstellung und Vereinfachung von Karten. Dort trifft man ständig auf polygonale Kantenzüge im zwei-, selten auch dreidimensionalen Raum: Straßen, Küstenlinien, Höhenlinien, Stadtumrisse und Landesgrenzen sind nur einige Beispiele. Leider sind in den meisten Fällen nicht alle dieser Kantenzüge so einfach darzustellen wie beispielsweise die Grenze zwischen Libyen und Ägypten; betrachtet man die Weltkarte, fällt auf, dass die meisten Grenzen sogar sehr komplizierte Formen annehmen, die aus Tausenden, wenn nicht Zehntausenden, einzelnen Punkten bestehen. Möchte man nun eine Karte erstellen, ist klar, dass es im Allgemeinen nicht möglich und auch nicht wünschenswert ist, alle dieser Punkte darzustellen. Genau an dieser Stelle liegt eine Anwendung der Kantenzugapproximation: Die Landesgrenze bzw. die anderen oben aufgeführten Beispiele werden durch einen

Kantenzug approximiert, der aus deutlich weniger Punkten besteht, ohne dass sich ein wichtiger Parameter ändert, wie zum Beispiel die Länge.

In dieser Arbeit soll es um Algorithmen gehen, die einen polygonalen Kantenzug unter ungefährer Einhaltung der Länge approximieren. Dabei betrachten wir zunächst zwei exakte Algorithmen und später noch zwei approximative, die eine deutliche bessere Laufzeit aufweisen als die exakten.

Bevor wir uns den Algorithmen zuwenden, müssen wir jedoch einige Grundbegriffe definieren.

1.1 Definitionen

Seien $u, v \in \mathbb{R}^d$. Wir definieren $|uv|$ als den euklidischen Abstand von u und v . Ist $p_i, p_j \in P$, dann ist $\delta(p_i, p_j) := \sum_{k=i}^{j-1} |p_k p_{k+1}|$, also der euklidische Abstand dieser beiden Punkte entlang des Pfades P .

Lemma 1.1. *Für alle Knoten $p_i, p_j \in P$ gilt $|p_i p_j| \leq \delta(p_i, p_j)$*

Beweis. Die Strecke zwischen zwei Punkten in \mathbb{R}^d ist immer die kürzeste Verbindung dieser beiden Punkte. \square

Bis jetzt haben wir nur über distanzerhaltende Approximationen eines Kantenzugs geredet, ohne festzulegen, was distanzerhaltend überhaupt bedeutet. Wir nennen eine Kante distanzerhaltend, wenn ihre Länge nur um einen bestimmten Faktor von der Länge des Pfades abweicht. Genauer:

Definition 1.2 (*t*-distanzerhaltend). *Sind $p_i, p_j \in P$, dann ist die Kante (p_i, p_j) genau dann t-distanzerhaltend, wenn $\delta(p_i, p_{i+1}) \leq t \cdot |p_i p_j|$.*

Damit können wir jetzt definieren, was eine distanzerhaltende Approximation ist.

Definition 1.3 (*t*-distanzerhaltende Approximation). *Ein Kantenzug $Q = (p_{i_1}, p_{i_2}, \dots, p_{i_k})$ ist genau dann eine t-distanzerhaltende Approximation von $P = (p_1, p_2, \dots, p_n)$, wenn beide der folgenden Bedingungen gelten.*

- (1) $1 = i_1 < i_2 < \dots < i_k = n$
- (2) Für alle $1 \leq l < k$ ist die Kante $(p_{i_l}, p_{i_{l+1}})$ des Kantenzugs *t*-distanzerhaltend

Wir nennen eine t-distanzerhaltende Approximation eines Pfades *minimal* bzw. eine *minimale*, falls sie die geringst mögliche Zahl an Knoten besitzt. Der Quotient $\frac{\delta(p_i, p_j)}{|p_i p_j|}$ heißt *Abweichung* vom Kantenzug.

Korollar 1.4. *Sei $1 \leq t < t'$. Dann ist jede t-distanzerhaltende Approximation eines Pfades P auch eine t'-distanzerhaltende Approximation von P .*

In Zusammenhang mit der distanzerhaltenden Kantenzugapproximation stellen sich im Wesentlichen die folgenden zwei Probleme:

Das **Minimum-Vertex-Path-Simplification Problem (MVPS)**: Liegt ein polygonaler Kantenzug P und eine reelle Zahl $t \geq 1$ vor, soll eine kürzeste *t*-distanzerhaltende Approximation von P berechnet werden.

Das **Minimum-Dilation-Path-Simplification Problem (MDPS)**: Liegt ein polygonaler Kantenzug P und eine natürliche Zahl k vor, soll der kleinste Wert t bestimmt werden, für den eine *t*-distanzerhaltende Approximation mit maximal k Knoten existiert.

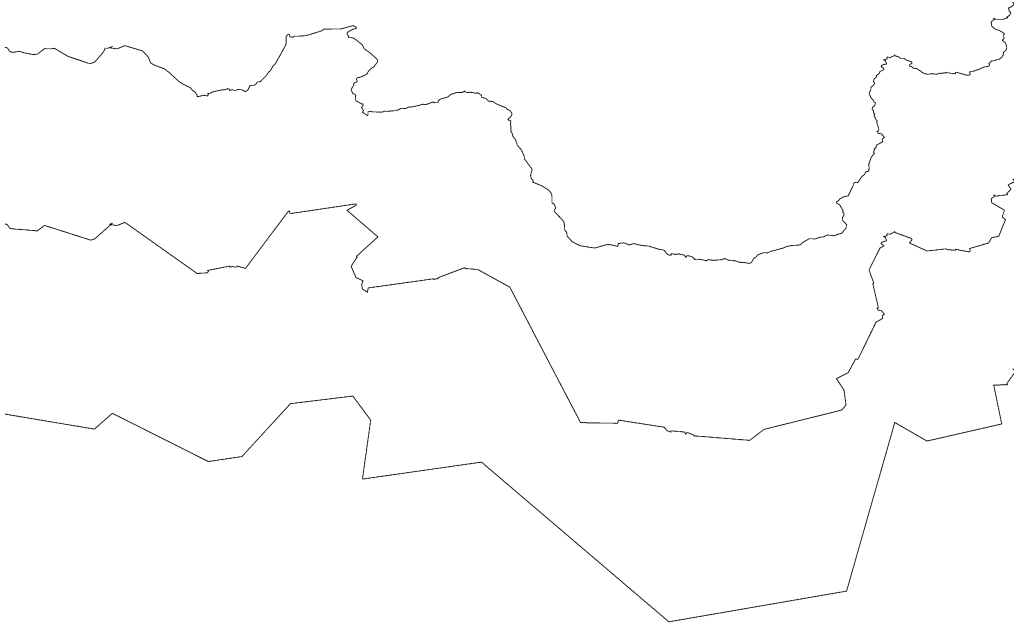


Abb. 1: Kantenzug mit 430 Punkten (oben) und zwei Approximationen mit 126 bzw. 22 Punkten (mitte und unten), die aus dem approximativen Algorithmus für das MVPS-Problem mit jeweils $\epsilon = 0.05$ und $t = 1.05$ bzw. $t = 1.2$ berechnet wurden (Quelle: [4])

2 Exakte Algorithmen für MVPS und MDPS

Zu Beginn betrachten wir zwei einfache exakte Algorithmen. Seien wieder $d \geq 1$, $t > 1$ und $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug in \mathbb{R}^d . Sei weiter P^* die Menge aller minimalen t -distanzerhaltenden Approximationen

Wir konstruieren jetzt den gerichteten Graphen $G_t = (V, E_t)$, wobei V genau aus den Knoten des Pfades P besteht und $E_t = \{(p_i, p_j) \in V \times V \mid i < j \text{ und } (p_i, p_j) \text{ ist } t\text{-distanzerhaltend}\}$. E_t ist also gerade die Menge aller t -distanzerhaltenden Kanten zwischen Knoten aus V . Zunächst beobachten wir, dass jede t -distanzerhaltende Approximation von P einem Pfad in G_t entspricht, da G_t alle t -distanzerhaltenden Kanten zwischen Knoten von P enthält. Andererseits ist auch jeder Pfad $Q = (p_{i_1}, p_{i_2}, \dots, p_{i_k})$ mit $1 = i_1 < i_2 < \dots < i_k = n$ in G_t eine t -distanzerhaltende Approximation von P , da nur t -distanzerhaltende Kanten verwendet werden. Daraus folgt, dass auch ganz P^* in G_t liegt. Jetzt müssen wir also nur noch ein Element aus P^* ermitteln. Das ist leicht: Wir führen eine Breitensuche in G_t mit Startknoten p_1 durch, bei der wir jeden Knoten mit der Nummer des Knotens beschriften, von dem aus er zum ersten Mal entdeckt wurde (also mit der Nummer seines Vaters im BFS-Baum). Am Ende lesen wir diese Beschriftung bei p_n beginnend solange aus, bis wir p_1 erreichen. Der dadurch entstandene Pfad entspricht dann aufgrund der Eigenschaften der Breitensuche einem Kantenzug aus P^* . Nun betrachten wir noch die Laufzeit: Die Konstruktion von G_t gelingt uns in $O(n^2)$, da wir für maximal $\binom{n}{2} = O(n^2)$ Kanten überprüfen müssen, ob diese t -distanzerhaltend sind. Sei m die Zahl der Kanten in G_t , dann wissen wir aus [5], dass die Breitensuche $O(n + m)$ Zeit dauert. In unserem Fall (und auch sonst immer) ist $m = O(n^2)$, und somit kostet die Breitensuche $O(n^2)$ Zeit. Insbesondere haben wir:

Satz 2.1. *Das Minimum-Vertex-Path-Simplification Problem kann für Pfade mit n Knoten $O(n^2)$ gelöst werden.*

Als nächstes wollen wir uns überlegen, wie man das MDPS-Problem für eine feste Anzahl von Knoten k lösen kann. Sei im Folgenden κ_t die geringst mögliche Zahl von Knoten für eine

t -distanzerhaltende Approximation von P .

Lemma 2.2. *Sind $t, t' \in \mathbb{R}$ und $1 \leq t < t'$, dann ist $\kappa_t \geq \kappa_{t'}$.*

Beweis. Wäre $\kappa_{t'} < \kappa_t$, hätte eine minimale t' -distanzerhaltende Approximation echt weniger Knoten als eine minimale t -distanzerhaltende. Aber jede t -distanzerhaltende Approximation von P ist nach Korollar 1.4 auch eine t' -distanzerhaltende Approximation. Das ist ein Widerspruch. \square

Da G_t maximal $O(n^2)$ Kanten enthält, gibt es eine endliche Zahl von t -Werten. Wir müssen also nur noch aus diesen Werten den geringsten Wert t^* ermitteln, für den eine t^* -distanzerhaltende Approximation existiert, die gerade noch k Knoten oder weniger hat. Dazu definieren wir zunächst für $1 \leq i < j \leq n$ $t_{ij}^* := \frac{\delta(p_i, p_j)}{|p_i p_j|}$, also als die Abweichung der Kante (p_i, p_j) vom Pfad. Sei nun $M := \{t_{ij}^* \mid 1 \leq i < j \leq n\}$. Wir wissen, dass $t^* \in M$, da die gesuchte Approximation eine Kante mit maximalen t -Wert hat, und M gerade alle diese enthält. Aus Lemma 2.2 wissen wir außerdem, dass sich die κ_t umgekehrt proportional zu den t -Werten verhalten. Sortieren wir jetzt M zu M' , können wir in M' nach t^* suchen. Da M $O(n^2)$ Elemente enthält, können wir M bekanntermaßen (siehe [5]) in $O(n^2 \log n^2) = O(n^2 \log n)$ sortieren. Für die Suche verwenden wir eine Binärsuche, bei der wir jeweils für den aktuell betrachteten t -Wert das MVPS-Problem lösen und dann abhängig vom Ergebnis entweder im rechten oder linken Teilbereich weitersuchen. Eine gewöhnliche Binärsuche kostet $O(\log n)$ Zeit und mit Satz 2.1 ergibt sich auch hierfür eine Laufzeit von $O(n^2 \log n)$. Wir halten fest:

Satz 2.3. *Das Minimum Dilation Path Simplification Problem kann für Pfade mit n Knoten in $O(n^2 \log n)$ gelöst werden.*

Damit beschließen wir das Kapitel über exakte Algorithmen für das MVPS- und das MDPS-Problem und wenden uns approximativen Lösungen zu.

3 Approximative Algorithmen

Die beiden approximativen Algorithmen, die wir betrachten werden, basieren auf der sogenannten Zerlegung in wohl-separierte Paare. Dabei werden im weitesten Sinne die Punkte des Kantenzugs in verschiedenen Mengen zusammengefasst, die bestimmte Eigenschaften haben. Da diese Zerlegung einen wesentlichen Teil beider Algorithmen ausmacht, werden wir sie zunächst genauer definieren und einen Algorithmus zur Berechnung angeben.

3.1 Well-separated Pair Decomposition

Definition 3.1 (wohl-separiert). *Seien $s > 0$ und A und B zwei endliche Mengen von Punkten in \mathbb{R}^d . A und B heißen wohl-separiert in Bezug zu s (engl. well-separated), falls es zwei disjunkte Bälle C_A und C_B gibt, die denselben Radius R haben, sodass $A \subseteq C_A$ und $B \subseteq C_B$ und die euklidische Distanz zwischen C_A und C_B mindestens $s \cdot R$ beträgt.*

s nennen wir dabei die *Trennungsrate* der beiden Mengen A und B . Das folgende Lemma hält zwei wichtige Eigenschaften von zwei in Bezug zu s wohl-separierten Mengen A und B fest.

Lemma 3.2. *Seien $a, a' \in A$ und $b, b' \in B$. Dann gilt:*

$$(1) \quad |aa'| \leq \frac{2}{s} \cdot |a'b'|$$

```

compute_split_tree(i, j) {
  if i = j then
    erstelle neuen Knoten u;
    speichere das Intervall [i, i] zu u;
    return u
  else
    z := (S[i] + S[j])/2;
    k := Index eines Elementes von S, sodass S[k] ≤ z < S[k + 1];
    v := compute_split_tree(i, k);
    w := compute_split_tree(k+1, j);
    erstelle neuen Knoten u;
    speichere das Intervall [i, j] zu u;
    mache v zum linken Kind von u;
    mache w zum rechten Kind von u;
    return u
  end if
}

```

Abb. 2: Algorithmus zum Erstellen eines fairen Split-Trees zu einer gegebenen Menge S (nach [4])

$$(2) \quad |a'b'| \leq \left(1 + \frac{4}{s}\right) \cdot |ab|$$

Beweis. Zu 1. Ist r der Radius von C_A und C_B , so gilt $|aa'| \leq 2 \cdot r$. Da A und B wohl-separiert sind, gilt zudem $|a'b'| \geq s \cdot r$, was äquivalent ist zu $r \leq \frac{|a'b'|}{s}$. Durch Einsetzen folgt dann die Behauptung.

Zu 2. Da A und B wohl-separiert in Bezug zu s sind, und C_A und C_B beide denselben Radius r haben, gilt $|a'b'| \leq s \cdot r + 4 \cdot r$. Ausklammern rechts ergibt $(1 + \frac{4}{s}) \cdot s \cdot r$. Da ja auch $s \cdot r \leq |ab|$, folgt durch Einsetzen die Behauptung. \square

Definition 3.3 (WSPD). Sei $S \subseteq \mathbb{R}^d$ und $s > 0$. Eine Folge $\{A_i, B_i\}_{1 \leq i \leq m}$ von nicht-leeren Teilmengen von S ist genau dann eine Zerlegung in wohl-separierte Paare (engl. well-separated pair decomposition; WSPD), wenn gilt:

- (1) $A_i \cap B_i = \emptyset$
- (2) Für alle $p, q \in S$ gibt es genau einen Index $1 \leq i \leq m$, sodass entweder $p \in A_i$ und $q \in B_i$ oder $q \in A_i$ und $p \in B_i$.
- (3) A_i und B_i sind wohl-separiert in Bezug zu s

m nennen wir dabei die Größe der WSPD.

Die WSPD bildet eine wichtige Grundlage für die beiden Algorithmen, die wir im Folgenden betrachten werden. Callahan und Kosaraju haben in [2] gezeigt, dass man zu einer gegebenen Menge $L \subset \mathbb{R}^d$ der Größe n eine WSPD der Größe $m = O(s^d n)$ in $O(n \log n + s^d n)$ Zeit berechnen kann. Dabei wird zunächst in $O(n \log n)$ ein sogenannter (fairer) Split-Tree berechnet, aus dem dann in $O(s^d n)$ Zeit eine WSPD erstellt werden kann. Wir werden sehen, dass es für unseren Anwendungsfall genügt, eine WSPD für Mengen von Punkten aus \mathbb{R} zu erstellen. Für diesen eindimensionalen Fall kann der faire Split-Tree mit Hilfe eines einfachen Algorithmus berechnet werden (Abb. 2).

Sei S nun eine endliche Teilmenge von \mathbb{R} und $|S| = n$. Wir können davon ausgehen, dass uns diese Menge sortiert in einem Array $S[1..n]$ vorliegt und werden später sehen, dass das bei unserem Algorithmus auch tatsächlich der Fall ist. Sei T der Split-Tree, der durch das Ausführen von `compute_split_tree(1, n)` entstanden ist. Bei T handelt es sich um einen Binärbaum, dessen Blättern die Werte von S in von links nach rechts aufsteigend sortierter Reihenfolge speichern. Für

```

compute_wspd(T, s) {
  for each innerer Knoten u in T do
    v := linkes Kind von u
    w := rechtes Kind von u
    find_pairs(v, w)
  }

find_pairs(v, w) {
  if  $S_u$  und  $S_v$  sind in Bezug zu  $s$  nicht wohl-separiert then
    Seien  $[i, j]$  und  $[k, l]$  die Intervalle die mit  $u$  bzw.  $v$  gespeichert sind;
    if  $S[j] - S[i] \leq S[l] - S[k]$  then
       $w_1$  := linkes Kind von  $w$ ;
       $w_2$  := rechtes Kind von  $w$ ;
      find_pairs(v,  $w_1$ );
      find_pairs(v,  $w_2$ );
    else
       $v_1$  := linkes Kind von  $v$ ;
       $v_2$  := rechtes Kind von  $v$ ;
      find_pairs( $v_1$ , w);
      find_pairs( $v_2$ , w);
    end if
  else if
    Speichere in  $u$  und  $v$ , dass deren Blätter die Teilmengen  $A$  und  $B$  einer WSPD bilden;
  end if
}

```

Abb. 3: Algorithmus zum Erstellen einer WSPD aus einem gegebenen Split-Tree T und einer Trennungsrates s (nach [4])

jeden inneren Knoten, also für jeden Knoten, der kein Blatt ist, wird zusätzlich das Intervall, in dem die Blattwerte des von ihm induzierten Teilbaumes liegen, gespeichert. Die Wurzel speichert also zum Beispiel das Intervall $(1, n)$, und das i -te Blatt das Intervall (i, i) . Da T n Blätter hat, erstellen wir insgesamt $O(n)$ Knoten. Dabei müssen wir aber für die Bestimmung von k jedes mal eine Binärsuche durchführen, die $O(\log n)$ Zeit kostet. Somit ergibt sich für das Erstellen von T eine Gesamtlaufzeit von $O(n \log n)$. Betrachten wir jetzt zwei innere Knoten p und q von T . Seien $[i, j]$ und $[k, l]$ die Intervalle, die wir mit p und q gespeichert haben und

$$R := \max(S[i] - S[j], S[k] - S[l])$$

Nach Definition 3.1 sind die beiden Intervalle genau dann wohl-separiert, wenn

$$S[k] - S[j] \geq R \cdot s \text{ oder } S[i] - S[l] \geq R \cdot s$$

Der in Abbildung 3 dargestellte Algorithmus berechnet dann aus dem eben erstellten fairen Split-Tree T eine WSPD. Betrachten wir diesen doch etwas genauer.

Beim Aufruf von `compute_wspd(T, s)` werden für jeden Knoten k dessen linke und rechte Kindknoten u und v betrachtet, und dann `find_pairs(u, v)` aufgerufen. Da die Elemente von S in den Blättern gespeichert sind, ist klar, dass die der linke Kindknoten u und der rechte v disjunkte Teilmengen von S repräsentieren. Somit ist Forderung (1) der WSPD erfüllt. `find_pairs(u, v)` überprüft, ob die mit u und v gespeicherten Intervalle S_u und S_v wohl-separiert sind; ist dies der Fall, speichern wir mit u , dass seine Blätter das Element A_i einer WSPD bilden, und mit v , dass seine Kinder das Element B_i darstellen. Sind die Intervalle nicht wohl-separiert, steigen wir solange in Richtung des größeren Intervalls im Baum herab, bis wir auf zwei wohl-separierte Intervalle treffen. Wir sehen also, dass die erste und die dritte Forderung der Definition der WSPD durch den Algorithmus erfüllt werden. Man kann auch zeigen, dass er die zweite erfüllt, was wir

an dieser Stelle allerdings überspringen werden. Interessierte können einen vollständigen Beweis auf Seite 72ff in [2] nachlesen. Callahan und Kosaraju, die Autoren dieses Artikels, haben auch bewiesen, dass `compute_wspd(T, s)` eine zu T gehörende WSPD der Größe $O(sn)$ in $O(sn)$ Zeit berechnet. Halten wir also fest:

Satz 3.4. *Sei $S \subset \mathbb{R}$ endlich und $n = |S|$. Dann kann in $O(n \log n + sn)$ Zeit ein Split-Tree T und eine dazugehörige WSPD $\{A_i, B_i\}_{1 \leq i \leq m}$ der Größe $m = O(sn)$ berechnet werden.*

3.2 Algorithmus für das MVPS-Problem

Nach dem wir jetzt einige Vorarbeit geleistet haben, werden wir in diesem Kapitel sehen, wie man das MVPS-Problem mit Hilfe einer WSPD bis auf ein ϵ genau approximieren kann - und dass (für festes t und ϵ) in $O(n \log n)$ Zeit.

Die Theorie

Sei $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug in \mathbb{R}^d . Für unseren Anwendungsfall genügt es, eine ein-dimensionale Version $S = (x_1, x_2, \dots, x_n)$ dieses Kantenzugs zu betrachten. Diese erhalten wir, indem wir für alle $1 \leq i \leq n$ $x_i = \delta(p_1, p_i)$ setzen. Als nächstes berechnen wir für ein festes $s > 0$ zunächst den Split-Tree T und danach eine zugehörige WSPD $\{A_i, B_i\}_{1 \leq i \leq m}$ von S . Wegen Eigenschaft (3) der WSPD sind für alle i entweder alle Elemente in A_i kleiner als die in B_i oder umgekehrt. Wir werden o.B.d.A annehmen, dass alle Elemente, die in A_i enthalten sind, kleiner sind, als alle Elemente in B_i , da wir einfach bei der Erstellung der WSPD die beiden Mengen passend benennen.

Lemma 3.5. *Seien $p, p', q, q' \in P$ und sei i ein solcher Index, dass für $x = \delta(p_1, p)$, $x' = \delta(p_1, p')$, $y = \delta(p_1, q)$ und $y' = \delta(p_1, q')$ $x, x' \in A_i$ und $y, y' \in B_i$ sind. Ist weiter $1 \leq t < \frac{s^2}{4s+16}$ und ist das Tupel (p, q) t -distanzerhaltend, dann ist (p', q') t' -distanzerhaltend, wobei t' gegeben ist durch*

$$t' = \frac{(1 + \frac{4}{s}) \cdot t}{1 - 4(1 + \frac{4}{s}) \cdot \frac{t}{s}}$$

Beweis. Wegen unserer speziellen Wahl von t ist der Nenner von t' immer positiv, und genauso der Zähler. Dann können wir rechnen:

$$\begin{aligned} \delta(p', q') &= |x'y'| \\ &\leq (1 + \frac{4}{s}) \cdot |xy| && \text{(Lemma 3.2 (2))} \\ &= (1 + \frac{4}{s}) \cdot \delta(p, q) \\ &\leq (1 + \frac{4}{s})t \cdot |pq| && ((p, q) \text{ ist } t\text{-distanzerhaltend}) \\ &\leq (1 + \frac{4}{s})t \cdot (|pp'| + |p'q'| + |q'q|) && \text{(Dreiecksungleichung in } \mathbb{R}) \\ &\leq (1 + \frac{4}{s})t \cdot (\delta(p, p') + |p'q'| + \delta(q', q)) && \text{(Lemma 1.1)} \\ &= (1 + \frac{4}{s})t \cdot (|xx'| + |p'q'| + |yy'|) \\ &\leq (1 + \frac{4}{s})t \cdot (\frac{2}{s} \cdot |x'y'| + |p'q'| + \frac{2}{s} \cdot |x'y'|) && \text{(Lemma 3.2 (1))} \\ &= (1 + \frac{4}{s})t \cdot (\frac{4}{s} \cdot \delta(p', q') + |p'q'|) \end{aligned}$$

$$= 4(1 + \frac{4}{s})\frac{t}{s} \cdot \delta(p', q') + (1 + \frac{4}{s})t \cdot |p'q'|$$

Also ist

$$\begin{aligned} \delta(p', q') &\leq 4(1 + \frac{4}{s})\frac{t}{s} \cdot \delta(p', q') + (1 + \frac{4}{s})t \cdot |p'q'| \\ \Leftrightarrow \delta(p', q') \cdot (1 - 4(1 + \frac{4}{s})\frac{t}{s}) &\leq (1 + \frac{4}{s})t \cdot |p'q'| \\ \Leftrightarrow \delta(p', q') &\leq t' \cdot |p'q'| \end{aligned}$$

□

Sei jetzt $0 < \epsilon < \frac{1}{3}$ und $1 \leq t$. Sei

$$s = \frac{12 + 24(1 + \frac{\epsilon}{3}) \cdot t}{\epsilon}$$

Durch diese Wahl von s ist die Einschränkung $t < \frac{s^2}{4s+16}$ von Lemma 3.5 für alle $t \geq 1$ erfüllt.

Lemma 3.6. *Seien $p, p', q, q' \in P$ wie in Lemma 3.5. Dann gilt*

- (1) *Ist (p, q) t -distanzerhaltend, dann ist (p', q') $(1 + \frac{\epsilon}{3})t$ -distanzerhaltend.*
- (2) *Ist (p, q) $(1 + \frac{\epsilon}{3})t$ -distanzerhaltend, dann ist (p', q') $(1 + \epsilon)t$ -distanzerhaltend.*

Beweis. Zu 1. Sei (p, q) t -distanzerhaltend. Dann ist (p', q') t' -distanzerhaltend, wobei t' in Lemma 3.5 gegeben ist. Da die Einschränkung $1 \leq t < \frac{s^2}{4s+16}$ aus selbigem Lemma immer noch gilt, ist $0 < s^2 - 4st - 16t$. Daraus folgt, dass $s \geq 4t \geq 4$ ist. Es ergibt sich damit und durch unsere spezielle Wahl von s

$$t' = \frac{(1 + \frac{4}{s}) \cdot t}{1 - 4(1 + \frac{4}{s}) \cdot \frac{t}{s}} \leq \frac{(1 + \frac{4}{s})t}{1 - 8\frac{t}{s}} = (1 + \frac{\epsilon}{3})t$$

Zu 2. Sei (p, q) $(1 + \frac{\epsilon}{3})t$ -distanzerhaltend. Dann ist (p', q') t'' -distanzerhaltend, wobei sich t'' aus Lemma 3.5 ergibt als

$$t'' = \frac{(1 + \frac{4}{s}) \cdot (1 + \frac{\epsilon}{3})t}{1 - 4(1 + \frac{4}{s}) \cdot (1 + \frac{\epsilon}{3})\frac{t}{s}}$$

Da $1 < \epsilon < \frac{1}{3}$, ist

$$s = \frac{12 + 24(1 + \frac{\epsilon}{3}) \cdot t}{\epsilon} \geq \frac{4(1 + \frac{\epsilon}{3})t}{\epsilon} \geq \frac{4(1 + \frac{\epsilon}{3})}{\epsilon} \geq \frac{4(1 + \frac{\epsilon}{3})}{1 - \frac{\epsilon}{3}}$$

Desweiteren ist

$$\begin{aligned} s &\geq \frac{4(1 + \frac{\epsilon}{3})}{1 - \frac{\epsilon}{3}} \quad \Leftrightarrow \quad s \cdot (2 - (1 + \frac{\epsilon}{3})) \geq 4 \cdot (1 + \frac{\epsilon}{3}) \\ \Leftrightarrow \quad 2s &\geq 4 \cdot (1 + \frac{\epsilon}{3}) + s \cdot (1 + \frac{\epsilon}{3}) \quad \Leftrightarrow \quad 2 \geq (1 + \frac{4}{s}) \cdot (1 + \frac{\epsilon}{3}) \end{aligned}$$

Also ist

$$t'' \leq \frac{(1 + \frac{4}{s}) \cdot (1 + \frac{\epsilon}{3})t}{1 - 8\frac{t}{s}} = (1 + \frac{\epsilon}{3})^2 \cdot t \leq (1 + \epsilon) \cdot t$$

□

Als nächstes konstruieren wir aus der WSPD einen gerichteten Graphen H und zeigen dann, dass jede Approximation von P einem Pfad in H entspricht.

Seien dafür für alle $1 \leq i \leq m$ a_i und b_i zwei feste Elemente aus A_i und B_i . Seien weiter α_i und β_i die Elemente von P , für die $a_i = \delta(p_1, \alpha_i)$ und $b_i = \delta(p_1, \beta_i)$. Wir nennen im Folgenden die Tupel (A_i, B_i) (t, ϵ) -distanzerhaltend, falls jeweils (α_i, β_i) $(1 + \frac{\epsilon}{3})t$ -distanzerhaltend ist.

Die Knoten von H sind genau die $2m$ Mengen A_i und B_i , und die Kanten sind wie folgt definiert:

- (1) Für alle $1 \leq i \leq m$ ist (A_i, B_i) genau dann eine Kante, wenn (A_i, B_i) (t, ϵ) -distanzerhaltend ist und $x_n \in B_i$
- (2) Für alle $1 \leq i < j \leq m$ ist (A_i, A_j) genau dann eine Kante, wenn (A_i, B_i) (t, ϵ) -distanzerhaltend ist und $A_j \cap B_i \neq \emptyset$

Satz 3.7. *Jede t -distanzerhaltende Approximation $Q = (q_1, q_2, \dots, q_k)$ von P entspricht einem Pfad R der Länge k in H von einer Menge A_i , die x_1 enthält, zu einer Menge B_j , die x_n enthält*

Beweis. Sei y_i das Element der Menge S , für das $y_i = \delta(p_1, q_i)$ gilt. Da nach Bedingung ja $q_1 = p_1$ gilt, ist also auch $y_1 = x_1$. Sei weiter i_1 ein solcher Index, für den $y_1 \in A_{i_1}$ und $y_2 \in B_{i_1}$. Dann hat der Pfad R A_{i_1} als ersten Knoten.

Nehmen wir jetzt an, dass wir bereits für ein l mit $1 \leq l < k - 1$ den Kantenzug (q_1, \dots, q_l) zu dem Teilpfad $(A_{i_1}, \dots, A_{i_l})$ von R umgewandelt haben, sodass $y_l \in A_{i_l}$ und $y_{l+1} \in B_{i_l}$. Wir wählen jetzt i_{l+1} als den Index, für den $y_{l+1} \in A_{i_{l+1}}$ und $y_{l+2} \in B_{i_{l+1}}$ ist. Solch ein Index existiert nach Definition der WSPD und ist eindeutig. Wir wissen, dass (q_l, q_{l+1}) t -distanzerhaltend ist und $y_l \in A_{i_l}$ und $y_{l+1} \in B_{i_l}$ liegt. Aus Lemma 3.6 (1) folgt dann, dass das Tupel (A_i, B_i) (t, ϵ) -distanzerhaltend ist. Des Weiteren ist der Schnitt von $A_{i_{l+1}}$ mit B_{i_l} nicht leer, da y_{l+1} in beiden Mengen liegt. Also ist $(A_{i_l}, A_{i_{l+1}})$ eine Kante in H und wir wählen $A_{i_{l+1}}$ als nächsten Knoten. Wir haben damit also $(q_1, \dots, q_l, q_{l+1})$ zu dem Pfad $(A_{i_1}, \dots, A_{i_l}, A_{i_{l+1}})$ in H umgewandelt, sodass $y_{l+1} \in A_{i_{l+1}}$ und $y_{l+2} \in B_{i_{l+1}}$.

Nehmen wir an, dass wir bereits $(q_1, q_2, \dots, q_{k-1})$ zu dem Pfad $(A_{i_1}, \dots, A_{i_{k-1}})$ umgewandelt haben, wobei $y_{k-1} \in A_{i_{k-1}}$ und $y_k \in B_{i_{k-1}}$. Es ist $y_k = \delta(p_1, q_k) = \delta(p_1, p_n) = x_n \in B_{i_{k-1}}$. Da die Kante (q_{k-1}, q_k) t -distanzerhaltend ist, ist wieder wegen Lemma 3.6 (1) $(A_{i_{k-1}}, B_{i_{k-1}})$ (t, ϵ) -distanzerhaltend und es folgt, dass $(A_{i_{k-1}}, B_{i_{k-1}})$ eine Kante in H ist. Wir fügen $B_{i_{k-1}}$ zum Pfad hinzu, und erhalten als Gesamtergebnis $R = (A_{i_1}, \dots, A_{i_{k-1}}, B_{i_{k-1}})$. \square

Wir haben also gezeigt, dass jede t -distanzerhaltende Approximation von P einem Pfad in H entspricht, der dieselbe Zahl von Knoten hat. Der nächste Satz zeigt, dass dies auch umgekehrt der Fall ist, unter der Einschränkung, dass die Approximation um einen kleinen Teil vom gewünschten t -Wert abweichen kann.

Satz 3.8. *Jeder Pfad $R = (A_{i_1}, \dots, A_{i_{k-1}}, B_{i_{k-1}})$ in H mit $x_1 \in A_{i_1}$ und $x_n \in B_{i_{k-1}}$ entspricht einer $(1 + \epsilon)t$ -distanzerhaltenden Approximation Q von P , die k Knoten besitzt.*

Beweis. Sei wieder y_i das Element der Menge S , für das $y_i = \delta(p_1, q_i)$ gilt. Da x_1 in A_{i_1} , können wir als ersten Knoten von Q $q_1 = p_1$ wählen.

Nehmen wir an, dass wir bereits für ein l mit $1 \leq l < k - 1$ den Teilpfad $(A_{i_1}, \dots, A_{i_l})$ in den Kantenzug (q_1, \dots, q_l) umgewandelt haben, sodass $y_1 (= x_1) \in A_{i_1}$ und für alle $1 < j \leq l$ $y_j \in A_{i_j} \cap B_{i_{j-1}}$. Betrachten wir jetzt die Kante $(A_{i_l}, A_{i_{l+1}})$, gibt es ein $y_{l+1} \in A_{i_{l+1}} \cap B_{i_l}$, da der Schnitt nach der Konstruktion von H nicht leer ist. Dann fügen wir das zu y_{l+1} gehörende q_{l+1} zum Kantenzug hinzu und erhalten $(q_1, \dots, q_l, q_{l+1})$.

Nehmen wir an, dass wir bereits $(A_{i_1}, \dots, A_{i_{k-1}})$ bereits zu (q_1, \dots, q_{k-1}) konvertiert haben. Nach Voraussetzung ist $x_n \in B_{k-1}$. Wir wählen dann $q_k = p_n$ und fügen q_k zum Kantenzug Q hinzu. Insgesamt haben wir also gezeigt, wie man R zu einem Kantenzug Q mit gleich vielen Knoten umwandeln kann.

Jetzt bleibt zu zeigen, dass Q auch tatsächlich $(1 + \epsilon)t$ -distanzerhaltend ist. Dazu betrachten wir ein j mit $1 \leq j < k$. Nach unserer Konstruktion von Q ist $y_j \in A_{i_j}$ und $y_{j+1} \in B_{i_j}$. Die Kante A_{i_j} und B_{i_j} ist zudem (t, ϵ) -distanzerhaltend. Mit Lemma 3.6 (2) folgt dann, dass alle Tupel (a, b) mit $a \in A_{i_j}$ und $b \in B_{i_{j+1}}$ $(1 + \epsilon)t$ -distanzerhaltend sind, insbesondere ist also die Kante zwischen den zu y_j und y_{j+1} gehörigen Knoten (q_j, q_{j+1}) $(1 + \epsilon)t$ -distanzerhaltend. Diese Eigenschaft gilt für alle aufeinanderfolgenden Knoten in Q , woraus folgt, dass der von uns konstruierte Kantenzug Q eine $(1 + \epsilon)t$ -distanzerhaltende Approximation von P ist. \square

Durch diese beiden Sätze wird schnell klar, was wir tun müssen, um eine $(1 + \epsilon)t$ -distanzerhaltende Approximation eines Pfades P zu erhalten: Wir konstruieren einfach den Graphen H und führen darin eine Breitensuche aus, um einen kürzesten Pfad zu bestimmen. Das Problem daran ist aber, dass H $2m = O(sn)$ Knoten hat und somit unter Umständen $O(s^2n^2)$ Kanten besitzen kann, was wir uns aber nicht erlauben können, wenn wir eine bessere Laufzeit als die des exakten Algorithmus anstreben.

Bevor wir uns allerdings dem expliziten Algorithmus zuwenden, betrachten wir noch Kriterien, unter denen ein kürzester Pfad in H , der den Kriterien aus Satz 3.8 genügt, einer kürzesten t -distanzerhaltenden Approximation von P entspricht.

Satz 3.9. *Sei $P = (p_1, p_2, \dots, p_n)$ ein polygonaler Kantenzug in \mathbb{R}^d und H der oben definierte Graph. Sei $R = (A_{i_1}, \dots, A_{i_{k-1}}, B_{i_{k-1}})$ ein kürzester Pfad in H mit $x_1 \in A_{i_1}$ und $x_n \in B_{i_{k-1}}$. Gilt für alle $p, q \in P$ mit $p \neq q$ $\frac{\delta(p, q)}{|pq|} \leq t$ oder $\frac{\delta(p, q)}{|pq|} > (1 + \epsilon)t$, dann entspricht R einer kürzesten t -distanzerhaltenden Approximation von P .*

Beweis. Seien $p, q \in P$, $x = \delta(p_1, p)$ und $y = \delta(p_1, q)$ und i ein solcher Index, dass $x \in A_i$ und $y \in B_i$.

Ist $\frac{\delta(p, q)}{|pq|} > (1 + \epsilon)t$, so gilt nach Lemma 3.6, dass für alle $a \in A_i$ und $b \in B_i$ $\delta(a, b) > (1 + \frac{\epsilon}{3})t \cdot |ab| > t$. Also ist (A_i, B_i) nicht (t, ϵ) -distanzerhaltend und es gibt in H keine Kante, die A_i verlässt. \square

Folglich existieren in H nur Kanten (A_i, A_j) , sodass für alle Elemente $a_i \in A_i$ und $a_j \in A_j$ die zugehörigen Knoten des Kantenzugs t -distanzerhaltend sind. Analoges gilt für die Kanten (A_i, B_i) in H . Insgesamt sind also alle in der Approximation auftretenden Kanten t -distanzerhaltend. \square

Im folgenden Abschnitt werden wir sehen, wie man einen kürzesten Pfad in H finden kann, ohne H vollständig zu konstruieren.

Der Algorithmus

Der Algorithmus besteht aus fünf Teilen, von denen wir die ersten drei im Prinzip schon betrachtet haben. Sei $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug. Die ersten drei Teile sind folgende:

- (1) Berechne $S = (x_1, x_2, \dots, x_n)$, wobei $x_i = \delta(p_1, p_i)$ für alle $1 \leq i \leq n$
- (2) Berechne aus S den Split-Tree T und daraus eine WSPD $\{A_i, B_i\}_{1 \leq i \leq m'}$ mit der Trennungsrate $s = \frac{12 + 24(1 + \frac{\epsilon}{3})t}{\epsilon}$. Nehme wieder o.B.d.A. an, dass für alle $1 \leq i \leq m'$ alle Elemente aus A_i kleiner sind als alle aus B_i .

- (3) Seien $a_i \in A_i$ und $b_i \in B_i$ für alle $1 \leq i \leq m'$ feste Elemente und seien α_i und β_i die Knoten von P , für die $a_i = \delta(p_1, \alpha_i)$ und $b_i = \delta(p_1, \beta_i)$. Falls (α_i, β_i) nicht $(1 + \frac{\epsilon}{3})t$ -distanzerhaltend ist, verwirft das korrespondierende Tupel (A_i, B_i) , ansonsten behalte es.

Der Einfachheit halber beschreiben wir die „ausgedünnte“ WSPD durch $\{A_i, B_i\}_{1 \leq i \leq m}$, wobei m die Zahl der verbleibenden Tupel ist. Bevor wir zu Schritt (4) kommen, zeigen wir zunächst, wie man mit Hilfe einer Breitensuche in T einen kürzesten Pfad in H bestimmen kann.

Für alle $1 \leq i \leq m$ sei u_i der Knoten des Split-Trees T , der A_i repräsentiert, und v_i derjenige, der B_i repräsentiert. Wir nennen die u_i auch *A-Knoten*. Es ist nicht ausgeschlossen, dass ein Knoten von T mehrere A_i und B_i repräsentiert. Folglich speichert jeder Knoten eine Liste seiner A_i und B_i .

Die Breitensuche die wir verwenden, zeigt Ähnlichkeiten zu der, die in [5] erklärt wird. Für jeden Knoten v des Baumes T speichern wir dabei drei Variablen:

- $color[v]$, die einen Wert aus $\{white, gray, black\}$ hat.
- $dist[v]$, die der aktuellen Distanz von einem Knoten A_i in H , der x_1 enthält, zum Knoten v speichert.
- $parent[v]$, die den Vater von w im BFS-Wald speichert.

Sind die Knoten von T mit den Zahlen $1, 2, \dots, n'$ benannt sind, können wir $color$, $dist$ und $parent$ zum Beispiel durch drei Arrays der Größe n' realisieren. Die Breitensuche sieht dann so aus:

Schritt 1: Für alle Knoten k von T , setze $color[k] = white$, $dist[k] = \infty$ und $parent[k] = null$.

Schritt 2: Initialisiere eine leere Warteschlange W (z.B durch eine verkettete Liste). Starte bei dem Blatt, dass x_1 speichert (dem „linksten“ Blatt) und laufe im Baum aufwärts bis zur Wurzel. Für alle besichtigten Knoten k , tue Folgendes:

Setze $color[k] = gray$. Falls k ein A-Knoten ist, setze $dist[k] = 0$. Füge k in W ein.

Schritt 3: Entferne das erste Element k von W . Setze $color[k] = black$. Für alle $u_i = k$ tue Folgendes:

Falls $x_n \in B_i$, setze $dist[v_i] = dist[k] + 1$, $parent[v_i] = k$, $z = v_i$ und gehe zu Schritt 4.

Falls $x_n \notin B_i$ und $color[v_i] == white$, führe die Schritte 3.1 und 3.2 aus.

Schritt 3.1: Starte bei v_i und laufe im Baum aufwärts bis zum ersten nicht weißen Knoten.

Für alle besichtigten Knoten k' , tue Folgendes:

Setze $color[k'] = gray$. Falls k' ein A-Knoten ist, setze $dist[k'] = dist[k] + 1$, $parent[k'] = k$ und füge k' in W ein.

Schritt 3.2: Starte bei v_i und besuche alle Knoten des Teilbaums von T , dessen Wurzel v_i ist. Für alle besichtigten Knoten k' , tue Folgendes:

Setze $color[k'] = gray$. Falls k' ein A-Knoten ist, setze $dist[k'] = dist[k] + 1$, $parent[k'] = k$ und füge k' in W ein.

Schritt 4: Berechne den Pfad $Z = (z, parent[z], parent[parent[z]], parent^3[z], \dots, parent^{k-1}[z])$, wobei $k = dist[z] + 1$. Gib den umgekehrten Pfad $Z' = (parent^{k-1}[z], \dots, parent[z], z)$ zurück.

Dass diese Breitensuche einen kürzesten Pfad in H zurückgibt, ist nicht sofort klar. Auf einen vollständigen Beweis wollen wir an dieser Stelle zwar verzichten, aber die wichtigsten Punkte skizzieren.

Ist k das erste Element der Warteschlange, so hat es die momentan geringste Distanz zu einer Menge A_i , die x_1 enthält. Sei A_l eine Menge, für die $u_l = k$ gilt. Wird jetzt u_l bearbeitet, werden alle anderen von A_l aus (in H) erreichbaren Mengen A_j betrachtet. Diese Mengen sind genau die, die einen nicht-leeren Schnitt mit B_l haben.

Wir können beobachten, dass falls ein Knoten u weiß ist, auch alle Knoten im Unterbaum von u weiß sind. Ist v der erste nicht-weiße Knoten, der in Schritt 3.1 erreicht wird, sind alle Knoten auf dem Pfad von v zur Wurzel des Split-Trees nicht-weiß.

Dadurch ist klar, dass in den Schritten 3.1 und 3.2 alle Mengen A_j bearbeitet werden, die von A_l aus erreichbar sind. Für jedes A_j wird dann dessen Distanz zu A_i verringert (nämlich von ∞ auf $d[k] + 1$).

Ist jedoch x_n in zu A_l gehörigen B_l bereits enthalten, ist die Breitensuche beendet, da dann bereits ein kürzester Weg von A_i zu B_l bestimmt wurde.

Ein genauer, vollständiger Beweis kann unter Verwendung der obigen Beobachtungen analog zum Beweis der Breitensuche im Informatik III Skript [5] oder zu dem in Cormen et al. [3] durchgeführt werden. Ist m die Kantenanzahl in T und z die Knotenanzahl, ergibt sich dabei insbesondere die für eine Breitensuche übliche Laufzeit $O(m + z)$. Aber $O(m + z) = O(sn) = O(\frac{t}{\epsilon}n)$. Darum erhalten wir für die Laufzeit der Breitensuche insgesamt $O(\frac{t}{\epsilon}n)$.

Die letzten beiden Teile des Algorithmus sind dann folgende:

- (4) Führe die oben aufgeführte modifizierte Breitensuche in T durch, um einen kürzesten Pfad Z' von einer Menge A_{i_1} , die x_1 enthält, zu einer Menge $B_{i_{k-1}}$, die x_n enthält, zu bestimmen.
- (5) Konvertiere Z' , der den Kriterien von Satz 3.8 entspricht, wie im Beweis desselben Satzes zu einer Approximation von P .

Abschließend betrachten wir noch die Laufzeit des Algorithmus. S zu berechnen kostet uns $O(n)$ Zeit. Nach Satz 3.4 können wir Teil 2 in $O(n \log n + sn)$ schaffen, und da $O(sn) = O(\frac{t}{\epsilon} \cdot n)$, dauert dieser Teil also $O(n \log n + \frac{t}{\epsilon}n)$. Somit dauert auch Teil 3 $O(\frac{t}{\epsilon}n)$, da wir jedes Tupel der WSPD einmal betrachten. Wie oben gesehen, weißt die Breitensuche dieselbe asymptotische Laufzeit auf.

Nun bleibt noch der letzte Teil des Algorithmus. Um den Pfad in H zu konvertieren, wählen wir, wie gezeigt, p_1 als ersten und p_n als letzten Knoten. Einer der wichtigen Schritte für die restlichen Knoten besteht darin, ein Element aus $A_{i_{l+1}} \cap B_{i_l}$ auszuwählen. Da wir wissen, dass der Schnitt nicht leer ist, und $A_{i_{l+1}}$ und B_{i_l} durch Knoten des Baumes repräsentiert werden (nämlich $u_{i_{l+1}}$ und v_{i_l}), die wiederum die Blätter in ihrem jeweiligen Unterbaum zusammenfassen, muss entweder $A_{i_{l+1}} \subseteq B_{i_l}$ sein oder umgekehrt. Also wählen wir zum Beispiel das Minimum μ des Intervalls, das mit $u_{i_{l+1}}$ gespeichert ist, und überprüfen, ob es im Intervall von v_{i_l} liegt. Ist das der Fall, liegt dieses im Schnitt und wir wählen den zu μ gehörigen Knoten als nächsten. Sonst ist $B_{i_l} \subset A_{i_{l+1}}$ und wir können das Minimum des Intervalls von v_{i_l} wählen. Dieses Vorgehen dauert für jede Kante $(A_{i_l}, A_{i_{l+1}})$ jeweils konstante Zeit, insgesamt kostet und Teil 5 des Algorithmus also $O(\frac{t}{\epsilon}n)$ Zeit. Zusammenfassend können wir festhalten:

Satz 3.10. *Sei $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug in \mathbb{R}^d , sei $t \geq 1$ und $0 < \epsilon < \frac{1}{3}$ und sei κ die Knotenanzahl der minimalen t -distanzerhaltenden Approximationen von P .*

- (1) *Dann können wir in $O(n \log n + \frac{t}{\epsilon}n)$ eine $(1 + \epsilon)t$ -distanzerhaltende Approximation Q von P*

mit maximal κ Knoten berechnen.

- (2) Ist weiter für alle verschiedenen $p, q \in P$ $\frac{\delta(p,q)}{|pq|} \leq t$ oder $\frac{\delta(p,q)}{|pq|} > (1 + \epsilon)t$, dann ist Q sogar eine t -distanzerhaltende Approximation von P mit κ Knoten.

3.3 Algorithmus für das MDPS-Problem

Ähnlich wie in Kapitel 2 werden wir jetzt versuchen, mit Hilfe des eben vorgestellten approximativen Algorithmus für das MVPS-Problem eine Lösung des MDPS-Problems zu konstruieren.

Sei dazu k die Knotenzahl und $P = (p_1, p_2, \dots, p_n)$ der Kantenzug, für die wir das MDPS-Problem lösen wollen. Wie in Kapitel 2 sei $t^* = \min\{t \geq 1 \mid \kappa_t \leq k\}$ die Lösung des Problems. Unser Algorithmus basiert im Wesentlichen auf dem folgenden Lemma.

Lemma 3.11. *Sei $t \geq 1$, $0 < \epsilon < \frac{1}{3}$ und sei Q eine $(1 + \epsilon)t$ -distanzerhaltende Approximation, die durch Satz 3.10 entstanden ist. Ist k' die Anzahl der Knoten von Q , dann gilt:*

- (1) Falls $k' \leq k$, dann ist $t^* \leq (1 + \epsilon) \cdot t$
(2) Falls $k' > k$, dann ist $t^* > t$

Beweis. Zu 1. Sei $k' \leq k$. Sei t' die exakte Abweichung des Kantenzugs Q von P , also $\max\{\frac{\delta(p_i, p_{i+1})}{|p_i p_{i+1}|} \mid 1 \leq i < k\}$. Dann gilt auch $t' \leq (1 + \epsilon) \cdot t$. Aus Lemma 2.2 folgt damit, dass $\kappa_{(1+\epsilon)t} \leq \kappa_{t'}$. Da $\kappa_{t'}$ aber die Knotenzahl einer minimalen t' -distanzerhaltenden Approximation ist, muss $\kappa_{t'} \leq k'$ gelten. Zusammen mit der Annahme folgt dann, dass $\kappa_{(1+\epsilon)t} \leq k$. Wegen Lemma 2.2 muss dann aber $t^* \leq (1 + \epsilon) \cdot t$ gelten.

Zu 2. Sei nun $k' > k$. Nach Satz 3.10 gilt $k' \leq \kappa_t$. Also ist auch $k < \kappa_t$. Wie oben folgt nun aus Lemma 2.2, dass $t^* > t$ sein muss, da für t^* ja $\kappa_{t^*} \leq k$ gilt. \square

Eine erste Approximation von t^*

Sei nun $0 < \epsilon < \frac{1}{3}$ fest. Der folgende Algorithmus berechnet eine Annäherung τ von t^* bis auf einen Faktor von 2.

Schritt 1: Setze $\tau = 2$.

Schritt 2: Führe den Algorithmus von Satz 3.10 aus. Sei Q die dadurch erhaltende Approximation und k' deren Knotenzahl.

Schritt 3: Falls $k' > k$, setze $\tau = 2 \cdot \tau$ und gehe zu Schritt 2. Falls $k' \leq k$, gib τ zurück.

Dass nach der Terminierung des Algorithmus

$$\frac{\tau}{2} < t^* \leq (1 + \epsilon) \cdot \tau \quad (*)$$

gilt, folgt direkt aus Lemma 3.11. Betrachten wir nun noch die Laufzeit des Algorithmus. Der im vorangehenden Kapitel vorgestellte Algorithmus zur Lösung des MVPS-Problems berechnet zunächst einen Split-Tree T und erstellt dann daraus zur gegebenen Trennungsrates s eine WSPD. Da T aber nur vom Eingabe-Kantenzug P abhängt, genügt es, T nur einmal zu berechnen und dann daraus die verschiedenen WSPDs zu erstellen. Bei gegebenem T dauert die Lösung des MVPS-Problems für t und ϵ mit Satz 3.10 dann nur noch $O(\frac{t}{\epsilon} \cdot n)$. Im obigen Algorithmus wird Schritt 2 $O(\log \tau)$ mal ausgeführt, da τ ja bei jeder Iteration verdoppelt wird. Die Laufzeit für die Approximation von t^* bei gegebenem Split-Tree T ist die Folgende:

$$O\left(\sum_{i=1}^{\log \tau} \left(\frac{2^i}{\epsilon} \cdot n\right)\right) = O\left(\frac{\tau}{\epsilon} \cdot n\right) = O\left(\frac{t^*}{\epsilon} \cdot n\right)$$

Eine bessere Approximation von t^* mit Hilfe von binärer Suche

Bevor wir uns einem Algorithmus für eine bessere Approximation von t^* als oben zuwenden, müssen wir zunächst ein dafür essentielles Lemma beweisen.

Sei nun $S = (x_1, x_2, \dots, x_n)$ wie in Kapitel 3.2 und $\{A_i, B_i\}_{1 \leq i \leq m}$ eine WSPD mit einer Trennungsrate von

$$s = \frac{4 + 8(1 + \epsilon)^3 \cdot \tau}{\epsilon}$$

Seien für alle $1 \leq i \leq m$ a_i und b_i feste Elemente aus A_i bzw. B_i und α_i und β_i die Knoten von P , für die $a_i = \delta(p_1, \alpha_i)$ und $b_i = \delta(p_1, \beta_i)$ ist. Sei $t_i = \frac{\delta(\alpha_i, \beta_i)}{|\alpha_i \beta_i|}$, dann gilt:

Lemma 3.12. *Es gibt ein j mit $1 \leq j \leq m$ mit*

$$\frac{t_j}{1 + \epsilon} \leq t^* \leq (1 + \epsilon) \cdot t_j$$

Beweis. Wie wir gesehen haben, gibt es zwei Knoten p und q , für die $t^* = \frac{\delta(p, q)}{|pq|}$. Sei j der Index, sodass $\delta(p_1, p) \in A_j$ und $\delta(p_1, q) \in B_j$. Solch ein Index existiert nach Definition der WSPD. Da (p, q) t^* -distanzerhaltend ist und wie im Beweis von Lemma 3.5 $t^* < \frac{s^2}{4s+16}$, ist $s^2 > 4st^* + 16t^*$. Aus selbigem Lemma folgt dann, dass (α_j, β_j) t' -distanzerhaltend ist, wobei

$$t' = \frac{(1 + \frac{4}{s}) \cdot t^*}{1 - 4(1 + \frac{4}{s}) \frac{t^*}{s}}$$

Da außerdem wieder $s \geq 4$, ist wegen unserer speziellen Wahl von s

$$t' \leq \frac{(1 + \frac{4}{s}) \cdot t^*}{1 - 8 \frac{t^*}{s}}$$

Zusammen mit (*) folgt daraus, dass

$$s \geq \frac{4 + 8(1 + \epsilon)t^*}{\epsilon} \Leftrightarrow \frac{(1 + \frac{4}{s}) \cdot t^*}{1 - 8 \cdot \frac{t^*}{s}} \leq (1 + \epsilon) \cdot t^*$$

Insgesamt ist damit

$$t_j = \frac{\delta(\alpha_j, \beta_j)}{|\alpha_j \beta_j|} \leq t' \leq \frac{(1 + \frac{4}{s}) \cdot t^*}{1 - 8 \frac{t^*}{s}} \leq (1 + \epsilon) \cdot t^*$$

Nun zur zweiten Ungleichung. Wir wissen jetzt, dass $t_j \leq (1 + \epsilon) \cdot t^*$ ist. Aus der Beobachtung im ersten Teil des Beweises folgt daraus dann, dass $4st_j + 16t_j < s^2$, insbesondere ist damit wieder die Bedingung aus Lemma 3.5 erfüllt. Selbigem Lemma entsprechend ist dann (p, q) t'' -distanzerhaltend, wobei sich t'' ergibt als

$$t'' = \frac{(1 + \frac{4}{s}) \cdot t_j}{1 - 4(1 + \frac{4}{s}) \cdot \frac{t_j}{s}}$$

Wie oben ist $s \geq 4$, sodass sich der Ausdruck zu

$$t'' = \frac{(1 + \frac{4}{s}) \cdot t_j}{1 - 8 \cdot \frac{t_j}{s}}$$

vereinfacht. Da mit (*) $t_j \leq (1 + \epsilon) \cdot t^* \leq (1 + \epsilon)^2 \tau$ gilt, ist des Weiteren

$$s \geq \frac{4 + 8(1 + \epsilon) \cdot t_j}{\epsilon}$$

Durch Einsetzen erhalten wir dann $t^* \leq t'' \leq (1 + \epsilon) \cdot t_j$. □

Nun können wir uns dem eigentlichen Algorithmus zuwenden. Dieser lautet wie folgt:

Schritt 1: Berechne τ wie oben als Approximation von t^* bis auf einen Faktor von 2.

Schritt 2: Erstelle die Liste der t_i für $1 \leq i \leq m$. Setze $t_0 := 1$ und sortiere die t_i für $0 \leq i \leq m$.

Schritt 3: Entferne alle Duplikate und die Werte aus der Liste, die größer als $(1 + \epsilon)^2 \tau$ sind.

Sei L die übrigbleibende sortierte Liste $1 = t_0 < t_1 < \dots < t_{m'} \leq (1 + \epsilon)^2 \cdot \tau$.

Schritt 4: Führe eine binäre Suche nach t^* in L durch.

Dass uns eine binäre Suche in L zum gewünschten Ziel führt, ist nicht sofort klar. Deshalb werden wir jetzt zeigen, dass die folgende Invariante während der gesamten Suche erhalten bleibt und die Gelegenheit nutzen, die Suche genauer zu beschreiben.

Lemma 3.13. *l und r sind Zahlen mit $0 \leq l < r \leq m'$ und $t_l \leq t^* \leq (1 + \epsilon) \cdot t_r$.*

Beweis. Wir unterscheiden drei Fälle:

Fall 1: Sei $l = 0$ und $r = m'$. Wir betrachten das j aus Lemma 3.12. Da für dieses j ja $t_j \leq (1 + \epsilon) \cdot t^* \leq (1 + \epsilon)^2 \cdot \tau$ gilt, kommt t_j in L vor. Jetzt gilt aber auch $t_l = 1 \leq t^* \leq (1 + \epsilon) \cdot t_j \leq (1 + \epsilon) \cdot t_r = (1 + \epsilon)^2 \cdot \tau$, sodass in diesem Fall die Invariante erfüllt ist.

Fall 2: Sei $l < r - 1$. Wir wählen $h = \lfloor \frac{l+r}{2} \rfloor$. Zunächst berechnen wir die zu einer $(1 + \epsilon)t_h$ -distanzerhaltenden Approximation gehörende Knotenzahl k' . Auf diese können wir Lemma 3.11 anwenden und überprüfen, ob $k' \leq k$ ist. Ist dies der Fall, ist $t^* \leq (1 + \epsilon) \cdot t_h$. Dann gilt aber $t_l \leq t^* \leq (1 + \epsilon) \cdot t_h$ und wir können $r := h$ setzen. Sonst ist $t_h < t^* \leq (1 + \epsilon) \cdot t_r$ und wir setzen $l := h$.

Fall 3: Sei $l = r - 1$. Dann gilt zunächst $t_l \leq t^* \leq (1 + \epsilon) \cdot t_{l+1}$. Auch hier berechnen wir zunächst die zu einer $(1 + \epsilon)t_h$ -distanzerhaltenden Approximation gehörende Knotenzahl k' und überprüfen, ob $k' \leq k$. Ist dies der Fall, folgt aus Lemma 3.11, dass $t^* \leq (1 + \epsilon)^2 \cdot t_l$ ist. Somit gilt $t_l \leq t^* \leq (1 + \epsilon)^2 \cdot t_l$ und wir geben t_l zurück.

Sonst ist $t^* > (1 + \epsilon) \cdot t_l$. Jetzt betrachten wir wieder den Index j aus Lemma 3.12. Für das korrespondierende t_j gilt $t_l < \frac{t^*}{1+\epsilon} \leq t_j$. Also ist $t^* \leq (1 + \epsilon) \cdot t_j$ und aus der Invariante folgt $t_{l+1} \leq t_j$. Da aber auch $\frac{t_j}{1+\epsilon} \leq t^*$ ist, ergibt sich insgesamt

$$\frac{t_{l+1}}{1 + \epsilon} \leq \frac{t_j}{1 + \epsilon} \leq t^* \leq (1 + \epsilon) \cdot t_{l+1}$$

und wir geben $\frac{t_l}{1+\epsilon}$ zurück.

□

Lemma 3.13 bestätigt uns also, dass uns die binäre Suche in L einen solchen Wert t zurückliefert, für den $t \leq t^* \leq (1 + \epsilon)^2 \cdot t$ gilt.

Betrachten wir nun noch die Laufzeit des Algorithmus. Zunächst berechnen wir zu einem gegebenen Kantenzug P das Array S in $O(n)$, und als nächstes daraus einen fairen Split-Tree T , was, wie wir gesehen haben, in $O(n \log n)$ möglich ist. Die Berechnung von τ kostet uns dann nur noch $O(\frac{t^*}{\epsilon} \cdot n)$ Zeit. Die WSPD, die als nächstes bestimmt wird, hat eine Größe von $m = O(sn) = O(\frac{\tau}{\epsilon}n) = O(\frac{t^*}{\epsilon}n)$ und die Berechnung kostet dementsprechend auch $O(\frac{t^*}{\epsilon}n)$ Zeit. Da außerdem $m \leq n^2$ ist, kann das Sortieren der t_i , die wir aus der WSPD erhalten, bekanntermaßen in

$O(m \log m) = O(\frac{t^*}{\epsilon} n \cdot \log n)$ geschehen. Das Aussortieren der unbrauchbaren Werte ist offensichtlich in Linearzeit zu bewerkstelligen.

Die binärer Suche macht $O(\log m) = O(\log n)$ Iterationen. In jeder dieser Iterationen führen wir mit gegebenem Split-Tree den im Beweis von Satz 3.10 angegebenen Algorithmus mit einer Eingabegröße $t_h \leq (1 + \epsilon)^2 \cdot \tau = O(t^*)$ aus. Dies führt uns zu einer Laufzeit von $O(\frac{t^*}{\epsilon} n)$ pro Durchführung, und für die gesamte binäre Suche haben wir $O(\frac{t^*}{\epsilon} n \cdot \log n)$.

Nun können wir noch einen kleinen Trick anwenden: Verwenden wir im gesamten Algorithmus nicht ϵ , sonder stattdessen $\frac{\epsilon}{3}$, erhalten wir als Ergebnis ein t mit $t \leq t^* \leq (1 + \frac{\epsilon}{3})^2 \cdot t \leq (1 + \epsilon) \cdot t$. Das können wir festhalten:

Satz 3.14. *Sei $P = (p_1, p_2, \dots, p_n)$ ein Kantenzug in \mathbb{R}^d , sei $2 \leq k \leq n$ und $0 < \epsilon < \frac{1}{3}$. Sei weiter t^* die kleinste Zahl t , für die eine t -distanzerhaltende Approximation von P mit maximal k Knoten existiert. Dann können wir in $O(\frac{t^*}{\epsilon} n \cdot \log n)$ eine Zahl t berechnen, für die $t \leq t^* \leq (1 + \epsilon) \cdot t$ ist.*

4 Fazit

Wir haben uns damit befasst, einen gegebenen Kantenzug P so zu approximieren, dass die Länge der Approximation nur um einen bestimmten Faktor t von der von P abweicht. Im Rahmen dieser Zielsetzung haben wir zwei Probleme betrachtet: Die Minimierung der Knotenzahl der Approximation bei gegebenem t (MVPS) und die Minimierung von t bei gegebener Knotenzahl (MDPS). Dazu haben wir zunächst einen exakten Algorithmus für das MVPS-Problem betrachtet, der eine Laufzeit von $O(n^2)$ aufwies, und anschließend daraus einen exakten Algorithmus für das MDPS-Problem konstruiert, dessen Ausführung $O(n^2 \log n)$ Zeit kostet.

Die danach vorgestellten approximativen Algorithmen wiesen dagegen eine deutlich bessere Laufzeit auf. Das MVPS-Problem können wir dabei bis auf ein ϵ genau in $O(n \log n + \frac{t}{\epsilon} n)$ lösen. Auch hier haben wir dieses Ergebnis verwendet, um daraus einen - diesmal approximativen - Algorithmus für das MDPS-Problem zu konstruieren. Dieser wies eine asymptotische Laufzeit von $O(\frac{t^*}{\epsilon} n \cdot \log n)$ auf, wobei t^* die exakte Lösung ist.

Die Autoren des Artikels [4], auf dem diese Arbeit beruht, zeigten 2006 in einem Experiment, dass die Laufzeit der approximativen Algorithmen nicht nur asymptotisch, sondern auch praktisch die der exakten Algorithmen deutlich unterbietet. So gelang die exakte Lösung des MVPS-Problems für einen Kantenzug mit 20.000 Punkten und einer Abweichung $t = 1.1$ in etwa 97 Sekunden, während der approximative Algorithmus für das selbe t und $\epsilon = 0.05$ nur 7.2 Sekunden brauchte.

Literatur

- [1] Prosenjit Bose, Sergio Cabello, Otfried Cheong, Joachim Gudmundsson, Marc van Kreveld, and Bettina Speckmann. Area-preserving approximations of polygonal paths. *Journal of Discrete Algorithms* 4, pages 554–566, 2006.
- [2] Paul Callahan and S Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. ACM* 42, pages 67–90, 1995.
- [3] T.H. Cormen, C.E. Leiserson, R.L Rivest, and C. Stein. *Introduction to Algorithms*. 2001.

- [4] Joachim Gudmundsson, Giri Narasimhan, and Michiel Smid. Distance-preserving approximations of polygonal paths. *Computational Geometry* 36, pages 183–196, 2007.
- [5] Torben Hagerup. Vorlesungsskript Informatik III WS 17/18, 2017.