Diplomarbeit

Peter Schäfer

Untersuchungen zu Varianten des Fréchet-Abstands

Theorie und Implementierung

Mai 2019

FernUniversität in Hagen

Fakultät für Mathematik und Informatik

Lehrgebiet Theoretische Informatik

Prof. Dr. André Schulz • Dr. Lena Schlipf

Inhaltsverzeichnis

Zweiter Teil: Implementierung

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Definitionen und Sätze

Erster Teil

Theorie

1

Einleitung

Ähnlichkeit In vielen Anwendungsbereichen möchte man die Ähnlichkeit geometrischer Kurven beschreiben. Beispiele sind die Erkennung von Handschriften [92], die Analyse und das Erstellen von Landkarten [2, 3, 4, 16, 34], die Analyse von Fahrzeug-Bewegungsdaten [21, 30], oder der Laufwege von Sportlern [62] und vieles mehr. In der Biochemie haben wir es mit komplexen geometrischen Objekten zu tun, z. B. mit gefalteten Proteinmolekülen [73, 95].

Für alle diese Anwendungen ist es wichtig, die Ähnlichkeit zweier Objekte beschreiben zu können. Zu diesem Zweck wurden geometrische Abstandsmaße für Kurven und Flächen eingeführt. Eine Übersicht zu dem Thema bieten Alt und Guibas [14]. Mit einem der wichtigsten Abstandsmaße wollen wir uns im Folgenden beschäftigen: dem *Fréchet-Abstand* und einigen seiner Varianten.

Diese Arbeit In dieser Arbeit stellen wir zwei wichtige Varianten des Fréchet-Abstands vor: den Fréchet-Abstand für *Polygonzüge* [13] und den Fréchet-Abstand für *einfache Polygone* [27]. In den Kapiteln 1 bis 3 legen wir die theoretischen Grundlagen und beschreiben Algorithmen zur Berechnung des Fréchet-Abstands.

In Kapitel 4 stellen wir ein sehr junges Konzept vor, den *k-Fréchet-Abstand* [6]. Wir beschreiben algorithmische Ansätze und steuern eigene Ergebnisse bei.

Implementierungen Alle beschriebenen Algorithmen wurden praktisch implementiert. Die Implementierung des Algorithmus für *einfache Polygone* stellt unseres Wissens die erste praktische Umsetzung dieses Algorithmus dar, ebenso unsere Implementierung der Algorithmen für den *k*-Fréchet-Abstand.

Im zweiten Teil der Arbeit beschreiben wir unsere Implementierung: welche Konzepte wurden verwirklicht, welche Herausforderungen gelöst. Einen breiten Raum wird dabei die Boolesche Matrixmultiplikation einnehmen. Wir beschäftigen uns auch mit der Umsetzung der Algorithmen auf paralleler Hardware und auf GPU-Hardware.

Fréchet-View Im Rahmen dieser Arbeit entstand die PC-Anwendung Fréchet View. Mit dieser graphischen, interaktiven Oberfläche können die Algorithmen getestet und analysiert werden. Als wichtigstes Werkzeug gibt das *Free-Space-Diagramm* einen Einblick in die Arbeitsweise der Algorithmen. Sämtliche Free-Space-Diagramme in dieser Arbeit wurden mit Fréchet View erstellt.

1.1. Der Hausdorff-Abstand

Ein lange etabliertes Maß zum Vergleich von geometrischen Objekten ist der *Hausdorff-Abstand*. Er ist nach Felix Hausdorff[1] benannt und wurde 1914 eingeführt. Wir suchen für jeden Punkt den kleinsten Abstand zur anderen Menge und bilden das Supremum über alle Punkte.

Definition 1.1 Hausdorff-Abstand [13, S. 75]

Gegeben sind zwei Mengen von Punkten $P, Q \subset \mathbb{R}^d$. Der **Hausdorff-Abstand** der beiden Mengen ist definiert als

$$\delta_H(P,Q) \;=\; \max\{\; \bar{\delta}_H(P,Q),\; \bar{\delta}_H(Q,P)\;\}\;,$$

wobei

$$\bar{\delta}_H(P,Q) \;=\; \sup_{p\in P}\;\inf_{q\in Q}\;\; ||\,p-q\,||\;.$$

Mit $||p-q||$ bezeichnen wir den Abstand zweier Punkte. Die zugrundeliegende Norm ist meist die *Euklidische Norm*.

Zwei Mengen haben einen niedrigen Hausdorff-Abstand, wenn es in der näheren Umgebung jedes Punktes einen Punkt der anderen Menge gibt. Auf kompakten Mengen bildet der Hausdorff-Abstand selbst eine Metrik.

Der Hausdorff-Abstand kann aber die Ähnlichkeit von Kurven nicht immer gut beschreiben. In Abb. 1.1 zeigen wir zwei Kurven mit einem niedrigen Hausdorff-Abstand. Zu jedem Punkt einer Kurve gibt in der nahen Umgebung Punkte der zweiten Kurve. Wir würden diese Kurven aber nicht als ähnlich bezeichnen. Der Hausdorff-Abstand ist auf Punktmengen definiert, kann aber nicht den *Verlauf* der Kurven berücksichtigen.

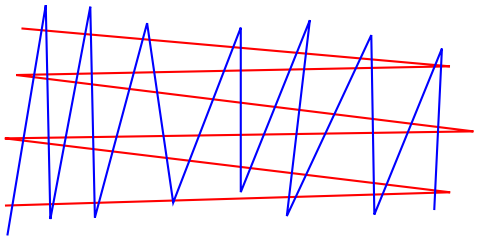


Abbildung 1.1.: Zwei Kurven mit niedrigem Hausdorff-Abstand: zu jedem Punkt lässt sich ein benachbarter Punkt auf der anderen Kurve finden. Aber wir würden diese Kurven nicht als ähnlich bezeichnen.

1.2. Parametrisierte Kurven

Um den Verlauf von Kurven zu beschreiben, führen wir zunächst *Parametrisierungen* ein.

Definition 1.2 Parametrisierte Kurven

Gegeben ist ein Wertebereich $A \subset \mathbb{R}^k$ mit $k = 1, 2, \ldots$ Eine **parametrisierte Kurve** ist eine stetige Abbildung

$$P\colon A \to \mathbb{R}^d$$

mit $d \geq k$.

[1] Felix Hausdorff, 1868-1942, ein Begründer von Topologie und Mengenlehre.

Für eindimensionale Wertebereiche, z. B. $A = [0, 1]$, beschreiben wir damit eine Kurve. Legen wir einen Kreis zugrunde, ist die Kurve geschlossen.

Zweidimensionale Wertebereiche, z. B. $A = [0, 1]^2$, beschreiben Flächen.

Das Ziel der Abbildung liegt oft in $\mathbb{R}^2$, d. h. wir betrachten Kurven oder Flächen in der Ebene. Aber auch Einbettungen von Flächen in $\mathbb{R}^3$ und $\mathbb{R}^4$ sind denkbar.

1.3. Der Fréchet-Abstand

Der Fréchet-Abstand wurde von Maurice Fréchet[2] 1906 vorgestellt und später verallgemeinert [58, 59]. Wir vergleichen zwei parametrisierte Kurven. Um von den zugrundeliegenden Parametrisierungen unabhängig zu werden, führen wir *Reparametrisierungen* ein.

Der Fréchet-Abstand versucht, über alle möglichen Reparametrisierungen den *maximalen* Abstand der Kurven zu *minimieren*.

Definition 1.3 Fréchet-Abstand

Gegeben sind zwei parametrisierte Kurven $P\colon A \to \mathbb{R}^d$ und $Q\colon B \to \mathbb{R}^d$. Der **Fréchet-Abstand**

$$\delta_F(P, Q) = \inf_{\sigma,\tau} \sup_{\substack{s\in A,\\ t\in B}} || P(\sigma(s)) - Q(\tau(t)) ||$$

ist das Infimum über alle orientierungs-erhaltende Homöomorphismen $\sigma\colon A \to A$ und $\tau\colon B \to B$.

$|| \cdot ||$ ist wiederum die zugrundeliegenede Norm, meist die Euklidische Norm. Andere Normen, z. B. L_1 oder L_∞ wurden auch erforscht [19, 33], sind aber nicht Gegenstand dieser Arbeit.

Von den Reparametrisierungen σ, τ verlangen wir, dass sie Homöomorphismen sind. Sie sollen stetig und bijektiv sein, ihre Umkehrungen sind ebenfalls stetig. Wir verlangen ebenso, dass sie die Orientierung der Kurven beibehalten. Eine Definition auf der Grundlage von orientierungsumkehrenden Homöomorphismen ist möglich, soll hier aber nicht stattfinden.

Zunächst erscheint die Definition schwer fassbar, da wir das Infimum über unendlich viele Reparametrisierungen bilden müssen. Für eindimensionale Wertebereiche (also für Kurven) können wir aber die Reparametrisierungen gut charakterisieren: die Homöomorphismen sind streng monotone Funktionen. In Kapitel 2 werden wir Algorithmen kennenlernen, die sich diese Eigenschaft zunutze machen, um den Fréchet-Abstand für Kurven zu berechnen.

Auf Flächen sind die Homöomorphismen schwieriger zu charakterisieren. Dieses Schwierigkeit wird uns in Kapitel 3 beschäftigen. Da die Berechnung des Fréchet-Abstands bereits für Flächen sehr schwierig ist, betrachten wir keine höherdimensionalen Wertebereiche.

Befinden sich die Kurven in $\mathbb{R}^1$, also auf einer Geraden, vereinfacht sich die Berechnung des Fréchet-Abstands. Für diesen Fall sind sehr effiziente Algorithmen bekannt [37]. Im Folgenden, und auch in unserer Implementierung Fréchet View, bewegen wir uns jedoch immer in $\mathbb{R}^2$.

1.3.1. Mann und Hund

Die Variablen s und t der Reparametrisierungen können wir als Zeitvariablen interpretieren. Dies inspiriert ein anschauliches Bild für den Fréchet-Abstand (Abb. 1.2).

[2] Maurice René Fréchet, 1878-1973

Ein Mann führt seinen Hund an der Leine. Der Mann bewegt sich auf einer Kurve, der Hund folgt der anderen Kurve. Beide können sich mit wechselnden Geschwindigkeiten bewegen, aber immer nur vorwärts (wegen der Monotonie der Reparametrisierungen).

Die Frage ist nun: wie lange muss die Leine *mindestens* sein? Der Fréchet-Abstand δ_F ist die minimale Länge der Leine, so dass Mann und Hund beide Kurven abschreiten können.

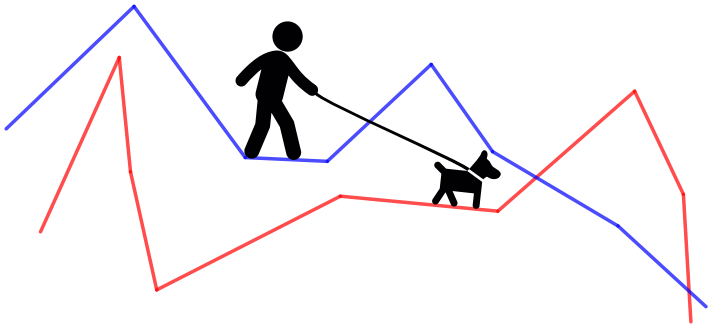


Abbildung 1.2.: Mann und Hund

1.4. Das Entscheidungsproblem

Oft müssen wir für eine Anwendung den Fréchet-Abstand nicht genau berechnen, sondern nur nach oben abschätzen. Das **Entscheidungsproblem** fragt für ein vorgegebenes ε, ob

$$\delta_F(P, Q) \leq \varepsilon \quad ?$$

Übertragen auf unser Bild: wir haben eine Leine mit Länge ε. Ist die Leine ausreichend lang, damit Mann und Hund beide Kurven abschreiten können?

Wir formulieren das Entscheidungsproblem zunächst etwas formaler, weil wir diese Definition später noch benötigen werden.

Definition 1.4 Realisierende Abbildungen [27, S. 4]
Wir bezeichnen zwei Abbildungen $\sigma : B \to A$ und $\tau : A \to B$ als *ε-realisierende* Abbildungen, wenn

$$\sup_{s \in A,\, t \in B} ||P(\sigma(s)) - Q(\tau(t))|| = \varepsilon .$$

Mit diesem Begriff können wir das Entscheidungsproblem etwas umformulieren:

Definition 1.5 Entscheidungsproblem [27, S. 4]
Es ist genau dann $\delta_F(P, Q) \leq \varepsilon$, wenn für alle $\varepsilon' > \varepsilon$ ein Paar von ε'-realisierenden Homöomorphismen existiert.

1.5. Das Optimierungsproblem

Die Berechnung des tatsächlichen Werts $\delta_F(P, Q)$ nennen wir das *Optimierungsproblem*, oder Berechnungsproblem.

Damit $\delta_F(P, Q) = \varepsilon$ ist, müssen nicht unbedingt ε-realisierende Homöomorphismen existieren. Es reicht eine Folge ε_i-realisierenden Homöomorphismen, die gegen ε konvergiert, d. h. $\lim_{i \to \infty} \varepsilon_i = \varepsilon$. Eine solche Folge nennen wir ebenfalls *ε-realisierend*.

Beim Entwurf unserer Algorithmen werden wir sehen, dass wir das Optimierungsproblem durch eine wiederholte Lösung des Entscheidungsproblems lösen können. Die Lösung des Entscheidungsproblems wird somit der wichtigste Baustein unserer Algorithmen.

1.6. Polygonzüge

Wir haben den Fréchet-Abstand bisher für beliebige parametrisierte Kurven definiert. Praktikable Algorithmen sind aber nur für stückweise lineare Kurven und Flächen bekannt, d. h. für Polygonzüge und für Polygone. Wir werden daher in dieser Arbeit immer Polygonzüge betrachten. Ein Polygonzug mit n Eckpunkten kann mit einer stückweise linearen Abbildung $P\colon [0,1] \to \mathbb{R}^d$ parametrisiert werden. Wir können dies auch dadurch rechtfertigen, dass sich empirische Daten meist durch Polygonzüge annähern lassen, oder ohnehin nur mit begrenzter Genauigkeit erfasst werden können.

Rote [89] hat sich mit dem Fréchet-Abstand für algebraische (nicht-lineare) Kurven beschäftigt. Für diskrete Wertebereiche eröffnet sich mit dem *diskreten* Fréchet-Abstand ein weites Forschungsfeld, das wir aber nicht betreten werden.

1.7. Eigenschaften des Fréchet-Abstands

Der Fréchet-Abstand δ_F ist symmetrisch bezüglich der Eingabekurven P und Q und er ist invariant für alle orientierungs-erhaltenden Homöomorphismen. Die Dreiecksungleichung gilt [57, S. 145]. Der Fréchet-Abstand bildet somit, genau wie der Hausdorff-Abstand, eine Metrik. Wir betrachten zwei Kurven als äquivalent, wenn $\delta_F(P,Q) = 0$ ist. Diese Kurven haben die gleiche Form und unterscheiden sich nur in ihrer Parametrisierung.

Nicht alle Varianten des Fréchet-Abstand bilden eine Metrik, z. B. ist dies für den schwachen Fréchet-Abstand (den wir später kennenlernen werden) nicht der Fall.

Die Unabhängigkeit von den zugrundeliegenden Parametrisierungen erlaubt uns beim Entwurf der Algorithmen einige Freiheiten. Wir können die Parametrisierungen so wählen, dass sie für den Algorithmus günstig sind. Für Polygonzüge mit n Eckpunkten bietet sich der Wertebereich $A = [0,n]$ an, mit stückweise linearen Parametrisierungen. Auch in Kapitel 3.1 werden wir die zugrundeliegenden Parametrisierungen vereinfachen.

1.8. Alternative Definitionen

Definition 1.3 wurde möglichst allgemein formuliert. In der Literatur finden sich, je nach Problemstellung, oft speziellere Definitionen des Fréchet-Abstands.

Wenn die Wertebereich A, B kompakt sind, so nehmen die ε-realisierenden Abbildungen ihr Supremum tatsächlich an. Wir können in diesem Fall in Definition 1.3 das Supremum „sup“ durch das Maximum „max“ ersetzen.

Alt und Godau [13] sind etwas weniger streng bei der Definition des Fréchet-Abstands. Sie verlangen von den Reparametrisierung keine strenge Monotonie, sondern nur schwache Monotonie. Auf die Berechnung des Fréchet-Abstands für Polygonzüge in Kapitel 2 hat dies aber keinen Einfluss: eine (schwach) monotone Abbildung kann immer durch eine Folge von (streng monotonen) Homöomorphismen angenähert werden, sodass die Definitionen gleichwertig sind. In Definition 1.3 haben wir uns hingegen für die etwas striktere Formulierung mit Homöomorphismen entschieden, um in Kapitel 3 darauf aufzubauen.

Für den naheliegenden Fall, dass die Wertebereiche A und B homöomorph sind, kann man die beiden Reparametrisierungen durch eine einzige ersetzen. Man findet in der Literatur daher sehr häufig die folgende Definition:

Definition 1.6 Fréchet-Abstand

$$\delta_F(P,Q) = \inf_{\sigma} \sup_{t \in A} || P(t) - Q(\sigma(t)) ||$$

über alle orientierungs-erhaltenden Homöomorphismen $\sigma : A \longrightarrow B$.

1.9. Varianten des Fréchet-Abstands

Im Laufe der Zeit wurden zahlreiche Varianten des Fréchet-Abstands für ganz unterschiedliche Anwendungsbereiche erforscht. Den *schwachen* Fréchet-Abstand und den k-Fréchet-Abstand werden wir in Kapitel 4 ausführlich behandeln. Wir stellen – ohne Anspruch auf Vollständigkeit – einige weitere kurz vor.

Diskreter Fréchet-Abstand Wenn wir diskrete Wertebereiche zugrunde legen, können wir den *diskreten* Fréchet-Abstand definieren. Der diskrete Fréchet-Abstand ist ein gut erforschtes Konzept mit Anwendungen z. B. bei der Analyse von Proteinmolekülen [73]. Er lässt sich auch algorithmisch sehr gut beherrschen [1, 5, 24, 32, 56]. Eine gute Übersicht über Algorithmen zum diskreten Fréchet-Abstand kann man sich bei Driemel [54] verschaffen.

Für Polygonzüge mit bestimmten Eigenschaften wird die Berechnung des Fréchet-Abstands einfacher. Für *c-packed* [23] oder *c-straight* Kurven sind effiziente Näherungsalgorithmen bekannt [55]. Der Fréchet-Abstand für *c-bounded* Kurven lässt sich z. B. durch den Hausdorff-Abstand annähern [17].

Der *geodesische* Fréchet-Abstand berücksichtigt die Einbettung von Flächen im Raum [44, 45, 46]. Mann und Hund bewegen sich in einem unebenem Gelände, die Hundeleine muss den Bodenerhebungen folgen.

Der *homotopische* [42, 67] und *isotopische* [36] Fréchet-Abstand berücksichtigt Hindernisse: die Hundeleine muss um einen Baum herumgeführt werden. Der isotopische Fréchet-Abstand hilft beim Berechnen von Übergängen zwischen Kurven („Morphing“) [42].

Der *richtungs-basierte* Fréchet-Abstand [50] ist unabhängig von der Lage und Größe der Kurven. Er betrachtet nicht Folgen von Punkten, sondern die Richtungswechsel der Polygonsegmente.

1.10. Vorverarbeitung

Der Fréchet-Abstand zweier Kurven hängt offenbar von der relativen Lage der Kurven zueinander ab. Je nach Anwendungsfall (z. B. bei der Handschrifterkennung) kann es notwendig sein, die Kurven durch Verschieben, Drehen oder Skalieren zunächst in eine günstige Position zu bringen, die den Fréchet-Abstand minimiert. Mit solchen Fragen werden wir uns in dieser Arbeit aber nicht beschäftigen. Wir setzen immer voraus, dass die Eingabedaten bereits in geeigneter Form vorliegen.

Bei Polygonzügen mit sehr vielen Punkten kann es nötig sein, die Datenmenge zu reduzieren. Driemel et al. [55] beschäftigen sich damit, wie man Polygonzüge so vereinfachen kann, dass die Berechnung des Fréchet-Abstands auch für große Eingabedaten lösbar bleibt.

2

Der Fréchet-Abstand für Polygonzüge

Wir werden uns in diesem Kapitel auf Polygonzüge in der Ebene konzentrieren, d. h. auf stückweise lineare Kurven. Für die meisten praktischen Anwendungen – wie z. B. die Analyse von Handschriften, oder von Wegen in Landkarten – kann man annehmen, dass sich Kurven durch Polygonzüge annähern lassen.

Polygonzüge Ein Polygonzug $P : [0, n] \to \mathbb{R}^2$ ist definiert durch ein Folge von Punkten $p_1, p_2, \ldots, p_n$ mit $p_i \in \mathbb{R}^2$ und den Strecken zwischen diesen Punkten $\overline{p_i p_{i+1}}$. Im Folgenden betrachten wir immer ein Paar von Polygonzügen P und Q und bezeichnen die Anzahl der Punkte in P mit $\mathbf{n}$ und die Anzahl der Punkte in Q mit $\mathbf{m}$. Geschlossene Polygonzüge mit $p_1 = p_n$ oder $q_1 = q_m$ behandeln wir in Kapitel 2.6.

1995 stellten Alt und Godau [13] einen Algorithmus zur Berechnung des Fréchet-Abstands für Polygonzüge vor. Diese Arbeit war der Anfang der weiteren Forschung zum Fréchet-Abstand. Seitdem wurden und werden zahlreiche Varianten des Fréchet-Abstands erforscht und Algorithmen entwickelt (siehe z. B. Kapitel 1.9). Alt und Godau [13] führten auch ein wichtiges Werkzeug ein, nämlich das *Free-Space-Diagramm*. Alle hier vorgestellten Algorithmen bauen auf dem Free-Space-Diagramm auf. In unserem Programm Fréchet View nimmt das Free-Space-Diagramm einen prominenten Platz ein.

In diesem Kapitel stellen wir nun die Algorithmen von Alt und Godau [13] für offene und geschlossene Polygonzüge vor.

2.1. Das Free-Space-Diagramm

Die Schwierigkeit bei der Berechnung des Fréchet-Abstands besteht darin, dass wir ein Minimum über alle Paare von Reparametrisierungen finden müssen. Die Grundidee des Free-Space-Diagramms ist nun, die Wertebereiche der Reparametrisierungen in eine zweidimensionale Struktur zu übertragen.

Im ersten Schritt definieren wir den *Free-Space* für zwei Strecken $P, Q : [0, 1] \to \mathbb{R}^2$.

Definition 2.1 Free-Space zweier Strecken [13, S. 78]

Der **Free-Space** zweier Strecken besteht aus allen Punktepaaren, deren Abstand kleiner oder gleich ε ist:

$$F_\varepsilon(P, Q) = \{(s, t) \in [0, 1]^2 \mid ||P(s) - Q(t)|| \leq \varepsilon\} .$$

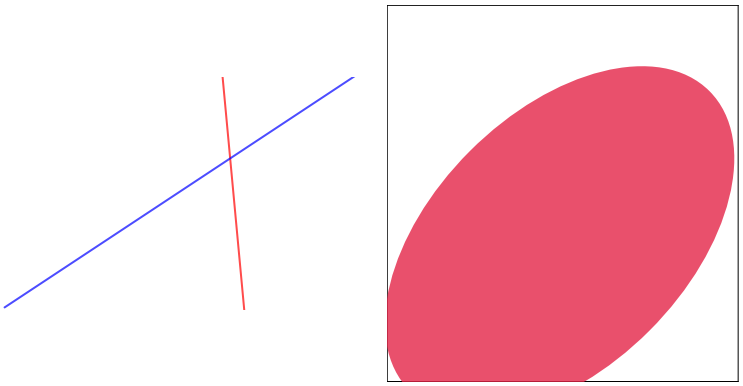


Abbildung 2.1.: Free-Space für zwei Strecken

Wir stellen zunächst fest:

Lemma 2.2 **[13, Lemma 3]**

Der Free-Space für zwei Strecken ist konvex und er besteht aus dem Schnitt einer Ellipse mit dem Einheitsquadrat (siehe Abb. 2.1).

Beweis. Seien $P', Q' : \mathbb{R} \to \mathbb{R}^2$ die Erweiterungen (als Linien) von P und Q auf $\mathbb{R}$. Weiter sei die Abbildung $f : \mathbb{R}^2 \to \mathbb{R}^2$ definiert durch $f(s,t) = P'(s) - Q'(t)$. Da P' und Q' affine Abbildungen sind, ist auch f eine affine Abbildung. Wir können F_ε als Urbild dieser Abbildung charakterisieren:

$$F_\varepsilon = f^{-1}(D_\varepsilon) \cap [0,1]^2 ,$$

wobei $D_\varepsilon = \{p \mid ||p|| \leq \varepsilon\}$ alle Punkte mit Abstand kleiner oder gleich ε beschreibt. D_ε bildet also einen Kreis. Das Bild eines Kreises unter einer affinen Abbildung ist eine Ellipse.

Für den Fall, dass P und Q parallele Segmente bilden, „degeneriert“ der Free-Space zu einer Fläche zwischen zwei Parallelen. □

Im nächsten Schritt erweitern wir den Free-Space auf Polygonzüge:

Definition 2.3 **Free-Space** **[13, S. 78]**

Der Free-Space für Polygonzüge P, Q ist

$$F_\varepsilon(P,Q) = \{(s,t) \in [0,n] \times [0,m] \mid ||P(s) - Q(t)|| \leq \varepsilon\} .$$

Für Polygonzüge wird damit ein Gitter aus $n \times m$ Zellen aufgespannt (siehe Abb. 2.2). Je ein Paar von Strecken bildet eine Zelle des Free-Space-Diagramms. Jede Zelle enthält, wenn sie nicht leer ist, eine angeschnittene Ellipse.

Wie wir weiter oben festgestellt haben, ist der Fréchet-Abstand invariant bezüglich der Parametrisierung der Kurven. Wir können daher eine für unsere Zwecke günstigste Parametrisierung wählen. In der Definition des Free-Space, und auch in der Implementierung von Fréchet View legen wir für Polygonzüge immer die Wertebereiche $[0,n]$ und $[0,m]$ zugrunde. Die Zellen des Free-Space sind Einheitsquadrate.

Graphische Darstellung Für die graphische Darstellung der Free-Space-Diagramme passen wir die Zellen hingegen an die Länge der Polygonsegmente an. Diese Darstellung ist etwas anschaulicher und wird auch in der Literatur meist so verwendet. Auf unsere Algorithmen hat diese Art der Darstellung aber keinen Einfluss.

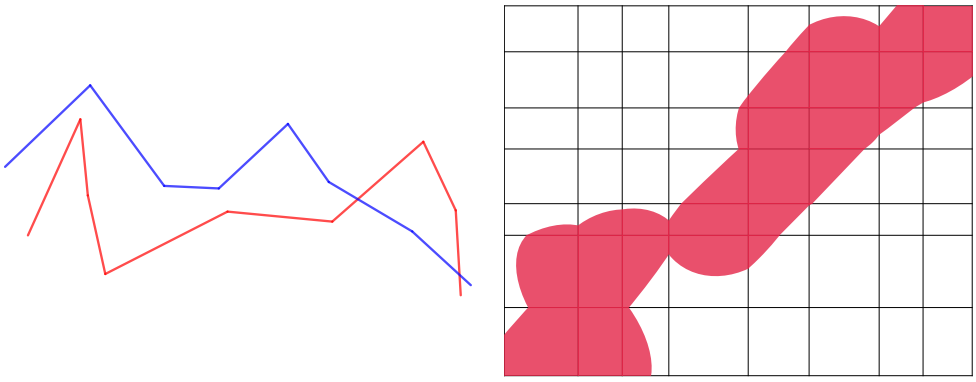


Abbildung 2.2.: Beispiel: zwei Polygonzüge und ein zugehöriges Free-Space-Diagramm

Mit Hilfe des Free-Space-Diagramms können wir nun das Entscheidungsproblem für offene Polygonzüge lösen:

Lemma 2.4 **[13, Lemma 4]**

Für zwei Polygonzüge P, Q ist genau dann $\delta_F(P, Q) \leq \varepsilon$, wenn im zugehörigen Free-Space-Diagramm F_ε ein stetiger, monotoner Pfad von $(0, 0)$ nach (n, m) existiert.
Wir bezeichnen einen solchen Pfad als **gültigen Pfad** (*feasible path*, siehe Abb. 2.3).

Beweis. Sei $f : [0, 1] \to [0, n] \times [0, m]$ ein gültiger Pfad im Free-Space-Diagramm. Indem wir f auf die beiden Wertebereiche projizieren, können wir jeweils zwei Reparametrisierungen $\sigma : [0, 1] \to [0, n]$ und $\tau : [0, 1] \to [0, m]$ ableiten.

Da f in beiden Koordinaten monoton ist, sind auch σ, τ monoton. Für alle $s, t \in [0, 1]$ ist $||P(\sigma(s)) - Q(\tau(t))|| \leq \varepsilon$, d. h. σ, τ sind ε-realisierende Abbildungen. Wir können σ und τ jeweils durch eine Folge von ε-realisierenden Homöomorphismen annähern, die Definition 1.5 genügen.

Ist andererseits $\delta_F(P, Q) \leq \varepsilon$, so gibt es zwei Reparametrisierungen σ, τ, aus denen wir einen gültigen Pfad f konstruieren können. □

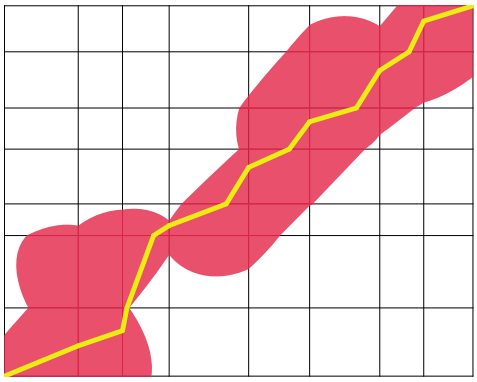


Abbildung 2.3.: Ein gültiger Pfad im Free-Space-Diagramm

Bemerkung Alt und Godau [13] definieren den Fréchet-Abstand geringfügig anders. Sie lassen bereits in der Definition (schwach) monotone Reparametrisierungen zu, während wir auf Homöomorphismen bestehen. Dies ändert aber wenig an unserer Darstellung, außer dass wir den oben stehenden Beweis etwas anders führen müssen.

Während wir von Reparametrisierungen verlangen, dass sie streng monoton sind, dürfen gültige Pfade auch horizontale und vertikale Abschnitte enthalten.

Mit dem schwachen Fréchet-Abstand werden wir in Kapitel 4.1 eine Variante des Fréchet-Abstands kennenlernen, welche auch nicht-monotone (aber surjektive) gültige Pfade zulässt.

2.2. Free-Space-Intervalle

Wie können wir nun einen solchen gültigen Pfad finden? Wir konstruieren ihn Zelle für Zelle im Free-Space-Diagramm. Da der Inhalt einer Free-Space-Zelle konvex ist, müssen wir auch nicht denn genauen Verlauf einer Ellipse innerhalb einer Free-Space-Zelle nachvollziehen. Es reicht zu wissen, an welchem Punkt die Zelle betreten und verlassen wird. Uns interessiert daher nur die Schnittmenge des Free-Space-Diagramms mit den Rändern der Zellen.

Für jede Zelle C_{ij} des Free-Space-Diagramm mit $i \in \{1, \dots, n\}$ und $j \in \{1, \dots, m\}$ bezeichnen wir den linken Rand mit L_{ij} und den unteren Rand mit B_{ij}. Der rechte Rand ist zugleich der linke Rand $L_{i+1,j}$ der benachbarten Zelle. Entsprechend ist $B_{i,j+1}$ der obere Rand.

Nun können wir die **Free-Space-Intervalle** definieren als

$$L_{ij}^F = L_{ij} \cap F_\varepsilon \qquad \text{und} \qquad B_{ij}^F = B_{ij} \cap F_\varepsilon \,.$$

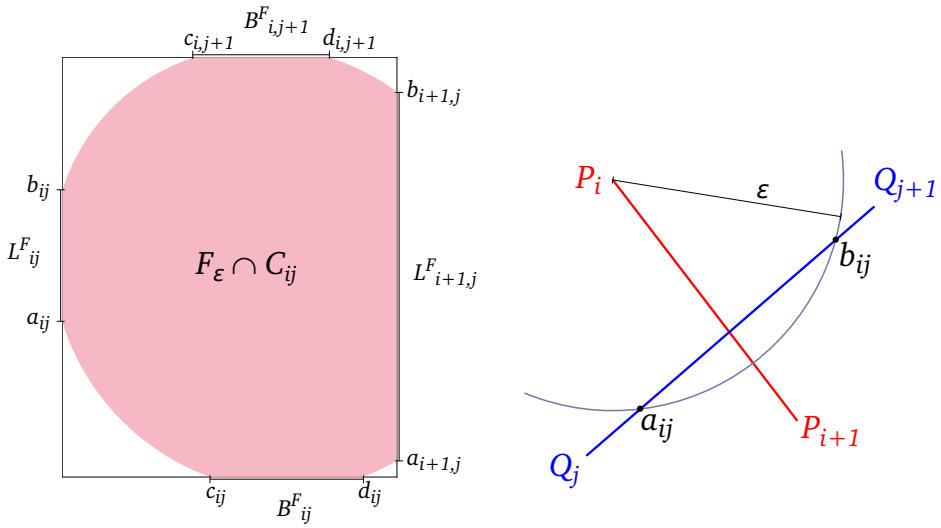


Abbildung 2.4.: Free-Space-Intervalle und ihre geometrische Konstruktion

Aufgrund der Konvexität des Inhalts einer Zelle sind L_{ij}^F und B_{ij}^F immer zusammenhängend. Wir stellen sie deshalb als Intervalle der Form $L_{ij}^F = [a_{ij}, b_{ij}]$, bzw. $B_{ij}^F = [c_{ij}, d_{ij}]$ dar. Wir bezeichnen sie als *Free-Space-Intervalle*.

Die Free-Space-Intervalle $[a_{ij}, b_{ij}]$ und $[c_{ij}, d_{ij}]$ enthalten alle Informationen, mit denen wir das Entscheidungsproblem lösen können. Wir können sie folgendermaßen aus den Eingabedaten berechnen: a_{ij} und b_{ij} ergeben sich, indem wir einen Kreis um den Punkt P_i mit Radius ε mit der Strecke $\overline{Q_jQ_{j+1}}$ schneiden. Entsprechend können wir c_{ij} und d_{ij} berechnen, indem wir einen Kreis um den Punkt Q_j mit Radius ε mit der Strecke $\overline{P_iP_{i+1}}$ schneiden.

Die Berechnung dieser Intervalle bildet die Grundlage aller Algorithmen, die wir im Folgenden beschreiben werden.

2.3. Erreichbare Intervalle

Bei der Konstruktion eines gültigen Pfads müssen wir auch beachten, dass der gültige Pfad monoton verläuft. Wir definieren den **erreichbaren Bereich** des Free-Space als diejenigen Punkte, die mit einem monoten Pfad vom Startpunkt $(0, 0)$ aus erreichbar sind.

Definition 2.5 erreichbarer Bereich

Der **erreichbare Bereich** R_ε sind diejenigen Punkte des Free-Space, für die ein monotoner Pfad vom Punkt $(0, 0)$ existiert. Die **erreichbaren Intervalle** sind die Schnittmengen mit den Rändern der Zellen:

$$L_{ij}^R = L_{ij} \cap R_\varepsilon \qquad \text{und} \qquad B_{ij}^R = B_{ij} \cap R_\varepsilon \ .$$

Das Entscheidungsproblem $\delta_F(P, Q) \leq \varepsilon$ ist lösbar, wenn der Punkt (n, m) erreichbar ist, also wenn $(n, m) \in L_{n+1,m}^R$. Auch ist leicht zu sehen, dass L_{ij}^R und B_{ij}^R wieder Strecken sind, und dass $L_{ij}^R \subseteq L_{ij}^F$ und $B_{ij}^R \subseteq B_{ij}^F$.

Um einen gültigen Pfad zu konstruieren, müssen wir wiederum nicht die genaue Form des erreichbaren Bereichs kennen. Wir müssen lediglich sicherstellen, dass der Austrittspunkt einer Zelle (in beiden Koordinaten) größer oder gleich dem Eintrittspunkt ist. Wenn wir die Intervalle $L_{ij}^R, B_{ij}^R, L_{i+1,j}^F$ und $B_{i,j+1}^F$ kennen, können wir damit leicht (in $O(1)$) die angrenzenden erreichbaren Intervalle $L_{i+1,j}^R$ und $B_{i,j+1}^R$ ermitteln (siehe Beispiel in Abb. 2.5).

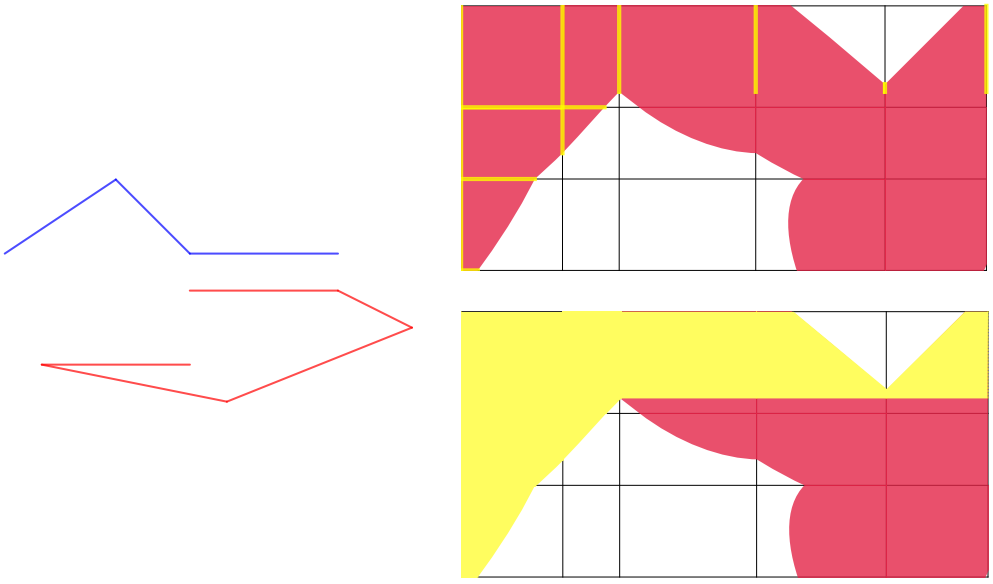


Abbildung 2.5.: Erreichbare Intervalle (oben) und erreichbarer Bereich (gelb) im Free-Space-Diagramm

2.4. Algorithmus für offene Polygonzüge

Schließlich können wir in Listing 2.1 einen Algorithmus formulieren [13, Algorithm 1]. Wir berechnen die erreichbaren Intervalle und lösen damit das Entscheidungsproblem für offene Polygonzüge.

```
for i = {1, ..., n},  j = {1, ..., m}
    berechne die Freespace-Intervalle L^F_ij und B^F_ij

if 0 ∉ L^F_1,1
    return false

L^R_1,1 := L^F_1,1
B^R_1,1 := B^F_1,1

for i = {2, ..., n}                  // unterer Rand des Free−Space
    ermittle B^R_i,1 aus B^R_i−1,1 und B^F_i,1
for j = {2, ..., m}                  // linker Rand des Free−Space
    ermittle L^R_1,j aus L^R_1,j−1 und L^F_1,j
for i ∈ {1, ..., n}                  // innere Zellen
    for j ∈ {1, ..., m}
        ermittle L^R_i+1,j und B^R_i,j+1
        aus L^R_ij,  B^R_ij,  L^F_i+1,j und B^F_i,j+1

return (n, m) ∈ L^R_n+1,m
```

Listing 2.1: Entscheidungsalgorithmus für offene Polygonzüge

2.4.1. Laufzeitanalyse

Als Modell für unsere Komplexitätsanalysen verwenden wir eine RAM-Maschine mit rationalen Zahlen. Wir nehmen an, dass die vier Grundrechenarten in konstanter Zeit ausgeführt werden können. Zur Berechnung der Free-Space-Intervalle müssen wir quadratische Gleichungen lösen. Wir nehmen an, dass der Aufwand zum Berechnen von Wurzeln linear zur Anzahl der Nachkommastellen ist. Bei begrenzter Genauigkeit ist er also ebenfalls konstant.

Alle Intervallberechnungen können somit in $O(1)$ durchgeführt werden. Wir haben zwei verschachtelte Schleifen mit n und m Iterationen. Der obenstehende Algorithmus kann damit das Entscheidungsproblem in $O(nm)$ Schritten lösen.

2.5. Das Optimierungsproblem

Wie können wir nun den tatsächlichen Wert von $\delta_F(P, Q)$ berechnen? Wir suchen das kleinste ε, welches einen gültigen Pfad im Free-Space-Diagramm ermöglicht.

Wenn wir das Free-Space-Diagramm (z. B. mit unserem Programm Fréchet View) beobachten, stellen wir fest, dass der Free-Space mit wachsendem ε auch wächst. Wir suchen zunächst diejenigen Werte von ε, an denen sich die Struktur des Free-Space-Diagramms so ändert, dass neue Pfade entstehen können.

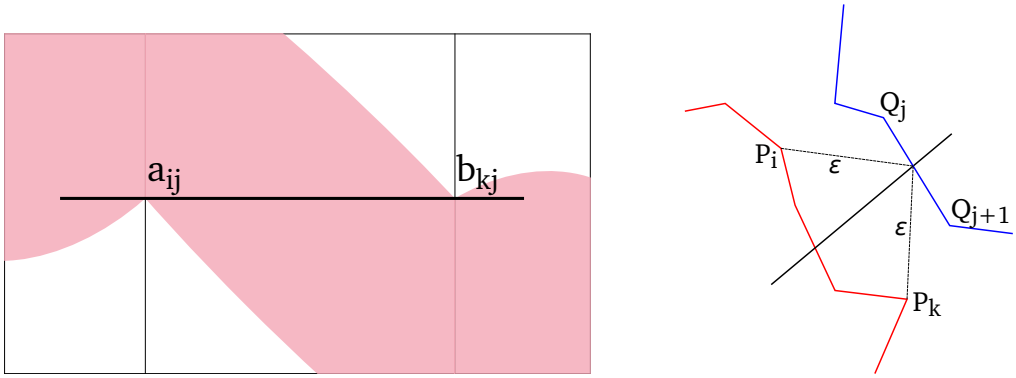
2.5.1. Kritische Werte

Wir unterscheiden die folgenden *kritischen Werte* von ε:

(a) Das kleinste ε, sodass $(0,0) \in F_\varepsilon$ oder $(n,m) \in F_\varepsilon$.

(b) Das kleinste ε, für welches L_{ij}^F oder B_{ij}^F nicht leer sind.
Damit öffnet sich ein neuer Weg zwischen benachbarten Zellen.

(c) Das kleinste ε mit $a_{ij} = b_{kj}$ oder $c_{ij} = d_{ik}$ für irgendein i, j, k.
Ein neuer horizontaler oder vertikaler Weg öffnet sich im Free-Space-Diagramm (siehe Abb. 2.6a).



(a) eine neue Passage öffnet sich im Free-Space-Diagramm (b) geometrische Konstruktion

Abbildung 2.6.: Kritischer Wert vom Typ (c)

Diese kritischen Werte lassen sich folgendermaßen geometrisch interpretieren:

Die kritischen Werte vom Typ (a) entsprechen den Abständen der Endpunkte der Polygonzüge. Es gibt somit zwei kritische Werte vom Typ (a).

Die kritischen Werte vom Typ (b) entsprechen der Entfernung eines Punktes P_i zu einer Strecke $\overline{Q_jQ_{j+1}}$ der anderen Kurve. Es gibt $2nm$ kritische Werte vom Typ (b).

Ein kritischer Wert vom Typ (c) entspricht dem gemeinsamen Abstand zweier Punkte P_i und P_k zu einem Punkt auf Q. Zwei Punkte besitzen den gleichen Abstand, wenn ihr Bisektor eine Strecke $\overline{Q_jQ_{j+1}}$ schneidet (Abb. 2.6b). Dies prüfen wir für jedes Paar von Punkten eines Polygonzugs und für jede Strecke des anderen Polygonzugs. Es gibt damit bis zu $n^2m + nm^2$ kritische Werte dieses Typs.

Insgesamt erhalten wir bis zu $2 + 2nm + n^2m + m^2n = O(n^2m + nm^2)$ kritische Werte. Da sie die kleinsten Werte sind, für die sich die Struktur des Free-Space-Diagramms ändert, ist einer dieser kritischen Werte gleich dem Fréchet-Abstand $\delta_F(P,Q)$.

2.5.2. Binärsuche

Wir können den Fréchet-Abstand berechnen, indem wir eine Binärsuche auf den kritischen Werten durchführen. Dazu muss die Liste der kritischen Werte zunächst sortiert werden. In jedem Schritt der Binärsuche wählen wir einen Wert ε_i aus der Mitte der Liste. Wir lösen das Entscheidungsproblem $\delta_F(P,Q) \leq \varepsilon_i$ und entscheiden, ob wir in der unteren oder oberen Hälfte der Liste weitersuchen. Zuletzt erhalten wir das Ergebnis $\delta_F(P,Q)$.

Laufzeitanalyse

Die Kosten für das Sortieren der Liste betragen

$$O((n^2m + nm^2) \cdot \log(n^2m + nm^2)) = O((n^2m + nm^2) \log nm) \; .$$

Wir führen höchstens $O(\log nm)$ Schritte der Binärsuche durch. In jedem Schritt lösen wir das Entscheidungsproblem in $O(nm)$. Die Gesamtlaufzeit zur Berechnung des Fréchet-Abstand wird somit durch das Sortieren der kritischen Werte dominiert.

2.5.3. Parametrische Suche

Eine weitere Strategie zum Lösen des Optimierungsproblems liefert die Parametrische Suche von Megiddo [79]. Alt und Godau [13] führen weiter aus, dass die Parametrische Suche für die kritischen Werte auf einen Sortieralgorithmus zurückgeführt werden kann und damit eine Technik von Cole [43] anwendbar ist.

Wir wollen hier nicht weiter auf die Parametrische Suche eingehen; diese Techniken sind auch schwierig zu implementieren. Es ist also fraglich, ob man damit den theoretischen Vorteil tatsächlich zur Geltung bringen kann. Alt und Godau [13] zeigen, dass der Fréchet-Abstand für Polygonzüge mit Hilfe der Parametrischen Suche in

$$O(nm \log nm)$$

berechnet werden kann.

Buchin et al. [35] zeigen einen anderen Ansatz zur Lösung des Optimierungsproblems, welcher die Methode der „Vier Russen“ verwendet. Sie erreichen damit eine Laufzeit von $O(n^2(\log \log n)^2)$. Wir werden die Methode der Vier Russen, allerdings in einem anderen Zusammenhang, in Kapitel 6.3 vorstellen.

2.5.4. Näherungsweise Berechnung

Wir geben hier noch einen Näherungsalgorithmus an, der auf einer einfachen Intervallschachtelung beruht. Aus den Eingabedaten können wir eine Obergrenze ε_{max} für $\delta_F(P, Q)$ ableiten, indem wir das umschreibende Rechteck (die Bounding Box) beider Polygonzüge ermitteln. Die Diagonale der Bounding Box ist der größtmögliche Abstand zweier Punkte und bildet damit eine Obergrenze für den Fréchet-Abstand. Eine bessere Obergrenze ε_{max} erhält man, indem man das Maximum aller paarweisen Abstände $||P_i - Q_j||$ berechnet. Der Aufwand hält sich mit $O(nm)$ im Rahmen.

Wir starten mit dem Intervall $[0, \varepsilon_{max}]$ und halbieren das Intervall in jedem Schritt. Ähnlich wie bei der Binärsuche lösen wir in jedem Schritt das Entscheidungsproblem und entscheiden, ob wir in der oberen oder unteren Hälfte weitersuchen. Dies können wir solange durchführen, bis eine vorgegebene Genauigkeit δ erreicht ist. Wir benötigen $\log \frac{\varepsilon_{max}}{\delta}$ Schritte, die Gesamtlaufzeit beträgt somit $O(nm \log \frac{\varepsilon_{max}}{\delta})$.

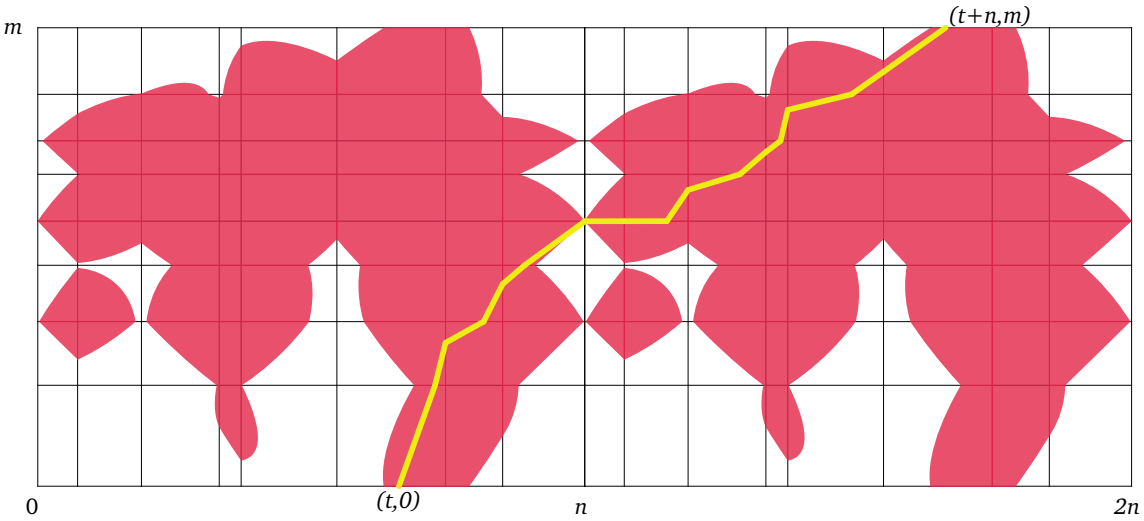
Wir sparen uns hierbei die Berechnung der kritischen Werte, die einen beträchtlichen Aufwand bedeuten kann, wie wir gesehen haben. In Kapitel 5.7.1 werden wir die Idee wieder aufgreifen, um Rundungsfehler in der Implementierung aufzuspüren.

2.6. Geschlossene Polygonzüge

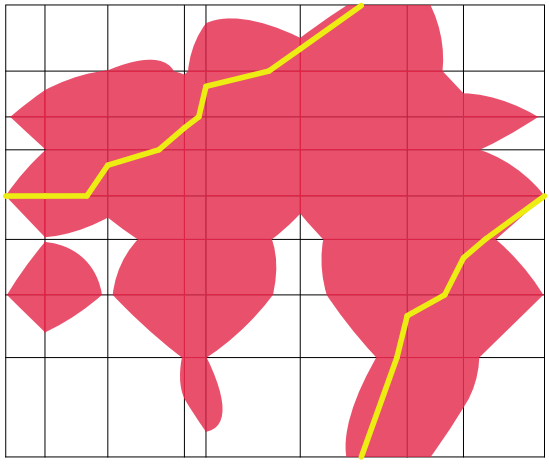
Sind einer oder beide Polygonzüge P und Q geschlossen, so wird das Entscheidungsproblem etwas schwieriger. Mann und Hund können ihren Startpunkt frei wählen. Es reicht nun nicht mehr, einen gültigen Pfad von $(0, 0)$ aus zu suchen; der gültige Pfad kann an jedem anderen Punkt der Kurve beginnen. Wir modellieren das Free-Space-Diagramm für geschlossene Polygonzüge als *doppeltes* Free-Space-Diagramm: die rechte Hälfte ist eine identische Kopie des einfachen Free-Space-Diagramms (Abb. 2.7a). Man kann sich auch vorstellen, den rechten Rand des Free-Space-Diagramm an den linken Rand zu kleben, um damit ein zylinderförmiges Free-Space-Diagramm herzustellen (Abb. 2.7b).

Lemma 2.6 Entscheidungsproblem für geschlossene Polygonzüge [13, Lemma 9]

Das Entscheidungsproblem für geschlossene Polygonzüge ist genau dann lösbar, wenn es ein $t \in [0, n]$ gibt und einen monotonen Pfad von $(t, 0)$ nach $(t + n, m)$ im doppelten Free-Space-Diagramm.



(a) doppeltes Free-Space-Diagramm



(b) einfaches Free-Space-Diagramm mit „Wrap-Around“

Abbildung 2.7.: Free-Space-Diagramm für geschlossene Polygonzüge: unterschiedliche Darstellungen

2.7. Die Erreichbarkeits-Struktur

Um das Entscheidungsproblem für geschlossene Polygonzüge effizient zu lösen, müssen wir die erreichbaren Intervalle anders aufbereiten. Wir führen hierzu eine Datenstruktur, die **Erreichbarkeits-Struktur** ein. Wenn sie fertig erstellt ist, können wir für jedes Intervall in $O(1)$ prüfen, ob ein gültiger Pfad zum oberen Rand des Free-Space existiert.

Die Ränder des Free-Space teilen wir in Intervalle ein, wobei wir für den rechten und linken Rand jeweils die gleiche Unterteilung wählen, ebenso für den unteren und oberen Rand. Im Folgenden bezeichnen wir die Ränder des Free-Space-Diagramms mit L für den linken Rand und R für rechten Rand. Entsprechend B und T für den unteren und oberen Rand. Jedem Intervall $I \subseteq L \cup B$ weisen wir eine von drei Eigenschaften – eine Markierung – zu:

n (*"nicht passierbar"*)	für *keinen* Punkt in I gibt es einen monotonen Pfad in F_ε nach $R \cup T$.
r (*"passierbar"*)	für je zwei Punkte aus I sind die gleichen Punkte in $R \cup T$ erreichbar.
s (*"see-through"*)	die horizontale (bzw. vertikale) Verbindungslinie zum gegenüberliegenden Rand verläuft vollständig in F_ε.

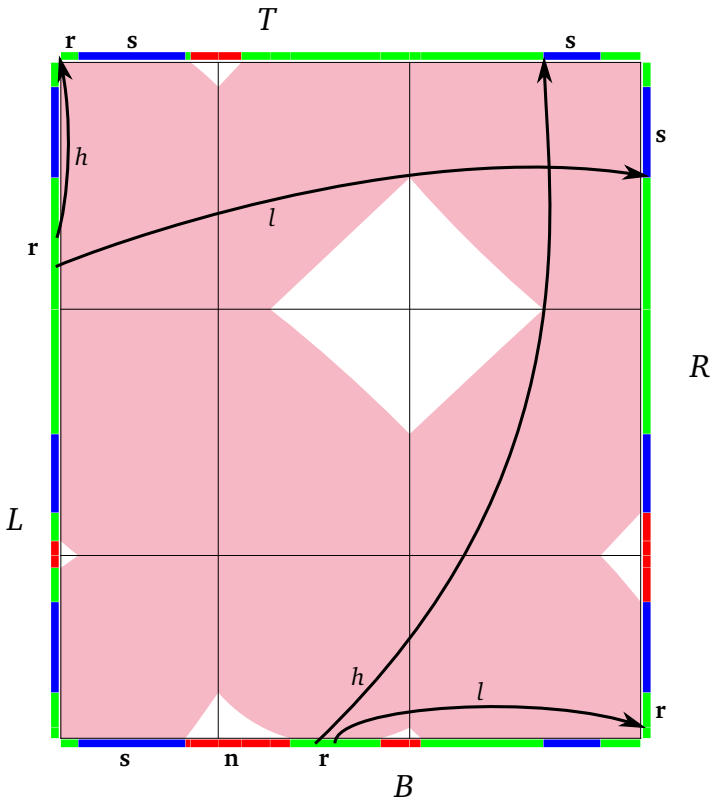


Abbildung 2.8.: Erreichbarkeits-Struktur: die drei Intervalltypen sind farblich markiert. l- und h-Zeiger markieren den erreichbaren Bereich und sind nur für einige Intervalle eingezeichnet. In der Gegenrichtung, von rechts und oben nach links und unten, verlaufen ebenfalls Zeiger.

Jedes *passierbare* **r**-Intervall I erhält einen Verweis auf den von I erreichbaren Bereich in $R \cup T$. Die Zeiger $l(I)$ und $h(I)$ verweisen auf das erste und letzte Intervall in $R \cup T$, das von I aus erreichbar ist. Die l- und h-Zeiger sind dabei immer gegen den Uhrzeigersinn orientiert, d. h. zeigen beide auf den rechten Rand R, so ist $h(I) \geq l(I)$. Verweisen die Zeiger jedoch auf den oberen Rand T, so ist $h(I) \leq l(I)$.

Für *see-through* **s**-Intervalle in L verweist der l-Zeiger auf das direkt gegenüberliegende Intervall in R. Für **s**-Intervalle in B verweist der h-Zeiger auf das direkt gegenüberliegende Intervall in T. Die Speicherung dieser Zeiger ist damit redundant.

Ganz analog teilen wir auch die Ränder R und T in Intervalle ein, markieren sie mit den Typen **n**, **r** und **s** und statten die **r**- und **s**-Intervalle mit Zeigern auf die Intervalle aus, von denen sie erreichbar sind. Die l- und h-Zeiger verweisen hier auf den gegenüberliegenden Rand $L \cup B$ und sind nun *mit dem* Uhrzeigersinn orientiert.

2.7.1. Konstruktion der Erreichbarkeits-Strukturen

Wir bauen die Erreichbarkeits-Strukturen in einem rekursiven Divide-and-Conquer-Verfahren auf. Wir erstellen zunächst Erreichbarkeits-Strukturen für jede Zelle des Free-Space. Dann verbinden wir schrittweise die Erreichbarkeits-Strukturen zu größeren Einheiten, bis wir schließlich eine Erreichbarkeits-Struktur für den gesamten Free-Space erhalten. Bei jedem Vereinigungsschritt achten wir darauf, dass die oben beschriebenen Eigenschaften der Intervalle erhalten bleiben. In Abb. 7.1 auf Seite 97 haben wir den Ablauf des Divide-and-Conquer-Verfahrens skizziert.

Zellen Für eine einzelne Zelle des Free-Space-Diagramms können wir die Intervalle und ihre Typen in $O(1)$ bestimmen. Wir benötigen dazu lediglich die Free-Space-Intervalle der Ränder L^F_{ij}, B^F_{ij}, $L^F_{i+1,j}$ und B^F_{ij+1}.

2.7.2. Zusammenfügen zweier Erreichbarkeits-Strukturen

In dem rekursiven Divide-and-Conquer-Verfahren fügen wir dann je zwei Erreichbarkeits-Strukturen zusammen (vergleiche auch Abb. 7.1 auf Seite 97). Wir beschreiben hier das Zusammenfügen zweier Strukturen D_1 und D_2 entlang der vertikalen Ränder (siehe Abb. 2.10). Entsprechend kann man sie auch entlang der horizontalen Ränder zusammenfügen.

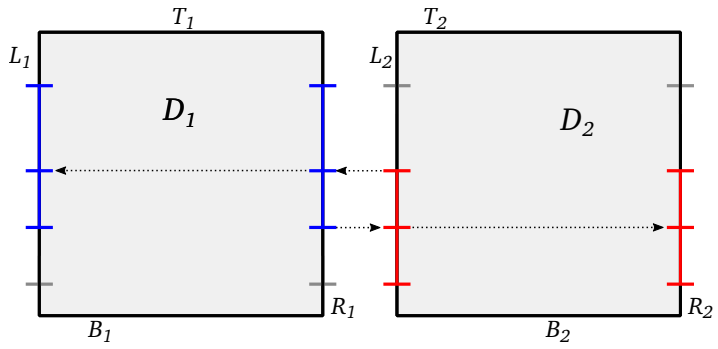


Abbildung 2.9.: Synchronisieren der Intervalleinteilung

Einteilung Zunächst vereinheitlichen wir die Einteilung der Intervalle an den Rändern R_1 und L_2. Die Intervalle werden, falls nötig, aufgeteilt, wobei die Teilintervalle ihren Typ und die l- und h-Zeiger beibehalten. Die Einteilung übertragen wir auch auf die gegenüberliegenden Ränder L_1 und R_2. Nach diesem Schritt besitzen alle vier vertikalen Ränder die gleiche Intervalleinteilung (siehe Abb. 2.9).

Aktualisieren In zweiten Schritt passen wir die Intervalle in $L_1 \cup B_1$ und in $R_2 \cup T_2$ an (die Abschnitte B_2 und T_1 bleiben unverändert).

Wir beschreiben hier, wie wir die Intervalle in $L_1 \cup B_1$ aktualisieren. Der Vorgang für $R_2 \cup T_2$ ist wieder ganz analog.

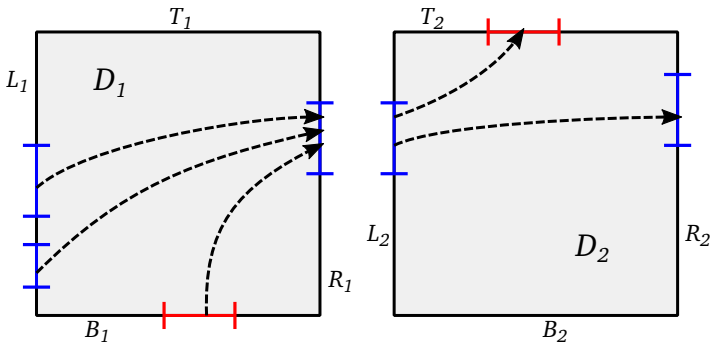


Abbildung 2.10.: Zusammenfügen zweier Erreichbarkeits-Strukturen

Anpassung der n-Markierungen Im ersten Durchgang betrachten wir alle Intervalle in R_1 und L_2, die mit dem Typ **n** (nicht-passierbar) markiert sind. Wir markieren die gegenüberliegenden Intervalle ebenfalls mit **n**.

Alle l- und h-Zeiger, die auf solche **n**-Intervalle verweisen, werden entsprechend angepasst. Die Zeiger werden auf das nächste passierbare (**r**- oder **s**-) Intervall aktualisiert. Wird bei der Anpassung der l- und h-Zeiger ein Bereich leer ($l > h$), so erhält das Intervall eine **n**-Markierung.

Befindet sich ein **s**-Intervall in L_1 direkt gegenüber einem **n**-Intervall, so wird das **s**-Intervall zu einem **r**-Intervall abgestuft.

Anpassung der r- und s-Markierungen Als nächstes betrachten wir die **r**- und **s**-Intervalle in $L_1 \cup B_1$. Alle Pfade von $L_1 \cup B_1$ nach $R_2 \cup T_2$ führen über die Ränder R_1 und L_2. Für jedes Intervall $K \subseteq L_1 \cup B_1$ aktualisieren wir den Typ und die Zeiger folgendermaßen:

1. wenn der h-Zeiger auf ein **r**- oder **s**-Intervall I verweist, so übernehmen wir dessen h-Zeiger: $h(K) := h(I)$.
2. wenn der l-Zeiger auf ein **r**-Intervall I verweist, so übernehmen wir auch den l-Zeiger: $l(K) := l(I)$.
3. wenn der l-Zeiger auf ein **s**-Intervall verweist, verlängern wir den l-Zeiger zum gegenüberliegenden Intervall in R_2.
4. wenn das Intervall K ein **s**-Intervall (*see-through*) ist, und wenn das gegenüberliegende Intervall K' eine **r**- Markierung besitzt, dann wird K ebenfalls zu einem **r**-Intervall herabgestuft. Den l-Zeiger übernehmen wir: $l(K) := l(K')$.
   Für denn Fall, dass sowohl K als auch K' **s**-Intervalle sind, müssen wir nichts tun. K behält seine Markierung, die l-Zeiger werden für **s**-Intervalle nicht gespeichert.

2.7.3. Komplexitätsanalyse

Die Zeit, die wir für das Zusammenfügen zweier Erreichbarkeits-Strukturen benötigen ist proportional zur Anzahl der Intervalle in D_1 und D_2. Da die Intervalle sämtlich aus Free-Space-Intervallen entstehen, ist ihre Anzahl durch $O(nm)$ beschränkt. Für das Zusammenfügen zweier Erreichbarkeits-Strukturen benötigen wir ebenfalls $O(nm)$.

Um alle $O(nm)$ Zellen im Divide-and-Conquer-Verfahren zu größeren Einheiten zusammenzufügen, werden $O(\log nm)$ mal Strukturen zusammengefügt. Wir benötigen somit insgesamt $O(nm \log nm)$ Schritte.

2.8. Lösung des Entscheidungsproblems

Nachdem wir eine Erreichbarkeits-Struktur D für den gesamten Free-Space konstruiert haben, können wir das Entscheidungsproblem für geschlossene Polygonzüge lösen.

Lemma 2.7 **[13, Lemma 10]**

Sei $p \in I \subseteq B$ ein Punkt in einem Intervall am unteren Rand von D und sei $q \in J \subseteq T$ ein Punkt in einem Intervall am oberen Rand. Dann existiert ein monotoner Pfad von p nach q im Free-Space F_ε genau dann, wenn

1. J den Typ **s** oder **r** hat, und wenn

2. a) I den Typ **r** hat und q innerhalb von $l(I)$ und $h(I)$ liegt, oder wenn

2. b) I den Typ **s** hat und q rechts von p und links von $l(I)$ liegt.

Beweis. Die Erreichbarkeits-Struktur wird so konstruiert, dass sie jederzeit die erreichbaren Intervalle wiedergibt. Wenn also der Punkt q von p aus erreichbar ist, müssen beide Intervalle I und J eine **r**- oder **s**-Markierung besitzen (Aussage 1. und 2.). Das Intervall J muss sich in dem von I erreichbaren Bereich befinden, der durch die l- und h-Zeiger markiert wird (Aussage 2a. und 2b.).

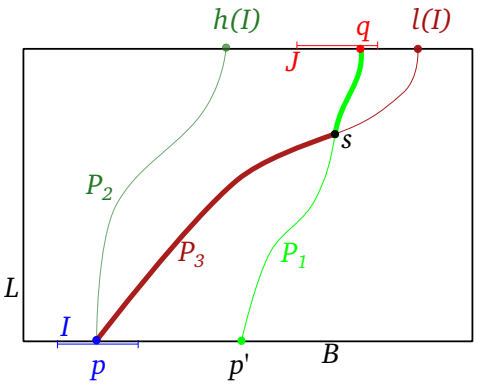


Abbildung 2.11.: Beweisskizze
Zeichnung nach Alt und Godau [13, S. 89]

Für die Umkehrrichtung nehmen wir an, dass die Aussagen 1. und 2. zutreffen. Aus 1. folgt, dass es einen Pfad P_1 nach q geben muss. Den Ausgangspunkt dieses Pfades bezeichnen wir mit $p' \in L \cup B$ (siehe Abb. 2.11). Für $p = p'$ haben wir den gesuchten monotonen Pfad schon gefunden. Ansonsten betrachten wir, ausgehend vom Intervall I, zwei Pfade: ein Pfad P_2 führt von p nach $h(I)$ und ein Pfad P_3 führt von p nach $l(I)$. Für **s**-Intervalle gilt dies ganz entsprechend, nur dass sich das Ziel von P_2 im direkt gegenüberliegenden Intervall befindet (anstelle von $h(I)$).

Einer dieser Pfade, P_2 oder P_3, muss P_1 schneiden (denn $p \neq p'$). Den Schnittpunkt bezeichnen wir mit s. Aus den Teilpfaden von p nach s und von s nach q können wir einen monotonen Pfad von p nach q zusammensetzen. □

Um nun das Entscheidungsproblem zu lösen, benötigen wir einen Pfad von einem Punkt $p \in L$ nach $p + n \in T$. Wir iterieren über die Intervalle in L. Für jedes **r**- oder **s**-Intervall wählen wir einen Punkt p (z. B. die Untergrenze des Intervalls) und prüfen die Bedingungen von Lemma 2.7.

2.8.1. Komplexitätsanalyse

Das Erstellen der Erreichbarkeits-Struktur kann in $O(nm \log nm)$ geschehen, das abschließende Finden einer Lösung in $O(nm)$. Wir können damit das Entscheidungsproblem für geschlossene Polygonzüge in $O(nm \log nm)$ lösen.

Das Optimierungsproblem können wir mit Hilfe der Parametrischen Suche in $O(nm \log^2 nm)$ lösen [13].

Für eine Binärsuche über die kritischen Werte benötigen wir ebenfalls $O(nm \log^2 nm)$ Schritte, wenn die kritischen Werte bereits sortiert vorliegen. Die Binärsuche führt $O(\log nm)$ Iterationen aus, wobei jeweils das Entscheidungsproblem gelöst wird (siehe Kapitel 2.5.2).

Das Berechnen und Sortieren der kritischen Werte ist mit $O((n^2m + nm^2) \log nm)$ aufwendiger als die anschließende Binärsuche – ebenso wie beim Algorithmus für offene Polygonzüge.

2.9. Untere Schranken?

In Kapitel 2.5.3 hatten wir die Laufzeit des Algorithmus von Alt und Godau [13] mit $O(n^2 \log n)$ angeben (mit der Annahme $m = O(n)$). Buchin et al. [35] konnten dieses Ergebnis geringfügig auf $O(n^2(\log \log n)^2)$ verbessern. Als untere Schranke für die Berechnung des Fréchet-Abstands wurde bisher $\Omega(n \log n)$ nachgewiesen [26].

Für die diskrete Variante des Fréchet-Abstands haben Agarwal et al. [1] einen Algorithmus mit $O(n^2 \frac{\log \log n}{\log n})$ vorgestellt. Er ist somit geringfügig schneller als $O(n^2)$. Trotz fortwährender Bemühungen ist bisher aber kein Algorithmus für den kontinuierlichen Fréchet-Abstand bekannt, der mit weniger als $O(n^2)$ Schritten auskommt.

Bringmann [22] hat gezeigt, dass es unter Voraussetzung der *Strong Exponential Time Hypothesis* keinen Algorithmus mit $O(n^d)$ für $d < 2$ geben kann. Wir sagen, es gibt keinen *stark subquadratischen* Algorithmus. Die Strong Exponential Time Hypothesis ist eine Annahme über die Lösung des CNF-SAT-Problems. Man nimmt an, dass sich das CNF-SAT-Problem nur mit einer erschöpfenden Suche in exponentieller Zeit lösen lässt.

Die Schwierigkeit des Fréchet-Abstands ist damit eng verknüpft mit dem CNF-SAT-Problem. Gäbe es einen stark subquadratischen Algorithmus für den Fréchet-Abstand, so müsste sich auch das CNF-SAT-Problem sehr viel schneller lösen lassen, als bisher angenommen wird.

Buchin et al. [38] zeigen, dass auch subquadratischen Näherungsalgorithmen enge Grenzen gesetzt sind.

Wir werten all diese Ergebnisse als Indizien dafür, dass sich für den Fréchet-Abstand die Grenze $O(n^2)$ nicht, oder nur geringfügig (wie bei Agarwal et al. [1]) unterschreiten lässt.

Gudmundsson et al. [63] zeigen einen stark subquadratischen Algorithmus für Kurven mit besonderen Eigenschaften.

3

Der Fréchet-Abstand für einfache Polygone

Die Definition des Fréchet-Abstands lässt sich leicht auf Flächen und andere mehrdimensionale Objekte anwenden. Es zeigt sich aber, dass die Berechnung wesentlich schwieriger wird. Die Frage, ob der Fréchet-Abstand für Flächen im Allgemeinen überhaupt berechenbar ist, ist bisher noch nicht beantwortet. In Kapitel 3.15 werden etwas näher darauf eingehen.

Für einige Klassen von Flächen ist die Situation aber glücklicherweise nicht hoffnungslos. Wir werden z. B. in Kapitel 3.3.6 sehen, dass der Fréchet-Abstand für *konvexe* Polygone sehr einfach zu berechnen ist; er ist nämlich identisch mit dem Hausdorff-Abstand.

In diesem Kapitel werden wir die Arbeit von Buchin et al. [27] vorstellen, die einen polynomiellen Algorithmus für *einfache Polygone* entwickelt haben. Einfache Polygone sind Polygone ohne Löcher und Selbstüberschneidungen. Sie bilden eine natürliche Klasse von Flächen, die in vielen Anwendungen ihren Platz haben. Auf die praktische Implementierung dieses Algorithmus werden wir im zweiten Teil der Arbeit, in den Kapiteln 5 bis 9 eingehen. Soweit wir wissen, ist unsere Implementierung die erste ihrer Art.

Schwierigkeiten Erinnern wir uns an die Definition 1.6 des Fréchet-Abstands. Beim Fréchet-Abstand für Kurven lag ein eindimensionaler Wertebereich zugrunde (z. B. $A = [0, 1]$). Die Reparametrisierungen der Kurven sind streng monotone, bijektive Funktionen.

Beim Fréchet-Abstand für Polygone haben wir zweidimensionale Wertebereiche. Die Reparametrisierungen lassen sich nun nicht mehr so leicht charakterisieren. Auch die bekannte Mann-Hund-Analogie schlägt nun fehl, da wir die Variable t nicht mehr als zeitliche Variable interpretieren können.

Die Idee Die Berechnung wird nur dann beherrschbar, wenn wir es schaffen, die Menge der möglichen Reparametrisierungen zu vereinfachen. Buchin et al. führen das Problem auf Homöomorphismen der Randkurve zurück, die auf die Diagonalen eines Polygons erweitert werden. Die Diagonalen werden auf *kürzeste Wege* im anderen Polygon abgebildet. Für die Darstellung des Algorithmus und seiner Grundlagen folgen wir nun der Arbeit von Buchin et al. [27].

Quellen Zwei vorhergende Veröffentlichungen [25, 28] beschreiben den Algorithmus in etwas anderer Form, nämlich ohne konvexe Zerlegung. Wir beziehen uns im Folgenden immer auf die aktuelleren Arbeiten von Buchin et al. [27, 39].

3.1. Vereinfachung des Fréchet-Abstands

Wenn die beiden Flächen P und Q homöomorph zu ihrem Wertebereich sind, dann können wir die Definition des Fréchet-Abstands etwas vereinfachen. Dies wird für alle Flächen möglich, die sich nicht selbst überschneiden. Wir betrachten im Folgenden zwei ebene, einfache Polygone P und Q. Die Ränder der Polygone sind geschlossene Polygonzüge mit n bzw. m Eckpunkten. Die Polygone enthalten keine Überschneidungen und keine Löcher. Die Polygone dürfen in verschiedenen Ebenen liegen.

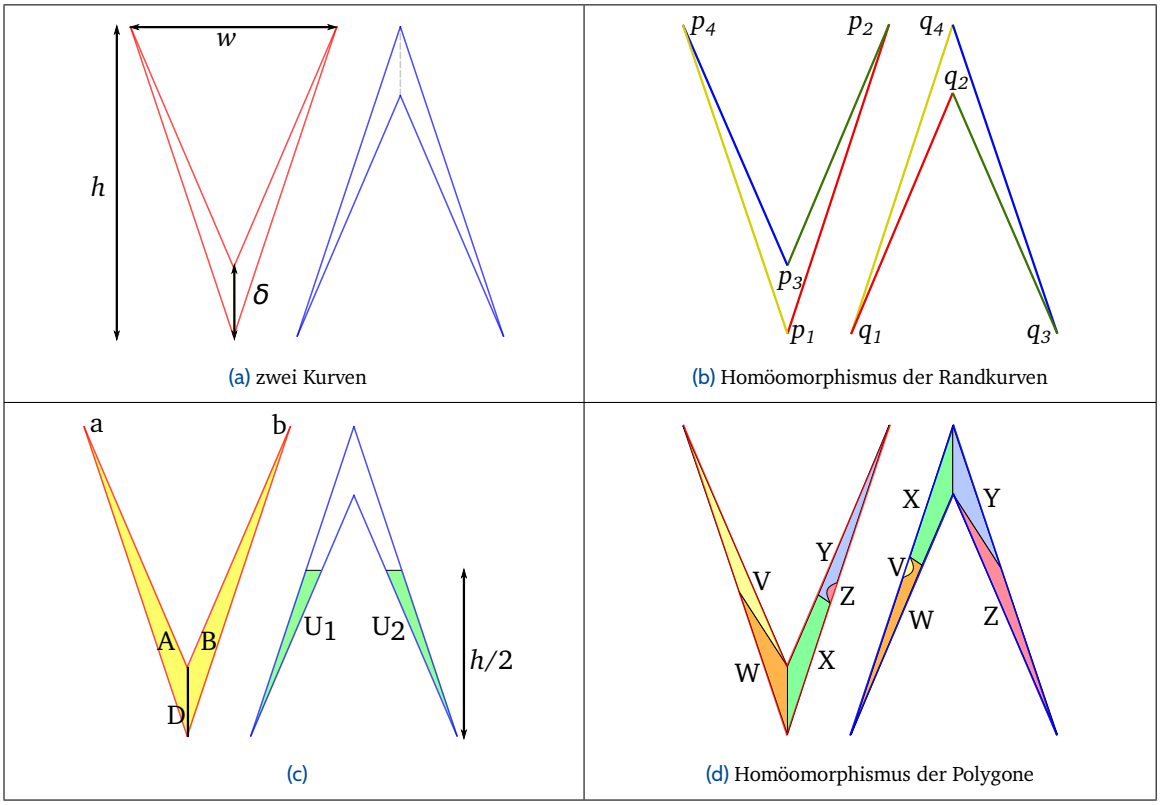
Wir verwenden als naheliegende Parametrisierungen die identischen Abbildungen $f : P \to P$ und $g : Q \to Q$. Damit können wir die Definition des Fréchet-Abstands auch ohne Parametrisierungen unmittelbar auf den Polygonen formulieren:

$$\delta_F(P, Q) = \inf_{\sigma\,:\,P \to Q} \max_{t \in P} ||t - \sigma(t)||$$

über alle orientierungs-erhaltenden Homöomorphismen σ. Die Orientierung eines Polygons wird durch seine Eckpunkte gegeben und bleibt unter σ erhalten. Die Erweiterung auf orientierungsumkehrende Homöomorphismen ist möglich, soll aber hier nicht stattfinden.

3.2. Unterschied zum Fréchet-Abstand der Randkurve

Zunächst stellt sich die Frage, ob sich der Fréchet-Abstand zweier einfacher Polygone von dem Fréchet-Abstand ihrer Randkurven (also der Polygonzüge) unterscheidet. Buchin et al. [27] demonstrieren dies an dem Beispiel in Abb. 3.1. Die Polygone überlagern sich und sind nur der Übersichtlichkeit halber nebeneinander gezeichnet.



(a) zwei Kurven

(b) Homöomorphismus der Randkurven

(c)

(d) Homöomorphismus der Polygone

Abbildung 3.1.: Beispiel: δ_F des Polygons unterscheidet sich von δ_F der Randkurve
(Zeichnung nach Buchin et al. [27, S. 4])

Randkurven Abb. 3.1b zeigt einen Homöomorphismus auf den Randkurven: die Punkte p_i werden auf q_i abgebildet. Strecken werden auf Strecken gleicher Farbe abgebildet. Die maximale Entfernung zweier Punkte liegt zwischen p_3 und q_3. Für $\delta \to 0$ konvergiert $||p_3 - q_3||$ gegen $\frac{w}{2}$. Für $w \to 0$ konvergiert $||p_3 - q_3||$ gegen 0 und damit konvergiert auch der Fréchet-Abstand der Randkurven $\delta_F(P, Q)$ gegen 0.

Polygone Abb. 3.1c zeigt, dass der Fréchet-Abstand der Polygone hingegen nicht kleiner als $\frac{h-\delta}{2}$ werden kann,[1] vorausgesetzt $4\delta < h$. Damit δ_F kleiner werden könnte, müsste die Diagonale D entweder vollständig in den Bereich U_1 oder in den Bereich U_2 abgebildet werden. Dann müsste der Bereich A oder B ebenfalls nach U_1 oder U_2 abgebildet werden, insbesondere einer der Punkte a und b. Aber die Entfernungen von a und b zu U_1 und U_2 betragen mehr als $\frac{h-\delta}{2}$. Tatsächlich konvergiert auch δ_F gegen $\frac{h}{2}$, wie man beispielsweise an dem Homöomorphismus in Abb. 3.1d sehen kann (Flächen werden auf Flächen gleicher Farbe abgebildet).

Der Fréchet-Abstand der Randkurven konvergiert also gegen 0, wohingegen der Fréchet-Abstand der Polygone gegen $\frac{h}{2}$ konvergiert. Das Beispiel lässt sich übrigens mit unserem Programm Fréchet View nachvollziehen, eine Beispieldatei haben wir beigelegt (siehe Anhang A.1).

[1] Buchin et al. [27] geben die Untergrenze mit $\frac{h}{2}$ an, was nicht ganz korrekt ist.

3.3. Vorbereitungen

3.3.1. Vereinfachung von Kurven

Der Fréchet-Abstand zwischen einer Kurve und einer Strecke wächst nicht, wenn wir einen Teil der Kurve durch eine gerade Strecke ersetzen (siehe Abb. 3.2).

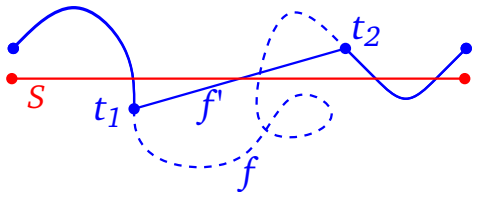


Abbildung 3.2.: Vereinfachung von Kurven
Der Fréchet-Abstand wächst nicht, wenn wir einen Kurvenabschnitt durch eine Strecke ersetzen.
(Zeichnung nach Buchin et al. [27, S. 6])

Lemma 3.1 **[27, Lemma 3]**
Gegeben sind eine Kurve $f\colon [0,1] \to \mathbb{R}^d$ und eine Strecke $s\colon [0,1] \to \mathbb{R}^d$. Außerdem sind $0 \le t_1 < t_2 \le 1$. Für die vereinfachte Kurve $f'\colon [0,1] \to \mathbb{R}^d$ ersetzen wir den Abschnitt $[t_1, t_2]$ durch eine gerade Strecke:

$$f'(t) = \begin{cases} f(t) & \text{für } t \in [0,t_1] \cup [t_2,1] \\ \dfrac{f(t_1)(t_2-t) + f(t_2)(t-t_1)}{t_2-t_1} & \text{für } t \in [t_1,t_2] \end{cases}$$

Der Fréchet-Abstand zwischen f' und s ist nicht größer als der ursprüngliche Fréchet-Abstand zwischen f und s:

$$\delta_F(f',s) \le \delta_F(f,s)\,.$$

Beweis. Wir betrachten einen Homöomorphismus $\sigma\colon [0,1] \to [0,1]$. Daraus konstruieren wir – analog zur Definition von f' – einen Homöomorphismus σ'. In den Intervallen $[0,t_1]$ und $[t_2,1]$ ist σ' identisch mit σ. Den Abschnitt zwischen t_1 und t_2 ersetzen wir durch eine lineare Abbildung. Dann ist

$$\max_{t\in[0,1]} ||f(t) - s(\sigma(t))|| \quad \ge \quad \max_{t\in[0,t_1]\cup[t_2,1]} ||f'(t) - s(\sigma'(t))|| \tag{3.1}$$

$$= \quad \max_{t\in[0,1]} ||f'(t) - s(\sigma'(t))||\,. \tag{3.2}$$

Sowohl f und f', als auch σ und σ' sind im Bereich $[0,t_1] \cup [t_2,1]$ identisch. Daraus folgt die erste Ungleichung 3.1.

Im Intervall $[t_1,t_2]$ bildet f' eine Strecke. Der maximale Abstand zweier Strecken wird an den Endpunkten angenommen. Damit ist für den Abstand zwischen f' und s im Intervall $[t_1,t_2]$:

$$\max_{t\in[t_1,t_2]} ||f'(t) - s(\sigma'(t))|| = \max\{\, ||f'(t_1) - s(\sigma'(t_1))||,\ ||f'(t_2) - s(\sigma'(t_2))||\,\}$$

Daraus ergibt sich die zweite Gleichung 3.2. Wenn der Fréchet-Abstand $\delta_F(f,s)$ durch eine Folge von Homöomorphismen σ_i realisiert wird, dann realisiert die entsprechende Folge σ_i' den Fréchet-Abstand $\delta_F(f',s)$ und mit Ungleichung 3.1 ist deshalb: $\delta_F(f,s) \ge \delta_F(f',s)$. □

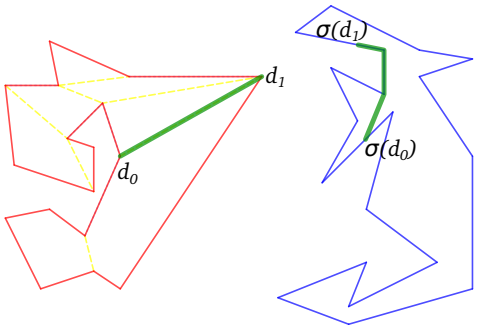


Abbildung 3.3.: Abbildung einer Diagonalen von P auf einen kürzesten Weg in Q.

3.3.2. Shortest-Path-Abbildungen

Am Anfang des Kapitels haben wir gesehen, dass die Schwierigkeit der Berechnung des Fréchet-Abstands für Flächen in der unübersehbar großen Menge von Reparametrisierungen besteht. Die zentrale Idee von Buchin et al. [27] ist nun, das Problem für einfache Polygone auf eine kleine Menge von beherrschbaren Abbildungen zurückzuführen.

Konvexe Zerlegung Wir gehen aus von den Homöomorphismen der Randkurve, die wir schon aus Kapitel 2 kennen. Um diese Abbildungen auf das Innere der Polygone zu erweitern, orientieren wir uns an gewissen Diagonalen, nämlich den Diagonalen einer *konvexen Zerlegung* von P. Für eine konvexe Zerlegung C von P bezeichnen wir mit E_C die Menge aller Punkte auf den Kanten von C, also die Punkte auf der Randkurve und den Diagonalen (siehe Abb. 3.9a auf Seite 45).

Shortest-Path-Abbildung Die Endpunkte der Diagonalen werden bereits durch den Homöomorphismus der Randkurve abgebildet. Die inneren Punkte der Diagonalen bilden wir homöomorph auf den *kürzesten Weg* zwischen den Bildern der Diagonal-Endpunkte in Q ab (siehe Abb. 3.3). Wir nennen eine solche erweiterte Abbildung $\sigma\colon E_C \to Q$ eine *Shortest-Path-Abbildung*. Wenn der Homöomorphismus die Orientierung der Randkurve erhält, dann bezeichnen wir die Shortest-Path-Abbildung ebenfalls als *orientierungs-erhaltend*.

Die Bestandteile einer Shortest-Path-Abbildung, nämlich die Abbildung der Randkurve und die Abbildungen der Diagonalen, sind –für sich genommen– Homöomorphismen. Ingesamt ist eine Shortest-Path-Abbildung aber nicht notwendigerweise ein Homöomorphismus, denn die kürzesten Wege in Q können sich überschneiden, oder teilweise auf der Randkurve verlaufen. Eine Shortest-Path-Abbildung ist also in der Regel nicht surjektiv.

Induzierte Shortest-Path-Abbildung Ausgehend von einem beliebigen Homöomorphismus auf den Polygonen können wir eine Shortest-Path-Abbildung ableiten, indem wir den Homöomorphismus zunächst auf die Randkurve einschränken und dann auf die Diagonalen der konvexen Zerlegung erweitern. Wir nennen eine solche Abbildung eine (durch einen Homöomorphismus) *induzierte* Shortest-Path-Abbildung.

In Lemma 3.2 zeigen wir, dass der durch eine induzierte Shortest-Path-Abbildung realisierte Fréchet-Abstand nicht größer als der vom ursprünglichen Homöomorphismus realisierte Fréchet-Abstand ist.

Danach zeigen wir in Lemma 3.3, dass es umgekehrt zu jeder Shortest-Path-Abbildung σ eine Folge von Homöomorphismen gibt, deren Fréchet-Abstand beliebig nahe an den von σ realisierten Wert herankommt.

Aus diesen beiden Sätzen folgern wir schließlich in Satz 3.4, dass wir den Fréchet-Abstand für einfache Polygone mit Hilfe von Shortest-Path-Abbildungen berechnen können.

3.3.3. Diagonalen auf kürzeste Wege abbilden

Lemma 3.2 **[27, Lemma 4]**

Gegeben ist eine Diagonale D von P und ein Homöomorphismus $\sigma\colon P \to Q$. Sei $\sigma'\colon P \to Q$ eine Abbildung, welche die Diagonale D homöomorph auf den kürzesten Weg zwischen den Bildern ihrer Endpunkte unter σ abbildet. Dann ist

$$\delta_F(D, \sigma'(D)) \;\leq\; \delta_F(D, \sigma(D)) \,.$$

Beweis. Wir skizzieren den Beweis anhand von Abb. 3.4. Die Zeichnung zeigt das Polygon Q. Die Endpunkte der Diagonalen D von P bezeichnen wir mit d_0 und d_1. Mit S bezeichnen wir den kürzesten Weg zwischen den Bildern der Diagonal-Endpunkte (in der Skizze blau). S besteht aus einem Polygonzug mit l Segmenten. Somit ist $s_0 = \sigma'(d_0) = \sigma(d_0)$ und $s_l = \sigma'(d_1) = \sigma(d_1)$.

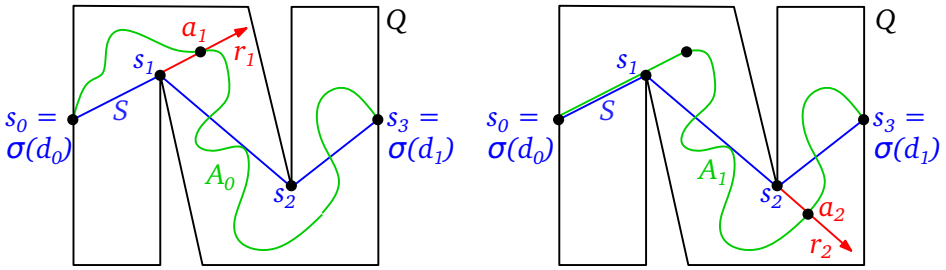


Abbildung 3.4.: Vereinfachung der Abbildung zum kürzesten Weg. Schritt von A_1 nach A_2.
(Zeichnung: Buchin et al. [27, S. 7])

Die Kurve $A = \sigma(D)$ ist das Bild von D unter σ (in der Skizze grün). Mit Hilfe von Lemma 3.1 werden wir schrittweise die Kurve A in S überführen. Wir ersetzen jeweils einen Abschnitt von A durch eine Strecke von S. Die durch die schrittweise Ersetzung entstehende Folge von Kurven bezeichnen wir mit $A_0 = A = \sigma(D),\ A_1, A_2, \ldots, A_l = S = \sigma'(D)$.

Wir beschreiben einen Schritt von A_{i-1} nach A_i. Dazu verlängern wir die Strecke $\overline{s_{i-1}s_i}$ von S. Die Verlängerung r_i (der rote Pfeil in der Skizze) zerlegt das Polygon Q so in zwei Teile, dass s_0 und s_l auf unterschiedlichen Seiten von r_i liegen. Damit wissen wir, dass r_i die Kurve A_{i-1} schneidet. Mit a_i bezeichnen wir den ersten Schnittpunkt zwischen r_i und A_{i-1}.

Die vereinfachte Kurve A_i entsteht, indem wir den Abschnitt von s_{i-1} nach a_i durch die Strecke $\overline{s_{i-1}a_i}$ ersetzen. Mit Lemma 3.1 ist dann $\delta_F(D, A_i) \leq \delta_F(D, A_{i-1})$. Wenn wir nach l Schritten A vollständig ersetzt haben, ist

$$\delta_F(D, S) \;\leq\; \delta_F(D, A) \,.$$

□

3.3.4. Shortest-Path-Abbildung durch Homöomorphismen annähern

Lemma 3.3 **[27, Lemma 5]**

Gegeben sind eine konvexe Zerlegung C von P und eine Shortest-Path-Abbildung $\sigma'\colon E_C \to Q$. Dann existiert für alle $\delta > 0$ ein Homöomorphismus $\sigma_\delta\colon P \to Q$, der einen Fréchet-Abstand realisiert, welcher nicht größer ist als der von σ' realisierte Fréchet-Abstand plus δ:

$$\max_{t \in P} ||t - \sigma_\delta(t)|| \;\leq\; \max_{t \in E_C} ||t - \sigma'(t)|| + \delta \,.$$

Beweis. Wir skizzieren hier nur die Grundidee des Beweises. Für die Details sei der interessierte Leser auf Buchin et al. [27, S. 8-10] verwiesen.

Wir erstellen zwei Triangulierungen der Polygone P und Q. Wir beginnen zunächst mit einer konvexen Zerlegung von P. Die Endpunkte der Diagonalen der konvexen Zerlegung bilden wir mit σ' auf Q ab. Die Diagonalen selbst bilden wir auf die kürzesten Wege zwischen den Bildern der Endpunkte in Q ab. Diejenigen Abschnitte der kürzesten Wege, die auf der Randkurve verlaufen, werden um einen kleinen Betrag ins Innere des Polygons verschoben (siehe Abb. 3.5). Das Ziel ist, dass wir mit diesen (evtl. leicht veränderten) Wegen eine Zerlegung von Q erhalten.

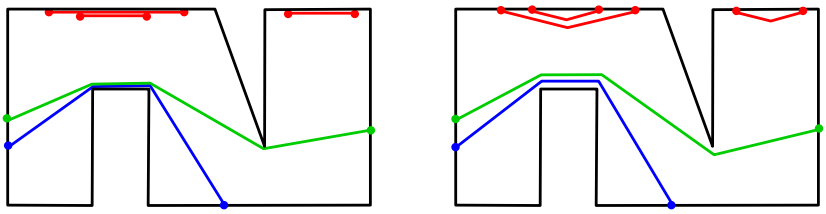


Abbildung 3.5.: Zerlegung von Q: Anpassung der kürzesten Wege.
(Zeichnung: Buchin et al. [27, S. 8])

In den nächsten Schritten werden die Zerlegungen von P und Q zu Triangulierungen ausgebaut, indem wir neue Diagonalen einfügen. Bei der Erstellung der Triangulierungen sind einige Details zu beachten, die bei Buchin et al. [27] beschrieben werden.

Gleichzeitig mit den Triangulierungen entsteht der Homöomorphismus σ_δ. Für die Eckpunkte von P ist σ_δ identisch mit σ', d. h. $\sigma_\delta(p) = \sigma'(p)$. Ebenso ist σ_δ identisch mit σ' für alle Diagonal-Endpunkte, die wir neu in die Triangulierungen einfügen.

Nachdem wir σ_δ für die Eckpunkte der Triangulierung festgelegt haben, können wir σ_δ linear auf die Diagonalen erweitern und dann auf das Innere der Dreiecke. Wir erhalten so einen Homöomorphismus zwischen P und Q, der die Bedingung von Lemma 3.3 erfüllt. □

3.3.5. Fréchet-Abstand mit Shortest-Path-Abbildungen

Satz 3.4 **[27, Proposition 6]**

Der Fréchet-Abstand für einfache Polygone P und Q ist

$$\delta_F(P,Q) \;=\; \inf_{\sigma' :\, E_C \to Q} \; \max_{t \in E_C} \; ||t - \sigma'(t)||$$

über alle orientierungs-erhaltenden Shortest-Path-Abbildungen $\sigma' : E_C \to Q$. C ist eine beliebige konvexe Zerlegung von P.

Beweis.

$$\begin{aligned} \delta_F(P,Q) \le \varepsilon \;&\Leftrightarrow\; \text{für alle } \varepsilon' > \varepsilon \text{ existiert ein } \varepsilon'\text{-realisierender Homöomorphismus} \\ &\Leftrightarrow\; \text{für alle } \varepsilon' > \varepsilon \text{ existiert eine } \varepsilon'\text{-realisierende Shortest-Path-Abbildung.} \end{aligned}$$

Die erste Äquivalenz folgt aus der Definition 1.5 des Fréchet-Abstands. Für die zweite Äquivalenz sei nun σ ein ε'-realisierender Homöomorphismus. Nach Lemma 3.2 realisiert die induzierte Shortest-Path-Abbildung ebenfalls einen Fréchet-Abstand von höchstens ε'. Die Gegenrichtung der zweiten Äquivalenz folgt aus Lemma 3.3. □

Damit haben wir unser wichtigstes Ziel erreicht, die Berechnung der Fréchet-Abstands für einfache Polygone auf Shortest-Path-Abbildungen zu reduzieren.

3.3.6. Konvexe Polygone

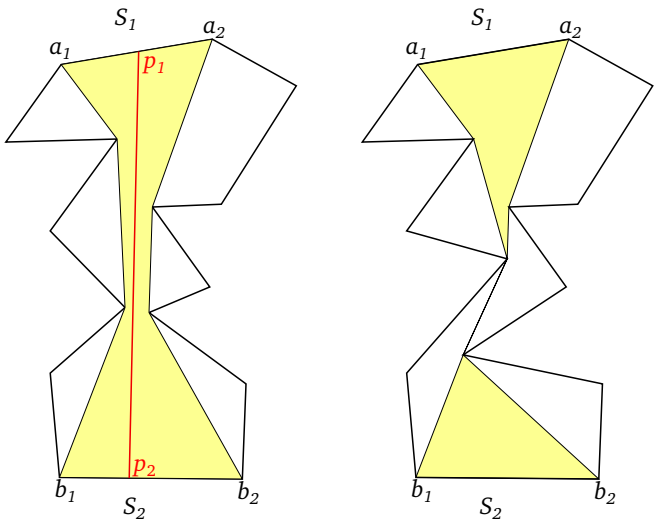
Ist eines der beiden Polygone P oder Q konvex, wird die Berechnung sehr viel einfacher. Bei einem konvexen Polygon besteht die konvexe Zerlegung nur aus einem Teil, nämlich dem Polygon selbst. Eine Shortest-Path-Abbildung ist in diesem Fall (per Definition) identisch mit einem Homöomorphismus der Randkurve.

Mit Satz 3.4 reicht es also, die Randkurve des konvexen Polygons auf die Randkurve des zweiten Polygons abzubilden. Anders gesagt, müssen wir nur den Fréchet-Abstand der Randkurven berechnen.

Der Fréchet-Abstand eines konvexen Polygonzugs ist wiederum identisch mit dem Hausdorff-Abstand [13, 60]. Der Hausdorff-Abstand kann in $O((m+n)\log(m+n))$ berechnet werden [15].

3.3.7. Sanduhren

Für unseren Algorithmus werden wir später kürzeste Wege in Polygonen berechnen. Wir führen zunächst ein wichtiges Konzept von Guibas et al. [64, 65] ein, nämlich *Sanduhren*.



(a) Offene Sanduhr (b) Geschlossene Sanduhr

Abbildung 3.6.: Sanduhren

Definition 3.5 Sanduhr [64, S. 20]

Wir betrachten zwei (unterschiedliche) Seiten eines Polygons S_1 und S_2. Die *Sanduhr* zwischen S_1 und S_2 beinhaltet alle kürzesten Wege zwischen allen Punkten von S_1 und S_2 (siehe Abb. 3.6). Wenn es zwei Punkte $p_1 \in S_1$ und $p_2 \in S_2$ gibt, die sich sehen können, sprechen wir von einer *offenen* Sanduhr (Abb. 3.6a), ansonsten von einer *geschlossenen* Sanduhr (Abb. 3.6b).

Die Sanduhr ist das Polygon, welches durch die Seiten S_1, S_2 und durch die kürzesten Wege zwischen den Endpunkten von S_1 und S_2 aufgespannt wird (im Beispiel: die kürzesten Wege von a_1 nach b_1 und von a_2 nach b_2). Wir werden später das Konzept etwas verallgemeinern, indem wir anstelle der Seiten S_1 und S_2 *Streckenzüge* auf dem Rand der Polygone verwenden. Diese Erweiterung ist aber intuitiv verständlich.

3.4. Ein Algorithmus in polynomieller Zeit

Nachdem die theoretische Vorarbeit geleistet ist, machen wir uns nun an die Konstruktion eines Algorithmus, der den Fréchet-Abstand für einfache Polygone berechnet. Wir folgen wieder Buchin et al. [27]. Wir entwickeln zunächst den Algorithmus zur Lösung des Entscheidungsproblems. Das Optimierungsproblem behandeln wir in Kapitel 3.14; wir werden hierfür ganz ähnliche Techniken wie in Kapitel 2.5 anwenden.

Die praktische Umsetzung der Algorithmen und die dabei auftretenden Herausforderungen diskutieren wir ausführlich in den Kapiteln 5 bis 8. Soweit wir wissen, ist unsere Implementierung die erste dieses Algorithmus. Zuletzt stellen wir in Kapitel 9 einige experimentelle Ergebnisse vor, die zeigen sollen, dass unsere Implementierung praxistauglich ist.

Der Algorithmus baut auf der Arbeit von Alt und Godau [13] auf, die wir in Kapitel 2 beschrieben haben. Wir werden vom Free-Space-Diagramm und von den Erreichbarkeits-Strukturen Gebrauch machen.

3.5. Gültige Pfade

Nach Satz 3.4 kann das Entscheidungsproblem des Fréchet-Abstands mit Hilfe von Shortest-Path-Abbildungen gelöst werden. Können wir das Free-Space-Diagramm benutzen, um solche Shortest-Path-Abbildungen zu konstruieren?

Ein monotoner Pfad im Free-Space-Diagramm, wie wir ihn in Lemma 2.4 kennengelernt haben, bildet die Randkurve von P auf die Randkurve von Q ab. Er entspricht einem ε-realisierenden Homöomorphismus der Randkurve (genauer: dem Grenzwert einer Folge von Homöomorphismen). Um den Homöomorphismus zu einer Shortest-Path-Abbildung zu erweitern, müssen wir zusätzlich die Diagonalen der konvexen Zerlegung auf kürzeste Wege in Q abbilden. Wenn dies möglich ist, bezeichnen wir den zugehörigen monotonen Pfad als *gültigen Pfad*. Die gültigen Pfade definieren wir also ganz ähnlich wie in Kapitel 2, aber mit einer zusätzlichen Bedingung.

Zur Überprüfung dieser Bedingung müssen wir für jede Diagonale der konvexen Zerlegung entscheiden, ob der Fréchet-Abstand zwischen der Diagonalen und dem zugehörigen kürzesten Weg kleiner oder gleich ε ist. Leider gibt es unendlich viele gültige Pfade, welche den Fréchet-Abstand ε für die Randkurven realisieren. In Kapitel 3.6 werden wir jedoch zeigen, dass wir für diese Entscheidung nur *einen* Punkt aus einem Free-Space-Intervall wählen müssen.

3.5.1. Freie Intervalle und Platzierungen

Wir ermitteln für jeden Diagonal-Endpunkt die zusammenhängenden vertikalen Intervalle im Free-Space-Diagramm. Wir nennen solche zusammenhängenden Intervalle *freie Intervalle* (siehe Abb. 3.7). Freie Intervalle können auch mehrere Free-Space-Zellen überspannen. Für jeden Diagonal-Endpunkt gibt es höchstens m freie Intervalle (denn das Free-Space-Diagramm hat m Zeilen).

Die Zuordnung von zwei Diagonal-Endpunkten zu einem Paar von freien Intervallen nennen wir eine *Platzierung*. Die beiden freien Intervalle einer Platzierung bilden in Q die Enden einer Sanduhr. Ein Platzierung ist *gültig*, wenn der Fréchet-Abstand zwischen der Diagonalen und dem kürzesten Weg in dieser Sanduhr kleiner oder gleich ε ist.

Mit dem Ergebnis aus Kapitel 3.6 werden wir sehen, dass wir das Entscheidungsproblem für eine Sanduhr lösen können, indem wir zwei *beliebige* Punkte aus den freien Intervallen wählen und das Entscheidungsproblem für diese Punkte lösen.

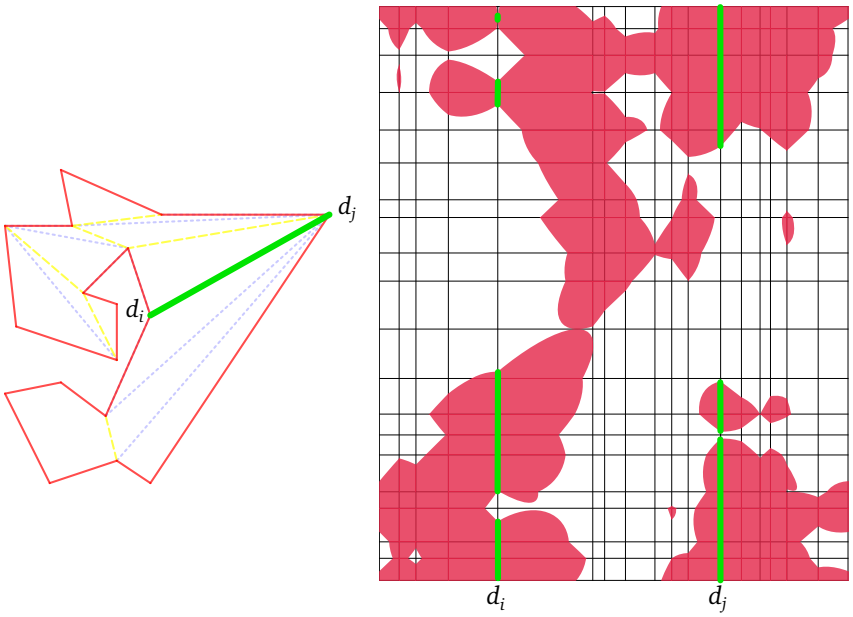


Abbildung 3.7.: Freie Intervalle für zwei Diagonal-Endpunkte im Free-Space-Diagramm

Randfälle Da ein gültiger Pfad nicht notwendigerweise bijektiv ist, enthält er u. U. vertikale Abschnitte. Deswegen ist es möglich, dass die Abbildung eines Diagonal-Endpunkts im gültigen Pfad nicht eindeutig ist. Auch hier hilft uns wieder das Ergebnis aus Kapitel 3.6.

Zur Lösung des Entscheidungsproblems reicht es, einen gültigen Pfad mit gültigen Platzierungen zu finden; wir müssen nicht zwangsläufig einen (streng monotonen) Homöomorphismus konstruieren [27, Korollar 8].

3.6. Fréchet-Abstand zwischen einer Diagonalen und einer Sanduhr

In Kapitel 3.5 müssen wir ein Entscheidungsproblem lösen. Wir wollen wissen, ob $\delta_F(D, s) \leq \varepsilon$ ist für eine Diagonale D und einen kürzesten Weg s in Q. Wir wissen allerdings noch nicht genau, wie wir die Diagonal-Endpunkte auf Q abbilden sollen, und welcher kürzeste Weg sich damit ergibt. Mit Hilfe des folgenden Lemmas werden wir sehen, dass wir das Entscheidungsproblem lösen können, indem wir die Endpunkte in den freien Intervallen *beliebig* wählen.

Lemma 3.6 **[27, Lemma 11]**

Gegeben sind eine Sanduhr und eine Diagonale D. Die Enden der Sanduhr liegen in ε-Kreisen um die Endpunkte der Diagonalen. Wenn es *einen* kürzesten Weg s in der Sanduhr gibt mit $\delta_F(D, s) \leq \varepsilon$, dann gilt dies für *alle* kürzesten Wege in der Sanduhr.

Beweis. Sei $A = a_1, a_2, \ldots, a_l$ ein kürzester Weg in der Sanduhr mit $\delta_F(A, D) \leq \varepsilon$ und sei $B = b_1, b_2, \ldots, B_k$ ein anderer kürzester Weg zwischen den Enden der Sanduhr, wie in Abb. 3.8 angedeutet. Die beiden ε-Kreise um die Endpunkte von D müssen je einen Endpunkt von A und B enthalten. O.B.d.A. nehmen wir an, dass a_1 und b_1 im linken ε-Kreis liegen und a_l und b_k im rechten ε-Kreis.

Wir konstruieren einen Weg B', der von b_1 über $a_1, \ldots, a_l$ nach b_k führt (der gestrichelte Weg in der Skizze). Für diesen Weg ist ebenfalls $\delta_F(B', D) \leq \varepsilon$.

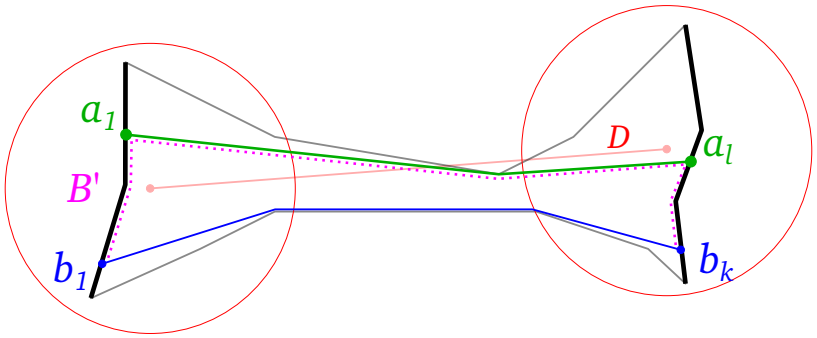


Abbildung 3.8.: Fréchet-Abstand für alle kürzesten Wege einer Sanduhr
(Zeichnung nach Buchin et al. [27, S. 14]))

Wenn B' *nicht* der kürzeste Weg von b_1 nach b_k ist, so gibt es zwei Punkte in B', die wir durch eine gerade Strecke ersetzen können. Wir erhalten damit einen kürzeren Weg B'' innerhalb der Sanduhr. Nach Lemma 3.1 ist $\delta_F(B'', D) \leq \delta_F(B', D)$. Indem wir diesen Prozess fortsetzen, erhalten wir eine Folge von Wegen, die gegen B konvergieren und damit ist $\delta_F(B, D) \leq \delta_F(B', D) \leq \varepsilon$. □

In einer offenen Sanduhr gibt es stets eine Verbindungsstrecke zwischen den Enden der Sanduhr. Wenn diese Enden nun, wie in Lemma 3.6, in einem ε-Kreis um die Endpunkte von D liegen, dann ist auch der Fréchet-Abstand zu D höchstens ε, denn die maximale Entfernung wird an den Enden der Strecken angenommen.

In unserem Algorithmus werden die Enden der Sanduhren von freien Intervallen (Kapitel 3.5.1) gebildet. Alle Punkte eines freien Intervalls sind nicht weiter als ε von einem Diagonal-Endpunkt entfernt, damit können wir Lemma 3.6 anwenden. Wir können das Entscheidungsproblem für alle kürzesten Wege einer Sanduhr lösen, indem es für *irgendeinen* kürzesten Weg lösen. Wir können also beliebige Punkte in den freien Intervallen wählen.

3.6.1. Fréchet-Abstand zwischen einer Diagonalen und mehreren Sanduhren

Als Nächstes wollen wir das Entscheidungsproblem für eine Diagonale und eine Menge von Sanduhren lösen, die jeweils ein Endstück gemeinsam haben.

Wir wählen einen beliebigen Punkt auf dem gemeinsamen Ende der Sanduhren. Dann können wir mit Lemma 3.6 und dem nun folgenden Lemma 3.7 das Entscheidungsproblem in linearer Zeit lösen.

Lemma 3.7 **[27, Lemma 12]**

Gegeben sind eine Diagonale, ein Polygon Q mit m Eckpunkten und eine Menge $\{w_1, w_2, \ldots, w_m\}$ von Punkten auf dem Rand des Polygons. Das Entscheidungsproblem für den Fréchet-Abstand zwischen der Diagonalen und den $m - 1$ kürzesten Wegen $\pi(w_1, w_i)$ für $i = 2, \ldots, m$ kann in $O(m)$ gelöst werden.

Beweis. Zuerst fügen wir die Punkte $w_1, w_2, \ldots, w_m$ zu den Eckpunkten des Polygons hinzu. Dann bestimmen wir die kürzesten Wege von w_1 zu allen anderen Eckpunkten des Polygons. Der Algorithmus von Guibas et al. [64] kann dies in $O(m)$ leisten. Während der Algorithmus die Eckpunkte des Polygons besucht, prüfen wir, ob der Fréchet-Abstand zwischen der Diagonalen und den kürzesten Wegen von w_1 nach w_i (für $i = 2, \ldots, m$) kleiner oder gleich ε ist. Wir erstellen zu diesem Zweck ein Free-Space-Diagramm zwischen der Diagonalen und den kürzesten Wegen. Dieses Free-Space-Diagramm besteht nur aus einer Spalte.

Wir wollen hier nicht auf die Details des Algorithmus von Guibas et al. [64] eingehen. In Kapitel 5.5 haben wir unsere Implementierung beschrieben und auch die wichtigste Datenstruktur des Algorithmus skizziert (Abb. 5.4). Der Algorithmus benutzt diese Datenstruktur, um Punkt für Punkt kürzeste Wege zu konstruieren. Gleichzeitig mit dieser trichterförmigen Datenstruktur können wir unser Free-Space-Diagramm aktualisieren. Mit jedem Punkt, der zu dem kürzesten Weg hinzugefügt wird, berechnen wir eine weitere Zelle des Free-Space-Diagramms. Dies kann in konstanter Zeit erfolgen. Ebenso können die erreichbaren Intervalle in konstanter Zeit aktualisiert werden.

Immer, wenn der Algorithmus einen der Punkte $w_2, \ldots, w_m$ erreicht, können wir ermitteln, ob dieser Punkt im Free-Space-Diagramm erreichbar ist. Damit entscheiden wir, ob der Fréchet-Abstand zwischen der Diagonalen und dem kürzesten Weg $\pi(w_1, w_m)$ kleiner oder gleich ε ist. □

3.7. Erreichbarkeits-Graphen

In Kapitel 2.7 haben wir Erreichbarkeits-Strukturen erstellt, um gültige Pfade im Free-Space-Diagramm zu finden. Im Algorithmus für einfache Polygone suchen wir ebenfalls gültige Pfade, müssen aber zusätzlich die gültigen Platzierungen von Diagonal-Endpunkten berücksichtigen (siehe Kapitel 3.5).

Zu diesem Zweck überführen wir die Erreichbarkeits-Struktur in eine Datenstruktur, die für unseren Algorithmus besser zu handhaben ist. Wir übertragen die Erreichbarkeits-Struktur in einen gerichteten Graphen, den *Erreichbarkeits-Graphen* (Reachability Graph, RG).

Jedes Intervall der Erreichbarkeits-Struktur bildet einen Knoten im Erreichbarkeits-Graphen. Kanten des Graphen entstehen zwischen erreichbaren Intervallen. Ein Erreichbarkeits-Graph enthält $O(mn)$ Knoten und $O((mn)^2)$ Kanten.

Der *Combined Reachability Graph* (CRG) ist ein Teilgraph des Erreichbarkeits-Graphen, welcher nur die Kanten mit *gültigen Platzierungen* der Diagonal-Endpunkte enhält. Der Combined Reachability Graph setzt also die geforderte Bedingung aus Kapitel 3.5 um. In Kapitel 5.6 werden wir uns ausführlich mit der effizienten Implementierung von Combined Reachability Graphs beschäftigen.

Im Laufe des Algorithmus für einfache Polygonen werden wir Erreichbarkeits-Strukturen für Teile des Free-Space erstellen, in Erreichbarkeits-Graphen überführen und dann die Erreichbarkeits-Graphen weiter verarbeiten.

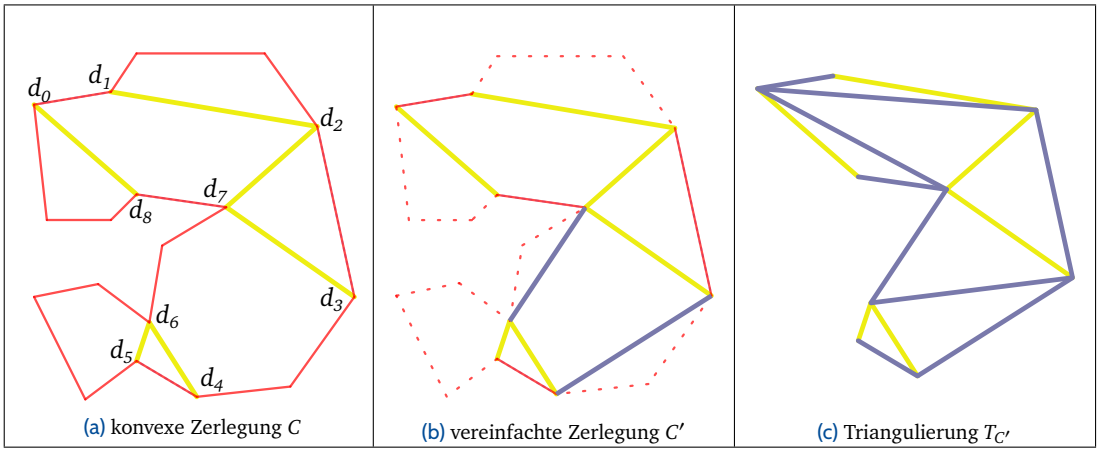
Wie wir in Kapitel 2.7.1 gesehen haben, entstehen die Intervalle der Erreichbarkeits-Struktur durch einen fortgesetzten Prozess der Unterteilung von Free-Space-Intervallen. Für mehrere Erreichbarkeits-Strukturen können damit auch unterschiedliche Intervall-Einteilungen entstehen. Bei der Abbildung auf Erreichbarkeits-Graphen benötigen wir hingegen eine einheitliche Unterteilung, um die Intervalle eindeutig den Knoten der Graphen zuordnen zu können. Damit werden wir uns in Kapitel 5.6 beschäftigen.

3.8. Konvexe Zerlegung und Triangulierung

Einen weiteren Baustein für den Algorithmus bildet die Zerlegung eines der beiden Polygone in konvexe Teile. In Kapitel 3.13.1 werden wir sehen, dass die Anzahl der konvexen Teile einen wichtigen Faktor für die Gesamtlaufzeit des Algorithmus darstellt. Wir machen uns deshalb die Mühe, *beide* Polygone zu zerlegen und wählen dasjenige mit den wenigsten Teilen.

Ein optimale konvexe Zerlegung ist erstrebenswert, aber die Algorithmen sind entsprechend aufwendig. Wir können anstelle der optimalen Zerlegung auch mit einer Näherungslösung leben. Verfügbare Algorithmen zur konvexen Zerlegung und deren praktische Umsetzung diskutieren wir in Kapitel 5.3.

Mit $d_0, d_1, \ldots, d_{l-1}$ bezeichnen wir die l Endpunkte der Diagonalen einer konvexen Zerlegung C von P. Die Diagonalen dieser Zerlegung nennen wir *c-Diagonalen* (c für *convex*). In Abb. 3.9 sind die c-Diagonalen gelb markiert.



(a) konvexe Zerlegung C (b) vereinfachte Zerlegung C' (c) Triangulierung $T_{C'}$

Abbildung 3.9.: Konvexe Zerlegung, c-Diagonalen (gelb) und t-Diagonalen (blau)

Vereinfachte Zerlegung Im Beweis von Lemma 3.8 werden wir sehen, dass nicht alle Bereiche der konvexen Zerlegung C für die Berechnung benötigt werden. Die *vereinfachte* Zerlegung C' enthält nur die Diagonal-Endpunkte $d_0, d_1, \ldots, d_{l-1}$. Alle übrigen Eckpunkte von P werden entfernt (Abb. 3.9b).

Um das Entscheidungsproblem zu lösen, werden wir Erreichbarkeits-Graphen für die Diagonalen zwischen den Punkten $d_0, d_1, \ldots, d_{l-1}$ berechnen. Einige dieser Erreichbarkeits-Graphen werden wir aus den Erreichbarkeits-Strukturen ableiten (Kapitel 3.9). Andere Erreichbarkeits-Graphen entstehen durch Zusammenfügen (Kapitel 3.12). Die Berechnung baut auf einer Triangulierung des Polygons P auf.

Triangulierung Zu diesem Zweck erstellen wir eine Triangulierung $T_{C'}$ der vereinfachten Zerlegung. Eine Triangulierung kann leicht auf Basis der konvexen Zerlegung erstellt werden, indem wir Fächer von Diagonalen in die konvexen Teile einfügen. Wir nennen diese Diagonalen *t-Diagonalen* (t für *Triangulierung*). Auch die Randsegmente der vereinfachten Zerlegung C' bezeichnen wir als t-Diagonalen. In Abb. 3.9c sind die t-Diagonalen blau markiert. Die vereinfachte Triangulierung $T_{C'}$ enthält somit c-Diagonalen und t-Diagonalen.

3.9. Aufteilung des Free-Space

Die Diagonal-Endpunkte $d_0, d_1, \ldots, d_{l-1}$ nummerieren wir entsprechend der Orientierung von P. Im Folgenden schreiben wir $+_l$ und $-_l$ für die Addition und Subtraktion modulo l. Zwei Punkte d_i und $d_{i+_l 1}$ sind in der vereinfachten Zerlegung C' benachbart, ebenso sind $d_{i-_l 1}$ und d_i benachbart.

Wir betrachten nun das Free-Space-Diagramm der Randkurven von P und Q. Wenn wir die Diagonal-Endpunkte $d_0, d_1, \ldots, d_{l-1}$ in das doppelte Free-Space-Diagramm eintragen, erhalten wir wie in Abb. 3.10 eine Aufteilung des Free-Space-Diagramms in vertikale Spalten. (Dass in der Zeichnung der linke Rand des Free-Space-Diagramms mit einem Diagonal-Endpunkt zusammenfällt, ist nur der vereinfachten Darstellung geschuldet. Da beide Randkurven geschlossen sind, können wir das doppelte Free-Space-Diagramm an jedem beliebigen Eckpunkt von P beginnen lassen. Hier ist uns lediglich wichtig, dass die Diagonal-Endpunkte aufsteigend (modulo l) nummeriert sind.)

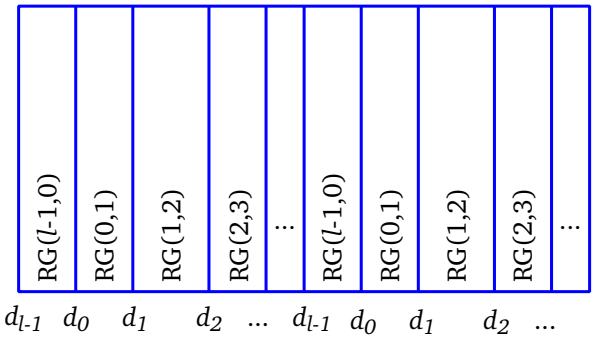


Abbildung 3.10.: Aufteilung des *doppelten* Free-Space in RG-Spalten

Eine vertikale Spalte repräsentiert den Free-Space zwischen einem Abschnitt der Randkurve von P und der gesamten Randkurve von Q. Wir berechnen für jede Spalte die Erreichbarkeits-Struktur und übertragen sie in einen Erreichbarkeits-Graphen. Für zwei benachbarte Diagonal-Endpunkte d_i und $d_{i+_l 1}$ bezeichnen wir den Erreichbarkeits-Graphen mit $\mathbf{RG}(i, i +_l 1)$. Diese Erreichbarkeits-Graphen bilden die Bausteine, aus denen wir schrittweise Erreichbarkeits-Graphen für die gesamte Randkurve erstellen werden.

Wir werden dabei ähnlich vorgehen wie in Kapitel 2, indem wir benachbarte Erreichbarkeits-Graphen zusammenfügen. Dabei müssen wir zusätzlich beachten, dass die Abbildungen der c-Diagonalen (der Diagonalen der konvexen Zerlegung) auf kürzeste Wege in Q *gültig* sind. D. h. der Fréchet-Abstand zwischen Diagonalen und kürzestem Weg muss kleiner oder gleich ε sein. Wir berechnen die *Combined Reachability Graphs* (Kapitel 3.7), die nur gültige Platzierungen der Diagonalen enthalten.

Den Combined Reachability Graph für einen Abschnitt des Free-Space zwischen d_i und d_j bezeichnen wir mit $\mathbf{CRG}(i, j)$. Für denn Fall, dass $j < i$ ist, meinen wir damit den Abschnitt zwischen d_i und dem zweiten Auftreten von d_j in der rechten Hälfte des (doppelten) Free-Space-Diagramms.

3.10. Die MERGE-Operation

Die MERGE-Operation verbindet zwei benachbarte Erreichbarkeits-Graphen, bzw. Combined Reachability Graphs.

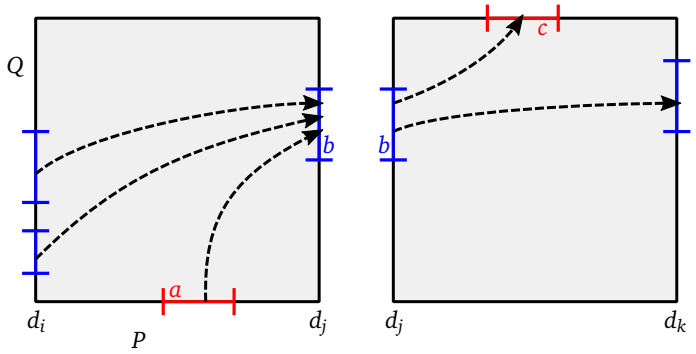


Abbildung 3.11.: MERGE-Operation zweier Erreichbarkeits-Graphen

In Kapitel 2.7.2 hatten wir das Zusammenfügen zweier Erreichbarkeits-Strukturen beschrieben. Die MERGE-Operation arbeitet auf Graphen ganz entsprechend. In Abb. 3.11 haben wir die MERGE-Operation skizziert. Die horizontalen Ränder stehen für benachbarte, disjunkte Abschnitte der Randkurve von P. Sie werden begrenzt durch die Diagonal-Endpunkte d_i, d_j und d_k. Die vertikalen Ränder stehen für die gesamte Randkurve von Q. Die Intervalle an den Rändern entsprechen den Knoten der Graphen. Erreichbare Intervalle werden im Graphen durch gerichtete Kanten dargestellt.

Um zwei Erreichbarkeits-Graphen zu verbinden, bilden wir zunächst deren Vereinigung. In den vertikalen Rändern verbinden wir die Kanten: je zwei Kanten (a, b) und (b, c) ersetzen wir durch eine neue Kante (a, c). Für die vertikalen Ränder bilden wir somit den *transitiven Abschluss*.

In Kapitel 5.6.3 und in den Kapiteln 6 bis 8 werden wir uns sehr ausführlich mit der Implementierung der MERGE-Operation befassen.

3.11. Die COMBINE-Operation

Die COMBINE-Operation wendet die gültigen Platzierungen auf einen Graphen an.

Für einen Combined Reachability Graph CRG (i, j) wollen wir sicherstellen, dass die Diagonale $\overline{d_i d_j}$ gültig platziert ist. Die Diagonale wird durch eine Shortest-Path-Abbildung auf einen kürzesten Weg in Q abgebildet. Die Platzierung ist gültig, wenn der Fréchet-Abstand zwischen Diagonale und kürzestem Weg kleiner oder gleich ε ist.

In Kapitel 3.6.1 haben wir die Berechnung der gültige Platzierungen beschrieben. Wir berechnen initial die gültigen Platzierungen für alle c-Diagonalen und speichern sie in geeigneter Weise ab.

Das Ergebnis der COMBINE-Operation ist ein Teilgraph, in dem nur gültige Platzierungen von Diagonal-Endpunkten enthalten sind. Auch hier werden wir uns Gedanken über die Implementierungen machen; die gültigen Platzierungen sollten in einer Datenstruktur abgelegt werden, die sich leicht auf Erreichbarkeits-Graphen anwenden lässt.

3.12. Berechnung der Combined Reachability Graphs

Wir beschreiben nun, wie wir die Combined Reachability Graphs rekursiv konstruieren können. Ausgehend von den Erreichbarkeits-Graphen für einzelne Spalten des Free-Space entstehen schrittweise Combined Reachability Graphs, die immer größere Teile des Free-Space repräsentieren.

Lemma 3.8 **[27, Lemma 9]**

Für jede c-Diagonale oder t-Diagonale (d_i, d_j) von $T_{C'}$ trifft einer der vier folgenden Fälle zu:

(C1) (d_i, d_j) ist eine c-Diagonale mit $j = i +_l 1$. Dann ist

$$\textsc{crg}(i,j) \;=\; \textsc{combine}(\textsc{rg}(i,j)) \,.$$

(T1) (d_i, d_j) ist eine t-Diagonale mit $j = i +_l 1$. Dann ist

$$\textsc{crg}(i,j) \;=\; \textsc{rg}(i,j) \,.$$

(C2) (d_i, d_j) ist eine c-Diagonale mit $j \neq i +_l 1$. Dann ist

$$\textsc{crg}(i,j) \;=\; \textsc{combine}(\textsc{merge}(\textsc{crg}(i,h), \textsc{crg}(h,j))) \,,$$

wobei (d_i, d_h, d_j) ein Dreieck in $T_{C'}$ darstellt.

(T2) (d_i, d_j) ist eine t-Diagonale mit $j \neq i +_l 1$. Dann ist

$$\textsc{crg}(i,j) \;=\; \textsc{merge}(\textsc{crg}(i,h), \textsc{crg}(h,j)) \,,$$

wobei (d_i, d_h, d_j) ein Dreieck in $T_{C'}$ darstellt.

Beweis. Die Fälle (C1) und (C2) decken die c-Diagonalen ab (die Diagonalen der konvexen Zerlegung), die Fälle (T1) und (T2) die t-Diagonalen der vereinfachten Triangulierung $T_{C'}$.

Die Fälle (C1) und (T1) folgen aus der Definition der Combined Reachability Graphs in Kapitel 3.7. Beide Fälle beschreiben in $T_{C'}$ benachbarte Diagonal-Endpunkte (modulo l).

Für benachbarte Diagonal-Endpunkte berechnen wir die Erreichbarkeits-Graphen aus den Erreichbarkeits-Strukturen, die wiederum auf dem Free-Space-Diagramm der Randkurven basieren.

c-Diagonalen werden durch eine Shortest-Path-Abbildung auf kürzeste Wege in Q abgebildet, deshalb müssen wir zusätzlich sicherstellen, dass die Diagonal-Endpunkte gültig platziert sind, d. h. dass der Fréchet-Abstand zwischen der Diagonalen und dem kürzesten Weg kleiner oder gleich ε ist. Dies leistet die COMBINE-Operation.

Für t-Diagonalen ist die Prüfung auf gültige Platzierungen nicht erforderlich. Der Wertebereich einer Shortest-Path-Abbildung $\sigma : E_C \to Q$ besteht aus der Randkurve und den c-Diagonalen. Die t-Diagonalen werden von σ entweder gar nicht abgebildet, oder sie werden auf die Randkurve abgebildet.

Für die Fälle (C2) und (T2) betrachten wir eine Diagonale (d_i, d_j), mit $j \neq i +_l 1$. d_i und d_j sind in der Triangulierung $T_{C'}$ nicht benachbart. Somit gibt es in $T_{C'}$ mindestens ein an $\overline{d_i d_j}$ angrenzendes Dreieck $\Delta = (d_i, d_h, d_j)$. Für den Fall, dass zwei Dreiecke an $\overline{d_i d_j}$ angrenzen, wählen wir d_h so, dass d_h zwischen d_i und d_j liegt („modulo l“). Falls $i < j$ ist, wählen wir d_h so, dass $i < h < j$. Falls $j < i$ ist, wählen wir d_h so, dass $h < j$ oder $i < h$ ist.

Ein Pfad im Free-Space-Diagramm zwischen d_i und d_j ist genau dann gültig, wenn die beiden Teilpfade zwischen d_i und d_h und zwischen d_h und d_j gültig sind. Der erste Teilpfad muss in $\textsc{crg}(i,h)$ liegen, der zweite Teilpfad muss in $\textsc{crg}(h,j)$ liegen. Indem wir die beiden Graphen mit der MERGE-Operation verbinden, kommen wir zu Fall (T2).

Im Fall (C2) muss außerdem noch die Platzierung von (d_i, d_j) gültig sein. Dies stellen wir wieder durch die COMBINE-Operation sicher, die nur gültige Platzierungen übernimmt. □

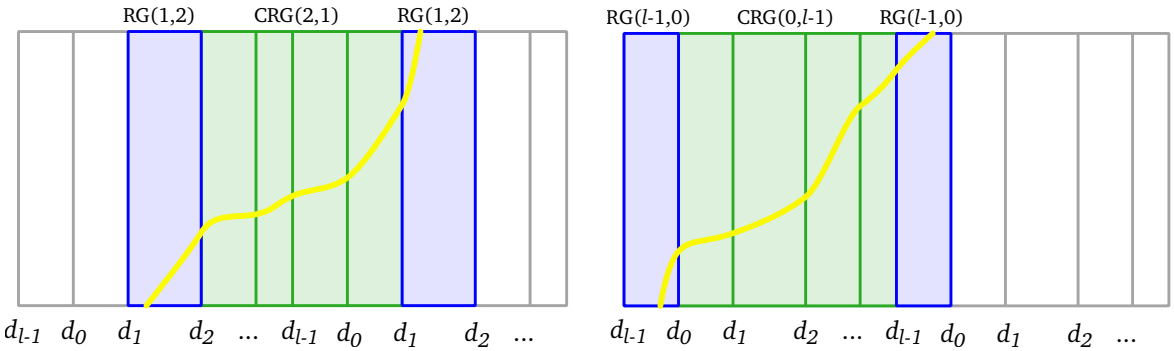
Man beachte, dass die Triangulierung $T_{C'}$ nur zum Einsatz kommt, wenn $j \neq i +_l 1$. Falls d_i und d_j benachbart sind, wird $T_{C'}$ nicht benötigt. Aus diesem Grund konnten wir in Kapitel 3.8 die Randbereiche von E_C zwischen d_i und $d_{i+_l 1}$ aus der Triangulierung entfernen.

Mit Lemma 3.8 können wir nun schließlich die Lösung des Entscheidungsproblems angehen. Ähnlich wie in Kapitel 2.8 suchen wir einen gültigen Pfad im (doppelten) Free-Space-Diagramm. Ein gültiger Pfad beginnt in einem Intervall $[d_{i-_l 1}, d_i)$ am unteren Rand (der linken Hälfte) des Free-Space-Diagramms und er endet im Intervall $[d_{i-_l 1}, d_i)$ am oberen Rand (der rechten Hälfte) des Free-Space-Diagramms (siehe Abb. 3.12).

Ob es eine Verbindung zwischen diesen Intervallen gibt, ermittelt der Erreichbarkeits-Graph

$$G \;=\; \textsc{merge}(\; \textsc{rg}(i -_l 1, i),\; \textsc{crg}(i, i -_l 1),\; \textsc{rg}(i -_l 1, i)\;)$$

Wenn G eine Kante zwischen $[d_{i-_l 1}, d_i)$ und $[d_{i-_l 1}, d_i)$ enthält, so gibt es einen gültigen Pfad und eine positive Lösung des Entscheidungsproblems.



(a) $G = \textsc{merge}(\; \textsc{rg}(1,2),\; \textsc{crg}(2,1),\; \textsc{rg}(1,2)\;)$ (b) $G = \textsc{merge}(\; \textsc{rg}(l-1,0),\; \textsc{crg}(0,l-1),\; \textsc{rg}(l-1,0)\;)$

Abbildung 3.12.: Zwei mögliche Lösungen im Free-Space-Diagramm. Gelb: die gültigen Pfade.

Memoisierung

Um das wiederholte Berechnen von Combined Reachability Graphs zu vermeiden, wenden wir die Technik der *Memoisierung* an. Bereits berechnete Combined Reachability Graphs werden in einer Datenstruktur vorgehalten und bei Bedarf aus dieser Datenstruktur abgerufen.

3.13. Der Entscheidungsalgorithmus

Nun haben wir alle Voraussetzungen, um mit Listing 3.1 den Entscheidungsalgorithmus für einfache Polygone zu formulieren [27, S. 15].

```
1  bool decide−frechet−distance(P, Q, ε)
2  {
3      berechne zwei näherungsweise optimale konvexe Zerlegungen von P und Q
4      C = die Zerlegung mit den wenigsten Teilen
5      l = Anzahl der Diagonal−Endpunkte in C
6      C' = vereinfachte Zerlegung von C
7      T_C' = Triangulation von C'
8
9      berechne das Free−Space−Diagramm(P, Q, ε) der Randkurven
10
11     for i ∈ {0, ..., l − 1}
12         berechne und memoisiere RG(i, i +_l 1)
13
14     for d ∈ { c−Diagonalen von C }
15         for Platzierungen der freien Intervalle im Free−Space
16             if δ_F(d, s) ≤ ε,    s = kürzester Weg in einer Sanduhr
17                 speichere eine gültige Platzierung
18
19     for i ∈ {0, ..., l − 1)
20         berechne und memoisiere CRG(i, i −_l 1) anhand T_C'
21         G = MERGE( RG(i −_l 1, i), CRG(i, i −_l 1), RG(i −_l 1, i) )
22         if G enthält eine Kante von [d_{i−_l 1}, d_i) nach [d_{i−_l 1}, d_i)
23             return true
24
25     return false
26 }
```

Listing 3.1: Entscheidungsalgorithmus für einfache Polygone

In den Zeilen **3-17** werden die Datenstrukturen vorbereitet.

In Zeile **3** berechnen wir konvexe Zerlegungen von P und Q. Da die Anzahl der konvexen Teile einen wichtigen Faktor für die Gesamtlaufzeit darstellt, wählen wir die Zerlegung mit den wenigsten Teilen. Die Zerlegungen sollten näherungsweise optimal sein, d. h. sie sollten sich nur um einen konstanten Faktor von einer optimalen Lösung unterscheiden. Wenn wir die Größe der Zerlegung mit k bezeichnen und die Größe einer optimalen Zerlegung mit $c(P)$, dann ist $k \in O(\min\{c(P), c(Q)\})$. Für l, die Anzahl der Diagonal-Endpunkte ist $l \leq 2k$.

O.B.d.A. bezeichnen wir im Weiteren das Polygon mit den wenigsten konvexen Teilen mit P. Wenn Q weniger Teile hat, tauschen wir einfach die Rollen.

In den Zeilen **6-7** berechnen wir eine (fächerförmige) Triangulation der vereinfachten Zerlegung nach Kapitel 3.8.

In Zeile **9** berechnen wir das Free-Space-Diagramm für die Randkurven. Anschließend (in den Zeilen **11-12**) berechnen wir Erreichbarkeits-Strukturen und überführen sie in Erreichbarkeits-Graphen (siehe Kapitel 3.7). Wir speichern die Graphen in einer Memoisierungs-Struktur.

In den Zeilen **14-17** prüfen wir für alle Platzierungen von c-Diagonalen, ob es sich um gültige Platzierungen handelt. Dazu betrachten wir die freien Intervalle und wählen jeweils einen beliebigen Punkt aus einem freien Intervall. Dann lösen wir, wie im Beweis von Lemma 3.7 beschrieben, das Entscheidungsproblem und speichern die gültigen Platzierungen.

Die Zeilen **19-23** bilden den Hauptteil des Algorithmus. Hier werden nach Lemma 3.8 die Combined Reachability Graphs berechnet. In Zeile **22** geben wir eine positive Antwort, falls ein gültiger Pfad existiert. Der Algorithmus liefert uns keine konkrete Instanz eines gültigen Pfads. Falls gewünscht, können wir aber einen gültigen Pfad aus den memoisierten CRGs rekonstruieren (siehe Kapitel 5.8).

3.13.1. Komplexitätsanalyse

Laufzeit Wir summieren die Laufzeit des Algorithmus zur Lösung des Entscheidungsproblems aus Listing 3.1.

Die Berechnung der konvexen Zerlegung kostet für eine optimale Lösung $O(n^4)$. Näherungsalgorithmen liefern Lösungen z. B. in $O(n)$ oder $O(n \log n)$. In Kapitel 5.3 diskutieren wir die Wahl eines Näherungsalgorithmus unter praktischen Aspekten.

Die Anzahl der konvexen Teile bezeichnen wir mit k. Die Anzahl der Diagonal-Endpunkte ist $l = O(k)$. Die konvexe Zerlegung C und die vereinfachte Zerlegung C' haben $O(k)$ Elemente. Die Triangulierung $T_{C'}$ kann in $O(k)$ berechnet werden, indem wir fächerförmig t-Diagonalen in die konvexe Zerlegung C' einfügen.

Das Free-Space-Diagramm der Randkurven in Zeile **9** kann in $O(mn)$ berechnet werden.

Für die Berechnung der Erreichbarkeits-Graphen in Zeile **12** benötigen wir zunächst die Erreichbarkeits-Strukturen zwischen benachbarten Diagonal-Endpunkten. Jeder Streifen des Free-Space (Kapitel 3.9) besteht aus $m \times n_i$ Zellen mit $i = 1, \dots l$ und $\sum n_i = n$. Die Erreichbarkeits-Strukturen können damit in

$$O\left(\sum_{i=1}^{l} mn_i \log(mn_i)\right) \;=\; O(mn \log mn)$$

berechnet werden (siehe Kapitel 2.7.3).

Zur Abbildung der Erreichbarkeits-Strukturen auf Erreichbarkeits-Graphen berechnen wir eine Unterteilung der Free-Space-Intervalle (siehe Kapitel 5.6.1). Dies kann in $O(mn \log mn)$ geschehen. Mit Hilfe dieser Unterteilung und den Erreichbarkeits-Strukturen können die Erreichbarkeits-Graphen insgesamt in $O(k(mn)^2)$ Schritten berechnet werden.

In den Zeilen **14-17** berechnen wir für alle c-Diagonalen die gültigen Platzierungen. Für einen einzelnen Diagonal-Endpunkt kann dies nach Lemma 3.7 in $O(m)$ geschehen. Für eine Diagonale kann die Prüfung in $O(m^2)$ durchgeführt werden, für alle Diagonalen in $O(km^2)$.

In den Zeilen **19-23** berechnen wir rekursiv die Graphen $\text{CRG}(i, j)$. Dank der Memoisierungs-Technik wird jeder Graph für die c-Diagonalen und die t-Diagonalen von $T_{C'}$ nur einmal berechnet. Es gibt $O(k)$ t-Diagonalen. Insgesamt werden $O(k)$ Combined Reachability Graphs berechnet und memoisiert. Jeder rekursive Aufruf enthält höchstens eine MERGE- und eine COMBINE-Operation. In Zeile **21** benötigen wir eine weitere MERGE-Operation. Insgesamt kommen wir damit auf $O(k)$ MERGE- und COMBINE-Operationen.

Die zu einer Diagonalen benachbarten Dreiecke (Lemma 3.8, Fälle (T2) und (C2)) können in konstanter Zeit ermittelt werden, wenn die Triangulierung $T_{C'}$ in einer geeigneten Datenstruktur abgelegt ist (siehe Kapitel 5.4).

Als Teil der MERGE-Operation bilden wir den transitiven Abschluss zweier Graphen. Der Aufwand zur Berechnung des transitiven Abschlusses entspricht dem einer Multiplikation zweier Matrizen mit $O(mn)$ Zeilen und Spalten. Dies wird z. B. plausibel, wenn man den Graphen mit Hilfe einer Adjazenzmatrix realisiert.

Den Aufwand für die Multiplikation zweier Matrizen mit N Zeilen bezeichnen wir mit $T(N)$. Die theoretische Untergrenze für die Matrizenmultiplikation liegt bei $\Omega(N^2)$. Die besten bekannten Algorithmen kommen auf eine asymptotische Komplexität von $O(N^{2.376})$. Für die praktische Umsetzung eignen sich aber vermutlich eher Algorithmen mit $O(N^3/\log N)$. Wir werden darauf in Kapitel 6 genauer eingehen.

Die COMBINE-Operation prüft die gültigen Platzierungen eines Erreichbarkeits-Graphen mit $O((mn)^2)$ Kanten.

Für die Schleife in Zeile **19-23** benötigen wir damit $O(k \cdot T(mn))$ Schritte. Die Iteration über die Diagonal-Endpunkte in Zeile **19** erhöht die Komplexität nicht, da wir zuvor berechnete Resultate aus der Memoisierungs-Struktur abrufen. Die Abfrage eines gültigen Pfads in Zeile **22** kann in $O(mn)$ geschehen.

Zusammenfassend können wir feststellen, dass die Laufzeit des Algorithmus durch die Schleife in Zeile **19-23** mit

$$O(k \cdot T(mn))$$

dominiert wird. Dadurch wird auch der Aufwand für die Berechnung der konvexen Zerlegung dominiert. Die entscheidenden Faktoren für die Laufzeit des Algorithmus sind also zum einen die Anzahl der konvexen Komponenten und zum anderen der Aufwand für den transitiven Abschluss der Erreichbarkeits-Graphen.

Für unsere Implementierung haben wir in Kapitel 9.1 Messungen durchgeführt, die sich mit dieser Analyse decken.

Speicherplatz Wir benötigen Speicherplatz für $O(k)$ Erreichbarkeits-Graphen und $O(k)$ Combined Reachability Graphs. Jeder diese Graphen hat $O((mn)^2)$ Einträge. Der Speicherplatz für das Free-Space-Diagramm benötigt dagegen nur $O(mn)$. Damit beträgt der insgesamt benötigte Speicherplatz $O(k \cdot (mn)^2)$.

3.14. Der Optimierungsalgorithmus

Zur Berechnung des Fréchet-Abstands für einfache Polygone verwenden wir das gleiche Verfahren wie in Kapitel 2.5 für Polygonzüge. Wir bestimmen eine Menge von kritischen Werten und führen dann eine Binärsuche durch, wobei wir in jedem Schritte der Suche das Entscheidungsproblem lösen.

3.14.1. Kritische Werte

Kritische Werte der Randkurven Die kritischen Werte der Randkurve vom Typ (a) bis (c) sind ebenso kritische Werte für den Fréchet-Abstand der Polygone. Es gibt $O(m^2n + n^2m)$ solche kritischen Werte, ihre Berechnung haben wir in Kapitel 2.5.1 beschrieben.

Kritische Werte für Diagonalen und kürzeste Wege Zusätzlich kommen noch kritische Werte für den Fréchet-Abstand zwischen einer Diagonalen und einem kürzesten Weg in Q hinzu. Die kritischen Werte vom Typ (a) betreffen die Endpunkte der Diagonalen und des kürzesten Weges. Sie werden bereits in den kritischen Werten vom Typ (b) der Randkurve erfasst.

Wir erinnern uns, dass das Free-Space-Diagramm für eine Diagonale und einen kürzesten Weg nur aus einer Spalte besteht (Kapitel 3.6.1). Ein monotoner Pfad in diesem Free-Space-Diagramm kreuzt somit nur horizontale Zellenränder. Die kritischen Werte vom Typ (b) und (c) müssen deshalb nur für die inneren Punkte des kürzesten Weges berechnet werden. Die inneren Punkte eines kürzesten Weges sind immer *konkave* Eckpunkte des Polygons Q. Ein konkaver Eckpunkt hat einen Innenwinkel, der größer als π ist.

Wir berechnen also die kritischen Werte vom Typ (b) und (c) für jede Diagonale von $T_{C'}$ und für alle konkaven Eckpunkte von Q. Für jede Diagonale gibt es $O(m)$ kritische Werte vom Typ (b), die alle jeweils in konstanter Zeit berechnet werden können. Es gibt $O(m^2)$ kritische Werte vom Typ (c), die wiederum in konstanter Zeit berechnet werden können. Die Berechnung der kritischen Werte vom Typ (b) und (c) erfolgt wieder so, wie wir sie in Kapitel 2.5.1 beschrieben haben.

Insgesamt erhalten wir $O(km^2)$ kritische Werte für Diagonalen und kürzeste Wege.

Parametrische Suche? Die Parametrische Suche von Megiddo [79] und Cole [43] bringt uns hier keine Vorteile, denn die Gesamtlaufzeit wird durch die Aufrufe zur Lösung des Entscheidungsproblems dominiert. Beim Algorithmus von Alt und Godau [13] lag die Situation etwas anders, da dort der Aufwand zum Sortieren der kritischen Werte dominierend war. Anstelle der Parametrischen Suche können wir also auch eine Binärsuche (Kapitel 2.5.2) verwenden. Die Binärsuche ist im Übrigen auch sehr viel leichter zu implementieren (Kapitel 5.7).

3.14.2. Komplexitätsanalyse

Wir berechnen $O(m^2n + n^2m)$ kritische Werte. Zum Sortieren der kritischen Werte benötigen wir $O((m^2n + n^2m)\log mn)$ Schritte. Die Binärsuche führt $O(\log mn)$ Schritte aus, in denen jeweils das Entscheidungsproblem in $O(kT(mn))$ gelöst wird. Zur Berechnung des Fréchet-Abstands für einfache Polygone benötigen wir somit

$$O(k \cdot T(mn) \cdot \log mn)$$

Schritte. Der Platzbedarf wird durch die Daten für das Entscheidungsproblem dominiert und beträgt $O(k \cdot (mn)^2)$.

3.15. Weitere Ergebnisse

Die Frage, ob der Fréchet-Abstand für Flächen im Allgemeinen berechenbar ist, wurde noch nicht abschließend beantwortet. Godau [60] zeigt, dass das Problem NP-schwer ist. Alt und Buchin [10, 12] zeigen, dass das Problem *semi-berechenbar* ist: der Wert des Fréchet-Abstands lässt sich von oben annähern, man kann aber nicht sagen, wie schnell diese Annäherung konvergiert.

Einen weiteren großen Schritt hin zur Beantwortung der Frage haben Nayyeri und Xu [81, 82] gemacht. Sie zeigen die Berechenbarkeit für Polygone mit bestimmten Eigenschaften in $\mathbb{R}^3$. Park et al. [87] und Neumann [83] kommen mit einem anderen Berechnungsmodell zu interessanten Ergebnissen über die Berechenbarkeit des Fréchet-Abstands für zweidimensionale Flächen.

Der Algorithmus von Buchin et al. [27] für einfache Polygone ist der erste erfolgreiche Ansatz, um den Fréchet-Abstand für eine bestimmte Klasse von zweidimensionalen Flächen effizient zu berechnen. Buchin et al. [29] erweitern den vorliegenden Algorithmus auf Polygone mit *einem* Loch. Nayyeri und Sidiropoulos [80] zeigen, dass das Problem auch mit einer konstanten Anzahl von Löchern effizient lösbar ist. Hingegen ist das Problem für Polygone mit beliebig vielen Löchern bereits NP-schwer [29].

Für einige andere Klassen von Flächen ist die Berechnung ebenfalls NP-schwer [29]. Damit bleibt nur wenig Spielraum für effiziente Algorithmen. Cook et al. [46, 47] stellen polynomielle Algorithmen für bestimmte Klasse von gefalteten Polygonen vor. Buchin et al. [31] betrachten zeitlich veränderliche Kurven, die sich auch als Flächen interpretieren lassen.

3.15.1. Eine Variante des Algorithmus

Cordell [49] stellt eine Variante des Algorithmus von Buchin et al. [27] vor, die auf einer anders konstruierten Erreichbarkeits-Struktur aufbaut. Diese Datenstruktur kann die MERGE-Operation, die einen wichtigen Faktor für die Gesamtlaufzeit darstellt, in $O(n^2m^3 + n^3m^2)$ durchführen.

Theoretisch liegt unser Algorithmus mit $T(mn) = O((mn)^{2.376})$ zwar geringfügig besser, realistisch ist aber eher $T(mn) = O((mn)^3/\log mn)$, wie wir in Kapitel 6 darlegen werden. Der Algorithmus von Cordell [49] verspricht also eine Beschleunigung um nahezu einen linearen Faktor. Andererseits sind seine Datenstrukturen deutlich komplexer als unsere Booleschen Matrizen, bringen also auch hohe konstante Faktoren mit sich. Ein Vergleich zwischen Implementierungen der beiden Algorithmen könnte interessant sein. Als interessantes „Nebenprodukt“ kann die Datenstruktur von Cordell [49] einen gültigen Pfad unmittelbar aus der Erreichbarkeits-Struktur ableiten.

4

Der k-Fréchet-Abstand

Wir befassen uns nun mit einer weiteren Variante des Fréchet-Abstands, die kürzlich von Buchin und Ryvkin [6, 7, 40] vorgestellt wurde. Der *k-Fréchet-Abstand* eignet sich als Maß für Kurven, deren Ähnlichkeit sich besser stückweise beschreiben lässt. Außerdem stellt der k-Fréchet-Abstand einen natürlichen Übergang zwischen zwei bereits bekannten Abstandsmaßen dar. Als Anwendungsbeispiel kommt die Erkennung von Handschriften in Frage. In Abb. 4.1 haben die mittlere und rechte Kurve einen hohen 2-Fréchet-Abstand, aber einen niedrigen 3-Fréchet-Abstand. Die Ähnlichkeit der Kurven wird erst ersichtlich, wenn wir sie in drei Teile zerlegen.

Eine andere mögliche Anwendung ist die Analyse von Touristenströmen. Touristen steuern die gleichen Punkte (Sehenswürdigkeiten) an. Sie machen dies oft, aber nicht immer, in der gleichen Reihenfolge. Der k-Fréchet-Abstand kann ein Maß dafür sein, wie häufig die Touristen gemeinsamen Wegen folgen.

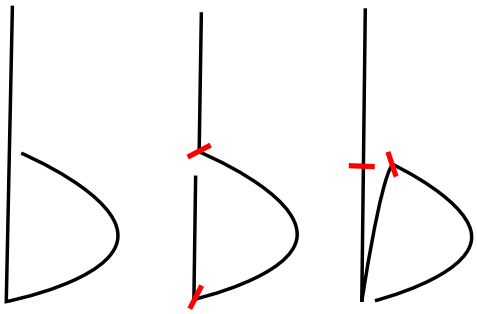


Abbildung 4.1.: Stückweise Ähnlichkeit; drei Schreibweisen des Buchstabens „b“

Viele Fragen zum k-Fréchet-Abstand sind noch Gegenstand der Forschung. Wir zeigen einige aktuelle Ergebnisse und Algorithmen. Die Ausführungen dieses Kapitels folgen den Arbeiten von Buchin und Ryvkin [6, 7, 40].

Unser Beitrag Als Ergänzung der Arbeiten von Buchin und Ryvkin [6, 7, 40] können wir zwei eigene Ergebnisse beisteuern. In Kapitel 4.5.1 beantworten wir eine offene Frage zum Näherungsfaktor eines Algorithmus. In Kapitel 4.6.2 stellen wir zwei parametrisierte Algorithmen vor. Unsere Implementierung (siehe auch Kapitel 5.10) ist die erste Umsetzung der hier vorgestellten Algorithmen.

4.1. Der schwache Fréchet-Abstand

Als Vorbereitung für den k-Fréchet-Abstand führen wir zunächst den schwachen Fréchet-Abstand ein. Der Unterschied zum (starken) Fréchet-Abstand besteht darin, das die Reparametrisierungen von P und Q nicht monoton sein müssen.

Definition 4.1 schwacher Fréchet-Abstand

Gegeben sind zwei Kurven $P, Q : A \to \mathbb{R}^d$ mit $A \subset \mathbb{R}^k$; z. B. $A = [0, 1]$.
Der **schwache Fréchet-Abstand** ist definiert als:

$$\delta_{wF}(P, Q) = \inf_{\sigma,\tau} \max_{s,t \in A} || P(\sigma(s)) - Q(\tau(t)) ||$$

über alle *stetigen* und *surjektiven* Reparametrisierungen $\sigma, \tau : A \to A$.

Übertragen auf das Bild vom Mann und der Hundeleine bedeutet dies: beim (starken) Fréchet-Abstand hatten wir darauf bestanden, dass sich Mann und Hund immer vorwärts bewegen. Beim *schwachen* Fréchet-Abstand dürfen sowohl Mann als auch Hund beliebig oft die Richtung wechseln. Sie dürfen auch ihren Startpunkt frei wählen; sie müssen aber beide Kurven vollständig abschreiten. Ebenso fordern wir von einem *gültigen Pfad* im Free-Space nicht, dass er monoton sein muss. Der gültige Pfad muss beide Wertebereiche abdecken, d. h. er muss alle vier Ränder des Free-Space-Diagramms berühren. Dazu suchen wir eine *zusammenhängende Komponente* im Free-Space, deren Projektion auf die Wertebereiche von P und Q beide Wertebereiche abdeckt. Es ist also wichtig, zusammenhängende Komponenten im Free-Space-Diagramm zu finden.

Die Erweiterung des schwachen Fréchet-Abstands auf Flächen erweist sich als schwierig. Mehrere Definitionen sind möglich, aber sie erfüllen nicht die Dreiecksungleichung, d. h. sie bilden keine vollwertige Metrik. Der schwache Fréchet-Abstand zwischen triangulierten Oberflächen ist in polynomieller Zeit berechenbar [11].

Variante Gelegentlich findet man in der Literatur eine etwas strengere Definition des schwachen Fréchet-Abstands. In dieser strengeren Variante fordert man, dass die Endpunkte der Kurven aufeinander abgebildet werden; Start- und Endpunkt des gültigen Pfads sind dann auf $(0, 0)$ und $(1, 1)$ festgelegt.

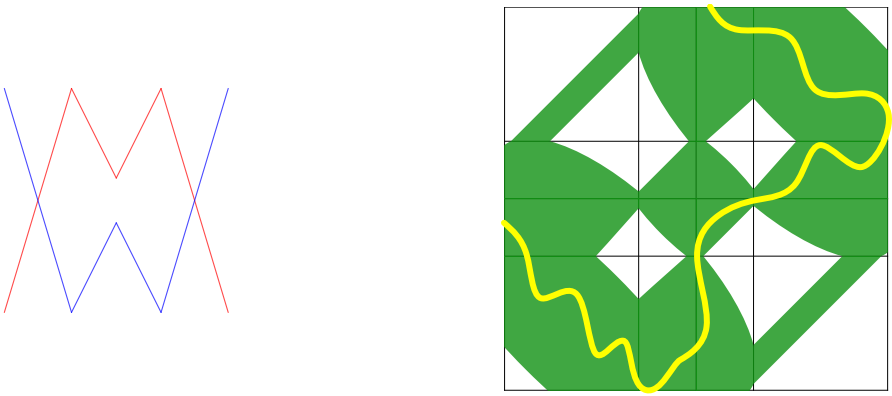


Abbildung 4.2.: Beispiel: zusammenhängende Komponente, nicht monotoner gültiger Pfad

Da die Voraussetzungen für den (starken) Fréchet-Abstand strikter sind, kann er nicht kleiner sein als der schwache Fréchet-Abstand:

$$\delta_{wF}(P, Q) \leq \delta_F(P, Q)$$

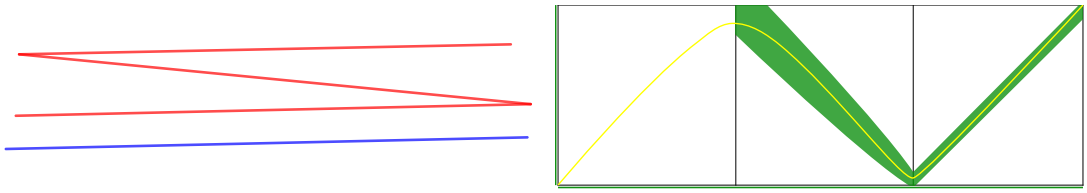


Abbildung 4.3.: Zwei Kurven mit kleinem δ_{wF}, aber großem δ_F.

Nachteile des schwachen Fréchet-Abstands Der schwache Fréchet-Abstand kann die Ähnlichkeit zwischen Kurven weniger gut erfassen als der (starke) Fréchet-Abstand. In Abb. 4.3 zeigen wir ein Beispiel, das einen niedrigen schwachen Fréchet-Abstand besitzt, obwohl die Kurven nicht sehr ähnlich sind.

4.1.1. Unterschiede zum Hausdorff-Abstand

Beim Hausdorff-Abstand suchen wir zu jedem Punkt von P und Q einen Punkt in der anderen Menge, der nicht weiter als ε entfernt ist. Im Free-Space-Diagramm bedeutet dies, dass die Vereinigung aller zusammenhängenden Komponenten beide Wertebereiche abdeckt. Auf die Mann-Hund-Analogie übertragen: Mann und Hund dürfen sich in beliebige Richtungen bewegen und auch unbegrenzt oft *springen*. Beim schwachen Fréchet-Abstand sind hingegen keine Sprünge erlaubt.

Die Motivation für ein neues Abstandsmaß besteht nun darin, die Anzahl der Komponenten (und damit die Anzahl der Sprünge) zu begrenzen. Daraus ergibt sich ein Abstandsmaß, welches einen natürlichen Übergang zwischen Hausdorff-Abstand und schwachem Fréchet-Abstand darstellt.

4.2. Der k-Fréchet-Abstand

Die Definition des k-Fréchet-Abstands basiert auf dem schwachen Fréchet-Abstand. Wir erlauben außerdem, dass wir die Kurven P und Q stückweise vergleichen. Der Parameter k beschreibt hierbei die maximale Anzahl von „Stücken“, in die wir P und Q zerlegen dürfen.

Definition 4.2 ***k*-Fréchet-Abstand** **[40, S. 2]**

Der ***k*-Fréchet-Abstand** $\delta_{kF}(P, Q)$ ist der kleinste Wert ε, so dass k zusammenhängende Komponenten die Wertebereiche des Free-Space F_ε von P und Q abdecken.

Daraus ergibt sich eine Unterteilung von P und Q in k Teilkurven. Um wieder das Bild von Mann und Hund zu bemühen: wir erlauben beiden Protagonisten gelegentlich zu einem anderen Punkt der Kurve zu springen. Mann und Hund dürfen jeweils $k - 1$ Sprünge ausführen.

Buchin und Ryvkin [6, 7, 40] unterscheiden weiter zwischen zwei Varianten des k-Fréchet-Abstands:

- Bei der *Cover*-Variante des k-Fréchet-Abstands dürfen sich die Teilkurven überschneiden.
- Bei der *Cut*-Variante wird gefordert, dass die Teilkurven disjunkt sind.

Wir werden uns im Folgenden hauptsächlich mit der Cover-Variante beschäftigen. Auch die Algorithmen beziehen sich jeweils immer auf die Cover-Variante. Für die Cut-Variante haben Buchin und Ryvkin [40] die NP-Vollständigkeit gezeigt. Weitere Ergebnisse und Algorithmen für die Cut-Variante sind noch nicht bekannt.

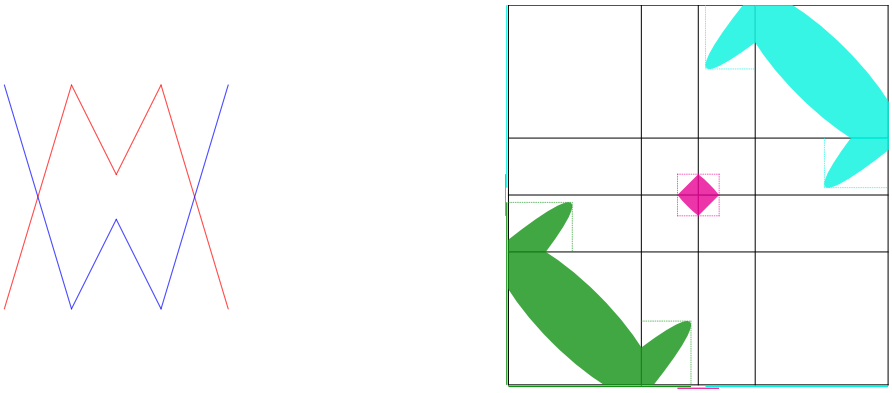


Abbildung 4.4.: Beispiel: k-Fréchet-Abstand mit $k = 3$
Die Projektionen der drei Komponenten decken beide Wertebereiche ab.

Das Entscheidungsproblem fragt, ob $\delta_{kF}(P, Q) \leq \varepsilon$ für ein gegebenes k und ε. Wir suchen k zusammenhängende Komponenten, die beide Wertebereiche des Free-Space F_ε abdecken.

Umgekehrt ist es auch interessant, für ein vorgegebenes ε das kleinstmögliche k zu finden, also die kleinste Zahl an Teilkurven, in die wir P und Q zerlegen müssen. Dieses Optimierungsproblem wird uns im Folgenden beschäftigen. Dazu haben wir zwei Algorithmen in Fréchet View implementiert. In Kapitel 5.10 gehen wir auf die Details dieser Implementierung ein.

Eigenschaften Für wachsendes k wird δ_{kF} kleiner:

$$\delta_{(k+1)F}(P, Q) \quad \leq \quad \delta_{kF}(P, Q)\,.$$

Für $k = 1$ ist der k-Fréchet-Abstand identisch mit dem schwachen Fréchet-Abstand. Für hinreichend großes k ($k \geq n^2$) ist der k-Fréchet-Abstand identisch mit dem Hausdorff-Abstand. Der k-Fréchet-Abstand bildet somit einen Übergang zwischen Hausdorff-Abstand und schwachem Fréchet-Abstand.

Zusammenfassend können wir die verschiedenen Abstandsmaße in folgende Ordnung bringen:

$$\underbrace{\delta_H(P, Q)}_{\text{Hausdorff-Abstand}} \leq \underbrace{\delta_{kF}(P, Q)}_{k\text{-Fréchet-Abstand}} \leq \delta_{1F}(P, Q) = \underbrace{\delta_{wF}(P, Q)}_{\substack{\text{schwacher} \\ \text{Fréchet-Abstand}}} \leq \underbrace{\delta_F(P, Q)}_{\substack{\text{(starker)} \\ \text{Fréchet-Abstand}}}$$

4.3. NP-Vollständigkeit

Interessanterweise ist aber der k-Fréchet-Abstand trotz der Ähnlichkeiten deutlich schwieriger zu berechnen als der schwache Fréchet-Abstand. Während sowohl der schwache Fréchet-Abstand als auch der Hausdorff-Abstand in polynomieller Zeit berechenbar sind, ist die Berechnung des k-Fréchet-Abstands NP-vollständig. Diese zunächst überraschende Tatsache wird dadurch verständlich, dass die Berechnung des k-Fréchet-Abstands ein kombinatorisches Teilproblem enthält, denn wir müssen aus einer Menge von Intervallen eine Auswahl treffen.

Zunächst kann man sich leicht klar machen, dass das Entscheidungsproblem des k-Fréchet-Abstands in NP enthalten ist. Wir können eine Lösung in polynomieller Zeit verifizieren, indem wir die Komponenten des Free-Space auf die Wertebereiche projizieren und prüfen, ob beide Wertebereiche durch die ausgewählten Komponenten abgedeckt werden.

4.3.1. NP-Reduktionen

Akitaya et al. [6, 7] zeigen die NP-Vollständigkeit der *Cover*-Variante des k-Fréchet-Abstands, indem sie eine Reduktion von einer Variante des bekannten 3-SAT-Problems durchführen. Die Konstruktion dieser Reduktion ist zu aufwendig, um sie hier vorzustellen. Wir verweisen den interessierten Leser auf Akitaya et al. [6, 7].

Cut-Variante Für die *Cut*-Variante des Problems zeigen Buchin und Ryvkin [40] eine Reduktion vom *Minimum Common String Partition Problem* (MCSP, siehe auch Jiang et al. [72]). Sie ist leichter verständlich und gibt einen guten Einblick in die Komplexität des k-Fréchet-Abstands.

Definition 4.3 Minimum Common String Partition Problem

Gegeben sind zwei endliche Zeichenketten A und B.
Die Zeichenketten bestehen aus Buchstaben des Alphabets $\Sigma = \{1, \ldots, c\}$.
Gesucht sind Partitionierungen $A = A_1 A_2 \ldots A_n$ und $B = B_1 B_2 \ldots B_n$, so dass wir zu jedem $i = 1, \ldots n$ ein zugehöriges $j \in \{1, \ldots n\}$ finden mit $A_i = B_j$. Die Anzahl n der Partitionen soll minimiert werden.

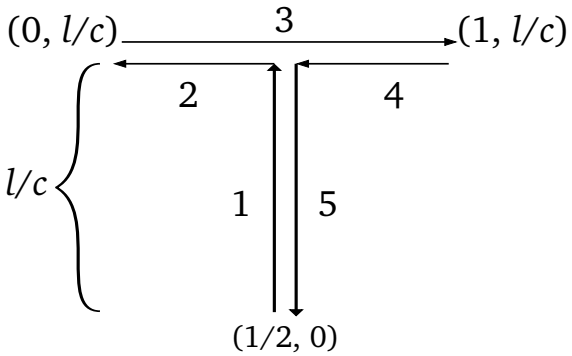


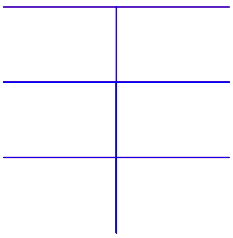
Abbildung 4.5.: MCSP-Reduktion: Buchstaben als T-förmige Teilkurve.
Die Höhe l/c kodiert den Buchstaben l.
(Zeichnung nach Buchin und Ryvkin [40])

Die Reduktion auf den k-Fréchet-Abstand wird möglich, indem wir die beiden Zeichenketten A und B in Kurven P und Q kodieren. Jeder Buchstabe des Alphabets wird auf eine T-förmige Teilkurve abgebildet (siehe Abb. 4.5). Da die T-förmigen Teilkurven im gleichen Punkt starten und enden, können wir sie beliebig oft verketten. Eine derartige Verkettung von T-förmigen Kurven bildet damit eine Zeichenkette ab. In Anhang A.1 haben wir eine Beispieldatei angegeben, mit der sich diese Konstruktion in unserem Programm Fréchet View nachvollziehen lässt.

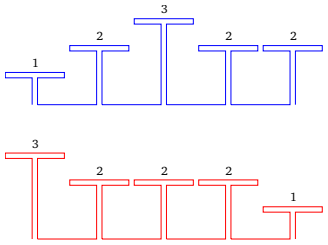
Den Parameter ε wählen wir mit $\varepsilon < 1/c$. Damit stellen wir sicher, dass nur gleiche Buchstaben im Free-Space-Diagramm Komponenten bilden. Die T-Kurven von unterschiedlichen Buchstaben haben einen größeren Abstand. Damit ist für alle Paare von (Teil-)Zeichenketten P_i und Q_j:

$$P_i = Q_j \quad \Leftrightarrow \quad \delta_{wF}(P_i, Q_j) \leq \varepsilon$$

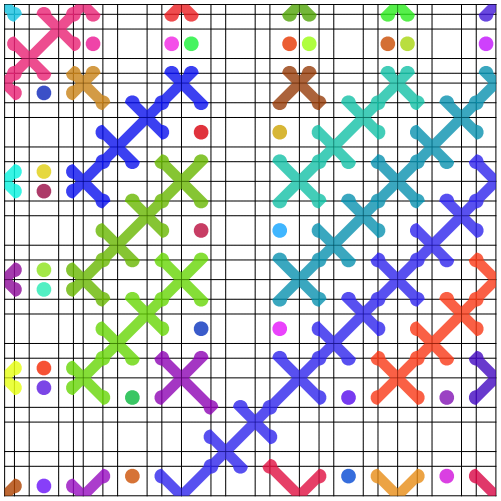
Im Free-Space-Diagramm stellt sich dies so dar, dass identische Teil-Zeichenketten eine zusammenhängende Komponente bilden (Abb. 4.6c). Je länger die identischen Zeichenketten sind, desto größer ist die Komponente. Die Aufgabe des Entscheidungsproblems, k Komponenten zu finden, die beide Wertebereiche abdecken, ist damit gleichbedeutend mit dem Finden von identischen Teilzeichenketten in A und B.



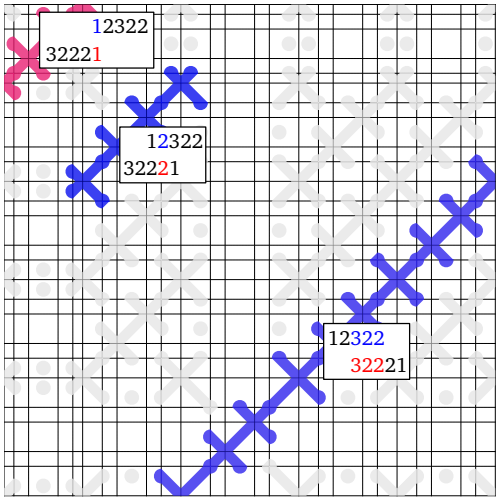
(a) P und Q übereinander liegend



(b) „Explosionszeichnung“



(c) zugehöriges Free-Space-Diagramm



(d) Lösung mit $k = 3$ Komponenten

Abbildung 4.6.: Beispiel für die MCSP-Reduktion.
(Zeichnung nach Buchin und Ryvkin [40])

Damit haben wir das NP-vollständige Problem MCSP auf das Entscheidungsproblem des k-Fréchet-Abstands abgebildet. Die Optimierungsvarianten des k-Fréchet-Abstands sind damit ebenfalls NP-vollständig.

4.4. Brute-Force-Algorithmus

Nachdem wir die NP-Vollständigkeit gezeigt haben, wissen wir, dass es keinen Algorithmus in polynomieller Zeit geben kann (es sei denn, NP=P). Wir stellen zunächst einen Brute-Force-Algorithmus vor, eine erschöpfende Suche in exponentieller Zeit.

Das Free-Space-Diagramm enthält höchstens nm zusammenhängende Komponenten. Daraus müssen wir k Komponenten auswählen, die beide Wertebereiche abdecken. Die Anzahl der möglichen Kombinationen beträgt $\binom{nm}{k}$. Die Prüfung, ob eine Menge die Wertebereiche abdeckt, können wir in $O(k)$ ausführen.

Für ein gegebenes k benötigen wir damit

$$O\left(k \cdot \binom{nm}{k}\right) = O(k \cdot (nm)^k)$$

Schritte. Da $\binom{m}{k} \leq 2^m$ für alle $m > k$ können wir $O(n \cdot 2^{mn})$ als allgemeine Obergrenze, unabhängig von k, angeben. In Kapitel 5.10.4 stellen wir eine Implementierung eines Brute-Force-Algorithmus vor.

Ein Algorithmus in exponentieller Zeit ist für praktische Anwendungen wenig geeignet. Wir zeigen zwei Wege, wie man den k-Fréchet-Abstand dennoch mit vertretbarem (d. h. polynomiellem) Aufwand anwenden kann: einen Näherungsalgorithmus in Kapitel 4.5 und parametrisierte Algorithmen in den Kapiteln 4.6.1 und 4.6.2, die für gewisse Eingabedaten eine exakte Berechnung in polynomieller Zeit möglich machen.

4.5. Näherungsalgorithmus: Greedy

Buchin und Ryvkin [6, 7, 40] beschreiben einen Näherungsalgorithmus zur Berechnung (der *Cover*-Variante) des k-Fréchet-Abstands und geben als Näherungsfaktor 2 an: die Ergebnisse des Näherungsalgorithmus sind höchstens um das Doppelte schlechter als die optimale Lösung.

Wir erinnern uns: die Aufgabe besteht darin, die zusammenhängenden Komponenten des Free-Space auf die Wertebereiche zu projizieren und dann k Intervalle auszuwählen, die beide Wertebereiche abdecken. Wir bezeichnen mit L_P die Projektion der Komponenten auf die P-Achse, mit L_Q die Projektion auf die Q-Achse. Jede Komponente erzeugt zwei Intervalle und hat damit jeweils einen Eintrag in L_P und einen weiteren in L_Q. Die Information, welche Intervalle zu einer gemeinsamen Komponente gehören, können wir mitführen.

Mit einem „Greedy“-Ansatz können wir eine optimale Lösung für jeweils einen Wertebereich finden. Dazu sortieren wir zunächst die Mengen L_P und L_Q nach ihren Intervall-Untergrenzen.

Wir starten am linken Rand des Free-Space und wählen aus allen Intervallen, die den Punkt 0 enthalten das größte, d. h. dasjenige mit der größten Obergrenze, nennen wir sie r_1. Dann setzen wir die Auswahl bei r_1 fort und wählen unter allen Intervallen, die r_1 enthalten wieder das mit der größten Obergrenze. Diese Auswahl setzen wir („greedy“) fort, bis wir den rechten Rand des Free-Space erreicht haben. Zu jedem Zeitpunkt des Algorithmus haben wir die minimale Anzahl von Intervallen ausgewählt, um den Bereich links der Auswahl abzudecken. Wenn wir den rechten Rand des Free-Space erreichen, haben wir damit eine *optimale* Lösung für diesen *einen* Wertebereich gefunden.

Die ausgewählten Intervalle für die Wertebereiche bezeichnen wir mit S_L und S_Q. Die optimale Lösung S_{opt} für beide Wertebereiche kann nicht besser sein als die beiden optimalen Teillösungen:

$$|S_{opt}| \;\geq\; \max\{|S_P|, |S_Q|\}\ .$$

Wir bilden die Vereinigung von S_L und S_Q und bestimmen die Anzahl der zugehörigen Komponente (jede Komponente kann bis zu zwei Intervalle beisteuern). Im ungünstigsten Fall wählt unser Greedy-Algorithmus Intervalle aus völlig unterschiedlichen Komponenten, so dass

$$|S_{greedy}| \;\leq\; |S_P| + |S_Q| \;\leq\; 2 \cdot \max\{|S_P|, |S_Q|\} \;\leq\; 2 \cdot |S_{opt}|\ .$$

Die Lösung des Greedy-Algorithmus ist also höchstens um das Doppelte schlechter als die optimale Lösung.

Laufzeit-Abschätzung Die Berechnung des Free-Space kann in $O(nm)$ geschehen. Das Erstellen und Sortieren der Listen L_P und L_Q kostet $O(nm \log nm)$. Für die „greedy“-Auswahl iterieren wir in linearer Zeit über die Listen, die jeweils $O(nm)$ groß sind. Die Laufzeit des Greedy-Algorithmus wird somit dominiert vom Aufwand, der zum Sortieren notwendig ist: $O(nm \log nm)$.

Anmerkung zur Implementierung Man kann die Greedy-Auswahl wahlweise bei der P-Achse oder der Q-Achse starten. In unserer Implementierung in Kapitel 5.10.3 berechnen wir immer *beide* Werte, in der Hoffnung, etwas bessere Abschätzungen zu erhalten. Auf die Argumentation von oben hat dies aber keinen Einfluss.

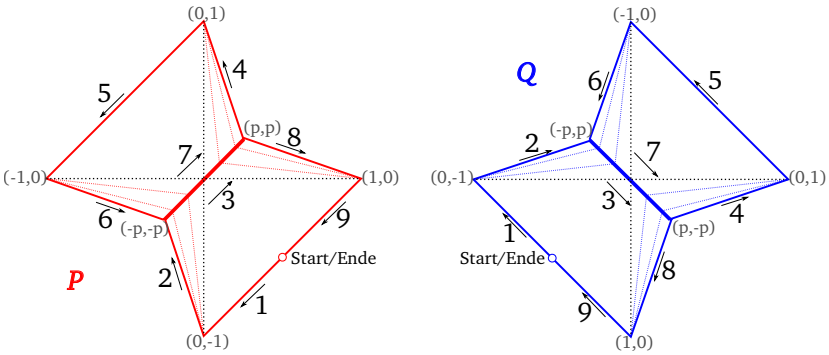
4.5.1. Beispiele für den Greedy-Näherungsfaktor

Wir haben gezeigt, dass die Greedy-Lösung höchstens um den Faktor 2 schlechter als eine optimale Lösung ist. Kann man diesen Faktor noch weiter eingrenzen, oder kann man zeigen, dass er exakt ist?

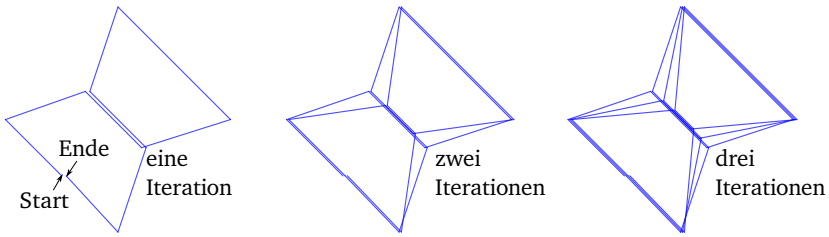
Bei Buchin und Ryvkin [6, 7, 40] blieb diese Frage offen. Wir können sie hier beantworten. Wir konstruieren Beispiele, mit denen wir uns dem Faktor 2 beliebig annähern (wenn auch nicht erreichen) können. Der Näherungsfaktor 2 ist also für den Greedy-Algorithmus tatsächlich der bestmögliche.

Konstruktion Wir bilden zwei Kurven, die sich jeweils im Nullpunkt kreuzen. Die Kurven beginnen und enden an den Punkten $(1/2, -1/2)$ bzw. $(-1/2, -1/2)$. Sie können beliebig oft verkettet werden. Bei jeder Iteration der Verkettung verschieben wir schrittweise den Punkt (p, p), beginnend bei $(p, p) = (1/4, 1/4)$ bis hin zu $(p, p) = (0, 0)$. Die zweite Kurve Q ergibt sich durch Drehung um 90° im Uhrzeigersinn. In Anhang A.1 haben wir eine Beispieldatei angegeben, mit der sich die Konstruktion in Fréchet View nachvollziehen lässt.

In Abb. 4.7a wird der Aufbau der Kurven beschrieben. Abb. 4.7b zeigt die Kurve nach mehreren sich überlagernden Iterationen. In Abb. 4.9a sind schließlich alle Iterationen dargestellt.



(a) Koordinaten



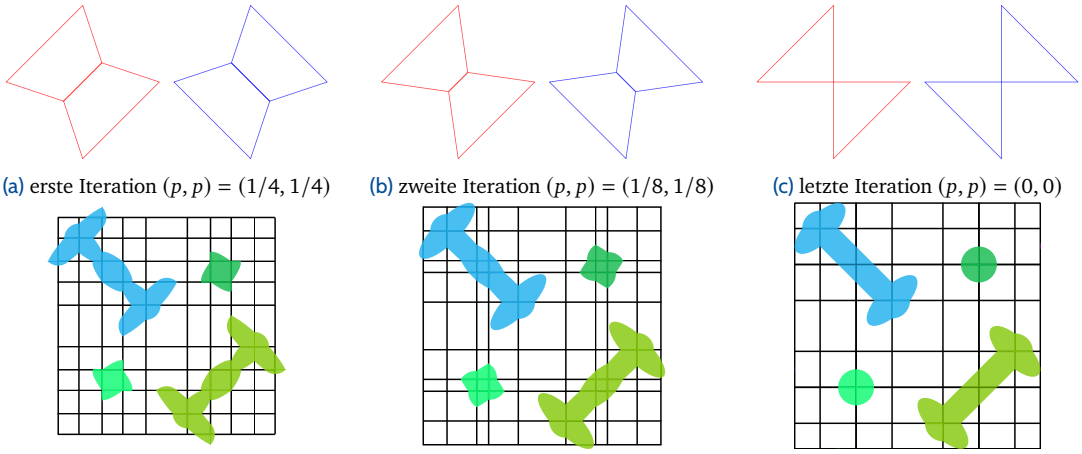
(b) mehrere Iterationen überlagern sich.

Abbildung 4.7.: Konstruktion der Kurven P und Q

Den Parameter ε wählen wir mit $1/2 \leq \varepsilon < 1$; für $\varepsilon < 1/2$ gibt es keine Lösung, für $\varepsilon \geq 1$ nur eine triviale. Der Wert $p \leq 1/4$ ist nicht zufällig gewählt. Zwei korrespondierende Punkte (p, p) und $(\pm p, \mp p)$ dürfen nicht weiter als $\varepsilon = 1/2$ voneinander entfernt liegen.

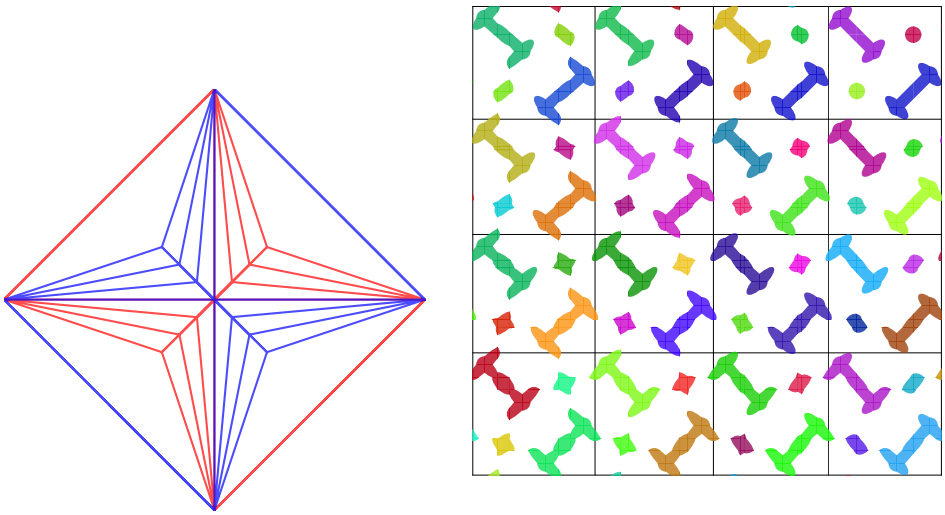
Das zugehörige Free-Space-Diagramm in Abb. 4.9b weist eine sehr regelmäßige Struktur auf. In Abb. 4.8 sind die Free-Space-Komponenten jeweils einer Iteration dargestellt. Je eine Iteration der Kurven erzeugt vier Komponenten, die zusammen ein Quadrat des Free-Space abdecken.

Die Größe der Komponenten ändert sich geringfügig. Diese kleinen Änderungen dienen lediglich dazu, dass die Komponenten von der Greedy-Suche in einer festgelegten Reihenfolge ausgewählt werden. Für unsere Argumentation ist diese Größenänderung ansonsten nicht weiter bedeutsam.



(a) erste Iteration $(p, p) = (1/4, 1/4)$ (b) zweite Iteration $(p, p) = (1/8, 1/8)$ (c) letzte Iteration $(p, p) = (0, 0)$

Abbildung 4.8.: man sieht, dass die Komponenten schrumpfen



(a) verkettete, übereinander liegende Kurven (b) das zugehörige Free-Space-Diagramm

Abbildung 4.9.: Alle Kurven zusammengesetzt.
Zusammenhängende Free-Space-Komponenten sind farblich markiert.

Wenn wir die Kurven x mal verketten, findet der Greedy-Algorithmus eine Lösung mit

$$k_{greedy} = 4x - 2$$

Komponenten (Abb. 4.10a und 4.10c), wohingegen die optimale Lösung

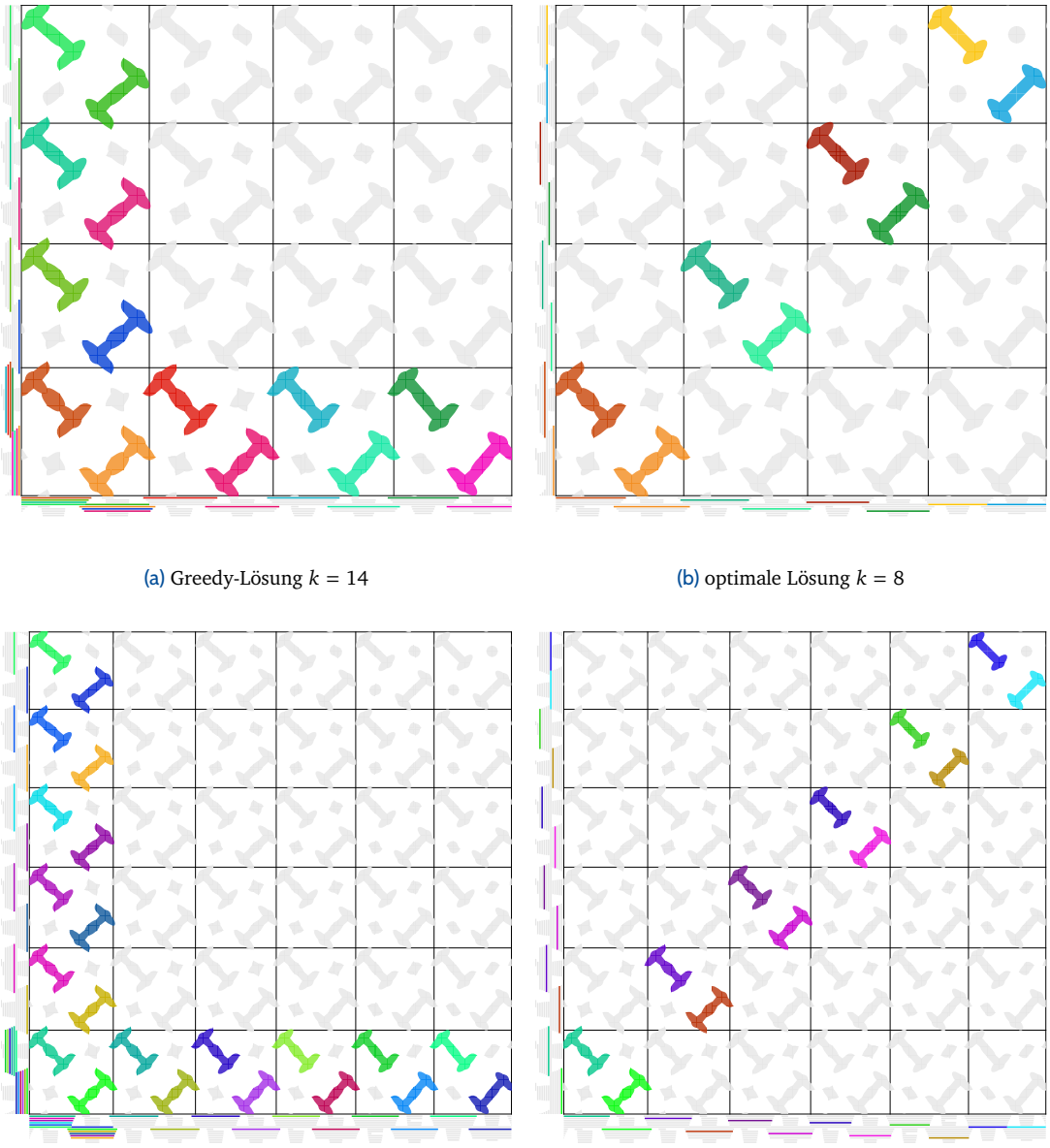
$$k_{opt} = 2x$$

beträgt (Abb. 4.10b und 4.10d). Bei größeren Beispielen wird eine Brute-Force-Suche vermutlich keine Lösung in akzeptabler Zeit finden. Man kann sich die optimale Lösung aber leicht herleiten: je zwei hantelförmige Komponenten decken ein Quadrat ab. Die optimale Lösung besteht aus den Quadraten auf der Diagonalen des Free-Space-Diagramms. Der Näherungsfaktor

$$\frac{4x - 2}{2x} = 2 - \frac{1}{x}$$

konvergiert mit wachsendem x (= Anzahl der verketteten Kurven) gegen 2.

Zuletzt bleibt noch die Frage offen, ob man auch ein Beispiel konstruieren kann, mit dem der Näherungsfaktor 2 *exakt* erreicht wird.



(a) Greedy-Lösung $k = 14$

(b) optimale Lösung $k = 8$

(c) Greedy-Lösung $k = 22$

(d) optimale Lösung $k = 12$

Abbildung 4.10.: Greedy- und optimale Lösungen.

Am unteren und linken Rand sind die Projektionen der Komponenten dargestellt.
Man beachte die Sortierung der Intervalle

4.6. Parametrisierungen

Ein weiterer Weg, um ein NP-vollständiges Problem beherrschbar zu machen, besteht im Finden einer Parametrisierung.[1] Man versucht, das exponentielle Wachstum des Problems auf Parameter zu begrenzen, die sich besser kontrollieren lassen.

Definition 4.4 Fixed Parameter Tractability [53, Def. 2.1.1., S. 15]

Ein Problem mit Eingabegröße n heißt **parametrisierbar** (*fixed parameter tractable*), wenn es eine Konstante c, einen Parameter k und eine Funktion f gibt, so dass das Problem in

$$f(k) \cdot n^c$$

Schritten lösbar ist.

Die Lösung des Problems bleibt polynomiell in Bezug auf die Eingabegröße n, während die Funktion f schneller wachsen darf (z. B. exponentiell). Wir haben das expontielle Wachstum auf einen Parameter k eingegrenzt, der unabhängig von n ist. Wenn es uns gelingt, den Parameter k zu fixieren, so bleibt die Lösung insgesamt in polynomieller Zeit („tractable“). Eine Einführung in die parametrisierte Komplexitätstheorie liefern Downey und Fellows [53].

Die Schwierigkeit besteht nun im Finden von brauchbaren Parametern, die nicht von der Eingabegröße n abhängig sind.

4.6.1. Eine Parametrisierung für den k-Fréchet-Abstand

Akitaya et al. [6, 7] stellen eine Parametrisierung für die Cover-Variante des k-Fréchet-Abstands vor. Als Parameter wählen sie den durch das Problem vorgegebenen Parameter k (die Anzahl der Komponenten) und zusätzlich z, die *neighborhood complexity* der Eingabekurven. Um die Analysen etwas zu vereinfachen, nehmen wir an, dass die Eingabekurven ähnlich groß sind, also $m = O(n)$.

Definition 4.5 Neighborhood Complexity

Die **neighborhood complexity** ist die maximale Anzahl von Segmenten einer Kurve, die in einer ε-Umgebung eines Punktes der anderen Kurve verlaufen.

Andere Autoren (de Berg et al. [51]) bezeichnen solche Kurven auch als *low-density* Kurven. Im Free-Space-Diagramm zeigt sich die neighborhood complexity so, dass jede horizontale oder vertikale Linie höchstens z Komponenten schneidet. Wenn wir die Komponenten auf die Wertebereiche projizieren, so überlappen sich nie mehr als z Intervalle.

Suchbäume

Die erschöpfende Suche über die Intervalle modellieren wir als *Suchbaum* (Abb. 4.11). Ein Knoten des Baum entspricht einer Komponente des Free-Space (bzw. deren Projektion auf einen Wertebereich).

Einen solchen Baum können wir z. B. erstellen, indem wir eine Sweep Line über den Free-Space laufen lassen. Wir führen Buch über die Komponenten, die von der Sweep Line geschnitten werden. Wir nennen diese Komponenten *aktiv*. Es gibt zu jedem Zeitpunkt höchstens z aktive Komponenten.

[1] Nicht zu verwechseln mit den parametrisierten Kurven aus Kapitel 1.

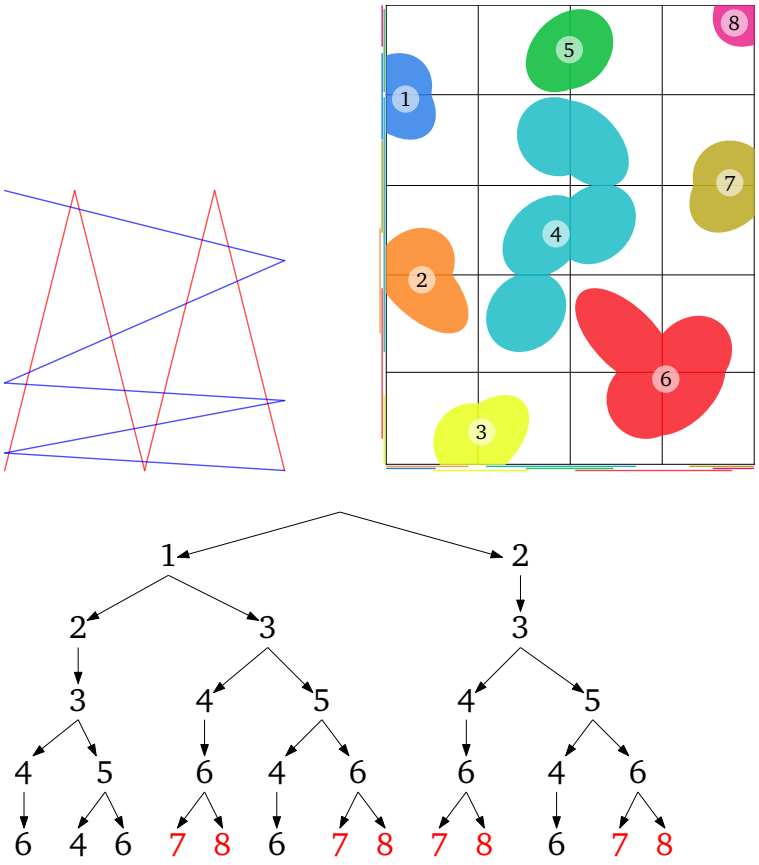


Abbildung 4.11.: Free-Space-Komponenten und Suchbaum für den Wertebereich P (Zeichnung nach Akitaya et al. [6, 7])

Wenn die Sweep Line an den rechten Rand einer Komponente r stößt, so fügen wir alle gerade aktiven Komponenten als Nachfolger von r in den Baum ein. Ein Knoten kann nie mehr als z Nachfolger erhalten. Der Baum wird bei Tiefe k gekappt und der Prozess hält an, wenn die Sweep Line den rechten Rand des Free-Space erreicht. Auf ganz ähnliche Weise könnte man den Suchbaum durch eine Tiefen- oder Breitensuche realisieren. In unserer Implementierung in Kapitel 5.10.4 arbeiten wir z. B. mit einer Tiefensuche. Auf diese Weise erstellen wir zwei Suchbäume T_P und T_Q für die beiden Wertebereiche von P und Q.

Ein Pfad im Baum beschreibt eine Auswahl von (bis zu k) Komponenten. Wir nennen einen Pfad *gültig*, wenn er den Wertebereich mit k Komponenten vollständig abdeckt. Das letzte Element eines gültigen Pfades (also das Blatt) berührt den rechten Rand des Free-Space. In Abb. 4.11 sind die Blätter der gültigen Pfade rot markiert.

Wir berechnen die gültigen Pfade von T_P und T_Q und speichern sie in sortierten Listen ab. Wir prüfen jedes Paar von gültigen Pfaden $S_{P,i}$ und $S_{Q,j}$ (mit $1 \leq i, j \leq z^k$). Wenn $|S_{P,i} \cup S_{Q,j}| \leq k$, so haben wir eine Lösung gefunden.

Damit können wir das Entscheidungsproblem des k-Fréchet-Abstands in

$$O(nz + kz^{2k}) \tag{4.1}$$

Schritten lösen [6, 7, Theorem 3].

Beweis. Der oben beschriebene Prozess findet alle Kombinationen von k Komponenten und führt also eine erschöpfende Suche durch. Die Berechnung des Free-Space kann in $O(nz)$ durchgeführt werden.

Die Ermittlung der Komponenten und ihrer Projektionen auf die Wertebereiche kann ebenfalls in $O(nz)$ geschehen.

Die Höhe eines Suchbaums ist durch k beschränkt. Der Verzweigungsgrad (die maximale Anzahl von Nachfolgerknoten) ist durch z beschränkt. Ein Suchbaum enthält damit höchstens

$$1 + z + z^2 + \ldots + z^k = \frac{z^{k+1} - 1}{z - 1} = O(z^k)$$

Knoten, höchstens z^k Blätter und $O(z^k)$ Pfade. Wir prüfen alle Paare von Pfaden, wobei eine Prüfung in $O(k)$ durchgeführt werden kann.

Insgesamt erhalten wir als Laufzeit

$$O(nz + z^k + z^k \log k + k(z^k)^2) = O(nz + kz^{2k})$$

□

4.6.2. Weitere Parametrisierungen

In Ergänzung zur Arbeit von Akitaya et al. [6, 7] stellen wir zwei weitere Parametrisierungen vor. Sie kombinieren die Suchbäume aus Kapitel 4.6.1 mit der bereits bekannten Greedy-Suche aus Kapitel 4.5.

Baumsuche + Greedy Für den Wertebereich von P erstellen wir, wie oben beschrieben, einen Suchbaum T_P. Für den Wertebereich von Q ist es aber nicht notwendig, alle Kombinationen von k Komponenten ermitteln. Wir machen uns die Eigenschaft der Greedy-Suche zunutze, eine optimale Lösung für *je einen* Wertebereich zu finden.

Immer wenn wir während der Baumsuche den rechten Rand des Free-Space erreichen, führen wir eine Greedy-Suche auf dem Wertebereich von Q durch und prüfen, ob es sich um eine gültige Lösung handelt. Bei dieser Greedy-Suche werden die bereits im Suchbaum selektierten Komponenten übergangen.

Der Suchbaum T_P hat $O(z^k)$ Knoten. Die Kosten einer einzelnen Greedy-Suche auf dem Wertebereich von Q können wir mit $O(nz)$ abschätzen. Das initiale Erstellen und Sortieren der Intervalle kostet $O(nz \log nz)$.

Ingesamt erhalten wir damit als Abschätzung für diese Parametrisierung

$$O(nz \log nz + z^k + z^k \cdot nz) \;=\; O(nz \log nz + nz^{k+1}) \tag{4.2}$$

Greedy-Suche mit Intervallbaum Wir können die Greedy-Suche beschleunigen, indem wir anstelle einer sortierten Liste einen *Intervallbaum* verwenden. In Listing 4.1 auf Seite 68 haben wir eine solche Greedy-Suche skizziert. Das Erstellen des Baums kostet – ähnlich wie das Sortieren der Liste – $O(nz \log nz)$.

Auf dem Intervallbaum können wir Anfragen ausführen, welche alle Intervalle liefern, die einen gesuchten Wert enthalten. Die Kosten einer Anfrage sind $O(\log N + M)$, wobei N die Größe des Baums bezeichnet und M die Anzahl der Treffer. Nach Definition 4.5 erhalten wir höchstens z Treffer. In unserem Fall kostet eine Anfrage also

$$O(\log nz + z) = O(\log n + \log z + z) = O(\log n + z)$$

Im Laufe einer Greedy-Suche führen wir höchstens k Anfragen aus. Damit können wir die Abschätzung von Gleichung (4.2) etwas verbessern:

$$O(nz \log nz + k \cdot (\log n + z) \cdot z^k) \tag{4.3}$$

4.7. Offene Fragen, weiterführende Ideen

Der k-Fréchet-Abstand ist ein sehr neues Konzept. Viele Fragen sind noch Gegenstand der Forschung. Z. B. ist noch sehr wenig über die Cut-Variante bekannt. Akitaya et al. [6] vermuten, dass das Entscheidungsproblem für die Cut-Variante $\exists\mathbb{R}$-vollständig ist. Die Komplexitätsklasse $\exists\mathbb{R}$ ist aus anderen geometrischen und Graphen-Problemen bekannt [91].

Es wäre auch interessant, weitere Parametrisierungen für andere Klassen von Eingabekurven zu finden. Mögliche Kandidaten sind monotone Kurven, k-straight-, k-bounded- oder c-packed-Kurven (siehe z. B. Driemel [54, S. 18] für eine kurze Übersicht über verschiedene Klassen von Kurven).

Ein anderer Ansatz, um das exponentielle Wachstum einer Brute-Force-Suche beherrschbar zu machen, sind Heuristiken. Heuristiken könnten sich die Eigenschaften der Eingabedaten zunutze machen, um den in Kapitel 4.6.1 beschriebenen Suchbaum einzuschränken.

Mecke und Wagner [76], Mecke et al. [77, 78], Ruf und Schöbel [90] beschreiben Heuristiken für ähnlich strukturierte Probleme. Durch Vorverarbeitung und Umsortierung der Eingabedaten können sie die Verzweigungen des Suchbaums drastisch reduzieren. Ob sich solche Techniken auch auf den k-Fréchet-Abstand anwenden lassen, und wie erfolgversprechend sie im Einzelnen sind, dürfte hauptsächlich von der Beschaffenheit der Eingabedaten abhängen.

```
bool greedy-search(IntervalTree tree, Set<Interval> selected)
{ // "selected" contains the intervals that have already been selected by the tree search
    // (maybe realised by a Bit−Set with O(1) access)
    float x = 0
    while x < m and |selected| < k
    { // query intervals that intersect the sweep line
        Set<Interval> I = tree.query(x) // note that |I| ≤ z
        if I = ∅
            return false
        if I ∩ selected ≠ ∅ // skip intervals that have already been selected
            x = sup (I ∩ selected)
        else { // pick the interval with the largest upper bound
            select i ∈ I such that sup i = sup I
            x = sup i
            selected = selected ∪ {i}
        }
    }
    return x ≥ m
}
```

Listing 4.1: Pseudocode: Greedy-Suche mit Intervallbaum

Zweiter Teil

Implementierung

5

Implementierungs-Aspekte

Im zweiten Teil dieser Arbeit stellen wir eine prototypische Umsetzung der beschriebenen Algorithmen vor. Hierzu entstand das Programm Fréchet View, mit dem sich die Algorithmen testen und Ergebnisse visualisieren lassen.

Die wesentlichen Teile der Algorithmen habe ich selbst in der Programmiersprache C++ (nach Standard C++14) implementiert und dabei ggf. auf Datenstrukturen und Algorithmen aus Programmierbibliotheken zurückgegriffen. Wir werden an den entsprechenden Stellen darauf eingehen.

Die Sprache C++ eignet sich nicht zuletzt deshalb für die Umsetzung, weil es ein breites Angebot an Software-Bibliotheken gibt. Das CGAL-Projekt [41] hat sich die Implementierung von Algorithmen der Algorithmischen Geometrie zur Aufgabe gemacht. Wir werden an mehreren Stellen Algorithmen und Datenstrukturen aus CGAL nutzen. Überdies lassen sich in C++ sehr gut Programme parallelisieren. Auch hierfür stehen zahlreiche hochwertige Frameworks zur Verfügung. Wir haben uns bei unserer Implementierung für TBB [71] entschieden. Weiterhin ist C++ hervorragend geeignet für Laufzeit-kritische Aufgaben wie die Boolesche Matrixmultiplikation (Kapitel 6) und für numerische Berechnungen.

Die Benutzeroberfläche wurde mit Hilfe des Qt-Frameworks umgesetzt, mit dem sich einfach plattform-übergreifende Programme für die gängigen Betriebssysteme Windows, Linux und macOS erstellen lassen.

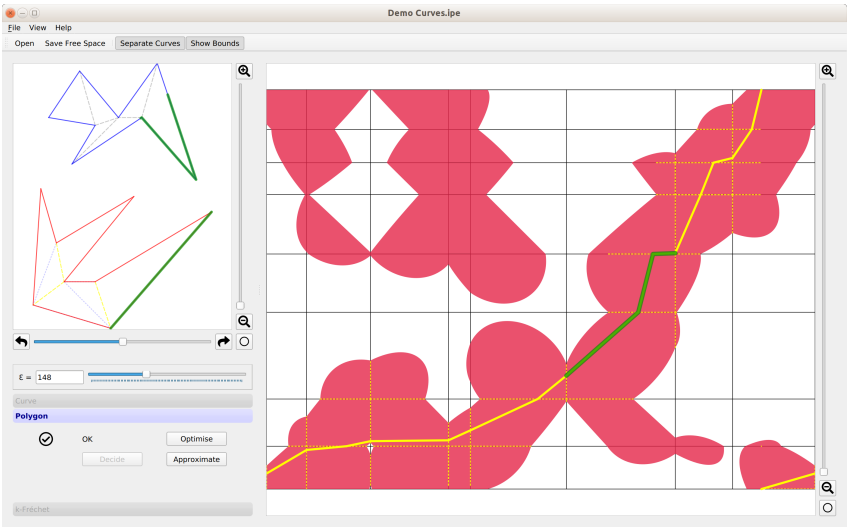


Abbildung 5.1.: Bildschirm-Ansicht von Fréchet View

Der Quellcode des Programms ist öffentlich zugänglich. Lauffähige Versionen liegen dieser Arbeit bei und können auch heruntergeladen werden (siehe Anhang A). Installation und Bedienung des Programms werden in Anhang B und auf den Webseiten kurz beschrieben.

Wir erläutern hier die grundlegenden Konzepte und begründen die wichtigsten Entscheidungen, die während der Implementierung getroffen wurden.

5.1. Die Berechnung des Free-Space-Diagramms

Die grundlegende Datenstruktur für alle Algorithmen bildet das Free-Space-Diagramm aus Kapitel 2.1. Wir berechnen die Intervalle L_{ij}^F und B_{ij}^F und speichern sie in einem zweidimensionalen Array.

Um Rundungsfehler während der Berechnung zu reduzieren, werden die Zwischenergebnisse mit erhöhter Genauigkeit berechnet. Während die Ergebnisse als 64-Bit-Fließkommazahlen gespeichert werden, berechnen wir die Zwischenergebnisse mit 80-Bit-Fließkommazahlen. Der zusätzliche Aufwand ist bei modernen Prozessoren gering, da sie entsprechende Fließkomma-Register besitzen. Rundungsfehler sind zwar für die Visualisierung der Algorithmen in der Regel vernachlässigbar, können aber beim Optimierungsalgorithmus durchaus relevant werden (siehe Kapitel 5.7.1). In Kapitel 7.1 werden wir sehen, wie sich die Berechnung des Free-Space-Diagramms auf Multi-Core-Rechnern beschleunigen lässt.

5.2. Darstellung der Erreichbarkeits-Strukturen

Die Erreichbarkeits-Struktur haben wir in Kapitel 2.7 beschrieben. Sie bildet die Grundlage für den Entscheidungsalgorithmus für geschlossene Kurven (Kapitel 2.6) und spielt auch im Algorithmus von Buchin et al. [27] eine wichtige Rolle, nämlich bei der Erstellung der Erreichbarkeits-Graphen (Kapitel 3.9).

Die Erreichbarkeits-Struktur besteht aus Intervallen an den vier Rändern des Free-Space-Diagramms. Zwischen den Intervallen bestehen Querverbindungen: die l- und h-Zeiger. Im Verlauf des Algorithmus von Alt und Godau [13] werden die Intervalle verfeinert und Erreichbarkeits-Strukturen werden schrittweise zusammengefügt.

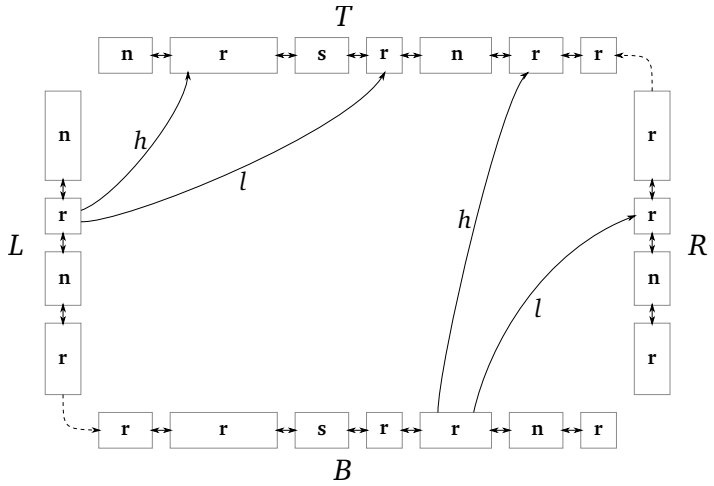


Abbildung 5.2.: Erreichbarkeits-Struktur, vier doppelt verkettete Listen L,B,R und T.

Jeder Eintrag steht für ein Intervall der Erreichbarkeits-Struktur mit Typ **r**, **s** oder **n**. Nur einige l- und h-Zeiger sind eingezeichnet.

Die Datenstruktur muss flexibel genug sein, um diese Operationen effizient durchzuführen. Die Ränder der Erreichbarkeits-Struktur implementieren wir als doppelt verkettete Listen. Das Einfügen von neuen Einträgen sowie das Zusammenfügen zweier Listen kann jeweils in $O(1)$ geschehen. Da diese Datenstruktur in unseren Algorithmen häufig verwendet wird, haben wir Wert auf eine speichereffiziente („cache-freundliche“) Implementierung gelegt. Z. B. haben wir die Verkettung selbst implementiert, anstatt auf eine Datenstruktur aus bekannten Bibliotheken wie STL oder Boost zurückzugreifen.

Die Unterteilung der Intervalle ist für den oberen und unteren Rand der Erreichbarkeits-Struktur jeweils identisch. Ebenso für den linken und rechten Rand. Wir könnten nun jeweils ein Paar von gegenüberliegenden Intervallen zu einem gemeinsamen Objekt zusammenfassen. Damit könnten wir Speicherplatz sparen und das Bearbeiten der verketteten Listen vereinfachen. Allerdings werden wir bei der MERGE-Operation sehen, dass eine solche kompakte Datenstruktur auch Nachteile hat. Unsere Implementierung speichert die Erreichbarkeits-Struktur also in Form von vier getrennten Listen (Abb. 5.2).

Der Algorithmus von Alt und Godau [13] berechnet zunächst Erreichbarkeits-Strukturen für die einzelnen Zellen des Free-Space-Diagramms. Danach werden die Zellen schrittweise zu größeren Strukturen zusammengefügt, bis wir eine Erreichbarkeits-Struktur erhalten, welche das gesamte Free-Space-Diagramm beschreibt.

5.2.1. MERGE-Operation auf Erreichbarkeits-Strukturen

Ein wichtiger Schritt im Algorithmus von Alt und Godau [13] besteht im Zusammenfügen von zwei Erreichbarkeits-Strukturen (siehe Kapitel 2.7). Zunächst wird die Unterteilung der Intervalle entlang der gemeinsamen Ränder synchronisiert. In Abb. 2.9 auf Seite 29 werden die Erreichbarkeits-Strukturen entlang der vertikalen Ränder R_1 und L_2 zusammengefügt. Wir iterieren über die verketteten Listen und teilen, wenn nötig, die Intervalle auf. Diese Unterteilung wird auch auf die gegenüberliegenden äußeren Ränder (L_1 und R_2) übertragen. Das Einfügen von neuen Einträgen geschieht in $O(1)$.

Die Ränder der verschmolzenen Erreichbarkeits-Struktur werden durch $B_1 \cup B_2$, $T_1 \cup T_2$, L_1 und R_2 gebildet. Die inneren Ränder R_1 und L_2 fallen weg. Mit vier verketteten Listen ist dies einfach in $O(1)$ durchzuführen. Mit der oben erwähnten kompakten Speicherung (mit nur zwei verketteten Listen) wäre dies schwieriger zu realisieren.

Danach werden die l- und h-Zeiger angepasst und die Erreichbarkeit der einzelnen Intervalle aktualisiert. Dies erfolgt wie in Kapitel 2.7.2 beschrieben in $O(nm)$.

5.3. Algorithmen zur konvexen Zerlegung von Polygonen

Im ersten Schritt des Algorithmus von Buchin et al. [27] werden die Eingabe-Polygone in konvexe Teile zerlegt (Kapitel 3.8). Die Anzahl der ermittelten konvexen Teile bildet einen Faktor für die Gesamtlaufzeit des Algorithmus (siehe Kapitel 3.13.1). Der Aufwand zum Finden einer besseren Zerlegung könnte sich also bezahlt machen.

Keil und Snoeyink [74] beschreiben einen Algorihtmus zur konvexen Zerlegung mit Laufzeit $O(n + r^2 \min\{n, r^2\})$, wobei r die Anzahl der konkaven Eckpunkte ist (das sind Eckpunkte, deren Innenwinkel größer als π ist). Darüberhinaus gibt es eine Vielzahl von optimalen und Näherungsalgorithmen.

Wir haben drei Algorithmen zur Berechnung von konvexen Zerlegungen in die engere Auswahl gezogen. Die Implementierungen stammen alle aus CGAL [41].

1. Ein Algorithmus von Greene [61] liefert ein optimales Ergebnis mit der kleinsten Zahl an konvexen Teilen. Seine Laufzeit ist mit $O(n^4)$ aber auch sehr hoch.
2. Ein Näherungsalgorithmus von Hertel und Mehlhorn [69] liefert Ergebnisse in $O(n)$. Die Anzahl der gefundenen konvexen Teile ist höchstens um den Faktor 4 größer als das optimale Ergebnis.
3. Ein Näherungsalgorithmus von Greene [61] besitzt eine Laufzeit von $O(n \log n)$, hat ebenfalls den Näherungsfaktor 4, liefert aber in der Regel bessere Ergebnisse als der Algorithmus von Hertel und Mehlhorn [69].

Letztendlich haben wir uns für den Näherungsalgorithmus von Hertel und Mehlhorn [69] entschieden. Ausschlaggebend war hierfür neben der günstigen Laufzeit vor allen Dingen die robuste Implementierung. Die Implementierungen der beiden anderen Algorithmen waren leider bei bestimmten Eingaben fehleranfällig, wie uns ein Mitarbeiter des CGAL-Projekts bestätigen konnte.

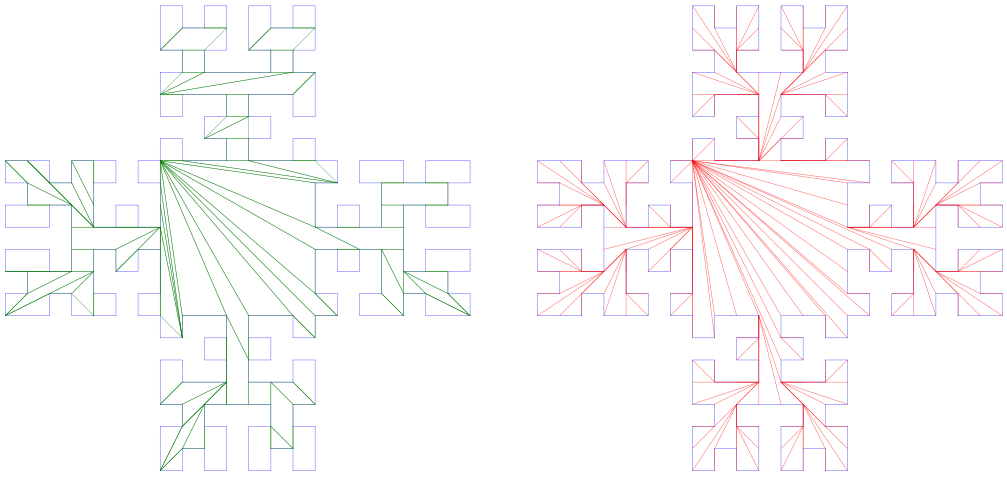


Abbildung 5.3.: Konvexe Zerlegung (links) und Shortest Path Tree (rechts)

5.4. Datenstruktur für die Triangulierung

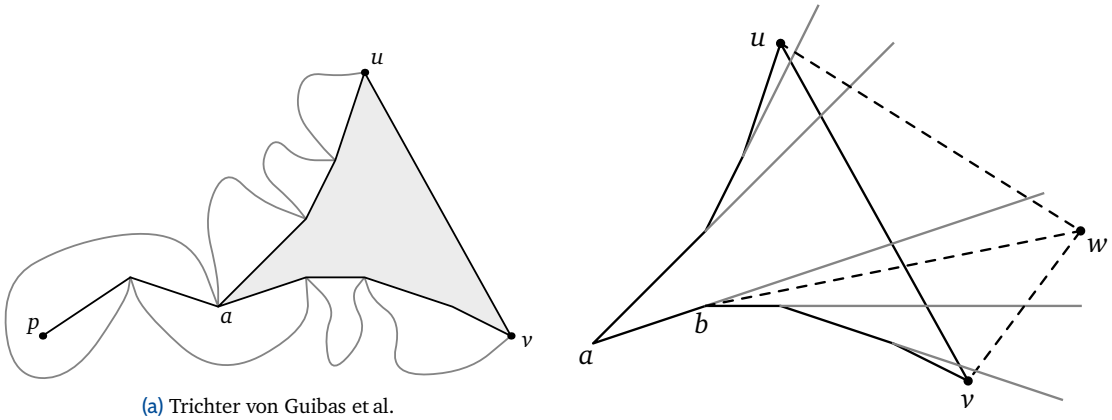
Aus der konvexen Zerlegung ergeben sich die *c-Diagonalen*. Die Endpunkte dieser Diagonalen werden im späteren Verlauf eine wichtige Rolle spielen. Die Anzahl dieser Endpunkte bezeichnen wir mit l.

Auf Basis der konvexen Zerlegung werden nun die Polygone fächerförmig trianguliert. Für P wird eine vereinfachte Triangulierung $T_{C'}$ erstellt, die nur die Diagonal-Eckpunkte enthält (siehe Kapitel 3.8). Für Q wird eine vollständige Triangulierung erstellt. Wir benötigen diese Triangulierung in Kapitel 5.5 für die Berechnung der kürzesten Wege.

Zur Speicherung der Triangulierung nutzen wir eine Datenstruktur aus dem CGAL-Projekt [41]. Die *Triangulation Data Structure* verwaltet eine verkettete Liste von Eckpunkten und Dreiecksflächen und eignet sich sehr gut dazu, über die Elemente der Triangulation in alle Richtungen zu iterieren. Wir werden diese Datenstruktur an zwei Stellen benötigen: beim Berechnen der kürzesten Wege und in der Hauptschleife des Algorithmus für einfache Polygone, bei der rekursiven Berechnung der Combined Reachability Graphs.

5.5. Shortest-Path-Tree-Suche und gültige Platzierungen

Für jede Diagonale der konvexen Zerlegung (c-Diagonale) werden *gültige Platzierungen* ermittelt, das sind Abbildungen der Diagonalen von P auf kürzeste Pfade in Q (siehe Kapitel 3.6). Die Platzierung der Endpunkte ergibt sich aus den *freien Intervallen* im Free-Space-Diagramm (Kapitel 3.5.1). Für jeden Endpunkt finden wir im Free-Space-Diagramm eine Menge von freien Intervallen. Nun muss geprüft werden, ob die Platzierungen der Endpunkte *gültig* sind, d. h. ob die Enden der kürzesten Pfade im Free-Space erreichbar sind (siehe Kapitel 3.6.1).



(a) Trichter von Guibas et al.
Startpunkt p. Scheitelpunkt a.

(b) Aufteilung des Trichters an einem Punkt w.

Abbildung 5.4.: Skizze zum Algorithmus von Guibas et al. [64]
(Zeichnungen: Hershberger und Snoeyink [68])

Hierzu berechnen wir für alle platzierten Endpunkte *Shortest-Path-Trees* nach dem Algorithmus von Guibas et al. [64]. Die Grundlage bildet die zuvor berechnete Triangulierung von Q, die um zusätzliche Punkte für die freien Intervalle erweitert wird. Diese Punkte werden markiert. Wenn die Shortest-Path-Tree-Suche auf einen solchen markierten Eckpunkt stößt, wird ermittelt, ob dieser Punkt im Free-Space-Diagramm erreichbar ist.

Den Algorithmus von Guibas et al. [64] haben wir für Fréchet View selbst implementiert. Wir wollen hier nur die Grundidee erläutern. Ausgehend von einem Startpunkt p wird eine trichterförmige Struktur (Abb. 5.4) erstellt. An einem Dreieck der Triangulation $\Delta(u, v, w)$ konstruieren wir eine neue Teilstrecke des kürzesten Wegs. Der Trichter teilt sich in zwei Teile und die Suche wird auf beiden Wegen fortgesetzt. Der Algorithmus führt damit eine Tiefensuche entlang der Triangulation von Q aus; gleichzeitig werden die kürzesten Wege vom Startpunkt zu allen übrigen Eckpunkten des Polygons konstruiert.

Als Datenstruktur für den „Trichter“ verwenden wir eine doppelseitig offene Warteschlange, wie sie bei Hershberger und Snoeyink [68] beschrieben wird. Für das Backtracking während der Tiefensuche wird dieser Trichter um einen Stack ergänzt.

Gleichzeitig mit den kürzesten Wegen wird ein Free-Space-Diagramm zwischen einer c-Diagonalen und einem kürzesten Weg in Q berechnet. Dieses Free-Space-Diagramm besteht nur aus einer einzigen Spalte: die x-Achse des Free-Space-Diagramms entspricht einer c-Diagonalen in P, während die y-Achse einem kürzesten Pfad in Q entspricht. Zusätzlich speichern wir an den Knoten dieses Pfades die nötigen Informationen für das Free-Space-Diagramm: die Intervalle L^F_{ij} und B^F_{ij}, sowie die erreichbaren Intervalle L^R_{ij} und B^R_{ij}. Das Free-Space-Diagramm folgt somit den Schritten des Algorithmus von Guibas et al. [64] und muss nicht in einer separaten Datenstruktur vorgehalten werden.

Stößt die Suche auf einen markierten Eckpunkt (also das Bild eines Diagonal-Endpunkts in Q), und ist dieser Punkt im Free-Space-Diagramm erreichbar, so haben wir eine *gültige Platzierung* ermittelt. Wir speichern die Menge der gültigen Platzierungen in Form einer Adjazenzmatrix, wie sie auch für Erreichbarkeits-Graphen verwendet wird. Diese Adjazenzmatrizen werden wir in Kapitel 5.6.3 bei der COMBINE-Operation benötigen.

Für jede c-Diagonale $\{d_i, d_j\}$ verwalten wir zwei Matrizen, denn die Diagonale kann wahlweise als (d_i, d_j), oder als (d_j, d_i) platziert werden.

5.6. Datenstruktur für Erreichbarkeits-Graphen

Der Algorithmus von Buchin et al. [27] berechnet zu Beginn eine Reihe von Erreichbarkeits-Strukturen und überträgt diese in Erreichbarkeits-Graphen (Kapitel 3.7 und 3.9).

Die Erreichbarkeits-Graphen sind gerichtete Graphen, deren Knoten den Intervallen der Erreichbarkeits-Struktur entsprechen. Wir speichern die Graphen in Form von Adjazenzmatrizen. Diese Adjazenzmatrizen sind quadratische Matrizen, jeweils ein Bit wird für eine gerichtete Kante benötigt.

Da die Intervalle der Erreichbarkeits-Strukturen unterschiedlich aufgeteilt sind, müssen wir sie auf eine einheitliche, die „feinste“ Aufteilung abbilden. Diese Aufteilung erhalten wir, indem wir alle horizontalen und vertikalen Intervalle des Free-Space übereinander projizieren [27, S. 13].

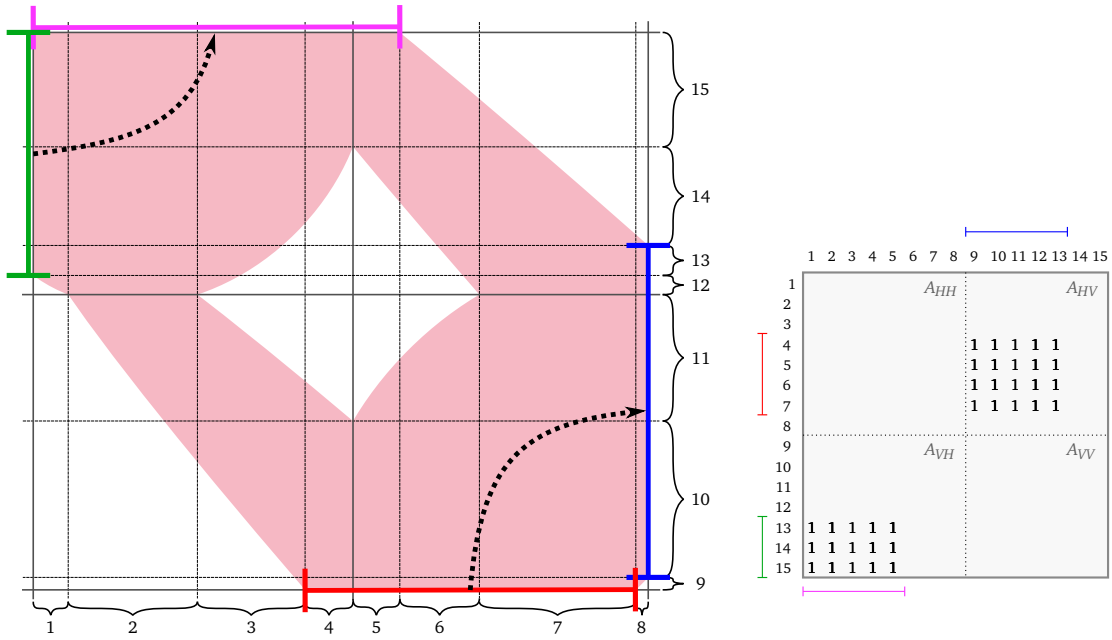
Ein größeres Intervall in der Erreichbarkeits-Struktur wird so auf eine Folge von kleineren Intervallen abgebildet. Jedes dieser (kleinen) Intervalle entspricht einem Knoten des Erreichbarkeits-Graphen. Zu einem ähnlichen Ergebnis käme man, wenn man mit dem Algorithmus von Alt und Godau [13] die Erreichbarkeits-Struktur für den kompletten Free-Space berechnet. In der Erreichbarkeits-Struktur von Alt und Godau [13] werden Strukturen schrittweise verschmolzen und bei jedem Schritt werden u. U. die Intervalle aufgeteilt. Am Ende erhält man ebenfalls die „feinste“ Unterteilung der Intervalle.

5.6.1. Von der Erreichbarkeits-Struktur zur Adjazenzmatrix

Wir zeigen dies an einem Beispiel in Abb. 5.5. Alle Free-Space-Intervalle werden überlagert und bilden damit die kleinstmögliche Aufteilung von Intervallen. Aus diesen Intervallen setzen sich die Erreichbarkeits-Strukturen zusammen. Jedes kleinste Intervall bildet einen Knoten im Erreichbarkeits-Graphen. Die Intervalle bzw. Knoten sind von 1 bis 15 durchnummeriert. Jedem Knoten entspricht eine Zeile und eine Spalte in der Adjazenzmatrix.

Nehmen wir an, die Erreichbarkeits-Struktur enthält eine Verbindung vom Intervall $[4, \ldots, 7]$ (im Beispiel das rote Intervall rechts unten) zum Intervall $[10, \ldots, 13]$ (das blaue Intervall). Der Erreichbarkeits-Graph enthält Kanten für alle Knoten, aus denen sich die Intervalle zusammensetzen. In der Adjazenzmatrix werden die Einträge in den Zeilen 4 bis 7 und in den Spalten 10 bis 13 gesetzt (Abb. 5.5b). Entsprechend verfahren wir mit der Verbindung von $[13, \ldots, 15]$ (das grüne Intervall links oben) nach $[1, \ldots, 5]$ (das pinkfarbene Intervall).

Jede Verbindung in der Erreichbarkeits-Struktur bildet also ein Rechteck von Einträgen in der Adjazenzmatrix. Die Fragmentierung der „feinsten“ Intervalle wächst schnell, nämlich mit $O(mn)$. Entsprechend erhält man auch Adjazenzmatrizen mit $O(mn)$ Zeilen und Spalten.



(a) Verfeinerung der Free-Space-Intervalle (b) Beispiel für eine Adjazenzmatrix

Abbildung 5.5.: Von der Erreichbarkeits-Struktur zur Adjazenzmatrix

5.6.2. Vier Teilmatrizen, Dreiecksmatrizen

Die wichtigsten Operationen auf den Adjazenzmatrizen sind die MERGE-Operation und die COMBINE-Operation. Wir werden nun sehen, dass diese Operationen jeweils nur gewisse Teile der Adjazenzmatrix betreffen. Wir teilen die Matrix in vier Teilmatrizen auf:

$$A = \left(\begin{array}{c|c} A_{HH} & A_{HV} \\ \hline A_{VH} & A_{VV} \end{array}\right)$$

A_{HH}	enthält die Kanten von	horizontalen	Intervallen zu	horizontalen	Intervallen
A_{HV}	-"-	horizontalen	-"-	vertikalen	-"-
A_{VH}	-"-	vertikalen	-"-	horizontalen	-"-
A_{VV}	-"-	vertikalen	-"-	vertikalen	-"-

Die Unterteilung in Teilmatrizen erlaubt uns im Folgenden eine Reihe von Vereinfachungen.

Die Lösungen des Entscheidungsproblems für einfache Polygone finden wir, indem wir einen gültigen Pfad im Combined Reachability Graph suchen. In Abb. 3.12 auf Seite 49 sind solche gültigen Pfade dargestellt.

1. Die Erreichbarkeits-Graphen $\mathrm{RG}(i, j)$ bilden auf der Horizontalen nur einen Ausschnitt des Wertebereichs ab. Damit ist in den Teilmatrizen A_{HH}, A_{HV} und A_{VH} auch nur ein Ausschnitt von Spalten bzw. Zeilen belegt. In unserer Implementierung der Teilmatrizen speichern wir nur die belegten Bereiche (siehe Abb. 5.6).

2. Da in der Erreichbarkeits-Struktur horizontale und vertikale Verbindungen immer monoton steigend sind, stellen A_{HH} und A_{VV} obere quadratische Dreiecksmatrizen dar. Wir werden diese Tatsache später verwenden, um die MERGE-Operation effizienter durchzuführen.

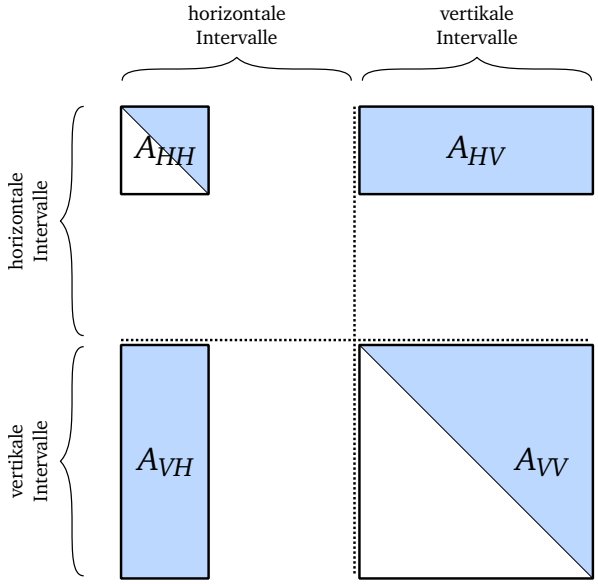


Abbildung 5.6.: Adjazenzmatrix mit kompakter Speicherung

3. Die MERGE-Operation verbindet nur horizontal direkt benachbarte Graphen (modulo l). Sie bildet hierzu den transitiven Abschluss zweier Erreichbarkeits-Graphen. Mit der eben beschriebenen Struktur der Adjazenzmatrizen können wir die MERGE-Operation wie in Abb. 5.7 durchführen. Da die Graphen im Free-Space direkt benachbart sind, können wir die Teilmatrizen nebeneinander bzw. untereinander kopieren. Nur für die vertikalen Kanten muss der transitive Abschluss berechnet werden. Dazu bilden wir die Produkte der Teilmatrizen $A_{HV} \cdot B_{VH}$, $A_{HV} \cdot B_{VV}$, $A_{VV} \cdot B_{VH}$ und $A_{VV} \cdot B_{VV}$.

4. Im mittleren Teil der Lösung in Abb. 3.12 verbinden wir nur vertikale Intervalle. Für die Combined Reachability Graphs $\textsc{crg}\ (i, i-1)$ wird deshalb ausschließlich die Teilmatrix A_{VV} berechnet. Die übrigen drei Teilmatrizen werden für die Lösung nicht benötigt und deshalb in den Combined Reachability Graphs$\textsc{crg}\ (i, j)$ auch nicht berechnet.

Die Teilmatrizen A_{HV} und A_{HV} werden lediglich in der abschließenden MERGE-Operation benötigt (Punkt 6. und 7.). Sie werden für die Erreichbarkeits-Graphen $\textsc{rg}\ (i -_l 1, i)$, berechnet.

5. Die COMBINE-Operation, die gültige Platzierungen filtert, arbeitet ebenfalls nur auf der A_{VV}-Teilmatrix. Dies muss auch so sein, denn die gültigen Platzierungen beschreiben nur vertikale Intervalle im Free-Space.

6. Der erste Teil der gültigen Pfade in Abb. 3.12 besteht aus einer HV-Kante in einem Erreichbarkeits-Graphen, der letzte Teil der Lösung besteht aus einer VH-Kante (im gleichen Erreichbarkeits-Graphen). Die Teilmatrix A_{HH} wird in den Erreichbarkeits-Graphen niemals benötigt und wird deshalb auch nicht berechnet.

7. Im abschließenden Schritt der Lösung wird der Graph

$$G = \textsc{merge}\,(\textsc{rg}(i-1, i), \textsc{crg}(i, i-1), \textsc{rg}(i-1, i))$$

berechnet und wir suchen in diesem Graphen nach einer HH-Kante. Wir müssen für diesen Erreichbarkeits-Graphen also nur die Teilmatrix G_{HH} berechnen.

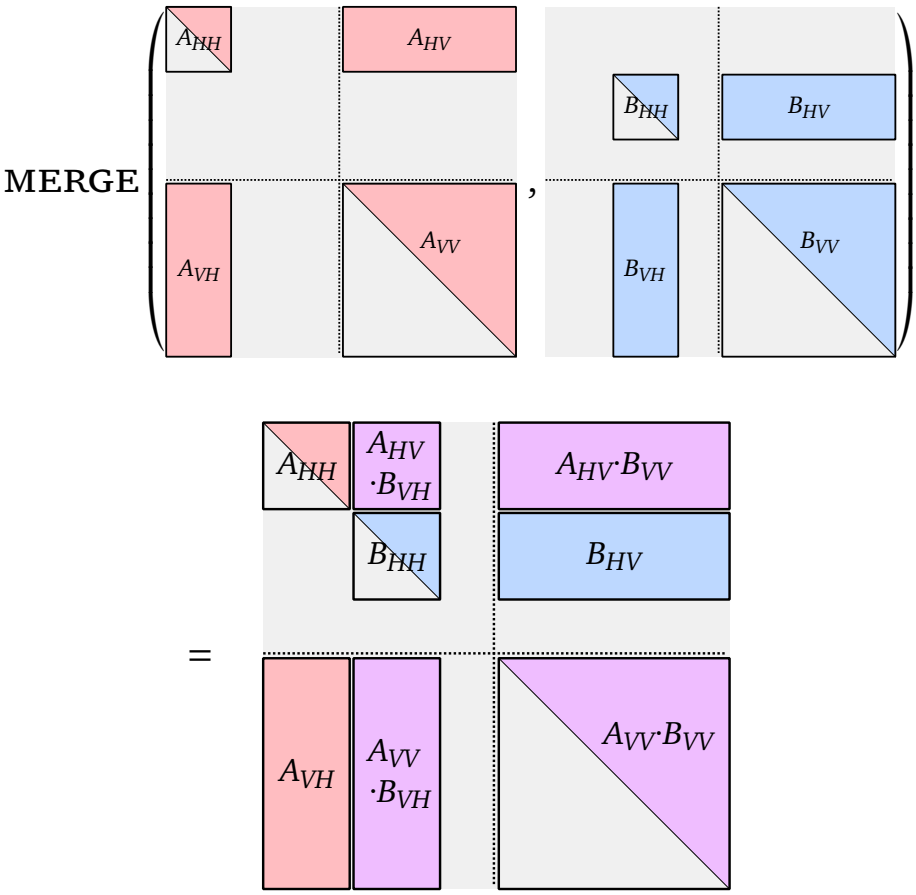


Abbildung 5.7.: MERGE-Operation zweier Adjazenzmatrizen

5.6.3. MERGE- und COMBINE-Operationen

Damit haben wir insgesamt die Operationen auf den Adjazenzmatrizen vereinfacht. Wir fassen zusammen:

MERGE von zwei Combined Reachability Graphs im inneren Teil der Lösung

$$C = \text{MERGE}(A, B) \qquad \text{wird implementiert als} \qquad C_{VV} = A_{VV} \odot B_{VV}$$

Mit $\odot$ beschreiben wir die Boolesche Matrixmultiplikation. Die elementweise Multiplikation entspricht hierbei der logischen AND-Operation und die Addition der logischen OR-Operation. Wie wir weiter oben festgestellt haben, handelt es sich hier um quadratische Dreiecksmatrizen, die wir effizienter multiplizieren können.

MERGE zwischen einem Erreichbarkeits-Graphen und einem Combined Reachability Graph im abschließenden Teil der Lösung

$$G = \text{MERGE}(A, B, A) \qquad \text{wird implementiert als} \qquad G_{HH} = A_{HV} \odot B_{VV} \odot A_{VH}$$

COMBINE auf einem Combined Reachability Graph

$$C = \text{COMBINE}(A, P) \qquad \text{wird implementiert als} \qquad C_{VV} = A_{VV} \otimes P_{VV}$$

P beschreibt eine Matrix mit gültigen Platzierungen, die wir zuvor berechnet haben (siehe Kapitel 5.5). Mit $\otimes$ bezeichnen wir die logische AND-Verknüpfung. Damit werden nur die gültigen Platzierungen übernommen.

Wir benötigen zur Realisierung der MERGE- und COMBINE-Operationen lediglich Boolesche Matrixmultiplikationen und elementweise AND-Operationen. Dies deckt sich mit der Komplexitätsanalyse in Kapitel 3.13.1, bei der die Matrixmultiplikation den zweiten wichtigen Faktor darstellt. Auf die Implementierung der Booleschen Matrixmultiplikation werden wir in Kapitel 6 und Kapitel 8 ausführlich eingehen.

5.6.4. Alternative Datenstrukturen für Erreichbarkeits-Graphen?

Wie wir gesehen haben, wächst die Größe der Adjazenzmatrizen mit $O(mn)$. Die Laufzeit für eine MERGE-Operation beträgt $O(T(mn))$, also die Zeit für eine Matrixmultiplikation (siehe Kapitel 3.13.1). Gibt es noch andere, kompaktere oder effizientere Möglichkeiten zur Speicherung von Erreichbarkeits-Graphen?

Adjazenzlisten Wenn die Erreichbarkeits-Graphen sehr dünn besetzt sind, kommen Adjazenzlisten in Frage. Die Dichte der Graphen hängt von den Eingabedaten ab und lässt sich nicht allgemein abschätzen. Unserer Beobachtung nach sind die Erreichbarkeits-Graphen oft vergleichsweise dicht besetzt (bis zu ca. 40%) und werden durch die folgenden MERGE-Operationen noch dichter. Hingegen werden sie durch die COMBINE-Operation auch wieder dünner.

R-Bäume Die Speicherung von rechteckigen Bit-Blöcken ist für sehr große Rechtecke nicht unbedingt vorteilhaft. Diese Rechtecke könnte man in einem R-Baum ggf. kompakter speichern. Allerdings gestaltet sich dann die MERGE-Operation aufwendiger. Wenn wir annehmen, dass die MERGE-Operation auf einem R-Baum ähnlich wie eine Matrixmultiplikation durchgeführt wird, und wenn wir weiter annehmen, dass das Einfügen in einen R-Baum im Mittel mindestens $O(\log n)$ kostet,[1] dann wären die Kosten für eine MERGE-Operation mit mehr als $O(n^3 \log n)$ zu hoch – von konstanten Faktoren ganz abgesehen. Deshalb haben wir diese Idee nicht weiter verfolgt.

Erreichbarkeits-Strukturen Die Erreichbarkeits-Struktur von Cordell [49] hatten wir in Kapitel 3.15.1 bereits angesprochen. Sie stellt eine interessante Alternative zu den Erreichbarkeits-Graphen dar. Sie kann eine MERGE-Operation in $O(n^5)$ durchführen und verspricht damit eine Beschleunigung um nahezu einen linearen Faktor gegenüber dem Algorithmus von Buchin et al. [27]. Wie sich diese Datenstruktur in einer praktischen Implementierung schlägt, ist eine offene Frage.

5.7. Implementierung der Optimierungsvariante

Wie wir in Kapitel 3.14.2 gesehen haben, kann das Optimierungsproblem für einfache Polygone durch eine Binärsuche auf den sortierten kritischen Werten gelöst werden. Die parametrische Suche nach Cole und Megiddo [43, 79] bringt keinen Vorteil für die Gesamtlaufzeit.[2] Erfreulicherweise ist die Binärsuche auch deutlich einfacher zu realisieren.

Wir berechnen zunächst eine sortierte Liste der kritischen Werte (Kapitel 2.5.1 und 3.14.1). Diese Berechnung kann auch parallel ausgeführt werden (Kapitel 7.5). Dann führen wir eine

[1] Eine genaue Komplexitätsabschätzung für R-Bäume ist recht schwierig und soll hier nicht stattfinden.

[2] Die parametrische Suche kommt beim Algorithmus von Alt und Godau [13] in Frage. Wir haben sie aber nicht implementiert.

Binärsuche auf dieser Liste durch und entscheiden in jedem Schritt, ob $d_F \leq \varepsilon$ ist, d. h. wir lösen in jedem Schritt der Binärsuche das Entscheidungsproblem.

Wir wissen, dass der Fréchet-Abstand für einfache Polygone durch den Fréchet-Abstand ihrer Randkurve nach unten begrenzt ist. Die untere Grenze der Binärsuche können wir deshalb mit dem – deutlich schnelleren – Algorithmus für Polygonzüge berechnen. Wenn der Fréchet-Abstand für Polygone mit dieser Untergrenze zusammenfällt, dann reicht ein einziger Aufruf des Entscheidungsalgorithmus. In praktischen Beispielen ist dies sogar sehr häufig der Fall. Nur wenn sich der Fréchet-Abstand für Polygone von dem der Randkurve unterscheidet (siehe z. B. Kapitel 3.2), müssen wir die Binärsuche auf den kritischen Werten fortsetzen.

5.7.1. Die kritischen Werte sind tatsächlich kritisch

Bei der praktischen Umsetzung der Optimierungsvariante stoßen wir nun auf numerische Probleme. Die kritischen Werte sind diejenigen Werte von ε, an denen sich neue Wege im Free-Space-Diagramm öffnen. Leider reagieren sie damit empfindlich auf Rundungsfehler. Wenn aufgrund von Rundungsfehlern ein kritischer Wert etwas zu klein berechnet wird, kann es sein, dass ein Weg im Free-Space nicht erkannt wird und der Algorithmus eine falsche Entscheidung trifft. Dann wird die Binärsuche den nächstgrößeren kritischen Wert als Lösung ausgeben. Die Abstände zwischen den kritischen Werten hängen lediglich von den Eingabedaten ab, sie können beliebig groß werden. Wir haben damit den ungünstigen Fall, dass kleine Rundungsfehler zu einem unbestimmt großen Fehler im Ergebnis führen können. Rundungsfehler nach oben stellen für den Entscheidungsalgorithmus hingegen kein Problem dar.

Eine genaue Analyse der Fehlergrenzen wäre sehr aufwendig und ist wiederum von den Eingabedaten abhängig, also nicht allgemein durchführbar.

Wir haben uns deshalb für eine pragmatische Lösung entschieden. Zunächst reduzieren wir Rundungsfehler, indem wir die Zwischenschritte der Berechnung möglichst präzise ausführen. Moderne Prozessoren und C++-Compiler bieten die Möglichkeit, interne Berechnungen mit 80-Bit-Fließkommazahlen durchzuführen. Die Eingabedaten bestehen aus 64-Bit-Fließkommazahlen. Damit haben wir eine Fehlermarge von 16 Bit gewonnen, das entspricht ungefähr 5 Dezimalstellen.

Zusätzlich schlagen wir bei den kritischen Werten eine fest vorgegebene Fehlermarge auf. Wir verlieren etwas Genauigkeit bei der Berechnung der kritischen Werte, verringern aber die Wahrscheinlichkeit von Fehlentscheidungen.

5.7.2. Intervallschachtelung

Als Alternative zur Binärsuche über kritische Werte haben wir in Kapitel 2.5.4 eine Intervallschachtelung vorgestellt. Die kritischen Werte werden hierzu nicht benötigt. Wir beginnen mit dem maximalen Intervall $[0, \varepsilon_{max}]$. Die Obergrenze ε_{max} können wir aus den Eingabedaten ableiten. In jedem Schritt halbieren wir das Intervall und befragen den Entscheidungsalgorithmus. Dies führen wir so lange durch, bis eine vorgegebene Genauigkeit erreicht wird.

Diese Intervallschachtelung liefert nur eine Näherungslösung. Wir können aber annehmen, dass die Genauigkeit der Lösung ungefähr in dem Maße wächst, in dem wir die Intervallgröße verringern. Wir können also die Genauigkeit der Lösung sehr viel besser beeinflussen. Damit haben wir ein Hilfsmittel, um die oben beschriebenen Schwierigkeiten der Optimierungsvariante in den

Griff zu bekommen. Wenn die Intervallschachtelung eine kleinere Lösung liefert als die Binärsuche auf den kritischen Werten, so ist dies ein Indiz dafür, dass die Binärsuche den korrekten Wert verpasst hat.

Die näherungsweise Intervallschachtelung kann in der graphischen Oberfläche mit den Schaltern Approximate berechnet werden, oder mit dem Kommandozeilenschalter `--approx`.

5.8. Visualisierung eines gültigen Pfads

Immer dann, wenn der Entscheidungsalgorithmus (für Polygonzüge oder Polygone) eine Lösung gefunden hat, stellen wir einen gültigen Pfad im Free-Space-Diagramm graphisch dar. In Abb. 2.7b auf Seite 27 zeigen wir ein Beispiel. Die Berechnung des gültigen Pfads ist nicht eigentlicher Bestandteil der Algorithmen, dient aber der besseren Visualisierung der Ergebnisse.

Die Entscheidungsalgorithmen liefern als Ergebnis lediglich Start- und Endintervall des gültigen Pfads. Wir können daraus den Verlauf eines gültigen Pfads rekonstruieren. Ausgehend vom Startpunkt des gültigen Pfads berechnen wir die erreichbaren Intervalle im Free-Space-Diagramm. Der Endpunkt des gültigen Pfads muss in einem erreichbaren Intervall liegen, ansonsten hätte der Entscheidungsalgorithmus einen Fehler gemacht. Ausgehend vom Endpunkt konstruieren wir dann rückwärts einen Pfad durch die erreichbaren Intervalle.

Für geschlossene Kurven (und für Polygone) wird in der Literatur meist ein doppeltes Free-Space-Diagramm verwendet (siehe Abb. 2.7a auf Seite 27). Hierbei ist die rechte Hälfte des Free-Space-Diagramms eine identische Kopie der linken Hälfte. Der gültige Pfad verläuft als monotone Kurve vom unteren Rand zu einem Zielpunkt am oberen Rand.

Bei der graphischen Darstellung in Fréchet View haben wir uns hingegen entschlossen, nur ein *einfaches* Free-Space-Diagramm anzuzeigen. Wir denken, dass diese Darstellung für den Benutzer übersichtlicher ist. Die Vervollständigung zum doppelten Free-Space-Diagramm bleibt der Vorstellungskraft des Betrachters überlassen. Der rechte Rand des Free-Space-Diagramms geht über in den linken Rand und man kann sich das Free-Space-Diagramm als geschlossenen Zylinder vorstellen. Der gültige Pfad verlässt das Free-Space-Diagramm am rechten Rand und setzt sich am linken Rand fort. Der gültige Pfad wird also in zwei Abschnitte aufgeteilt, die man sich verbunden vorstellen muss (siehe Abb. 2.7b auf Seite 27).

Sind beide Kurven geschlossen, so bildet das Free-Space-Diagramm gewissermaßen einen Torus. Für unsere Algorithmen ist dies aber nicht relevant. Start- und Endpunkte des gültigen Pfads befinden sich immer am *unteren* und *oberen* Rand des Free-Space-Diagramms.

Gültiger Pfad für einfache Polygone Beim Entscheidungsalgorithmus für einfache Polygone müssen wir zusätzliche Stützpunkte innerhalb des gültigen Pfads berücksichtigen (siehe Abb. 3.12 auf Seite 49). Wir ermitteln sie aus den Combined Reachability Graphs, welche die Lösung des Algorithmus bilden. Während der Berechnung der Combined Reachability Graphs mit Hilfe der MERGE-Operationen speichern wir zusätzliche Verweise auf die Vorgänger, aus denen sich die Lösung zusammensetzt. Aufgrund der Memoisierungs-Technik (Kapitel 3.12) werden ohnehin alle Graphen in einer Datenstruktur vorgehalten. Das Speichern der Lösung benötigt daher keinen zusätzlichen Platz. Den Pfad zwischen zwei Stützpunkten finden wir – wie oben beschrieben – durch das Propagieren der erreichbaren Intervalle.

Mouse Over Der gültige Pfad wird als gelbe Linie im Free-Space-Diagramm dargestellt. Wenn der Benutzer die Maus über den gültigen Pfad hält, wird ein Segment markiert sowie die Abbildung dieses Segments auf die Kurven P und Q. Somit kann der Benutzer die einzelnen Abschnitte der Reparametrisierungen nachvollziehen. Hält der Benutzer die Maus über ein Polygonsegment

oder eine Diagonale, so wird ebenfalls die Abbildung des Homöomorphismus und der zugehörige Abschnitt im gültigen Pfad angezeigt. Die erreichbaren Intervalle können in Fréchet View mit dem Schalter Show Bounds angezeigt werden.

Cordell [49] hat einen Algorithmus vorgestellt, der gleichzeitig mit dem Entscheidungsalgorithmus auch einen gültigen Pfad (quasi als Nebenprodukt) konstruiert.

5.9. Entscheidungsalgorithmus in Reverse-Level-Order

Im Hauptteil des Entscheidungsalgorithmus für einfache Polygone führen wir eine Suche über die c- und t-Diagonalen von P durch (Kapitel 3.12). Wir berechnen in einem rekursiven Verfahren die Combined Reachability Graphs.

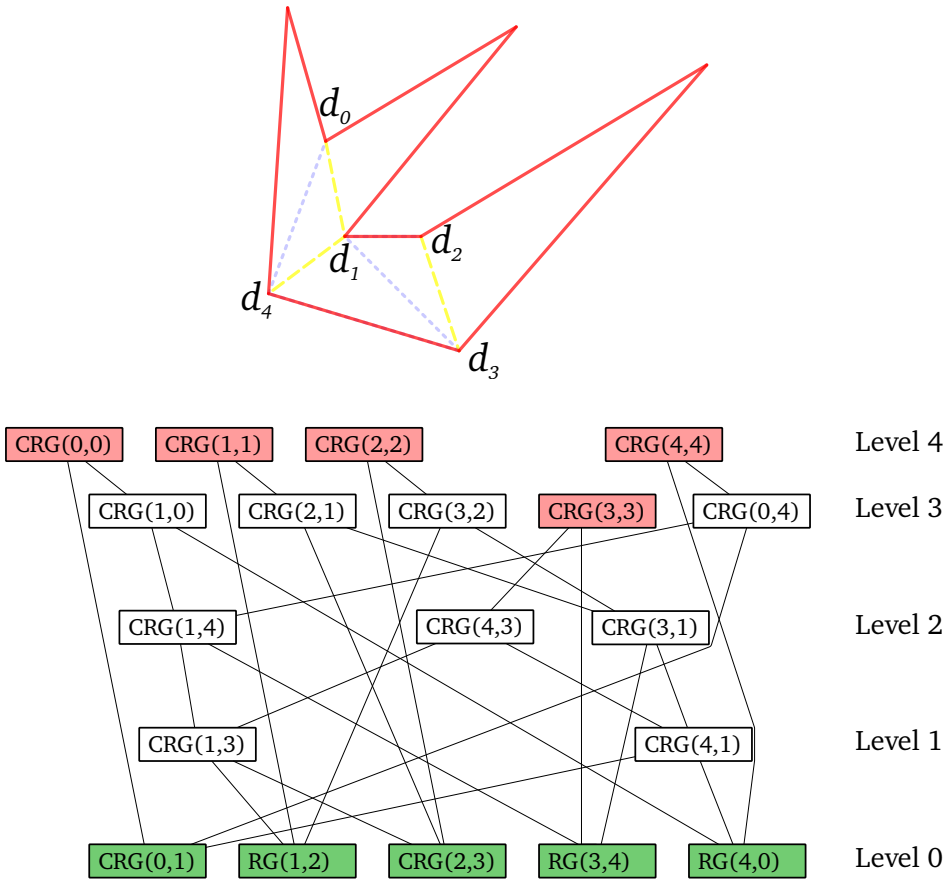


Abbildung 5.8.: Eine triangulierte Kurve und der zugehörige Aufruf-Graph, sortiert in Reverse-Level-Order.

Die Blätter (grün) stehen für die Erreichbarkeits-Graphen, mit oder ohne COMBINE-Operation.
Die Wurzeln (rot) sind potentielle Lösungen.
Innere Knoten entstehen durch MERGE-Operationen (und ggf. COMBINE-Operationen).

In Abb. 5.8 haben wir beispielhaft einen Aufruf-Graphen dargestellt. Die Kanten dieses gerichteten, azyklischen Graphen (DAG) stellen Aufruf-Abhängigkeiten dar. Ein Knoten kann erst bearbeitet werden, wenn alle seine Nachfolger berechnet wurden.

Die Blätter des Graphen sind die zu Anfang berechneten Erreichbarkeits-Graphen $\text{RG}\ (i, i +_l 1)$. Die Wurzeln des Graphen bilden die Combined Reachability Graphs, die eine potentielle Lösung darstellen. Um redundante Berechnung zu sparen, werden alle Zwischenergebnisse in einer Datenstruktur vorgehalten („memoisiert“).

Es ist naheliegend, diesen Aufruf-Graphen mit rekursiven Funktionsaufrufen zu implementieren. Der Graph wird dann in Post-Order abgearbeitet.

Wir zeigen nun, wie wir den Aufruf-Graphen auch in einer anderen Reihenfolge bearbeiten können. Dazu führen wir den *Reverse-Level* (oder Bottom-Level) eines Knotens ein. Er beschreibt die maximale Entfernung eines Knotens zu einem Blatt.

Der Reverse-Level eines Blattknotens ist 0. Der Reverse-Level eines inneren Knotens ist um eins größer als der maximale Reverse-Level seiner Nachfolger.

Wir stellen fest, dass zwischen Knoten mit gleichem Reverse-Level keine Abhängigkeiten bestehen. Damit eröffnet sich die Möglichkeit, alle Knoten mit gleichem Reverse-Level parallel zu bearbeiten. Wir werden diese Idee in Kapitel 7.7 und Kapitel 8.7 diskutieren.

Weiter können wir anhand des Aufruf-Graphen feststellen, ob alle Vorgänger eines Knotens bearbeitet wurden. Wir können nicht mehr benötigte Nachfolger aus der Memoisierungs-Struktur entfernen und damit Speicherplatz sparen. In Kapitel 8.7 werden wir darauf zurückkommen.

Zur Erstellung und Verwaltung des Aufruf-Graphen müssen wir keine neue Datenstruktur einführen; es reicht aus, die bereits vorhandenen Datenstrukturen zur Memoisierung geringfügig zu erweitern. Wir merken an, dass der Aufbau des Graphen nur von den Eingabedaten P und Q abhängig ist, nicht aber von ε. Wir müssen ihn also pro Eingabe nur einmal erstellen.

Unsere Versuche zeigen, dass der Aufruf-Graph oft eine Sanduhr-artige Form annimmt. Nahe der Wurzeln und an den Blättern ist er sehr breit, im mittleren Teil finden sich aber langgestreckte Abschnitte, die sehr schmal sind. In diesen schmalen Abschnitten ist die Parallelisierung weniger effektiv.

Die Berechnung nach Reverse-Level-Order lässt sich in Fréchet View über den Kommandozeilen-Schalter `--large` aktivieren (da diese Option in erster Linie für große Eingabedaten gedacht ist). Auf die Laufzeit des Entscheidungsalgorithmus hat dies keinen wesentlichen Einfluss.

5.10. Berechnung des k-Fréchet-Abstands

Die Berechnung des k-Fréchet-Abstands aus Kapitel 4 unterscheidet sich deutlich von der Berechnung des „klassischen“ Fréchet-Abstands. Die Grundlage bildet wieder das Free-Space-Diagramm, aber wir benötigen weder Erreichbarkeits-Strukturen, noch gültige Pfade. Zuerst ermitteln wir zusammenhängende Komponenten im Free-Space-Diagramm. Wir projizieren diese Komponenten auf die Wertebereiche (die P- und Q-Achse des Free-Space-Diagramms) und erhalten so eine Menge von Intervallen, die sich überlappen können. Wir wählen k Komponenten aus, deren Projektionen die Wertebereiche vollständig abdecken.

5.10.1. Zusammenhängende Komponenten

Zwei benachbarte Zellen des Free-Space-Diagramms gehören zur gleichen Komponente, wenn das gemeinsame Free-Space-Intervall (Kapitel 2.2) nicht leer ist. Anhand der Free-Space-Intervalle können wir somit leicht die zusammenhängenden Komponenten ermitteln.

Zur Ermittlung der Komponenten verwenden wir eine Union-Find-Struktur, das ist eine Datenstruktur zur Verwaltung von (disjunkten) Partitionen einer Menge. Die Boost-Bibliothek [20] stellt uns eine solche Datenstruktur zur Verfügung.

Jede Zelle des Free-Space-Diagramms erhält initial einen eindeutigen Eintrag in dieser Datenstruktur, dann fassen wir alle zusammenhängenden Zellen zu Komponenten zusammen. Eine Änderung der Union-Find-Struktur kann in nahezu konstanter Zeit durchgeführt werden.

Wenn wir nm Zellen im Free-Space haben, dann benötigt die Ermittlung der verbundenen Komponenten insgesamt $O(nm + nm \cdot \alpha(nm))$ Schritte. Mit $\alpha(nm)$ bezeichnen wir die Umkehrfunktion der Ackermann-Funktion. Sie wächst so langsam, dass sie für praktische Zwecke als konstant angenommen werden kann.

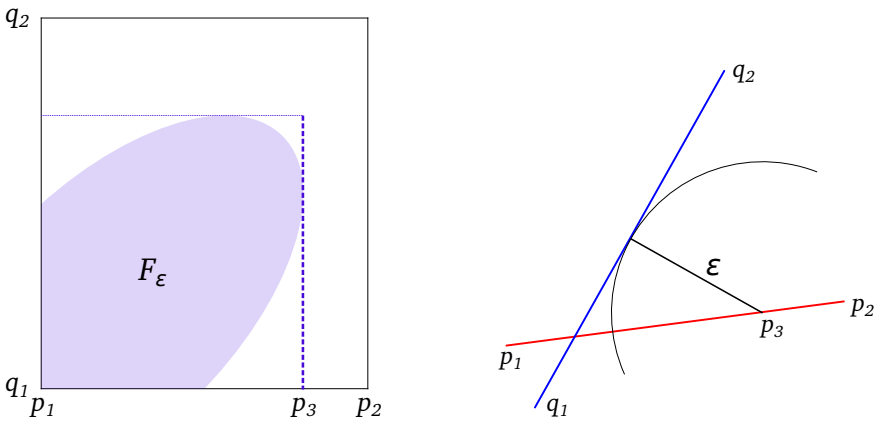
Alternativ könnten wir die Komponenten auch in $O(nm)$ mit einer Breiten- oder Tiefensuche ermitteln, aber der praktische Vorteil ist gering.

In der graphischen Darstellung haben wir die Komponenten farblich hervorgehoben. Wie man in Abb. 4.9b auf Seite 63 sieht, können solche Free-Space-Diagramme ziemlich bunt werden.

5.10.2. Bounding Boxes der Komponenten

Nun projizieren wir jede zusammenhängende Komponente auf die P- und Q-Achse des Free-Space-Diagramms. Hierzu müssen wir die Bounding Boxes (die umgebenden Rechtecke) der Komponenten berechnen. Im Programm Fréchet View lassen sich die Bounding Boxes der Zellen mit dem Schalter Show Bounds anzeigen.

Wir erinnern uns, dass die Zellen des Free-Space F_ε jeweils aus Ellipsen-Segmenten bestehen. Die Bounding Box einer solchen Ellipse berechnen wir wie in Abb. 5.9b angedeutet. Der Punkt p_3 ist derjenige Punkt auf $\overline{p_1p_2}$, welcher genau den Abstand ε zu $\overline{q_1q_2}$ hat.



(a) Ränder von F_ε

(b) geometrische Konstruktion

Abbildung 5.9.: Bounding Box einer Free-Space-Zelle

5.10.3. Greedy-Algorithmus

In Kapitel 4.5 haben wir einen Greedy-Näherungsalgorithmus für den k-Fréchet-Abstand beschrieben. Wir implementieren ihn so, dass wir zusätzlich noch nützliche Unter- und Obergrenzen für den Brute-Force-Algorithmus ermitteln können.

Der Greedy-Algorithmus liefert, bezogen auf einen einzelnen Wertebereich des Free-Space-Diagramms, jeweils ein optimales Ergebnis. Wir bezeichnen mit k_P das Greedy-Ergebnis für die P-Achse (d. h. für die Projektion der zusammenhängenden Komponenten auf den Wertebereich von P). k_Q bezeichnet entsprechend das Greedy-Ergebnis für die Q-Achse.

Beim Greedy-Algorithmus für beide Achsen können wir wahlweise zuerst die P-Achse bearbeiten, oder zuerst die Q-Achse. Wir berechnen immer *beide* Varianten und erhalten damit die Näherungswerte

$$k_{PQ} \quad = \quad \text{Greedy: erst } P\text{-Achse dann } Q\text{-Achse} \quad \leq \quad k_P + k_Q$$

und

$$k_{QP} \quad = \quad \text{Greedy: erst } Q\text{-Achse dann } P\text{-Achse} \quad \leq \quad k_P + k_Q \ .$$

In Kapitel 4.5.1 haben wir gezeigt, dass der Näherungsfaktor des Greedy-Algorithmus 2 beträgt. Wir können damit für den optimalen Wert k_{opt} folgende Abschätzungen vornehmen:

$$\max\{k_{PQ}/2,\ k_{QP}/2\} \quad \leq \quad \max\{\ k_P,\ k_Q\ \} \quad \leq \quad k_{opt} \quad \leq \quad \min\{\ k_{PQ},\ k_{QP}\ \}$$

5.10.4. Brute-Force-Algorithmus

Der Brute-Force-Algorithmus wurde weiter oben in Kapitel 4.4 vorgestellt. Wir suchen die kleinste Teilmenge von Intervallen, die beide Wertebereiche (P- und Q-Achse) abdeckt. Wir führen dazu eine erschöpfende Suche durch. Eine Breitensuche über alle $O(\binom{n^2}{k})$ Teilmengen verbietet sich, da der Suchraum sehr groß werden kann. Wir führen stattdessen eine Tiefensuche durch.

Wie wir gerade gesehen haben, liefert uns der Greedy-Algorithmus Unter- und Obergrenzen für die Größe des Ergebnisses. Diese Grenzen bezeichnen wir mit

$$k_{min} = \max\{k_P, k_Q\} \qquad \text{und} \qquad k_{max} = \min\{k_{PQ}, k_{QP}\} \ .$$

Den von der Tiefensuche durchsuchten Raum können wir uns als Baum vorstellen (siehe auch Kapitel 4.6.1). Seine Tiefe ist zwar durch k_{max} beschränkt, aber seine Größe wächst exponentiell mit der Suchtiefe. Wir behelfen uns mit einer *iterativen* Tiefensuche. Wir führen zunächst eine durch k_{min} beschränkte Suche durch. Wenn wir nicht fündig werden, erhöhen wir die Tiefe des Suchbaums schrittweise auf $k_{min} + 1, k_{min} + 2$, bis zu k_{max}.

Diese Vorgehensweise beinhaltet offenbar eine gewisse Redundanz. Der Suchbaum für k ist im nächsten Schritt $k + 1$ vollständig enthalten. Der Vorteil liegt aber darin, dass wir die Suche bei der ersten gefundenen Lösung abbrechen können. Der Suchbaum wird also niemals tiefer als nötig. Da seine Größe exponentiell mit der Suchtiefe wächst, ist der Preis für die redundanten Iterationen vertretbar.

Aus dem gleichen Grund wenden wir hier auch keine Binärsuche zum Finden von k_{opt} an. Die Kosten für große Werte von k wären höher als die Summe der iterativen Schritte.

5.11. Visualisierung

Wir beschreiben hier kurz einige Details der Implementierung der graphischen Oberfläche Fréchet View. Dieses Kapitel ersetzt keine Bedienungsanleitung. Eine kurze Bedienungsanleitung und weiteres Material zur Einführung findet sich im Anhang, auf der beiliegenden CD und im Internet.

Die graphische Oberfläche von Fréchet View stellt die Eingabekurven P und Q dar sowie das (einfache) Free-Space-Diagramm. Der Benutzer kann den Parameter ε verändern und die Auswirkungen im Free-Space-Diagramm unmittelbar beobachten. Man kann verschiedene Algorithmen – sowohl Entscheidungsproblem als auch Optimierungsproblem und Intervallschachtelung – ausführen und ihre Ergebnisse vergleichen.

Sofern das Entscheidungsproblem eine positive Antwort liefert, wird ein gültiger Pfad im Free-Space-Diagramm dargestellt. Im Free-Space-Diagramm können die erreichbaren Intervalle angezeigt werden. Die Homöomorphismen lassen sich nachvollziehen, indem der Benutzer die Maus über ein Polygonsegment oder einen Abschnitt des gültigen Pfads hält.

Beim Algorithmus für den k-Fréchet-Abstand hat der Benutzer die Wahl zwischen dem Greedy-Algorithmus und dem Brute-Force-Algorithmus. Die Ergebnisse des Greedy-Algorithmus werden unmittelbar angezeigt, der Brute-Force-Algorithmus muss vom Benutzer per Knopfdruck gestartet werden. Die zusammenhängenden Komponenten, ihre Bounding Boxes sowie die Projektion der Komponenten auf die Wertebereiche lassen sich anzeigen.

5.11.1. Eingabedateien

Die Kurven P und Q werden aus einer Datei eingelesen. Jede Datei sollte zwei Polygonzüge enthalten (keine Beziérkurven, oder Ähnliches). Für den Algorithmus von Buchin et al. [27] erwarten wir *einfache* Polygone (also geschlossene Polygonzüge ohne Überschneidungen).

Als Dateiformate akzeptieren wir die Vektorgraphikformate SVG (Scalable Vector Graphics) und IPE (vom gleichnamigen Graphikprogramm). Beides sind XML-basierte Dateiformate. Wir lesen sie mit dem in Qt [88] integrierten XQuery-Parser.

Skript-Dateien Zusätzlich können Kurven per JavaScript-Dateien „programmiert“ werden. Wir stellen hierfür eine einfache API zur Verfügung, eine Art „Turtle Graphics“, mit der sich die beiden Polygonzüge erzeugen lassen. Eine Beschreibung findet im Anhang B.1.2. Damit lassen sich komplexe Polygonzüge erstellen, die mit einem graphischen Editor nur schwer zu zeichnen wären. Die Beispiele in Kapitel 4.3.1 und 4.5.1 wurden durch solche Skripte erzeugt.

5.11.2. Integration der Algorithmen in die graphische Oberfläche

Einige Algorithmen haben sehr kurze Laufzeiten. Sie werden unmittelbar ausgeführt, wenn der Benutzer den Wert von ε ändert. Zu den kurzlaufenden Aufgaben zählen:

- die Berechnung des Free-Space-Diagramms,
- der Entscheidungsalgorithmus für Polygonzüge. Wenn das Entscheidungsproblem eine Lösung hat, wird sofort ein gültiger Pfad im Free-Space-Diagramm angezeigt.
- der Greedy-Algorithmus für den k-Fréchet-Abstand. Die Ergebnisse k_{PQ} und k_{QP} (aus Kapitel 5.10.3) werden angezeigt.

Länger laufende Algorithmen werden vom Benutzer per Knopfdruck gestartet:

- der Entscheidungsalgorithmus für einfache Polygone,
- die Optimierungsalgorithmen für Polygonzüge und für einfache Polygone,
- der Brute-Force-Algorithmus für den k-Fréchet-Abstand.

Um die Bedienung der graphischen Oberfläche nicht zu behindern, arbeiten die langlaufenden Algorithmen in separaten Threads (Arbeitsthreads). Der Benutzer hat jederzeit die Möglichkeit, einen langlaufenden Arbeitsthread abzubrechen (z. B. den Brute-Force-Algorithmus mit exponentieller Laufzeit).

Arbeitsthreads werden auf folgende Weise kontrolliert beendet:

1. wenn der Benutzer den Abbruch anfordert, wird zunächst ein Flag gesetzt (mit der Speicherklasse `volatile`).
2. der Arbeitsthread prüft dieses Flag in regelmäßigen Abständen.
3. wenn der Arbeitsthread feststellt, dass der Benutzer das Flag gesetzt hat, bricht er seine Arbeit ab. Dies kann am Einfachsten dadurch geschehen, dass der Arbeitsthread eine Exception wirft.
4. die Applikationslogik fängt die Exception, gibt die entsprechenden Speicher-Ressourcen frei und aktualisiert die Benutzeroberfläche.

Nur auf diese Art und Weise werden Threads zuverlässig und ohne Speicher-Leaks beendet. Die Arbeitsthreads wurden mit der Qt-Thread-API realisiert, da sie sich einfach in die Anwendung integrieren lässt. Für Multi-Core-Anwendungen in Kapitel 7 verwenden wir zusätzlich die TBB-Thread-API [71]. Erfreulicherweise lassen sich beide APIs (Qt-Thread und TBB) reibungslos miteinander kombinieren.

5.12. Bedienung in einem Kommandozeilen-Terminal

Für weitergehende Tests oder Benchmarks lässt sich Fréchet View über die Kommandozeile steuern. Im Kommandozeilen-Modus wird einer der Algorithmen auf eine Datei angewandt und die Ergebnisse ausgegebenen. Zusätzlich werden auch genaue Zeitmessungen gemacht. Die Ergebnisse in Kapitel 9 wurden so ermittelt. Im Anhang B.1.3 werden die Kommandozeilen-Schalter beschrieben.

5.13. Unit Tests

Die Entwicklung der Algorithmen wurde durch Unit Tests begleitet. Es wurden Tests erstellt, mit denen einzelne Komponenten automatisiert mit vorgegebenen oder zufällig erzeugten Daten getestet werden können. So entstanden zahlreiche Testfälle für Erreichbarkeits-Strukturen, Erreichbarkeits-Graphen, für die konvexe Zerlegung und Triangulierung von Polygonen, für den Algorithmus von Guibas et al. [64] sowie für die Boolesche Matrixmultiplikation.

Weiter wurden im Source-Code Testbedingungen (Assertions) eingebaut. Diese Testbedingungen werden in der veröffentlichten Programmversion ausgeschaltet. Während der Entwicklung sind sie aber ein wichtiges Hilfsmittel, um Fehler aufzuspüren.

Assertions und Unit Tests sind unerlässliche Entwicklungswerkzeuge, um komplexe Algorithmen wie die hier vorgestellten zu entwickeln. Eine Garantie für eine fehlerfreie Implementierung bieten sie aber selbstverständlich nicht.

6

Boolesche Matrixmultiplikation

In Kapitel 5.6 haben wir wir uns dafür entschieden, die Erreichbarkeits-Graphen in Form von Booleschen Adjazenzmatrizen zu realisieren. Die aufwendigste Operation auf Erreichbarkeits-Graphen ist die MERGE-Operation. Sie berechnet den transitiven Abschluss zweier Graphen. Diese Operation implementieren wir durch die Multiplikation zweier (Teil-) Matrizen (Kapitel 5.6.3). Die MERGE-Operationen beanspruchen einen großen Teil der Gesamtlaufzeit des Algorithmus für einfache Polygone. Wir widmen deshalb ihrer effizienten Implementierung besondere Aufmerksamkeit.

Definition 6.1 Boolesche Algebra

Der Wertebereich der Booleschen Algebra ist $\mathbb{B} = \{0, 1\}$.

Mit $\vee$ bezeichnen wir die logische Disjunktion (OR-Verknüpfung).
Mit $\wedge$ bezeichnen wir die logische Konjunktion (AND-Verknüpfung).

$\mathbb{B}$ bildet mit $\vee$ als Addition und $\wedge$ als Multiplikation einen algebraischen *Halbring*.
Wenn wir die Negation $\neg$ hinzunehmen, entsteht ein Boolescher *Verband*, oder eine Boolesche *Algebra*. Wenn wir die Addition mit XOR durchführen, so erhalten wir einen algebraischen *Ring*. Diesen Ring bezeichnen wir auch als $\mathbb{F}_2$, oder GF(2), den Galois-Körper mit zwei Elementen.

Wir sprechen im Folgenden der Einfachheit halber von „der“ Booleschen Algebra, auch wenn wir meist nur auf die Eigenschaften des Halbrings $(\mathbb{B}, \vee, \wedge)$ Bezug nehmen.

Definition 6.2 Boolesche Matrixmultiplikation

Sei $A \in \mathbb{B}^{m \times n}$ eine Boolesche Matrix mit m Zeilen und n Spalten.
Sei $B \in \mathbb{B}^{n \times l}$ eine Boolesche Matrix mit n Zeilen und l Spalten.

Das Produkt

$$C = A \odot B$$

ist eine Boolesche Matrix $C \in \mathbb{B}^{m \times l}$ mit m Zeilen und l Spalten. Ihre Elemente ergeben sich als

$$C_{ij} \quad = \quad \bigvee_{k=1}^{n} A_{ik} \wedge B_{kj}$$

für $1 \leq i \leq m,\ 1 \leq j \leq l$.

Für eine solche Matrixmultiplikation werden $m \cdot n \cdot l$ Konjunktionen und $m \cdot (n-1) \cdot l$ Disjunktionen durchgeführt. Wenn wir annehmen, dass sich m, n, l in der gleichen Größenordnung N bewegen, so haben wir $O(N^3)$ Operationen.

6.1. Ein „naiver“ Ansatz

In Listing 6.1 führen wir die Multiplikation auf naheliegende Weise mit drei verschachtelten Schleifen durch. Die logische Disjunktion (OR) erlaubt uns eine Vereinfachung: wir können die innere Schleife vorzeitig abbrechen, sobald ein Term den Wert 1 annimmt (Listing 6.2).

```
for i ∈ {1,...,m}
    for j ∈ {1,...,l}
    {
        bool sum = A_i1 ∧ B_1j
        for k ∈ {2,...,n}
            sum = sum ∨ (A_ik ∧ B_kj)
        C_ij = sum
    }
```

Listing 6.1: Pseudocode zur Matrixmultiplikation

```
for i ∈ {1,...,m}
    for j ∈ {1,...,l}
    {
        bool sum = 0
        for k ∈ {1,...,n}
            if (A_ik ∧ B_kj) {
                sum = 1
                break
            }
        C_ij = sum
    }
```

Listing 6.2: mit Schleifenabbruch

Wie oft die innere Schleife durchlaufen wird, hängt von der Verteilung der Einsen in A und B ab. Wenn wir annehmen, dass ein Element mit der Wahrscheinlichkeit $p < 1$ den Wert 1 annimmt, so ist die Wahrscheinlichkeit für den Abbruch der Schleife in jeder Iteration p^2. Die Anzahl der Schleifendurchläufe wird durch eine geometrische Verteilung beschrieben, ihr Erwartungswert ist $1/p^2$. Dieser Erwartungswert ist insbesondere nicht von N abhängig. Im Worst-Case bleibt der Gesamtaufwand jedoch bei $O(N^3)$.

Boolesche arithmetische Operationen können auf CPUs sehr effizient durchgeführt werden. Jede CPU besitzt Befehle, mit denen sich AND-Verknüpfungen bzw. OR-Verknüpfungen auf 64 Bits (oder sogar 128 Bits) in wenigen Taktzyklen berechnen lassen. Dazu ist es hilfreich, wenn die Elemente der Matrizen A und B passend im Speicher angeordnet sind. Um z. B. die oben beschriebenen Schleifen effizient zu bearbeiten, sollten die Elemente von A zeilenweise angeordnet sein, die Elemente von B hingegen spaltenweise. Ggf. muss also B erst transponiert werden, damit wir die benachbarten Bits von B effizient lesen können.

Während einer Multiplikation werden jeweils N^3 Lesezugriffe auf die Matrizen A und B durchgeführt, und N^2 Schreibzugriffe auf die Matrix C. Wir werden später sehen, dass für die Bewertung der Algorithmen nicht nur die Anzahl der arithmetischen Operationen wichtig ist, sondern auch die Anzahl (und die Verteilung) der Speicherzugriffe.

6.2. Warum nicht Coppersmith und Winograd?

Die besten bisher bekannten Algorithmen zur Matrixmultiplikation stammen von Strassen [93] und von Coppersmith und Winograd [48]. Sie erreichen eine asymptotische Komplexität von $O(N^{2.376})$. Weitere kleine Verbesserungen wurden in den letzten Jahren erzielt.

Allerdings setzen diese Algorithmen voraus, dass der Wertebereich der Matrixelemente einen algebraischen Ring bildet (z. B. die ganzen Zahlen $\mathbb{Z}$, oder die rationalen Zahlen $\mathbb{Q}$). Die Boolesche Algebra scheidet hier leider aus, sie bildet nur einen Halbring; kurz gesagt, gibt es in der Booleschen Algebra keine Subtraktion.

Um einen der Algorithmen von Coppersmith und Winograd [48] anzuwenden, müssen wir deshalb die Matrixelemente als ganze Zahlen oder Fließkommazahlen speichern. Das bringt einen

Nachteil mit sich: wir benötigen deutlich mehr Speicherplatz. Wir werden an anderer Stelle sehen, dass der Speicherbedarf ein wichtiges Kriterium zur Beurteilung von Algorithmen für die Boolesche Matrixmultiplikation darstellt (Kapitel 6.3.3).

Außerdem können wir die sehr effizienten CPU-Befehle für Boolesche Arithmetik nicht nutzen. Stattdessen müssen wir Ganzzahl- oder Fließkomma-Multiplikationen durchführen. Der Mehraufwand („konstante Faktoren“) ist erheblich.

In Kapitel 9.1 haben wir einige Vergleichsmessungen durchgeführt. Wir haben Matrixmultiplikationen mit einigen numerisch hochoptimierten Bibliotheken durchgeführt (Intel MKL [70] und andere). Einige dieser Bibliotheken nutzen die Rechenleistung einer GPU (CLBlast [84]). Die Eingaben liegen in der Größenordnung, die wir für unsere Probleme erwarten, und die auf üblicher PC-Hardware beherrschbar sind.

Zum Vergleich haben wir eine Bibliothek aufgelistet, die auf Boolesche Arithmetik spezialisiert ist (M4RI [8], wir werden sie bald genauer kennenlernen). Die Ergebnisse dieser Bibliothek sind um mehr als den Faktor 30 besser als die Ergebnisse der Ganzzahl- und Fließkomma-Algorithmen. Auch auf GPU-Hardware liefert die auf Boolesche Arithmetik spezialisierte Software bessere Ergebnisse (siehe Kapitel 8).

Diese Messungen erheben keinen Anspruch auf Allgemeingültigkeit. Wir denken aber, dass sie ausreichen, um das unterschiedliche Verhalten der Algorithmen zu demonstrieren.

Wir ziehen als *Fazit*: die Algorithmen von Coppersmith und Winograd [48] sind zwar asymptotisch besser, aber für unsere Anwendungsfälle und Problemgrößen nicht gut geeignet.

6.3. Die Methode der Vier Russen

Wir stellen nun eine Methode vor, welche den „naiven“ Algorithmus zur Booleschen Matrixmultiplikation beschleunigen kann. Den Ausgangspunkt bildet die Arbeit der „Vier Russen“ Arlazarov et al. [18].[1] Die Methode lässt sich in etwas verallgemeinerter Form auch auf andere Probleme anwenden [35, 66].

Eine Voraussetzung für die Methode der Vier Russen ist, dass die Elemente der Matrizen einen kleinen Wertebereich besitzen; in unserem Fall $\mathbb{B} = \{0, 1\}$.

Wir wählen zunächst eine *Blockgröße* $k \ll N$. Für einen *Blockstreifen* von k Zeilen aus B erstellen wir eine Lookup-Tabelle, die alle Linearkombinationen dieser k Zeilen enthält. Die Tabelle enthält somit 2^k Zeilen. Beim Füllen dieser Tabelle benötigen wir keine Konjunktionen, sondern müssen lediglich Zeilen addieren. Bei der Berechnung der Ergebniszeilen wird auf diese Lookup-Tabelle zurückgegriffen und man spart sich etliche Operationen. Wir iterieren über eine Zeile von A und bilden Blöcke aus jeweils k Einträgen (Bits). Ein Block von k Bits hat 2^k mögliche Werte; sie dienen als Index in die Lookup-Tabelle. Die Summe der Zeilen aus der Lookup-Tabelle ergibt dann die Ergebniszeile in C.

[1] Wir wissen nicht, ob es sich bei den vier Autoren wirklich um Russen handelt; der Begriff hat sich so eingebürgert.

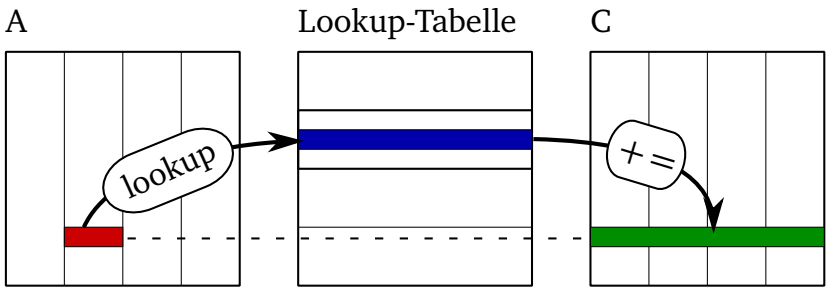


Abbildung 6.1.: Matrixmultiplikation mit der Methode der Vier Russen
Rot: ein Block aus A. Blau: die Lookup-Tabelle. Grün: Ergebniszeile in C.

```
1  for (block_offset = 0; block_offset < N; block_offset += k)
2  {
3      T = createLookupTable(block_offset)
4      for i ∈ {0, ..., N − 1}
5          for j ∈ {0, k, 2k, ..., N − k}
6          { // extrahiere k Bits aus einer Zeile von A
7              bool[k] a = A[i][j ... j + k − 1]
8              // und verwende sie als Index in die Lookup−Tabelle
9              C[i] = C[i] ∨ T[a]
10         }
11 }
```

Listing 6.3: Methode der Vier Russen: Hauptschleife

6.3.1. Komplexitätsanalyse

Erstellen der Lookup-Tabellen Wir teilen die Matrix B in N/k Blockstreifen auf. Für jeden Streifen muss eine Lookup-Tabelle erstellt werden. Eine Lookup-Tabelle besteht aus 2^k Zeilen mit jeweils N Einträgen. Mit

$$\frac{N}{k} \cdot 2^k \cdot N$$

Additionen können wir alle Lookup-Tabellen erstellen.

Berechnen der Ergebnisse Wir haben N Ergebniszeilen; für jede Zeile iterieren wir über N/k Blöcke in A und lesen eine Zeile der Länge N aus der Lookup-Tabelle. Zur Berechnung der Ergebnisse benötigen wir damit

$$N \cdot \frac{N}{k} \cdot N = \frac{N^3}{k}$$

Additionen.

Gesamtkosten Insgesamt kostet eine Matrixmultiplikation nach der Methode der Vier Russen

$$\frac{N^2 \cdot 2^k}{k} + \frac{N^3}{k}$$

Additionen. Wenn wir $k = \log N$ wählen, dann kommen wir auf Kosten von

$$O\left(\frac{N^3}{\log N}\right)$$

Es fällt auf, dass wir nur Additionen auf den Elementen ausführen, keine Multiplikationen. Bei der Booleschen Arithmetik bringt uns dies allerdings keinen wirklichen Vorteil. Eine AND-Verknüpfung ist nicht teurer als eine OR-Verknüpfung. Weiter stellen wir fest, dass wir nur zeilenweise

auf die Matrizen zugreifen. Im Gegensatz zur „naiven“ Multiplikation ist es nicht notwendig, die Matrix B zu transponieren.

Speicherplatz und Speicherzugriffe Zusätzlich zu den Matrizen A, B und C werden für die Lookup-Tabelle $N \cdot (2^k - 1)$ Bits an Speicher benötigt. Eine Zeile besteht aus N Bits, es gibt 2^k Linearkombinationen. Die Zeile an Stelle 0 kann man einsparen.

Jeder Eintrag in der Matrix A wird N/k mal gelesen, insgesamt haben wir N^3/k Lesezugriffe auf A und ebensoviele Schreibzugriffe auf C. Die Matrix B wird nur einmal gelesen und zeilenweise in die Lookup-Tabellen kopiert, also haben wir N^2 Lesezugriffe auf B.

6.3.2. Die Bibliothek M4RI

Die Software-Bibliothek M4RI [8, 9] implementiert die Matrixmultiplikation mit Hilfe der Vier-Russen-Methode. M4RI ist ein Open-Source-Projekt, das u. a. in den Mathematik-Anwendungen SageMath und PolyBoRi zum Einsatz kommt.

Wir verwenden diese Bibliothek in unserer Implementierung, mussten sie aber zunächst auf unsere Bedürfnisse anpassen. Ursprünglich war nur die Matrixmultiplikation auf $\mathbb{F}_2$ (dem Galois-Körper mit zwei Elementen, siehe Definition 6.1) implementiert. Glücklicherweise sind die Unterschiede nicht sehr groß. In $\mathbb{F}_2$ wird die Addition (und Subtraktion) durch eine XOR-Verknüpfung realisiert.

Unser Beitrag Um den Code von M4RI für unsere Zwecke nutzbar zu machen, ersetzten wir alle XOR-Verknüpfungen durch OR-Verknüpfungen. Außerdem musste auch das Befüllen der Lookup-Tabellen angepasst werden. Die ursprüngliche Version verwendet beim Befüllen der Tabellen eine Schleife über Gray-Codes, die in dieser Weise nur auf $\mathbb{F}_2$ funktioniert. Wir ersetzen sie durch eine etwas andere Konstruktion, die aber ähnlich effizient arbeitet (siehe Listing 6.4). Unsere Code-Erweiterungen zur Matrixmultiplikation auf dem Booleschen Halbring sind in das Open-Source-Projekt M4RI zurückgeflossen.

```
function createLookupTable(block_offset)
{ // Berechne alle 2^k Linearkombinationen von k Zeilen aus B
    bool T[2^k][N]
    // Linearkombinationen mit genau einem Bit (eine Zeile aus B kopieren)
    for j ∈ {0,...,k−1}
        T[2^j] = B[block_offset+j]
    // Linearkombinationen mit zwei und mehr Bits
    for h ∈ {2,...,k}
        for j | popcount(j) = h // alle Werte mit genau h Bits
            T[j] = T[lsb(j)] ∨ T[j − lsb(j)]
        // popcount(i) liefert die Anzahl gesetzter Bits (das Hamming−Gewicht)
        // lsb(i) liefert die Position des kleinsten gesetzten Bits einer Zahl
    return T
}
```

Listing 6.4: Methode der Vier Russen: Füllen einer Lookup-Tabelle

Dreiecksmatrizen Für Dreiecksmatrizen, wie wir sie in Kapitel 5.6.2 verwenden, können wir die Methode der Vier Russen etwas optimieren. Die Lookup-Tabellen passen sich den Zeilenlängen von A und B an. Leere Bereiche werden nicht in die Lookup-Tabelle übernommen. Auch hierfür haben wir den Source-Code von M4RI erweitert.

6.3.3. Speicherzugriffe bei der Booleschen Matrixmultiplikation

Die Methode der Vier Russen reduziert die Anzahl arithmetischer Operationen bei der Booleschen Matrixmultiplikation. Bei der Beurteilung des Verfahrens dürfen wir aber nicht nur die Anzahl der arithmetischen Operationen beachten; ähnlich wichtig (mitunter wichtiger) ist die Anzahl der Speicherzugriffe.

Probleme, deren Laufzeit vorwiegend durch arithmetische Operationen bestimmt wird, bezeichnen wir als **CPU-bound**. Probleme, deren Laufzeit durch Speicherzugriffe dominiert wird, nennen wir **Memory-bound**, oder Latency-bound. Während die Multiplikation von Fließkomma-Matrizen üblicherweise CPU-bound ist, ist die Boolesche Matrixmultiplikation in hohem Maße Memory-bound.

Tatsächlich sind bei modernen CPU-Architekturen Zugriffe auf den Hauptspeicher sehr viel teurer als arithmetische Operationen. CPUs werden daher mit schnellen Cache-Speichern ausgestattet, deren Nutzung aber sorgfältig geplant sein will.

Der schnellste Cache-Speicher ist der Level-1 (L1) Cache. Er hat üblicherweise eine Größe von 32KB. Der nächstgrößere Cache L2 hat eine übliche Größe von 256KB bis über 1MB. Der nächstgrößere Cache L3 hat eine übliche Größe von einigen MB. Jeder Kern eines Multi-Core-Prozessors hat in der Regel eigene L1- und L2-Caches, während der L3-Cache von allen Kernen gemeinsam genutzt wird.

M4RI [8] passt die Arbeitsmenge, d. h. die Menge aktuell bearbeiteter Daten, an die Größe der CPU-Caches an. Die Lookup-Tabelle wird so ausgelegt, dass sie komplett im L2-Cache Platz findet. Jeweils ein Blockstreifen von B und die zugehörige Lookup-Tabelle kann dann von einem Prozessorkern bearbeitet werden. Die Größe der Lookup-Tabellen kann über den Parameter k gesteuert werden. Um die Lookup-Tabelle vollständig im L2-Cache unterzubringen, müssen wir ggf. Abstriche bei der Wahl von k machen. Wir müssen in Kauf nehmen, dass k nicht den optimalen Wert $k = \log N$ annehmen kann. In vielen Fällen ist der L2-Cache schon bei $k = 8$ belegt. Der theoretisch erreichbare Gewinn durch die Vier-Russen-Methode sinkt durch diese Einschränkung.

Die Iteration über die Zeilen von A und C ist so ausgelegt, dass eine Arbeitsmenge im L3-Cache Platz findet. Idealerweise passen die Matrizen A und C vollständig in den L3-Cache. Sind sie größer, so werden sie in horizontale Bänder unterteilt und jeweils ein Band bearbeitet. Das führt ggf. dazu, dass die Lookup-Tabellen mehrfach berechnet werden müssen, wodurch wieder die theoretisch erreichbare Leistung sinkt.

Es ist wichtig zu beachten, dass die L2- und L3-Caches durch eine einzige Matrixmultiplikation bereits größtenteils belegt sind. Wird eine weitere Matrixmultiplikation in einem anderen Thread durchgeführt, dann machen sich die Threads gegenseitig den L3-Cache streitig. Die Leistung wird dadurch sehr stark beeinträchtigt. Wir werden bei der Parallelisierung des Algorithmus für einfache Polygone in Kapitel 7.6 auf diese wichtige Beobachtung zurückkommen.

Die hier gemachten Bemerkungen über Speicher-Architekturen gelten übrigens in ähnlicher Weise auch für GPUs in Kapitel 8. Dort werden wir ebenfalls feststellen, dass die Planung der Speicherzugriffe mindestens so viel Aufmerksamkeit erfordert, wie die arithmetischen Operationen.

7

Parallelisierung für Multi-Core-Rechner

In diesem Kapitel betrachten wir Ansätze, um die Algorithmen für Polygonzüge und für einfache Polygone durch parallele Programmierung zu beschleunigen. Wir konzentrieren uns auf weit verbreitete Multi-Core-Architekturen. Wir nehmen an, dass ein Prozessor mehrere CPU-Kerne (2, 4, 8 oder mehr) besitzt, die unabhängig voneinander Berechnungen durchführen. Alle Kerne greifen auf einen gemeinsamem Speicher zu (Shared Memory), wobei die Speicherzugriffe durch Cache-Speicher gepuffert werden (siehe auch Kapitel 6.3.3).

Für Anwendungen mit sehr großen Eingabedaten mögen auch andere parallele oder verteilte Architekturen in Frage kommen; diese sind aber nicht Gegenstand unserer Arbeit.

Die bekanntesten und am weitesten ausgereiften Werkzeuge zur parallelen Programmierung in C++ sind die Frameworks **OpenMP** und **Intel Threading Building Blocks** (TBB [71]). Wir verwenden in unserer Implementierung Letzteres, da TBB leicht integrierbar ist und eine gut verständliche API besitzt. Alle im Folgenden beschriebenen Ansätze lassen sich aber ebenso mit OpenMP umsetzen.

Wir identifizieren hier diejenigen Teile der Algorithmen, die für eine parallele Ausführung geeignet sind und die eine merkliche Beschleunigung versprechen.

7.1. Berechnung des Free-Space

Der Free-Space besteht aus $2 \cdot n \cdot m$ Intervallen, die unabhängig voneinander berechnet werden können. Als Eingabedaten dienen die Eckpunkte der Polygonzüge P und Q.

Wenn die Berechnung der Intervalle in zwei verschachtelten Schleifen stattfindet, so kann die äußere Schleife auf die CPU-Kerne verteilt werden. Jeder Kern arbeitet auf einem eigenen Abschnitt der Schleife. In der TBB-API kann dies z. B. wie in Listing 7.1 ausgedrückt werden. Es finden keine konkurrierenden Schreibzugriffe statt.

TBB wurde so konfiguriert, dass für jeden Prozessorkern genau ein Thread angelegt wird. Die Verwaltung der Threads und das Verteilen der einzelnen Arbeitspakete wird von der TBB-Laufzeit-Umgebung weitgehend transparent durchgeführt. Ebenso sorgt die TBB-Laufzeit-Umgebung dafür, dass der Hauptprozess auf die Ergebnisse der einzelnen Threads wartet.

Die Parallelisierung einer Schleife verursacht auch zusätzliche Kosten, da die Arbeit auf mehrere Threads verteilt und auf die Beendigung der Arbeit gewartet werden muss. Bei mehrfach verschachtelten Schleifen sollte daher stets darauf geachtet werden, die *äußerste* Schleife zu parallelisieren – wenn die Abhängigkeiten der Datenzugriffe dies erlauben.

```
tbb::static_partitioner sp;
tbb::parallel_for(0, n, [&] (int i)
{
    for(int j=0; j < m; ++j)
    {
        L_ij = calculateFreeSpaceSegment(P_i, Q_j,Q_j+1, ε);
        B_ij = calculateFreeSpaceSegment(Q_j, P_i,P_i+1, ε);
    }
}, sp);
/* static_partitioner teilt eine Schleife in gleich große Bereiche und
verteilt sie auf die Prozessorkerne. Andere Schemata sind bei Bedarf möglich. */
```

Listing 7.1: Beispiel: parallele Berechnung der Free-Space-Intervalle

Überdies sollte darauf geachtet werden, dass lesende Speicherzugriffe möglichst in benachbarten Bereichen stattfinden, um die Cache-Speicher möglichst gut auszunutzen. Im Beispiel Listing 7.1 werden die Ergebnisse L_{ij} und B_{ij} spaltenweise abgelegt, sodass die innere Schleife auf einem zusammenhängenden Speicherbereich arbeiten kann.

Wie man sieht, lässt sich die Berechnung des Free-Space recht leicht parallelisieren. Allerdings macht diese nur einen kleinen Teil der Gesamtlaufzeit der Algorithmen von Alt und Godau [13] und von Buchin et al. [27] aus. Der Gewinn ist für die Gesamtlaufzeit also nicht sehr groß.

7.2. Berechnung der Erreichbarkeits-Strukturen

Im Algorithmus von Alt und Godau [13] werden Erreichbarkeits-Strukturen nach einem Divide-and-Conquer-Verfahren rekursiv berechnet (Kapitel 2.7). In jedem Teilschritt werden zwei Erreichbarkeits-Strukturen miteinander verbunden. Die Teilschritte arbeiten auf voneinander unabhängigen Daten und lassen sich daher ebenfalls auf mehrere Prozessorkerne verteilen. Allerdings können die Arbeitspakete für die einzelnen Threads nicht so einfach geschnürt werden, wie dies bei der Free-Space-Berechnung der Fall war.

Wir bilden die Schritte des Divide-and-Conquer-Verfahrens in einem Aufruf-Graphen ab (Abb. 7.1). Solche Aufruf-Graphen lassen sich mit TBB modellieren. Jeder Arbeitsschritt (MERGE zweier Erreichbarkeits-Strukturen) bildet einen Knoten – einen *Task* – im Graphen. Die Kanten geben die Abhängigkeiten der Tasks wieder.

Wir müssen die Modellierung der Tasks nicht bis zu den Blättern des Aufruf-Graphen durchführen, sondern nur für die oberen Ebenen. Sobald wir ausreichend viele Tasks haben, um die Prozessorkerne vollständig auszulasten, können die darunterliegenden Teilbäume sequentiell abgearbeitet werden. Eine weitere Aufteilung in parallele Tasks ist nicht hilfreich, wenn schon alle Prozessorkerne beschäftigt sind.

Die Verteilung der Tasks auf die verfügbaren Prozessorkerne und die Verwaltung der Abhängigkeiten wird wiederum von der TBB-Laufzeit-Umgebung durchgeführt.

Im oberen Teil des Aufruf-Graphen, nahe bei der Wurzel, stellen wir fest, dass nur noch wenige Tasks parallel bearbeitet werden können. Sobald wir uns der Wurzel des Aufruf-Graphen nähern sinkt daher die Auslastung der Prozessorkerne.

Alternativen? Um die Auslastung der Prozessorkerne in diesen letzten Schritten zu verbessern, könnten wir eine Parallelisierung auf der nächst tieferen Ebene anstreben. Wir könnten versuchen,

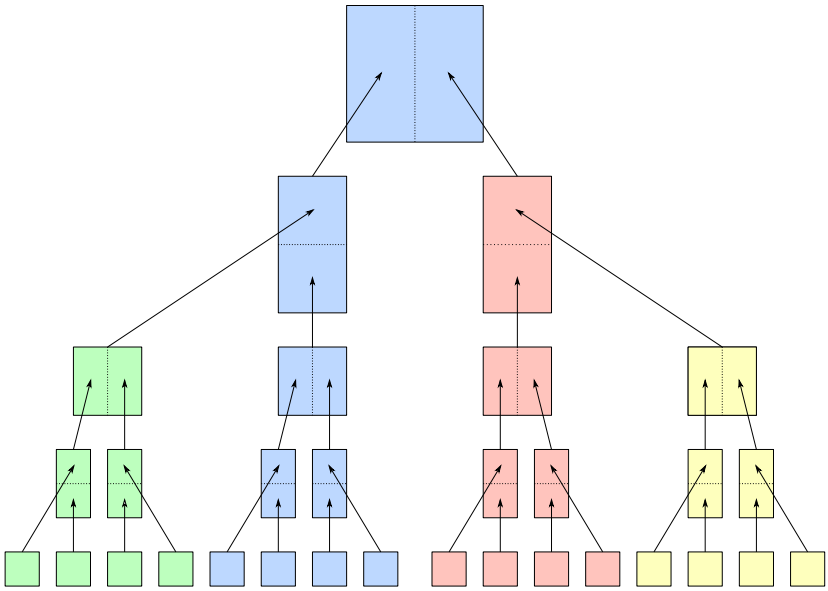


Abbildung 7.1.: Aufruf-Graph zur Berechnung einer Erreichbarkeits-Struktur

Die inneren Knoten entsprechen MERGE-Operationen.
Verteilung auf vier Prozessoren = vier Farben.

einen einzelnen MERGE-Task auf mehrere Kerne zu verteilen. Dies gestaltet sich aber sehr schwierig, da eine Erreichbarkeits-Struktur aus vielen voneinander abhängigen Datenstrukturen besteht: doppelt verkettete Listen mit Zeigern zwischen den Listen, siehe Kapitel 5.2.

Wenn nun mehrere parallele Threads dieselbe Datenstruktur lesen *und verändern*, müssen wir äußerst sorgfältig vorgehen, um konkurrierende Schreibzugriffe zu regeln. Vermutlich würde der zusätzliche Aufwand zur Synchronisierung von Schreibzugriffen den Vorteil der Parallelisierung zunichte machen. Wir haben diese Idee daher nicht weiterverfolgt. Wir halten die implementierte Lösung für ausreichend, da sie bereits den weitaus größten Teil des Aufruf-Graphen parallel verarbeiten kann.

7.3. Berechnung der Erreichbarkeits-Graphen

Im ersten Schritt des Algorithmus von Buchin et al. [27] wird eine Liste von Erreichbarkeits-Graphen berechnet (Kapitel 3.9). Die Daten dieser Liste sind voneinander unabhängig und deshalb können die Berechnungen leicht parallel durchgeführt werden. Ähnlich wie in Kapitel 7.1 wird eine Schleife auf die verfügbaren Prozessorkerne verteilt.

Die Ergebnisse werden in einer Memoisierungs-Struktur gespeichert. Um konkurrierende Schreibzugriffe auf diese Datenstruktur aus mehreren Threads durchzuführen, werden die Zugriffe durch Semaphore gesichert. TBB bietet bereits eine entsprechende Datenstruktur an: `tbb::concurrent_unordered_map` ist eine Hash-Tabelle, die konkurrierende Schreibzugriffe ermöglicht. Der zusätzliche Aufwand ist vertretbar, da im Verlauf des Algorithmus nur vergleichsweise wenige schreibende Zugriffe stattfinden, nämlich $O(k)$ (siehe Kapitel 3.13.1).

7.4. Berechnung der gültigen Platzierungen

Im nächsten Schritt des Algorithmus von Buchin et al. [27] werden gültige Platzierungen für alle Diagonalen der konvexen Zerlegung berechnet. Die berechneten Ergebnisse sind wiederum voneinander unabhängig, somit kann auch hier die Schleife über die Diagonalen auf die Prozessorkerne verteilt werden.

Der Algorithmus von Guibas et al. [64] zur Berechnung von Shortest-Path-Trees arbeitet auf einer Triangulierung der Kurve Q. Hier müssen wir beachten, dass die Datenstruktur der Triangulierung im Laufe des Algorithmus verändert wird. Deshalb erhält jeder parallele Thread zu Anfang eine identische Kopie der Triangulierungs-Datenstruktur. TBB bietet hierfür Unterstützung in Form von sogenannten *Thread-lokalen*-Datenstrukturen an. Wir müssen lediglich das Original berechnen und eine Funktion zur Verfügung stellen, welche identische Kopien erstellt (einen *Copy Constructor*). Ebenso werden die Trichter-Datenstrukturen aus Kapitel 5.5 als Thread-lokale Kopien vorgehalten.

7.5. Kritische Werte

Die kritischen Werte für das Optimierungsproblem können ebenfalls parallel berechnet werden. Ähnlich wie in Kapitel 7.1 bietet sich dafür wieder eine verteilte Schleife an. Allerdings müssen die Ergebnisse am Ende in einer gemeinsamen sortierten Liste vorliegen. Jeder Thread arbeitet zunächst auf einer Thread-lokalen Kopie und erstellt einen Teil der Ergebnisse. Danach müssen die berechneten Listen zusammengeführt und sortiert werden. Beim Zusammenführen ist uns wieder die TBB-Laufzeit-Software behilflich.

Das abschließende Sortieren der kritischen Werte erfolgt mit einer verteilten Sortierfunktion (`tbb::parallel_sort`), die wiederum alle Prozessorkerne nutzt. Die Kosten für das Berechnen der kritischen Werte werden – genau wie im sequentiellen Fall – von der abschließenden Sortierung dominiert. Mit p Prozessorkernen kann die parallele Sortierfunktion N Einträge in

$$O\left(\frac{N \log N}{p}\right)$$

Schritten sortieren.

7.6. Parallele Matrixmultiplikation

In Listing 6.3 auf Seite 92 haben wir gesehen, dass die Hauptfunktion der Vier-Russen-Multiplikation aus drei verschachtelten Schleifen besteht:

- Die äußere Schleife in Zeile **1** iteriert über die Blöcke von B.
  In jeder Iteration wird jeweils eine Lookup-Tabelle berechnet.
- Die mittlere Schleife in Zeile **4** iteriert über alle Zeilen von A und C.
- Die innere Schleife in Zeile **5** iteriert über die Spalten von A.

Die Frage ist nun, welche dieser Schleifen sich am besten zur Parallelisierung eignet?

Es wäre naheliegend, die äußere Schleife zu verteilen, aber dann hätten wir konkurrierende Schreibzugriffe auf die Ergebnismatrix C. Wir müssten einen erhöhten Aufwand treiben, um diese Zugriffe zu synchronisieren.

Wir entscheiden uns stattdessen, die mittlere Schleife in Zeile **4** zu parallelisieren. Jeder Prozessorkern arbeitet auf einem separaten Bereich von Zeilen, es finden keine konkurrierenden Schreibzugriffe statt.

Damit auch die Berechnung der Lookup-Tabellen in Zeile **3** von der Parallelisierung profitieren kann, passen wir den Vorgang etwas an. Wir erstellen in jedem Durchlauf der äußeren Schleife nicht nur eine Lookup-Tabelle, sondern *acht* Lookup-Tabellen. Konzeptionell ändert dies aber nichts am Algorithmus, wie er in Listing 6.3 beschrieben wurde.

In Kapitel 6.3.3 haben wir die Speichernutzung von M4RI erläutert. Jeweils ein Satz von Lookup-Tabellen füllt den L2-Cache. Der L3-Cache wird von den Matrizen A und C belegt. Damit ist diese Implementierung gut auf die parallele Ausführung abgestimmt: jeder Prozessorkern arbeitet auf separaten Lookup-Tabellen, während sich alle Kerne die Ein- und Ausgabedaten teilen.

Wichtig ist aber, dass nur eine Matrixmultiplikation durchgeführt wird. Wenn zwei oder mehr Multiplikationen gleichzeitig stattfinden, geht die sorgfältig kalkulierte Belegung der Cache-Speicher nicht mehr auf. In diesem Fall müssen Daten wiederholt in den Cache geladen werden („Cache Thrashing“), dadurch sinkt die Leistung.

Aus dem gleichen Grund verzichten wir auf das bei Intel-Prozessoren mögliche *Hyperthreading*. Mit aktiviertem Hyperthreading kann ein CPU-Kern zwei Hardware-Threads gleichzeitig verarbeiten. Für CPU-lastige Probleme ist dies eine sinnvolle Option, nicht aber für unser Problem. Die beiden Threads teilen sich nämlich den gleichen Cache-Speicher – das wollen wir vermeiden. Wir empfehlen, pro CPU-Kern *genau einen* Thread einzusetzen.

7.7. Warum wir die Hauptschleife nicht parallelisieren

Die Hauptschleife des Algorithmus von Buchin et al. [27] berechnet rekursiv Combined Reachability Graphs. In Kapitel 5.9 haben wir einen Ansatz vorgestellt, mit dem wir die Aufgaben in sortierter Reihenfolge durchführen können. Wir erstellen dazu einen Aufruf-Graphen und arbeiten die Aufträge in Reverse-Level-Order ab.

Neben der besseren Speicherplatznutzung bietet dies auch die Möglichkeit, die Aufträge einer Ebene parallel abzuarbeiten. Nun bestehen diese Aufträge aber im Wesentlichen aus Matrixmultiplikationen. Wir haben im vorhergehenden Kapitel 7.6 gesehen, dass sich eine Matrixmultiplikation zwar parallelisieren lässt, aber nach Möglichkeit nicht mehrere Multiplikationen gleichzeitig ausgeführt werden sollten. Damit ist der Ansatz aus Kapitel 5.9 zum Scheitern verurteilt. Unsere Tests haben gezeigt, dass die Leistung der Matrixmultiplikationen tatsächlich sehr stark sinkt, wenn wir mehrere Multiplikationen gleichzeitig durchführen. In Kapitel 8.7 werden wir aber auf die Idee zurückkommen.

7.8. Welche Schritte haben wir nicht parallelisiert?

Nicht alle Schritte der Algorithmen eignen sich zur Parallelisierung. Die Binärsuche des Optimierungsalgorithmus ist dazu ungeeignet, denn jeder Schritt der Binärsuche ist vom vorhergehenden abhängig. Das Gleiche gilt für die Intervallschachtelung, die wir in Kapitel 5.7.2 beschrieben haben.

Bei den Algorithmen zur konvexen Zerlegung von Polygonen (Kapitel 5.3) greifen wir auf die sequentiellen Implementierungen von CGAL zurück. Ob sich diese Algorithmen zur Parallelisierung eignen, wäre eine interessante Frage, die wir aber an dieser Stelle nicht beantworten können und wollen.

k-Fréchet-Abstand Auch die Algorithmen für den k-Fréchet-Abstand basieren auf einem Free-Space-Diagramm. Dies wird parallel berechnet. Die zusammenhängenden Komponenten werden jedoch sequentiell berechnet. Der Aufwand fällt mit $O(mn \cdot \alpha(mn))$ nicht sehr ins Gewicht.

Die Greedy-Suche für den k-Fréchet-Abstand lässt sich nicht parallelisieren. Die Brute-Force-Suche könnte parallelisiert werden; aufgrund des exponentiellen Wachstum verspricht die Parallelisierung aber wenig Vorteile. Wir haben sie deshalb nicht umgesetzt.

7.9. Erwartungen und Ergebnisse

Wir haben gesehen, dass die meisten Möglichkeiten zur Parallelisierung in der Aufteilung von Schleifen liegen. Wenn wir p Prozessorkerne haben, so können wir bei gleichmäßiger Auslastung eine Beschleunigung um den Faktor p erwarten. Allerdings benötigt die Verwaltung und Synchronisierung der Threads auch Zeit. Außerdem können wir nicht in allen Fällen eine gleichmäßige Auslastung sicherstellen (Kapitel 7.2). Drittens lassen sich gewisse Teile der Algorithmen nicht parallelisieren (Kapitel 7.8).

In Kapitel 9.4 haben wir einige experimentelle Ergebnisse zusammengestellt. Beim Entscheidungsalgorithmus für einfache Polygone erreichen wir Beschleunigungen, die durchaus in der Größenordnung der theoretisch erreichbaren Werte liegen.

Bei den Optimierungsalgorithmen und beim Entscheidungsalgorithmus für Polygonzüge liegen die Beschleunigungen aus den oben angeführten Gründen etwas niedriger.

8

GPGPU-Ansätze

Ein weitere Möglichkeit, die Laufzeit unserer Algorithmen zu verbessern, ist der Einsatz von GPU-Hardware. GPUs (Graphics Processing Units) wurden ursprünglich zur Bildverarbeitung und zur Unterstützung von graphischen Berechnungen gebaut, sie eignen sich aber auch für viele rechenintensive Aufgaben (*General Purpose* GPUs, GPGPUs). Nach einer kurzen Einführung in die Hardware-Architektur stellen wir die wichtigsten Konzepte der GPU-Programmierung vor. Eine kurze, aber informative Einführung in die Programmierung von GPUs findet sich z. B. in der Dokumentation von Nvidia [85, 86].

Danach stellen wir unsere Implementierung der Booleschen Matrixmultiplikation für GPU-Hardware vor. Einige experimentelle Ergebnisse folgen in Kapitel 9.

Weise [94] und Gudmundsson und Valladares [62] haben in ihren Arbeiten die Berechnung des Free-Space erfolgreich auf GPU-Hardware umgesetzt. Wir werden uns hingegen auf die Boolesche Matrixmultiplikation konzentrieren, da sie den größten Anteil der Laufzeit des Algorithmus für einfache Polygone ausmacht.

8.1. GPU-Architektur

Der Aufbau einer GPU unterscheidet sich stark von CPU-Architekturen. GPUs bestehen aus einer großen Zahl von Recheneinheiten (**Prozessoren**), die aber nicht unabhängig voneinander arbeiten. Eine Gruppe von Prozessoren bildet einen **Multiprozessor**. Alle Prozessoren des Multiprozessors führen die gleichen Instruktionen auf unterschiedlichen Daten aus.

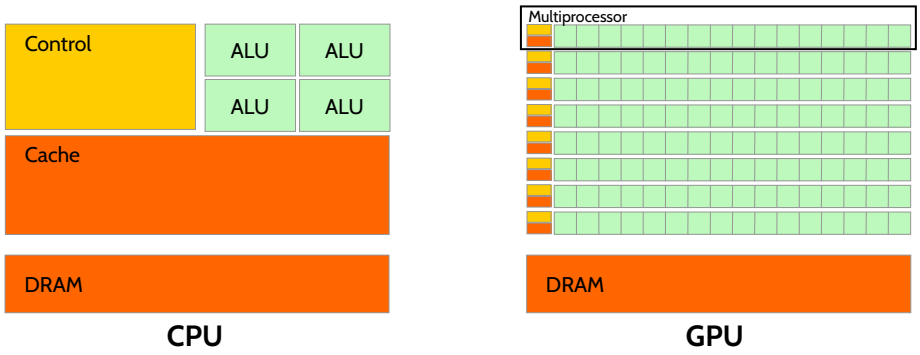


Abbildung 8.1.: Blockschaltbilder
(Zeichnung: Nvidia Corp. [85])

Wir sprechen deshalb von **datenparalleler** Ausführung (Single Instruction Multiple Data, SIMD). Ein GPU-**Device** besteht wiederum aus mehreren Multiprozessoren.

Die technische Entwicklung der GPU-Hardware ist nach wie vor rasant. Aktuelle hochwertige GPUs enthalten nicht selten viele tausend Prozessoren. Im Vergleich dazu besitzen CPUs selten mehr als 32 Prozessorkerne.

8.2. Speicher-Architektur

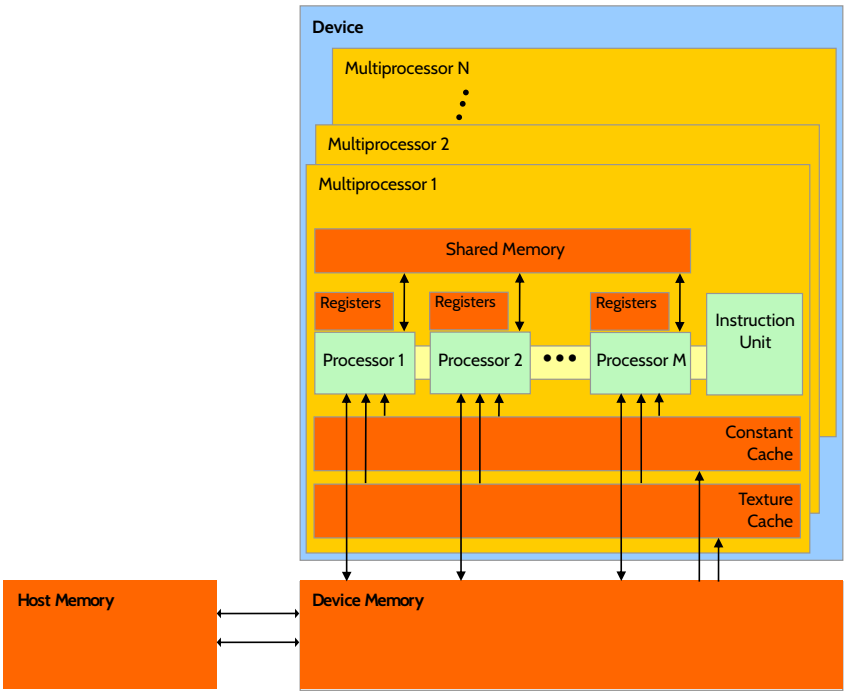


Abbildung 8.2.: GPU-Speicherhierarchie
(Zeichnung: Nvidia Corp. [85])

Auch die Speicher-Architektur unterscheidet sich erheblich. Während CPUs mit großen Cache-Speichern ausgestattet werden, besitzen GPUs nur sehr kleine Cache-Speicher, die jeweils bei einem Multiprozessor angesiedelt sind. Mit ca. 32KB bis 128KB sind sie deutlich kleiner als die Cache-Speicher einer CPU.

Die Eingabedaten müssen zunächst aus dem Hauptspeicher der CPU (dem **Host**-System) in den Hauptspeicher der GPU kopiert werden (siehe auch Abb. 8.2).

Ein Multiprozessor besitzt einen Speicherbereich (Shared Memory), der von allen Prozessoren dieses Multiprozessors gemeinsam genutzt werden kann. Zugriffe auf diesen gemeinsamen Speicher sind deutlich schneller als Zugriffe auf den Hauptspeicher.

Wir werden versuchen, so oft wie möglich Daten im Shared Memory unterzubringen, aber dessen Größe ist beschränkt. Prozessoren können über das Shared Memory Daten austauschen, nicht aber über die Grenzen von Multiprozessoren hinweg. Jeder Prozessor besitzt darüber hinaus noch einen Satz von Registern. Der Zugriff auf Prozessor-Register erfolgt sehr schnell in einem Taktzyklus.

Nicht alle Algorithmen lassen sich sinnvoll auf einer GPU umsetzen. Wichtig ist, dass der Algorithmus einen hohen Grad an Datenparallelität erlaubt. Algorithmen mit komplexen Datenstrukturen oder rekursive Algorithmen sind üblicherweise nicht für die Umsetzung auf einer GPU geeignet.

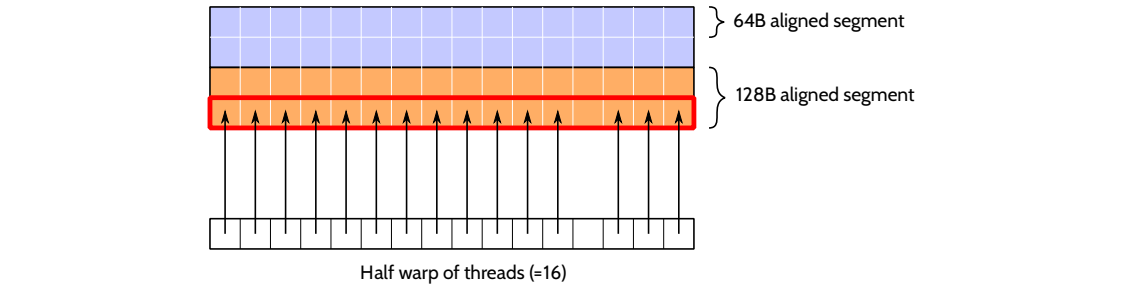
8.2.1. Speicherzugriffe sorgfältig planen

Wir haben vier Arten von Speicher kennengelernt:

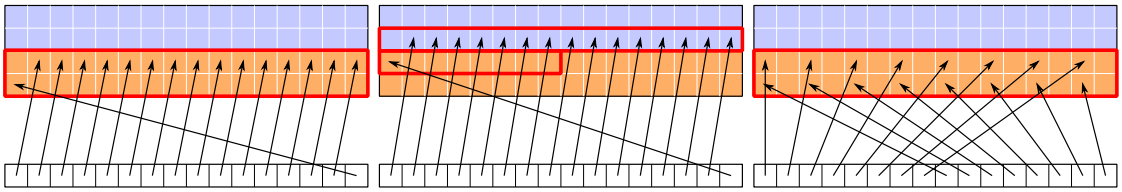
1. Hauptspeicher auf dem Host-System (CPU)
2. Hauptspeicher auf dem GPU-Device (globaler Speicher)
3. Shared Memory des Multiprozessors
4. Prozessor-Register

Host-Speicher und GPU-Speicher Die Übertragung von Daten vom Host-Hauptspeicher zur GPU (und zurück) wird über OpenCL-Aufträge durchgeführt. Die Übertragung erfolgt meist über Bus-Systeme wie z. B. PCI, und ist sehr viel langsamer als andere Zugriffsmethoden. Es sollten nicht mehr Daten als nötig zwischen diesen Speicherbereichen kopiert werden.

Globaler Speicher Zugriffe auf den globalen Hauptspeicher der GPU sind vergleichsweise langsam (mehrere hundert Taktzyklen). Pro Anfrage werden zudem üblicherweise 16 benachbarte Wörter aus dem Hauptspeicher gelesen. Wenn nun mehrere Threads gleichzeitig auf den Hauptspeicher zugreifen, ist es sehr wichtig, dass sich die angefragten Daten in benachbarten Speicherzellen befinden. Man sagt, die Zugriffe solten „vereinigt“ (*coalesced*) sein.



(a) günstiges Zugriffsmuster



(b) ungünstige Zugriffsmuster

Abbildung 8.3.: Globaler Speicher: vereinigte Zugriffe (coalesced access)
(Zeichnung nach Nvidia Corp. [85])

In Abb. 8.3 haben wir einige günstige und weniger günstige Zugriffsmuster dargestellt. Beim Entwurf eines GPU-Programms (eines *Kernels*, siehe Kapitel 8.3.1) ist die Planung von vereinigten Speicherzugriffen ein sehr wichtiger Gesichtspunkt.

Shared Memory Bei gängiger GPU-Hardware ist jeder Multiprozessor mit einem gemeinsamen Speicher in der Größe von 32KB bis etwa 128KB ausgestattet. Zugriffe auf das Shared Memory sind deutlich schneller (etwa 1-3 Taktzyklen). Pro Zugriff wird jeweils nur ein Wort übertragen, „vereinigte“ Zugriffe sind hier nicht zu beachten.

Das Shared Memory ist jedoch in Speicher-**Bänke** aufgeteilt. Gleichzeitige Zugriffe auf die gleiche Speicher-Bank (*Bank-Konflikte*) werden durch die Hardware serialisiert und sollten möglichst vermieden werden. Hier ist es wichtig, die Abstände (*Strides*) zwischen gleichzeitig angeforderten

Speicherzellen zu beachten. Die Strides sollten keine Vielfachen von 16 sein, da sonst gehäuft Zugriffe auf die gleichen Speicher-Bänke stattfinden.

Über das Shared Memory können die Prozessoren eines Multiprozessors Daten austauschen. Konkurrierende Lese- und Schreibzugriffe können – falls nötig – mit Synchronisationspunkten (*Barrieren*) abgesichert werden. Der Datenaustausch über die Grenzen von Multiprozessoren hinweg kann aber nur über den globalen Speicher stattfinden.

Eine detailliertere Einführung in die Schwierigkeiten von Speicherzugriffen findet sich z. B. im OpenCL *Best Practices Guide* [85].

Register Jeder Prozessor hat Zugriff auf einen Satz von Registern, die – ähnlich wie bei einer CPU– für lokale Variablen verwendet werden. Der Zugriff auf Register ist schnell, aber ihre Anzahl ist begrenzt. Üblicherweise ist jeder Multiprozessor mit 32KB an Registerspeicher ausgestattet, den sich die einzelnen Prozessoren teilen müssen. Stehen für einen auszuführenden Kernel nicht ausreichend viele Register zur Verfügung, so wird er vom Scheduler der GPU zu einem späteren Zeitpunkt eingeplant, was u. U. die Gesamtauslastung des Systems nachteilig beeinflusst.

8.2.2. Texturspeicher

Ein Teil des GPU-Hauptspeichers kann für Bilddaten (Texturen) verwendet werden. In diesem Speicherbereich können aber auch andere Daten abgelegt werden, vorzugsweise, wenn sie sich in ein zweidimensionales Layout einpassen lassen. Z. B. können wir sehr gut unsere Matrixdaten in den Texturspeicher legen. Jeder Multiprozessor verfügt über einen Cache für Texturdaten, sodass wiederholte Zugriffe günstiger sind.

Wir haben unseren Algorithmus zur Booleschen Matrixmultiplikation mit beiden Speicherarten (Hauptspeicher und Texturspeicher) getestet, dabei aber empirisch keine wesentlichen Unterschiede festgestellt. Vermutlich sind die Zugriffsmuster nicht günstig, um vom Cache des Texturspeichers profitieren zu können.

8.3. Das OpenCL-Framework

Frameworks wie OpenCL [75] oder CUDA unterstützen uns bei der Entwicklung und Ausführung von GPU-Programmen. Sie ermöglichen das Erstellen von ausführbaren Programme für GPUs. Sie stellen APIs bereit, um Programme und Daten in den Speicher der GPU zu laden. Daten und Algorithmen werden auf einer höheren Abstraktionsebene entworfen. Die Abbildung auf die konkrete Hardware erfolgt zur Laufzeit durch die Software von OpenCL oder CUDA.

Wir haben uns bei unserer Implementierung für das OpenCL-Framework entschieden, da es ein breites Spektrum an Hardware unterstützt. Das CUDA-Framework ist hingegen nur für GPUs des Herstellers Nvidia verfügbar. Konzeptionell sind beide Frameworks sehr ähnlich, verwenden aber etwas unterschiedliche Begriffe. Im vorliegenden Text verwenden wir die Terminologie von CUDA, da sie besser eingeführt und auch prägnanter ist. Wir haben die wichtigsten Begriffe in Tabelle 8.1 kurz gegenübergestellt. Dies soll den Lesern, die mit OpenCL vertraut sind, die Orientierung etwas erleichtern.

OpenCL eignet sich übrigens nicht nur zur Programmierung von GPUs, sondern kann auch andere parallele Architekturen abbilden (z. B. Intel Xeon Phi Prozessoren). Wir gehen hierauf jedoch nicht weiter ein. Die Programmierung von Multi-Core-Prozessoren haben wir in Kapitel 7 bereits mit anderen Werkzeugen behandelt.

OpenCL	CUDA
Compute Unit	Stream Multiprozessor, SM
Processing Element	Prozessor
Work Item	Thread
Work Group	Block
Wave Front	Warp
Local Memory	Shared Memory
Private Memory	Local Memory

Tabelle 8.1.: Kleine Übersetzungshilfe: OpenCL nach CUDA

8.3.1. Kernels

Ein **Kernel** ist ein auf der GPU ausführbares Programm. Das Programm wird in einer C-ähnlichen Sprache geschrieben. Es gibt einige Einschränkungen, die sich aus der GPU-Architektur ergeben, z. B. sind keine rekursiven Funktionen erlaubt. Außerdem lassen sich die Speicherklassen von Objekten genauer unterscheiden (globaler Speicher, Shared Memory, etc.)

Verzweigungen und Schleifen sind innerhalb eines Kernel-Programms möglich. Es sollte aber darauf geachtet werden, dass die Verzweigungen nicht von den Eingabedaten abhängen. Die Prozessoren innerhalb eines Multiprozessor führen stets identische Instruktionen aus. Wenn die Prozessoren unterschiedliche Verzweigungen nehmen, muss der Scheduler des Multiprozessors diese Zweige nacheinander ausführen. Darunter leidet die Auslastung des Multiprozessors. Um den Multiprozessor gleichmäßig auszulasten, ist es wichtig, dass alle Prozessoren den gleichen Instruktionszweigen folgen.

Mit OpenCL können Kernels übersetzt und in den Speicher der GPU kopiert werden. Dann werden die Eingabedaten in den GPU-Speicher kopiert und Aufrufe an die Kernel-Funktionen durchgeführt.

```
__kernel void boolean_matrix_and(
    __global unsigned int* C,
    __global unsigned int* A,
    __global unsigned int* B)
{
    const int nrows = get_global_size(0); // number of rows
    const int row = get_global_id(0); // index of work-item
    const int col = get_global_id(1);

    unsigned int a = A[col*nrows + row];
    unsigned int b = B[col*nrows + row];
    C[col*nrows + row] = a & b;
}
```

Listing 8.1: Beispiel: ein einfacher OpenCL-Kernel

8.3.2. Auftrags-Warteschlange

Die Berechnungen der GPU laufen unabhängig zur Arbeit der CPU. Aufträge für die GPU werden in eine Warteschlange eingereiht und dann von der GPU asynchron abgearbeitet. Ein Scheduler plant die Aufträge und verteilt sie auf die verfügbaren Multiprozessoren. OpenCL stellt Funktionen bereit, um den Zustand eines Auftrags zu prüfen, oder um auf die Ergebnisse eines Auftrags zu warten. Wenn nötig, lassen sich Abhängigkeiten zwischen Aufträgen modellieren. Beim Algorithmus für einfache Polygone können wir z. B. auf diese Weise den Aufruf-Graphen in Abb. 5.8 auf Seite 83 aufbauen.

Die Auftrags-Warteschlange enthält:

- Aufträge zum Kopieren von Daten zwischen GPU-Speicher und Host-Speicher,
- Aufträge zum Reservieren von Daten im GPU-Speicher,
- Kernel-Funktionen, die auf den Ein- und Ausgabedatenarbeiten.

8.3.3. Daten und Algorithmen modellieren

Es gibt eine Vielzahl unterschiedlicher GPU-Hardware, die sich in Architektur und Leistungsfähigkeit stark unterscheiden. Sehr einfache GPUs sind mit wenigen Prozessoren ausgestattet, während Hochleistungs-GPUs nicht selten mehrere tausend Prozessoren besitzen. Um Algorithmen für dieses breite Spektrum unterschiedlicher Hardware zu entwerfen, bieten die Frameworks OpenCL und CUDA Modellierungs-Werkzeuge auf einer abstrahierten Ebene.

Wir beginnen unseren Entwurf damit, dass wir ein Problem in datenparallele Arbeitspakete aufteilen. Ein Arbeitspaket entspricht einem **Thread**. Mehrere Threads werden zu einem **Block** zusammengefasst. Die Anzahl der Threads und die Größe der Blocks wird durch das vorliegende Problem definiert, nicht so sehr durch die konkrete GPU-Hardware.

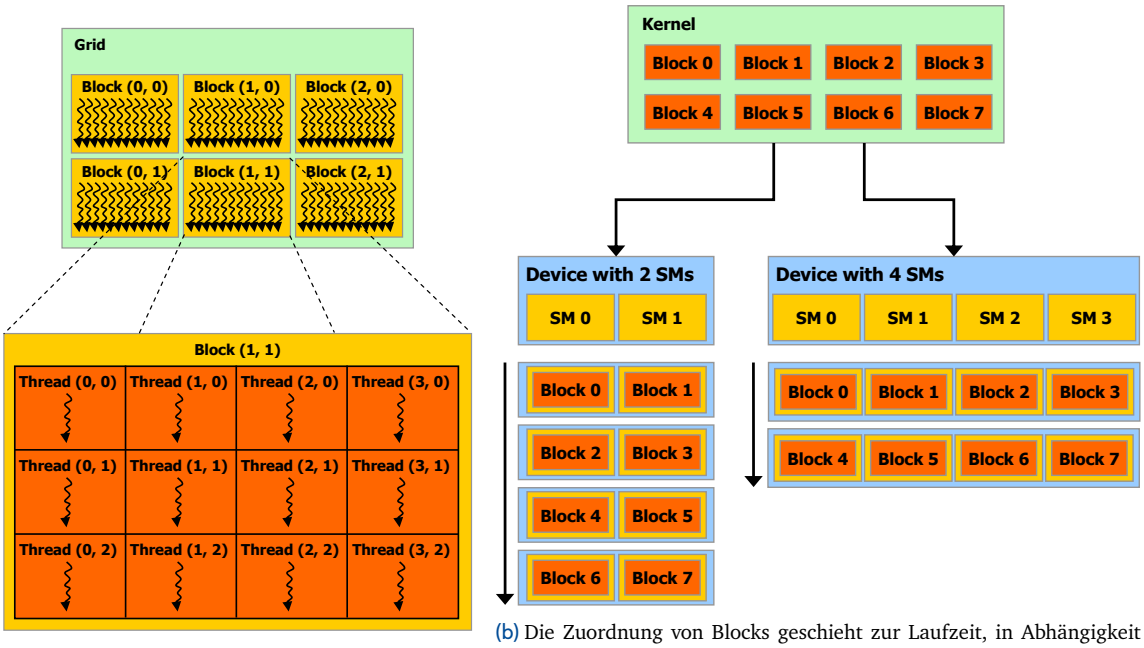
Erst zur Laufzeit entscheidet die GPU, in welcher Weise Threads und Blocks auf die verfügbaren Multiprozessoren verteilt werden. Ein Block von Threads wird dabei einem oder mehreren Multiprozessoren zugeteilt.

Da viele Aufgaben aus der Bildverarbeitung stammen, lassen sich die Threads in einem zwei- oder dreidimensionalen **Grid** anordnen. Die logische Anordnung der Threads folgt wieder der Modellierung des Problems, ist aber unabhängig von der Hardware-Architektur. Auch die Matrixmultiplikation, mit der wir uns gleich befassen werden, lässt sich auf naheliegende Weise in einem zweidimensionalen Grid modellieren.

8.4. Scheduling

Auf einem Multiprozessor können mehrere Blocks nebenläufig ausgeführt werden. Eine Menge von Threads, die gegenwärtig auf einem Multiprozessor ausgeführt wird, bezeichnen wir als **Warp** (in OpenCL: Wave Front). Hier müssen wir wieder zwischen der logischen Modellierung des Problems und der tatsächlichen Hardware unterscheiden. Die Größe eines Blocks wird durch die Problemstellung definiert. Die Anzahl der Threads in einem Warp jedoch durch die Hardware. Üblicherweise besteht ein Warp aus 32 Threads, die alle die gleichen Instruktionen auf unterschiedlichen Daten ausführen.

Während die Threads eines Warps auf einen langsamen Hauptspeicher-Zugriff warten, kann der Scheduler der GPU in der Zwischenzeit einem anderen Warp Rechenzeit zuteilen; in etwa so,



(a) logischer Aufbau: Grid, Blocks, Threads

(b) Die Zuordnung von Blocks geschieht zur Laufzeit, in Abhängigkeit von den verfügbaren Multiprozessoren.

Abbildung 8.4.: logischer Aufbau eines Problems und Zuordnung zur Laufzeit
(Zeichnungen: Nvidia Corp. [85])

wie auch eine CPU Rechenzeit an andere Threads abgeben kann. So lassen sich die Ressourcen der GPU besser nutzen und Wartezeiten kompensieren („verbergen“), die durch langsame Speicherzugriffe entstehen.

8.4.1. Arithmetic Intensity

Um die Wartezeiten bei Speicherzugriffen verbergen zu können, müssen ausreichend viele rechenbereite Aufgaben zur Verfügung stehen. Ansonsten können die Prozessoren nicht gut ausgelastet werden. Ein Algorithmus ist für die Ausführung auf GPU-Hardware dann am besten geeignet, wenn die Speicherzugriffe durch eine hohe Zahl von arithmetischen Operationen ausgeglichen werden können. Wir sprechen von einer hohen *Arithmetic Intensity*.

Die Multiplikation von Fließkommamatrizen besitzt z. B. eine hohe Arithmetic Intensity und ist deshalb gewissermaßen die Paradedisziplin von GPGPUs. In der Praxis werden GPUs häufig zur Multiplikation von Fließkommamatrizen eingesetzt.

An anderer Stelle hatten wir aber bereits festgestellt, dass der Aufwand für die Boolesche Arithmetik sehr viel niedriger ist. Wir werden daher im Folgenden die Erfahrung machen, dass die Arithmetic Intensity der Booleschen Matrixmultiplikation eher niedrig ist, und die Rechenleistung der GPU nicht optimal ausnutzt. Tatsächlich wird die Leistung unserer Implementierung der Booleschen Matrixmultiplikation in sehr viel höherem Maß durch die Speicherzugriffe bestimmt.

8.5. Free-Space-Berechnung auf GPUs

Die Berechnung des Free-Space ist ein datenparalleler Vorgang, der sich gut zur Umsetzung auf einer GPU eignet. In Kapitel 7.1 hatten wir schon bemerkt, dass sich die Free-Space-Intervalle unabhängig voneinander berechnen lassen. Weise [94] hat die Free-Space-Berechnung mit GPU-Unterstützung umgesetzt und ausführlich untersucht. Gudmundsson und Valladares [62] nutzen ebenfalls GPUs zur Berechnung des Fréchet-Abstands von Bewegungsdaten.

Bei unserem Algorithmus für einfache Polygone spielt die Berechnung des Free-Space aber nur eine sehr untergeordnete Rolle für die Gesamtlaufzeit. Wir verzichten auf eine GPU-Implementierung und berechnen die Free-Space-Daten konventionell (aber parallel) auf der CPU (siehe Kapitel 7.1).

Welche weiteren Teile unserer Algorithmen sind für die Umsetzung auf GPUs geeignet? Welche Algorithmen lassen sich als datenparallele Vorgänge modellieren?

Die Berechnung der Erreichbarkeits-Strukturen kommt dafür nicht in Frage. Datenstrukturen und Algorithmus sind viel zu komplex, um sie sinnvoll datenparallel zu modellieren. Das gleiche gilt für die Berechnung der gültigen Platzierungen. Der Algorithmus von Guibas et al. [64] zur Berechnung von Shortest-Path-Trees ist ebenfalls viel zu komplex, um ihn in eine datenparallele Form zu bringen. Wir werden uns deshalb im Folgenden auf die Boolesche Matrixmultiplikation konzentrieren.

8.6. Matrixmultiplikation auf GPUs

Operationen auf Matrizen sind in hohem Maße datenparallel und eignen sich sehr gut für die Umsetzung auf GPUs. Es gibt zahlreiche Bibliotheken für Matrizenrechnung auf GPU-Hardware. Insbesondere bei Fließkommamatrizen können GPUs ihre Stärken zur Geltung bringen. In Kapitel 9 stellen wir einige experimentelle Ergebnisse vor.

Es gibt hingegen nur sehr wenige Bibliotheken, die auf Boolesche Matrizen spezialisiert sind. Wir haben im Folgenden zwei Algorithmen zur Booleschen Matrixmultiplikation auf GPUs eigenständig implementiert und werden versuchen, sie zu bewerten.

8.6.1. Matrixmultiplikation mit Kacheln

Zunächst betrachten wir die „naive“ Matrixmultiplikation, wie wir sie in Kapitel 6.1 beschrieben haben. Bei der Umsetzung für GPUs versuchen wir, globale (teure) Speicherzugriffe zu reduzieren und möglichst viele Daten im Shared Memory zu halten. Zu diesem Zweck teilen wir die Matrix in gleichgroße Teilmatrizen (Kacheln) ein. Jede Kachel hält ihre Arbeitsdaten im Shared Memory und wird von einem Block von Threads bearbeitet. Jeder Thread bearbeitet einen Teil der Kachel – ein oder mehrere Speicherwörter. In Listing 8.2 haben wir ein Code-Beispiel in etwas vereinfachter Form wiedergegeben.

Im ersten Schritt werden die Eingabedaten einer Kachel aus dem globalen Speicher in das Shared Memory kopiert. Die Multiplikation einer einzelnen Kachel erfolgt nach dem Algorithmus aus Listing 6.1 mit verschachtelten Schleifen. Sie arbeitet ausschließlich auf Daten im Shared Memory. Zuletzt wird das Ergebnis zurück in den globalen Speicher kopiert.

Den Algorithmus aus Listing 6.2 mit vorzeitigem Abbruch der inneren Schleife wenden wir bewusst *nicht* an, da wir Verzweigungen vermeiden wollen, die lediglich von den Eingabedaten abhängen.

```
inline int buffer_address(int row, int col, int tile_n)
{ // ungerade Strides vermeiden Bank–Konflikte
    int rowd16 = row>>4;
    return col*(34*tile_n+1) + (rowd16<<4)+rowd16 + (row & 0x0f);
}

__kernel void clcubic_mul(
    __global unsigned int* C, __global unsigned int* A, __global unsigned int* B,
    __local unsigned int* A_buf, __local unsigned int* B_buf, // Shared Memory–Puffer
    int nrows, int ncols) // Größe der Matrix
{
    int tile_n = get_local_size(1); // Größe einer Kachel
    int lrow = get_local_id(0); // Zeile innerhalb der Kachel
    int lcol = get_local_id(1); // Spalte innerhalb der Kachel
    int row = get_global_offset(0) + get_group_id(0)*tile_n*32; // Zeile in A,C
    int col = get_global_offset(1) + get_group_id(1)*tile_n*32; // Spalte in B,C
    unsigned int Csum = 0; // Ergebnis

    for (int a0=0; a0 < ncols; a0 += tile_n)
    { // Kopiere eine Kachel vom globalen Speicher ins Shared Memory.
      // Jeder Thread liest zwei Wörter. Auf coalesced Zugriffe achten.
      A_buf[buffer_address(lrow,lcol,tile_n)] = A[a0*nrows + row];
      B_buf[buffer_address(lrow,lcol,tile_n)] = B[col*nrows + 32*a0];
      // Lese/Schreibzugriffe innerhalb eines Multiprozessors synchronisieren
      barrier(CLK_LOCAL_MEM_FENCE);
      // Bearbeite eine Zeile von A x eine Spalte von B
      for(int a1=0; a1 < tile_n; ++a1) {
          int ai = a0+a1;
          unsigned int a = A_buf[buffer_address(lrow,a1,tile_n)]
          a &= -(ai < A_width); // if (ai >= A_width) a = 0;

          for (int y=0; y < 32; ++y, a >>= 1) {
              unsigned int b = B_buf[buffer_address(32*a1+y,lcol,tile_n)];
              Csum |= -(a & 1) & b; // if (a & 1) Csum |= b
          }
      }
      barrier(CLK_LOCAL_MEM_FENCE);
    }
    // Ergebnis zurückkopieren. Jeder Thread schreibt ein Wort.
    C[col*nrows + row] = Csum;
}
```

Listing 8.2: OpenCL-Kernel zur Multiplikation mit Kacheln

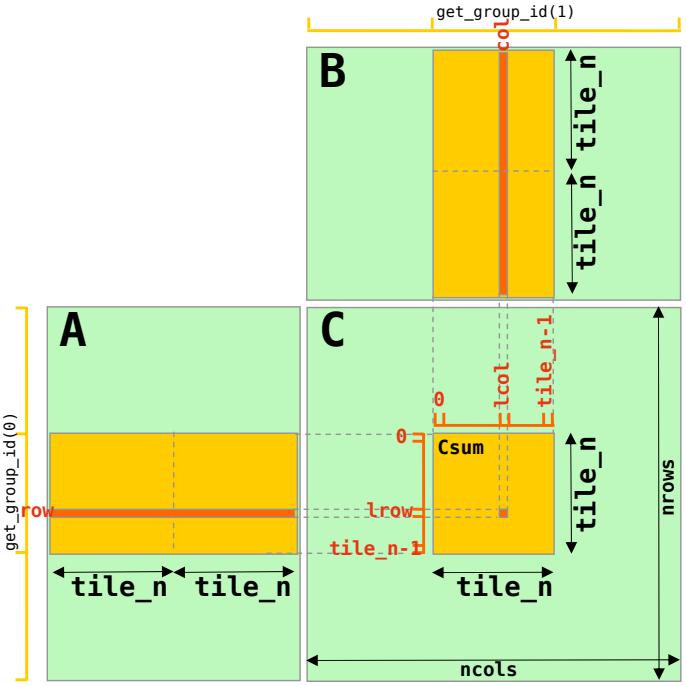


Abbildung 8.5: Matrixmultiplikation mit Kacheln

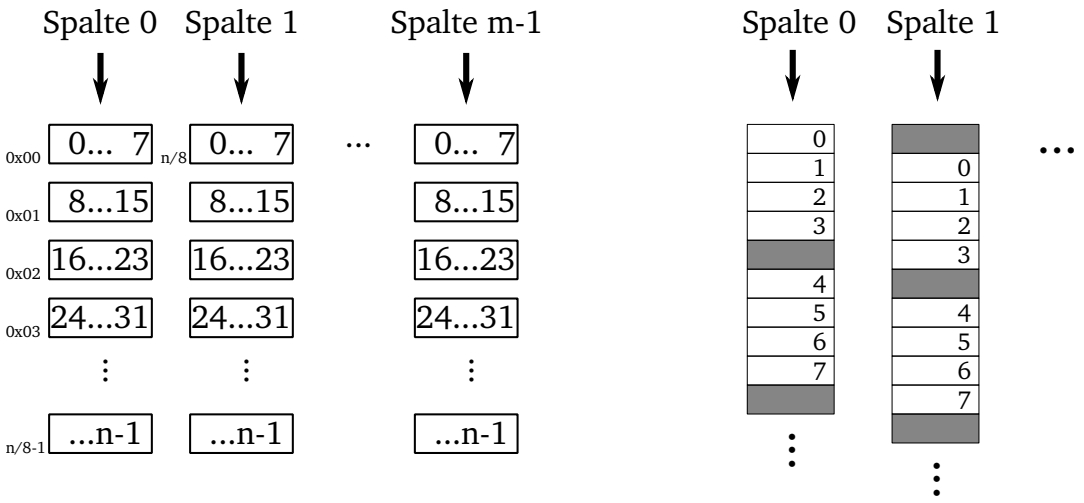
Komplexitätsanalyse

Der Aufwand für eine kachelweise Multiplikation unterscheidet sich übrigens nicht von der naiven Variante aus Listing 6.2. Wenn wir eine Matrix der Größe N in Kacheln der Größe T aufteilen, so benötigen wir für die Multiplikation einer einzigen Kachel $O(T^3)$ Operationen.

Die Multiplikation der gesamten Matrix benötigt $O((N/T)^3 \cdot T^3) = O(N^3)$ Operationen. Die kachelweise Multiplikation ist also nicht teurer. Sie bietet aber den Vorteil, dass wir die Berechnungen besser verteilen können, und dass die Arbeitsdaten eine höhere Lokalität aufweisen.

Daten Kopieren

Das Kopieren der Daten aus dem globalen Speicher wird von den Threads eines Blocks gemeinsam ausgeführt. Jeder Thread liest eine Reihe von Wörtern aus dem Speicher. Jedes Speicherwort enthält 32 Bits. Hierbei ist es sehr wichtig, auf das Muster der Speicherzugriffe zu achten, d. h. auf „vereinigte“ Zugriffe (Kapitel 8.2.1). Um sicherzustellen, dass die Zugriffe vereinigt (coalesced) ausgeführt werden, legen wir die Matrixdaten so im globalen Speicher ab, dass die Wörter einer Spalte in zusammenhängenden Speicherbereichen liegen (siehe Abb. 8.6a).



(a) Layout im globalen Speicher
Die Daten einer Spalte werden in fortlaufenden Byte-Adressen gespeichert. Große Zahlen geben Zeilenindizes wieder. Die kleinen Zahlen sind Speicheradressen.

(b) Layout im Shared Memory
Eingefügte Leerstellen sorgen für ungerade Zugriffsmuster und reduzieren Bank-Konflikte.

Abbildung 8.6.: Speicher-Layouts

Im Shared Memory passen wir das Speicherlayout so an, dass Bank-Konflikte reduziert werden (Kapitel 8.2.1). Bank-Konflikte treten häufig auf, wenn die Speicherzellen in geraden Abständen (Strides) gelesen werden. Wir fügen deshalb nach jeweils 16 Wörtern ein unbenutztes Wort ein, ebenso fügen wir zwischen jeder Spalte ein leeres Wort ein. Wir erreichen damit, dass die Speicherwörter in Strides von 17 Wörtern gelesen werden. Durch den Versatz verteilen sich die gleichzeitigen Zugriffe möglichst gleichmäßig auf die Speicher-Bänke. Wir vergeuden damit zwar einen Teil des ohnehin knappen Shared Memory, aber die Reduzierung von Bank-Konflikten ist ein lohnendes Ziel.

Die optimale Größe der Kacheln

Grundsätzlich ist es erstrebenswert, die Kacheln möglichst groß zu wählen, um die Anzahl der Blocks klein zu halten. Andererseits müssen die Arbeitsdaten einer Kachel im begrenzten Shared Memory eines Multiprozessors Platz finden. Darüber hinaus ist es sogar hilfreich, wenn *mehrere* Kacheln auf einem Multiprozessor Platz finden; nur dann kann der Scheduler Warps nebenläufig ausführen und damit langsame Speicherzugriffe kompensieren.

Die optimale Wahl der Kachelgröße hängt also von mehreren, teils schwer bestimmbaren Faktoren ab: der Größe des Shared Memory, der Zugriffszeit auf den globalen Speicher, der Auslastung der Prozessoren durch arithmetische Operationen.

In unserer Implementierung wird die Kachelgröße durch eine Heuristik bestimmt, die sich an der Größe des Shared Memory orientiert. Es besteht aber die Möglichkeit, die Kachelgröße über einen Kommandozeilenschalter gezielt zu steuern. Die empirischen Ergebnisse zeigen, dass die heuristische Wahl der Kachelgröße meist gute Ergebnisse liefert. Manuelles „Feintuning“ kann die Ergebnisse nur geringfügig verbessern.

8.6.2. Was ist mit der Methode der Vier Russen?

In Kapitel 6.3 haben wir gesehen, dass wir mit der Methode der Vier Russen gute Ergebnisse bei der Booleschen Matrixmultiplikation erzielen können. Allerdings haben wir auch gesehen, dass die Kosten der Speicherzugriffe die entscheidende Begrenzung für die Leistungsfähigkeit der Methode darstellen und deshalb sorgfältig geplant werden müssen. Es ist sehr wichtig, den Cache-Speicher gut zu nutzen. Bei der Umsetzung der Vier-Russen-Methode für GPUs gilt dies ebenso, unter ähnlichen Voraussetzungen.

Das begrenzte Shared Memory ist nun leider ein großes Problem. Im Kapitel 6.3.3 hatten wir schon gesehen, dass wir den optimalen Parameter $k = \log N$ reduzieren müssen, um die Lookup-Tabelle im Cache unterzubringen. Um zusätzlich noch die Arbeitsdaten von A und C im Shared Memory unterzubringen, müssten wir entweder k noch weiter reduzieren, oder wir müssten die Matrizen partitionieren. Wenn wir die Arbeitspakete zeilenweise partitionieren, müssen die Lookup-Tabellen mehrfach berechnet werden. Wenn wir hingegen die Arbeitspakete spaltenweise partitionieren, müssen die Eingabedaten von A öfter aus dem globalen Speicher gelesen werden.

Eine weitere Schwierigkeit: das zeilenweise Erstellen der Lookup-Tabelle ist inhärent sequentiell. In Kapitel 7.6 hatten wir das so gelöst, dass pro Iteration acht Tabellen parallel berechnet werden. Diese Idee können wir leider nicht bei unserer GPU-Implementierung anwenden. Sie scheitert auch hier wieder am fehlenden Shared Memory und an der Tatsache, dass Multiprozessoren ihre Daten nicht untereinander teilen können.

Weiter wäre es denkbar, die Lookup-Tabellen im globalen Speicher unterzubringen. Wir könnten sie zwar nebenläufig erstellen, allerdings benötigen wir dann sehr viel mehr globale Speicherzugriffe für die eigentliche Multiplikation.

Wie auch immer wir uns entscheiden, kann die Methode der Vier Russen ihre Vorteile nicht entfalten. Unsere empirischen Ergebnisse bestätigen dies leider auch. Die Methode der Vier Russen ist auf einer GPU nicht konkurrenzfähig gegenüber der kachelweisen Multiplikation. Demirel [52] hat die Umsetzung der Vier Russen auf GPU-Hardware untersucht und kommt zu vergleichbaren Ergebnissen.

8.7. Work Stealing zwischen CPU und GPU?

Zum Berechnen der Combined Reachability Graphs im Algorithmus für einfache Polygone führen führen wir MERGE-Operationen durch. Den Aufruf-Graphen haben wir in Abb. 5.8 auf Seite 83 skizziert. Jede MERGE-Operation wird durch den Aufruf eines Kernel-Programms realisiert. Die Aufrufe werden in eine Warteschlange eingereiht. Die Ergebnisse einer MERGE-Operation verbleiben im Hauptspeicher der GPU und dienen dem nächsten Auftrag als Eingabe.

Während die Aufträge von der GPU abgearbeitet werden, hat die CPU des Host-Systems nichts zu tun. Können wir die CPU besser auslasten, indem wir einen Teil der Aufträge von der CPU bearbeiten lassen? Das Verfahren ist unter dem Begriff „Work Stealing“ bekannt.

Damit die CPU Aufträge übernehmen kann, müssten aber die Eingabedaten vom GPU-Speicher zurück in den Host-Speicher kopiert werden. Diese zusätzlichen Kopiervorgänge sind teuer und müssen zudem ebenfalls als Aufträge in die Warteschlange eingereiht werden. Die Koordination der Kopier- und Multiplikations-Aufträge zwischen GPU und CPU wird dadurch sehr unübersichtlich und würde vermutlich den erhofften Gewinn durch „Work Stealing“ zunichte machen. Wir haben diese Idee nicht weiter verfolgt. Die Berechnung der Combined Reachability Graphs wird also entweder vollständig von der CPU bearbeitet, oder vollständig von der GPU. Eine Mischlösung wurde nicht realisiert.

Speicherplatz Der Aufruf-Graph aus Kapitel 5.9 bietet aber einen weiteren Vorteil, nämlich die Reduzierung des Speicherbedarfs. Anhand des Aufruf-Graphen können wir ermitteln, wann die Eingabedaten freigegeben werden können. Dies machen wir uns auch bei der GPU-Implementierung zunutze; die Freigabe des Speichers geschieht wiederum über die OpenCL-Warteschlange.

Über die Kommandozeilenoption `--large` kann die Bearbeitung in sortierter Reihenfolge eingeschaltet werden. Damit lässt sich der Bedarf an GPU-Speicher reduzieren. Auf die Gesamtlaufzeit hat diese Option aber keinen Einfluss.

8.8. Fazit zur Booleschen Matrixmultiplikation

Wir fassen unsere Ergebnisse zur Booleschen Matrixmultiplikation auf GPUs wie folgt zusammen:

- die Methode der Vier Russen ist für die Umsetzung auf GPUs nicht gut geeignet;
- die „naive“ Multiplikation mit Kacheln ist besser geeignet;
- im Vergleich zur Multiplikation von Fließkommamatrizen ist die Arithmetic Intensity des Problems nicht sehr hoch.

Bei den experimentellen Ergebnissen in Kapitel 9.1 können wir sehen, dass die Multiplikation von Fließkommamatrizen auf einer leistungsfähigen GPU um ca. den Faktor 100 schneller ist als auf einer CPU. Bei Booleschen Matrizen liegt der Beschleunigungs-Faktor „nur“ bei etwa 20. Auch wenn diese Messungen keinen Anspruch auf Allgemeingültigkeit haben, sind sie doch ein Indiz dafür, dass die Rechenleistung der GPU weniger gut ausgeschöpft wird. Der wichtigste bestimmende Faktor sind die Kosten der Speicherzugriffe. CPUs sind hier tendenziell etwas im Vorteil, weil sie größere und besser organisierte Cache-Speicher besitzen. Um eine signifikante Beschleunigung zu erzielen, braucht es sehr leistungsfähige GPU-Hardware mit guter Speicheranbindung.

9

Experimentelle Ergebnisse

Abschließend stellen wir einige experimentelle Ergebnisse vor. Die Messungen zur Booleschen Matrixmultiplikation helfen uns bei der Auswahl eines geeigneten Algorithmus in Kapitel 6. Mit den Messungen der Ergebnisse von Fréchet View wollen wir überprüfen, ob die parallelen Implementierungen ihr Ziel erfüllen, d. h. ob sich die erhoffte Beschleunigung gegenüber der seriellen Variante einstellt.

9.1. Boolesche Matrix-Multiplikation

In Abb. 9.1 führen wir einen Vergleich verschiedener Algorithmen zur Booleschen Matrixmultiplikation durch. Wir vergleichen hierzu einige Implementierungen mit Fließkomma-Arithmetik mit spezialisierten Implementierungen für Boolesche Matrizen und mit GPGPU-Implementierungen.

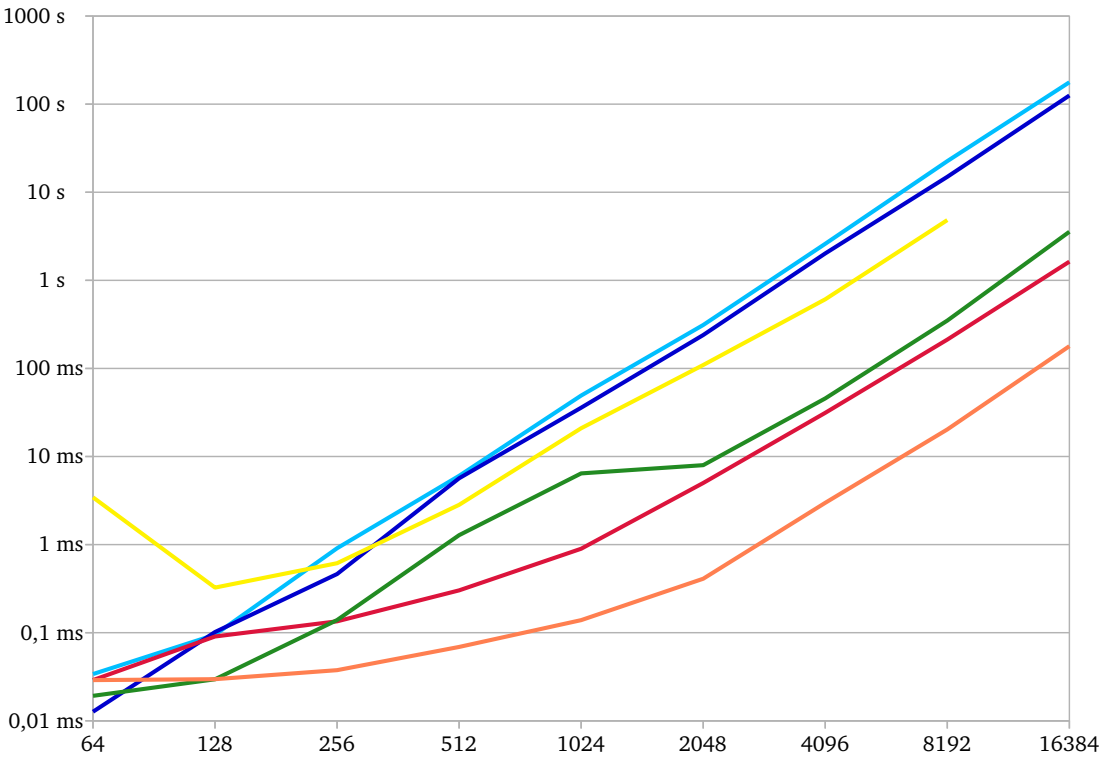
Hardware Wir verwenden handelsübliche PC- und Notebook-Hardware. Wir vergleichen eine (einfache) Notebook-GPU mit einer leistungsfähigen GPU vom Typ Nvidia Tesla V100. Solche High-End-Geräte sind sehr teuer, können aber bei Cloud-Dienstleistern angemietet werden.

- CPU: Intel Core i7-2600 mit 3,4 GHz Takt und 4 Kernen (ohne Hyperthreading).
- Low-End GPU: Intel HD Graphics.
- High-End GPU: Nvidia Tesla V100 mit 5.120 Prozessoren, 80 Multiprozessoren, 1,2 GHz Takt und 16 GB Hauptspeicher.

Software Wir vergleichen vier Implementierungen:

- Intel MKL [70] ist eine Bibliothek zur Multiplikation von Fließkomma- und Ganzzahlmatrizen auf CPUs. Sie verwendet einen der Algorithmen von Strassen [93] oder Coppersmith und Winograd [48].
- CLBlast [84] ist eine Implementierung für GPUs.
- Die Bibliothek M4RI [8] ist für Boolesche Matrizen konzipiert; wir haben sie in Kapitel 6.3.2 ausführlich vorgestellt.
- Zuletzt vergleichen wir unseren OpenCL-Kernel aus Kapitel 8.

Interpretation Die gemessenen Werte machen keine absoluten Aussagen über die Qualität der Algorithmen oder ihrer Implementierung. Dazu ist unsere Methodik auch nicht ausreichend. Sie sollen in erster Linie als Entscheidungshilfe für unsere Implementierung in Kapitel 6 dienen.



x-Achse: Größe der Matrizen: Anzahl der Zeilen und Spalten von $n = 2^6$ bis $n = 2^{14}$
y-Achse: Zeit für eine Multiplikation in Millisekunden bzw. Sekunden
(jeweils logarithmische Skalierung)

	Hardware	Zahlendarstellung	Software	
	CPU Intel Core i7	32 Bit Fließkomma	Intel MKL	[70]
	CPU Intel Core i7	32 Bit Ganzzahlen	Intel MKL	[70]
	GPU Intel HD Graphics	32 Bit Fließkomma	CLBlast	[84]
	CPU Intel Core i7	1 Bit	M4RI	[8]
	GPU Nvidia Tesla V100	32 Bit Fließkomma	CLBlast	[84]
	GPU Nvidia Tesla V100	1 Bit	Fréchet View	Kapitel 8.6

Abbildung 9.1.: Matrixmultiplikation auf CPU- und GPU-Hardware

Algorithmen Man sieht, dass die Algorithmen zur allgemeinen Matrixmultiplikation schlechter abschneiden als diejenigen, die für Boolesche Matrizen optimiert wurden. Obwohl der Algorithmus von Strassen [93] asymptotisch sehr gut ist, bringt er offenbar hohe konstante Faktoren mit sich. Es macht einen Unterschied, ob wir für einen Eintrag ein Bit benötigen oder 32 Bit. Ebenso ist die Fließkomma- oder Ganzzahl-Arithmetik aufwendiger als die Boolesche Arithmetik. Diese Ergebnisse hatten wir so auch erwartet.

GPU-Hardware Die Notebook-GPU bietet zwar eine gute Fließkomma-Leistung, kann aber mit der Booleschen Arithmetik nicht mithalten. Als Fazit kann man daraus ziehen, dass der Einsatz solcher GPUs für unser Problem keine Vorteile bietet. Eine deutliche Beschleunigung erreicht erwartungsgemäß die Hochleistungs-GPU mit Boolescher Arithmetik und unserem OpenCL-Kernel aus Kapitel 8.6. Dabei ist aber auch zu bedenken, dass solche GPUs sehr viel teurer als CPUs sind.

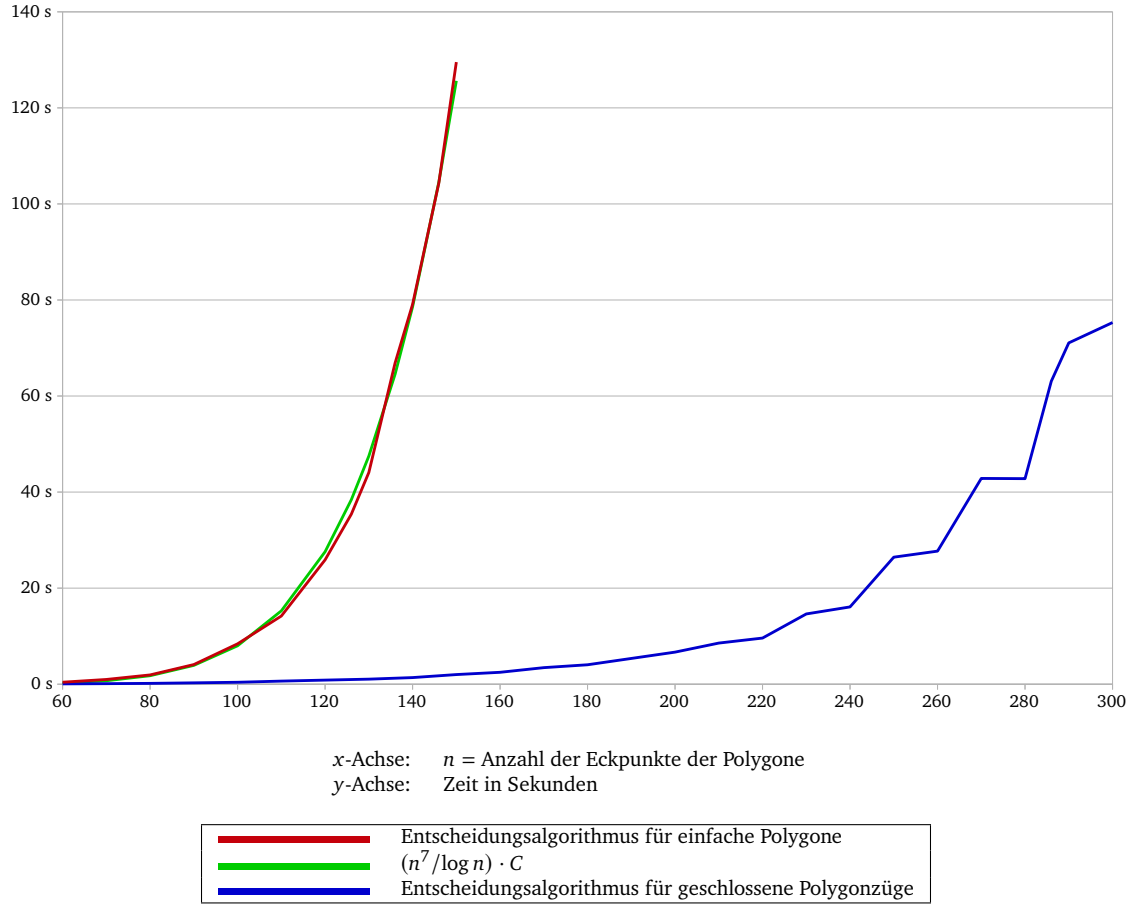


Abbildung 9.2.: Entscheidungsalgorithmus für einfache Polygone

9.2. Entscheidungsalgorithmen für Polygone und Polygonzüge

Wir haben die Laufzeiten der Entscheidungsalgorithmen in Abhängigkeit von der Eingabegröße gemessen. Die Eingabedaten (Abb. 9.3a) wurden so gewählt, dass die in Kapitel 3.13.1 gemachten Annahmen deutlich werden. Beide Polygone haben n Eckpunkte. Bei der konvexen Zerlegung entstehen $k = (n-1)/2 = O(n)$ Teile. Die Erreichbarkeits-Graphen bestehen aus ca. $0.63 \cdot n^2 = O(n^2)$ Knoten. Die Boolesche Matrixmultiplikation wurde mit der Methode der vier Russen realisiert, mit $T(N) = O(N^3/\log N)$.

Die erwarteten Kosten für den Entscheidungsalgorithmus für einfache Polygone sind damit

$$O(k \cdot T(n^2)) \;=\; O(n^7/\log n)\,.$$

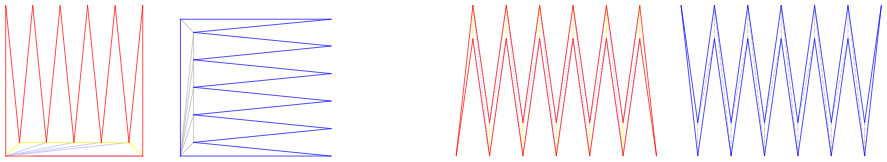
Die gemessenen Laufzeiten in Abb. 9.2 (rote Kurve) zeigen eine gute Übereinstimmung mit diesen Annahmen (grüne Kurve). Zum Vergleich sind die Laufzeiten des Entscheidungsalgorithmus für geschlossene Polygonzüge als blaue Kurve dargestellt. Die Messungen wurden auf einem Prozessorkern durchgeführt.

9.3. Algorithmen für den k-Fréchet-Abstand

Die Laufzeiten der Algorithmen für den k-Fréchet-Abstand sind nicht so leicht einheitlich darzustellen. In manchen Fällen liefert der Greedy-Algorithmus bereits eine optimale Lösung, in anderen Fällen wird der Brute-Force-Algorithmus benötigt. Die in Kapitel 5.10.4 beschriebene iterative Tiefensuche macht es zudem schwierig, von den tatsächlichen Laufzeiten auf die Wort-Case-Annahmen von $O(n^k)$ zu schließen. Wir geben einige Messungen für die weiter oben verwendeten Beispiele an.

	n	k	Zeit	
Abb. 4.6	26	3	0,3	ms
Abb. 4.10b	17	4	0,2	ms
”	26	6	0,5	ms
”	35	8	1,0	ms
”	44	10	21,8	ms
Abb. 4.10d	53	12	2,7	s
”	62	14	718	s

Tabelle 9.1.: Ergebnisse für den k-Fréchet-Abstand.
n = Größe der Polygonzüge.
k = Anzahl der Komponenten im Ergebnis.



(a) in Kapitel 9.1 (b) in Kapitel 9.4

Abbildung 9.3.: Eingabedaten für Messungen

```
frechet-view trouser-line.jscript -p --dec=36 --cores=4 --large --gpu --tile=3,4
Info:         using 4 threads on 8 CPU cores.
Info:         using Tesla V100-SXM2-16GB with 80 units.
              matrix tile = 3,4
Input:        trouser-line.jscript
Data:         P=525 vertices. Q=525 vertices.
Algorithm:    Fréchet Distance for Simple Polygons.
              Decision variant. Is d_F <= 36 ?
              Setup Curves: 42.386 ms = 0.042 s
Info:         Using extended floating point arithmetic (64 digits mantissa)
              for intermediate results.
Data:         matrix size (VV) = 9369
RGraphs:      5834.147 ms = 5.834 s
Decision:     YES. d_F <= 36.000000000000000
Elapsed time: 11981.018 ms = 11.981 s
```

Listing 9.1: Beispiel: Ausgaben von Fréchet View.

9.4. Parallelisierung

Im Folgenden zeigen wir einige Messungen zu den parallelen Algorithmen für einfache Polygone und für Polygonzüge. Wir führen Messungen mit einer unterschiedlichen Anzahl von Prozessorkernen durch, und auch mit unserer GPU-Implementierung aus Kapitel 8.

Die Messungen wurden auf einer Multi-Core-CPU Intel Xeon mit 16 Kernen und 2,30 GHz Takt gemacht. Als GPU kommt wieder die Nvidia Tesla V100 zum Einsatz.

9.4.1. Algorithmen für einfache Polygone

Wir testen den Entscheidungsalgorithmus, den Optimierungsalgorithmus und die näherungsweise Intervallschachtelung. Als Eingabe dienen zwei Polygone mit $n = 525$, bzw. $n = 365$ Kanten (Abb. 9.3b). Dazu wurde eine Variante des Polygons aus Abb. 3.1 gewählt. Die Adjazenzmatrizen der Erreichbarkeits-Graphen haben jeweils ca. 10.000 Spalten und Zeilen.

Im ersten Teil des Algorithmus für einfache Polygone werden Erreichbarkeits-Graphen berechnet. Dieser Teil läuft ausschließlich auf den CPUs. Die Berechnung der kritischen Werte erfolgt ebenfalls auf CPU-Hardware. Falls eine GPU vorhanden ist, werden alle Matrixmultiplikationen an die GPU delegiert. Wir messen die Gesamtlaufzeit und geben die Beschleunigung gegenüber der seriellen Variante an.

Entscheidungsalgorithmus			$n = 525$
CPU-Kerne	GPU	Zeit in Sekunden	Speed-Up
1		99,4	
2		56,3	1,8
4		30,8	3,2
8		19,5	5,1
12		15,1	6,6
16		14,3	7,0
4	✓	11,9	8,4

Optimierungsalgorithmus			$n = 365$
CPU-Kerne	GPU	Zeit in Sekunden	Speed-Up
1		101,5	
2		60,9	1,7
4		46,8	2,2
4	✓	37,2	2,7

Intervallschachtelung			$n = 365$
CPU-Kerne	GPU	Zeit in Sekunden	Speed-Up
1		314,2	
2		189,2	1,7
4		140,3	2,2
4	✓	55,9	5,6

Tabelle 9.2.: Ergebnisse: Algorithmen für einfache Polygone

9.4.2. Algorithmen für Polygonzüge

Der Algorithmus von Alt und Godau [13] für Polygonzüge ist schneller; wir testen hier mit größeren Eingabedaten. Als Eingabe dienen zwei Polygonzüge mit ca. $n = 16.000$, bzw. $n = 2.000$ Kanten.

Entscheidungsalgorithmus		$n = 16.005$
CPU-Kerne	Zeit in Sekunden	Speed-Up
1	372	
2	259	1,4
4	210	1,8

Optimierungsalgorithmus		$n = 2.005$
CPU-Kerne	Zeit in Sekunden	Speed-Up
1	168,6	
2	95,1	1,8
4	61,5	2,8

Intervallschachtelung		$n = 2.005$
CPU-Kerne	Zeit in Sekunden	Speed-Up
1	53,1	
2	32,1	1,7
4	22,5	2,4

Tabelle 9.3.: Ergebnisse: Algorithmen für Polygonzüge

9.4.3. Interpretation

Beim Entscheidungsalgorithmus für einfache Polygone liegt die erreichte Beschleunigung ungefähr in der Größenordnung des theoretisch Machbaren (Faktor 5,1 bei 8 Kernen). Bei mehr als 12 Kernen flachen die Werte etwas ab. Der Entscheidungsalgorithmus profitiert also gut von der Parallelisierung. Eine deutliche Beschleunigung erreicht erwartungsgemäß die Hochleistungs-GPU mit Boolescher Arithmetik. Der Optimierungsalgorithmus und die Intervallschachtelung profitieren weniger von der Parallelisierung.

A

Digitale Ressourcen

A.1. Fréchet View

Dieser Arbeit liegt eine CD mit zusätzlichem Material bei. Die CD enthält ausführbare Versionen des Programms Fréchet View sowie den vollständigen Quellcode. Die Datei `index.html` enhält eine Übersicht über den Inhalt der CD.

Alle Informationen finden Sie auch Online unter der Adresse:
`https://hrimfaxi.bitbucket.io/fv`

Ausführbare Programme

Für die Betriebssysteme Windows, Linux und macOS liegen ausführbare Versionen von Fréchet View bei. Die aktuellste Version kann außerdem von
`https://bitbucket.org/hrimfaxi/c-development/downloads`
heruntergeladen werden.

Falls Sie Probleme haben, diese Programme auf Ihren PC auszuführen, kontaktieren Sie bitte den Autor, oder erstellen Sie einen Bug Report auf
`https://bitbucket.org/hrimfaxi/c-development/issues/new`

Eine Installationsanleitung und eine Bedienungsanleitung finden Sie ebenfalls auf der beiliegenden CD oder Online.

Beispieldateien

Wir haben einige Beispiel-Dateien beigelegt. Sie sollen helfen, sich mit dem Programm vertraut zu machen. Einige Beispiele aus der vorliegenden Arbeit lassen sich mit den Dateien nachvollziehen:

`MCSP Reduction.jscript`	NP-Reduktion aus Kapitel 4.3.1
`Trousers.jscript`	Unterschied zwischen Fréchet-Abstand von Polygon und Randkurve aus Kapitel 3.2
`Greedy-Factor 2.jscript`	Konstruktion des Näherungsfaktors aus Kapitel 4.5.1

Quellcode

Der vollständige Quellcode für Fréchet View liegt bei. Die aktuellste Version ist zudem in einem öffentlichen Git-Repository verfügbar:

```
git clone https://bitbucket.org/hrimfaxi/c-development.git
```

Eine kommentierte Übersicht des Quellcodes ist ebenfalls enthalten. Falls Sie das Programm selbständig aus den Quellen erstellen möchten, finden Sie dazu eine kurze Anleitung. In der Regel ist dies aber nicht erforderlich.

Video

Für die Konferenz *Symposium on Computational Geometry 2019* entstand eine kleine Video-Präsentation, die ebenfalls auf der CD enthalten ist. In dem Video finden Sie eine kurze Einführung zum Fréchet-Abstand und zu den implementierten Algorithmen.

A.2. Werkzeuge und Bibiliotheken

Zur Entwicklung von Fréchet View wurden folgende Werkzeuge und Software-Bibliotheken verwendet. Sie sind nicht auf der CD enthalten, aber größtenteils frei verfügbar.

Software-Bibliotheken

- Computational Geometry Algorithms Library [41]
- M4RI [8]
- Boost C++ Libraries [20]
- C++ Standard Template Library
- Threading Building Blocks [71]
- OpenCL [75]
- Qt [88]
- Intel Math Kernel Library [70] und CLBlast [84] wurden für die Erstellung von Messwerten in Kapitel 9 verwendet.

Compiler

- GCC für Linux
- Intel C++ Compiler und MSVC für Windows
- Clang/LLVM für macOS

Entwicklungswerkzeuge

- Integrierte Entwicklungsumgebungen: Qt Creator und JetBrains CLion
- GDB Debugger
- git Source-Code-Verwaltung
- Build-Werkzeuge: qmake und CMake
- Google Test für Unit Tests
- Doxygen zur Dokumentation des Quellcodes

B

Instructions

B.1. How To use Fréchet View

- Open an input file and display the correspondig free-space diagram.
- Play with parameter ε and watch the free-space diagram change.
- Select an algorithm from the panel on the lower left part of the screen.

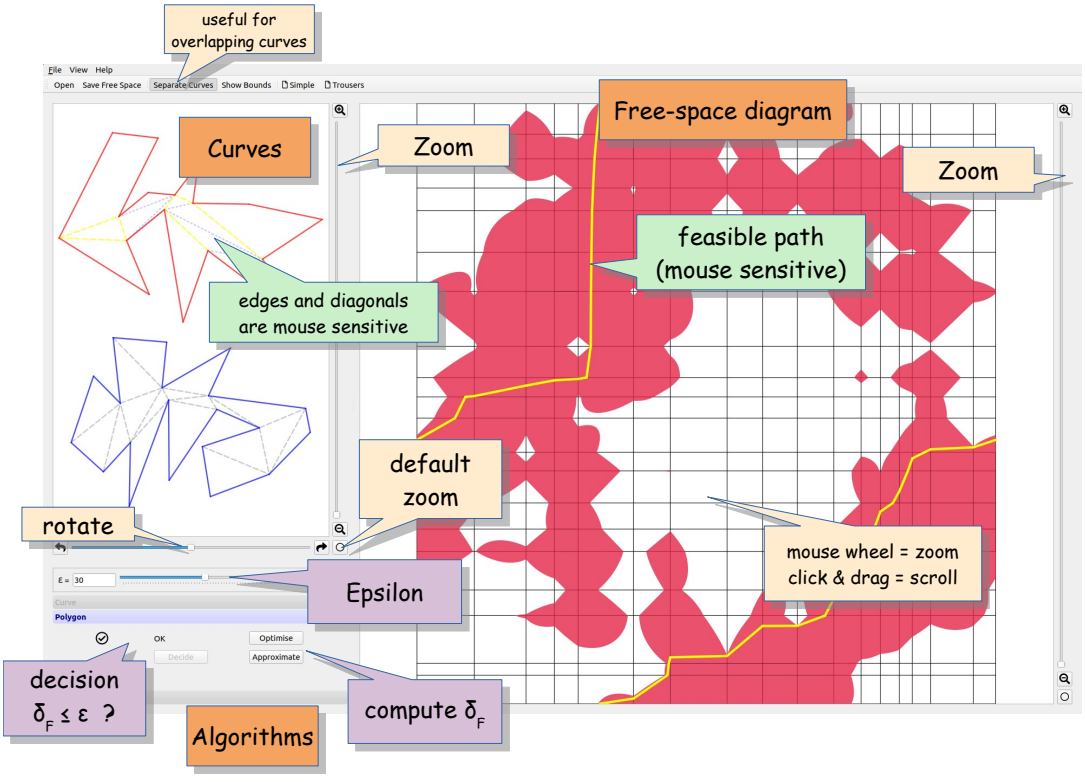


Abbildung B.1.: Fréchet View Window

Curve Algorithm

Computes the classical Fréchet distance for polygonal curves. The answer to the decision problem ("is $\delta_F \leq \varepsilon$?") is displayed in real-time.

The resulting **feasible path** is shown as a yellow line inside the free-space diagram. The path may wrap at the right edge of the diagram and continue at the left edge.

Show Bounds displays reachable intervals as dotted lines.

Separate Curves moves the input curves apart.

Move the mouse pointer over an edge (or a diagonal) to highlight the correspoding parts of the curves (mapped by a homeomorphism).

(**Optimise** und **Approximate** see below).

Polygon Algorithm

Computes the Fréchet distance for simple polygons (i. e. polygons without holes, and without self-intersections).

Decide computes the answer to the decision problem. If successful, the resulting feasible path is shown in the free-space diagram.

Optimize computes the Fréchet distance (i.e. the smallest possible epsilon) using binary search on a set of *critical values*.

Approximate computes an approximation result (may sometimes be faster).

k-Fréchet Algorithm

Computes the parameter k for the k-Fréchet distance.

- the **Greedy** result is shown immediately. Lower and upper bounds for k are shown.
- click **Brute-Force** to compute the exact result; **stop** the computation if it takes too long.

Connected components are indicated by colors.

Click **Show Bounds** to project components to the x- and y-axes. These projected intervals are the basis for the k-Fréchet algorithms.

B.1.1. Print and Save

Curves and the free-Space diagram can be printed, or saved to disk as a PDF files.
Note: saving to SVG files is available, but may not always be accurate. Do prefer PDF.

B.1.2. File Formats

Fréchet View expects two polygonal curves as input data. These can be read from SVG or IPE files. There are numerous applications for editing vector graphics files, like InkScape or Ipe.

The Curve algorithm accepts any pair of straight polygon paths (not Beziér Curves).
The Polygon algorithm expects closed, simple polygons (no intersections).

💡 While viewing a file in Fréchet View, you can edit the same file in any graphical or text editor. Saved changes are updated immediately in the Fréchet View window.

Use **JavaScript** to "program" complex curves. Have a look at some sample files.
Our JavaScript interface defines two `Path` variables `P` and `Q`.
All `Path` methods are "fluent", like: `P.M(20,40).H(100).m(30,10).Z();`
Other common JavaScript constructs (functions, loops, variables, etc.) are avaible.

`new Path()`	creates an empty Path object
`new Path(copy)`	copies a Path object
`M(x,y)`	draws a polygon edge to (x,y)
`H(x)`	horizontal edge to x
`V(y)`	vertical edge to y
`m(x,y)`	edge relative to the current location
`h(x,y)`	horizontal edge, relative to current location
`v(x,y)`	vertical edge, relative to current location
`Z()`	close polygon
`appendPath(path)`	appends one path to another
`appendString(string)`	appends a string-encoded path. Strings contain a series of M-H-V-Z instructions, like "M 10 10 H 90 V 90 H 10 Z". Roughly follows the SVG Specification.
"Turtle Graphics"	
`polar(angle,distance)`	draws an edge defined by angle (360° degrees) and distance
`forward(distance)`	move forward or backward
`left(angle)`	turn left (without drawing an edge)
`right(angle)`	turn right
`reset(angle)`	update angle
Transformations	
`scale(factor)`	scales the whole Path object
`scale(x,y)`	scales the whole Path object
`rotate(angle)`	rotates the whole Path object
`translate(x,y)`	translates the whole Path object
`mirrorx()`	mirrors Path object horizontally
`mirrory()`	mirrors Path object vertically

Tabelle B.2.: Path API

B.1.3. Command Line Interface

You can run Fréchet View in "headless" mode from a command line terminal. The program processes just one input file. Select an algorithm:

`-c`	Fréchet distance for curves
`-p`	Fréchet distance für simple polygons
`-k`	k-Fréchet distance

Options `-c` and `-p` support three variants:

`--dec=`ε	decision problem: is $\delta_F \leq \varepsilon$?
`--opt`	optimization problem: computes the exact value of δ_F
`--app=`δ	approximates δ_F by nested intervals

The k-Fréchet algorithm `-k` accepts only `--dec=`ε. It computes the parameter k for a given ε. There is no other optimization variant for the k-Fréchet algorithm.

`--cores` defines the number of parallel threads on a multi-core system. Results are usually best if you choose the number of physical CPU cores. Performance *does not* benefit from hyperthreading on Intel processors.
Example for a quad core processor: `--cores=4` is fine, but `--cores=8` does not improve.

`--large` minimizes memory usage (for really large input data).

`--gpu` enables GPGPU support. You need an OpenCL 1.2 compatible card and driver. Significant speedups require *really* powerful graphics cards.

`--tile=`*a,b* is a fine-tuning parameter for GPGPU support.

`--scale=`*x* adjusts to HiDPI screen resolutions (if necessary)

Type `--help` to view available options.

Examples

```
./Frechet_View "Demo Data/trousers.jscript" -p --dec=45

C:\Programs (x86)\Frechet_View\frechet-view "Demo Data/trousers.jscript" -c --approx=1-e6

Frechet-View.app/Contents/MacOS/frechet-view "Demo Data/trousers.jscript" -p --opt --cores=4

./Frechet_View "Demo Data/criss-cross-2.jscript" -k --dec=0.4
./Frechet_View "Demo Data/trousers.jscript" -p --dec60 --cores=4 --gpu
```

B.2. Installation Notes

Windows Installation

Unzip downloaded package and run `frechet-view.exe`.

Linux Installation

Make the downloaded file **executable**:

```
chmod +x Frechet_View_for_Linux
```

and run it by double clicking, or from a terminal:

```
./Frechet_View_for_Linux
```

Use "Help/Install Desktop Icon" to place a link onto your desktop (you may need to edit that desktop file, but it's easy).

macOS Instalation

Open the downloaded disk image and double click the Fréchet View application icon.

All binaries are 64-bit. If you *desperately* need a 32-bit version, try to build from sources.
I've never tried it myself, but I assume it's *possible*.

B.2.1. GPGPU Support

GPGPU support is optional. You need a graphics card driver that supports OpenCL 1.2 or later. Best results are achieved if the graphics card has high memory bandwidth. Graphics cards that share memory with the CPU (like most Notebook cards) are usually less effective.

B.3. Building from Sources

Clone Source Repository

```
git clone https://bitbucket.org/hrimfaxi/c-development.git
```

Download Third-Party Libraries

Qt 5.9 or later is required. Qt comes pre-installed with many Linux system, or can be downloaded from `https://www.qt.io/download-qt-installer`.
Check installed version with `qmake --version`.

Boost 1.53 or later (maybe earlier) is required. Boost comes pre-installed with most Linux systems, or can be downloaded from `https://www.boost.org`.
Binary libraries are not needed, Boost header files are sufficient.

Computational Geometry Algorithms Library
`git clone https://github.com/CGAL/cgal.git`
Note that we do not need CGAL binaries, header files are sufficient.

Intel Threading Building Blocks
Download a package for your system from `https://github.com/01org/tbb/releases`.

Unpack it. If you need to build TBB libraries from source, refer to their documentation. It is not very difficult, actually. The hardest part of TBB is finding the right documentation.

M4RI Boolean Matrix Multiplication

```
git clone https://bitbucket.org/malb/m4ri.git
cd m4ri
autoreconf --install
./configure --enable-thread-safe --disable-png
make
```

Next, open `m4ri/m4ri_config.h` and make sure that the following options are set correctly:

```
#define __M4RI_HAVE_MM_MALLOC       0
#define __M4RI_HAVE_POSIX_MEMALIGN  0
#define __M4RI_HAVE_SSE2            1
...
#define __M4RI_ENABLE_MZD_CACHE     0
#define __M4RI_ENABLE_MMC           0
```

This is very important, because some of these options are *not safe* for multi-threading!

OpenCL SDKs are available from Intel, AMD, and as part of Nvidia CUDA toolkit.
OpenCL is readily installed on macOS and many Linuxes.

Build Project

Adjust include and library paths as necessary in `projects/frechet-view.pro`.
Then run the build script:

```
cd projects/frechet-view
qmake -config release
make
```

(resp. `nmake` on Windows)

Alternatively, a **cmake** project script is available.
MSVC project files are available, but I won't promise to keep them up-to-date.

Windows Build Notes

Both **Microsoft Visual C++** and **Intel C++** compilers are supported.
We recommend the latter because it allows for better numerical results.
Typing `qmake -spec win32-icc` should do the trick.

Building M4RI from sources may not work on Windows. However, it is sufficient to edit the file `m4ri\m4ri_config.h`, as described above.

Linux Build Notes

gcc 5.4 or later is recommended.

macOS Build Notes

XCode Command Line Tools are required. Install with `xcode-select --install`.
A full-fledged XCode installation is not needed.

Literaturverzeichnis

[1] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan und Micha Sharir. Computing the discrete Fréchet distance in subquadratic time. *CoRR*, abs/1204.5333, 2012.

[2] Mahmuda Ahmed und Carola Wenk. Constructing street networks from GPS trajectories. In Leah Epstein und Paolo Ferragina, Hrsg., *Algorithms – ESA 2012*, S. 60–71. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33090-2.

[3] Mahmuda Ahmed, Kyle S. Hickmann und Carola Wenk. Path-based distance for street map comparison. *CoRR*, abs/1309.6131, 2013.

[4] Mahmuda Ahmed, Sophia Karagiorgou, Dieter Pfoser und Carola Wenk. *Fréchet Distance-Based Map Construction Algorithm*, S. 33–46. Springer International Publishing, 2015. doi: 10.1007/978-3-319-25166-0_3.

[5] Hee-Kap Ahn, Christian Knauer, Marc Scherfenberg, Lena Schlipf und Antoine Vigneron. Computing the discrete Fréchet distance with imprecise input. *International Journal of Computational Geometry & Applications*, 22(01):27–44, 2012. doi: 10.1142/S0218195912600023.

[6] Hugo A. Akitaya, Maike Buchin, Leonie Ryvkin und Jérôme Urhausen. The k-Fréchet distance. *CoRR*, abs/1903.02353, 2019.

[7] Hugo A. Akitaya, Maike Buchin, Leonie Ryvkin und Jérôme Urhausen. The k-Fréchet distance revisited and extended. In *35th European Workshop on Computational Geometry*, 2019. `http://www.eurocg2019.uu.nl/papers/41.pdf`.

[8] Martin Albrecht und Gregory Bard. *The M4RI Library*, 2009-18. `https://bitbucket.org/malb/m4ri`.

[9] Martin Albrecht, Gregory Bard und William Hart. Algorithm 898: Efficient multiplication of dense matrices over GF(2). *ACM Trans. Math. Softw.*, 37(1):9:1–9:14, January 2010. ISSN 0098-3500. doi: 10.1145/1644001.1644010.

[10] Helmut Alt und Maike Buchin. Semi-computability of the Fréchet distance between surfaces. In *21st European Workshop on Computational Geometry*, S. 45–48, 2005.

[11] Helmut Alt und Maike Buchin. Can we compute the similarity between surfaces? *Discrete & Computational Geometry*, 43(1):78, Mar 2009. ISSN 1432-0444. doi: 10.1007/s00454-009-9152-8.

[12] Helmut Alt und Maike Buchin. Can we compute the similarity between surfaces? *Discrete & Computational Geometry*, 43(1):78, Mar 2009. doi: 10.1007/s00454-009-9152-8.

[13] Helmut Alt und Michael Godau. Computing the Fréchet distance between two polygonal curves. *International Journal of Computational Geometry and Applications*, 5, 1995. doi: 10.1142/S0218195995000064.

[14] Helmut Alt und Leonidas J. Guibas. *Discrete Geometric Shapes: Matching, Interpolation, and Approximation*, S. 121–153. Elsevier, 2000.

[15] Helmut Alt, Bernd Behrends und Johannes Blömer. Approximate matching of polygonal shapes. *Annals of Mathematics and Artificial Intelligence*, 13(3):251–265, Sep 1995. ISSN 1573-7470. doi: 10.1007/BF01530830.

[16] Helmut Alt, Alon Efrat, Günter Rote und Carola Wenk. Matching planar maps. *Journal of Algorithms*, 49(2):262 – 283, 2003. ISSN 0196-6774. doi: https://doi.org/10.1016/S0196-6774(03)00085-3.

[17] Helmut Alt, Christian Knauer und Carola Wenk. Comparison of distance measures for planar curves. *Algorithmica*, 38(1):45–58, Jan 2004. ISSN 1432-0541. doi: 10.1007/s00453-003-1042-5.

[18] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod und I. A. Faradžev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics—Doklady*, 11(5):1209–1210, 1970.

[19] Boris Aronov, Sariel Har-Peled, Christian Knauer, Yusu Wang und Carola Wenk. Fréchet distance for curves, revisited. In Yossi Azar und Thomas Erlebach, Hrsg., *Algorithms – ESA 2006*, S. 52–63. Springer Berlin Heidelberg, 2006.

[20] Boost. *The Boost C++ Libraries*, 1.66 edition. `https://www.boost.org`.

[21] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas und Carola Wenk. On map-matching vehicle tracking data. In *31st VLDB Conference*, S. 853–864, 2005.

[22] Karl Bringmann. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. *CoRR*, abs/1404.1448, 2014.

[23] Karl Bringmann und Marvin Künnemann. Improved approximation for Fréchet distance on c-packed curves matching conditional lower bounds. *CoRR*, abs/1408.1340, 2014.

[24] Karl Bringmann und Wolfgang Mulzer. Approximability of the discrete Fréchet distance. *Journal of Computational Geometry*, 7(2):46–76, 2016. doi: http://dx.doi.org/10.20382/jocg.v7i2a4.

[25] Kevin Buchin, Maike Buchin und Carola Wenk. Computing the Fréchet distance between simple polygons in polynomial time. In *22nd Annual Symposium on Computational Geometry*, SCG '06, S. 80–87, New York, USA, 2006. ISBN 1-59593-340-9. doi: 10.1145/1137856.1137870.

[26] Kevin Buchin, Maike Buchin, Christian Knauer, Günter Rote und Carola Wenk. How difficult is it to walk the dog? In *23rd European Workshop on Computational Geometry*, S. 170–173, 2007. `http://page.mi.fu-berlin.de/rote/Papers/postscript/How+difficult+is+it+to+walk+the+dog.ps`.

[27] Kevin Buchin, Maike Buchin und Carola Wenk. Computing the Fréchet distance between simple polygons. *Computational Geometry*, 41:2–20, 2008. `https://doi.org/10.1016/j.comgeo.2007.08.003`.

[28] Kevin Buchin, Maike Buchin und Carola Wenk. Computing the Fréchet distance between simple polygons. *Computational Geometry*, 41(1):2 – 20, 2008. ISSN 0925-7721. doi: https:

//doi.org/10.1016/j.comgeo.2007.08.003. Special Issue on the 22nd European Workshop on Computational Geometry (EuroCG).

[29] Kevin Buchin, Maike Buchin und André Schulz. Fréchet distance of surfaces: Some simple hard cases. In Mark de Berg und Ulrich Meyer, Hrsg., *Algorithms – ESA 2010*, S. 63–74. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15781-3.

[30] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler und Jun Luo. Detecting commuting patterns by clustering subtrajectories. *International Journal of Computational Geometry & Applications*, 21(03):253–282, 2011. doi: 10.1142/S0218195911003652.

[31] Kevin Buchin, Tim Ophelders und Bettina Speckmann. Computing the similarity between moving curves. In Nikhil Bansal und Irene Finocchi, Hrsg., *Algorithms - ESA 2015*, S. 928–940, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[32] Kevin Buchin, Maike Buchin, Maximilian Konzack, Wolfgang Mulzer und André Schulz. Fine-grained analysis of problems on curves. In *32nd European Workshop on Computational Geometry*, 2016. http://www.win.tue.nl/~mkonzack/papers/lowerBoundsCurvesEuroCG.pdf.

[33] Kevin Buchin, Maike Buchin, Rolf van Leusden, Wouter Meulemans und Wolfgang Mulzer. Computing the Fréchet distance with a retractable leash. *Discrete and Computational Geometry*, 56(2):315–336, Sep 2016. ISSN 1432-0444. doi: 10.1007/s00454-016-9800-8.

[34] Kevin Buchin, Maike Buchin, David Duran, Brittany Terese Fasy, Roel Jacobs, Vera Sacristan, Rodrigo I. Silveira, Frank Staals und Carola Wenk. Clustering trajectories for map construction. In *25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, S. 14:1–14:10, New York,USA, 2017. ISBN 978-1-4503-5490-5. doi: 10.1145/3139958.3139964.

[35] Kevin Buchin, Maike Buchin, Wouter Meulemans und Wolfgang Mulzer. Four soviets walk the dog: Improved bounds for computing the Fréchet distance. *Discrete and Computational Geometry*, 58(1):180–216, Jul 2017. ISSN 1432-0444. doi: 10.1007/s00454-017-9878-7.

[36] Kevin Buchin, Erin Chambers, Tim Ophelders und Bettina Speckmann. Fréchet isotopies to monotone curves. In *33rd European Workshop on Computational Geometry*, S. 41–44, 2017. http://csconferences.mah.se/eurocg2017.

[37] Kevin Buchin, Jinhee Chun, Maarten Löffler, Aleksandar Markovic, Wouter Meulemans, Yoshio Okamoto und Taichi Shiitada. Folding free-space diagrams: Computing the Fréchet distance between 1-dimensional curves. In *33rd International Symposium on Computational Geometry, July 4-7, 2017, Brisbane, Australia*, S. 64:1–64:5, 2017. doi: 10.4230/LIPIcs.SoCG.2017.64.

[38] Kevin Buchin, Tim Ophelders und Bettina Speckmann. SETH says: Weak Fréchet distance is faster, but only if it is continuous and in one dimension. *CoRR*, abs/1807.08699, 2018.

[39] Maike Buchin. *On the Computability of the Fréchet Distance Between Triangulated Surfaces*. Doktorarbeit, Freie Universität Berlin, 2007. http://www.diss.fu-berlin.de/diss/receive/FUDISS_thesis_000000002618.

[40] Maike Buchin und Leonie Ryvkin. The *k*-Fréchet distance of polygonal curves. In *34th European Workshop on Computational Geometry*, 2018. https://conference.imp.fu-berlin.de/eurocg18/download/paper_43.pdf.

[41] CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.12 edition, 2018. https://doc.cgal.org/4.12/Manual/packages.html.

[42] Erin Chambers, Éric Colin de Verdière, Jeff Erickson, Sylvain Lazard, Francis Lazarus und Shripad Thite. Homotopic Fréchet distance between curves or, walking your dog in the woods in polynomial time. *Computational Geometry*, 43(3):295 – 311, 2010. ISSN 0925-7721. doi: https://doi.org/10.1016/j.comgeo.2009.02.008. Special Issue on 24th Annual Symposium on Computational Geometry (SoCG'08).

[43] Richard Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987. ISSN 0004-5411. doi: 10.1145/7531.7537.

[44] Atlas F. Cook und Carola Wenk. Geodesic Fréchet distance with polygonal obstacles. Technical Report CS-TR-2008-0010, University of Texas at San Antonio. Department of Computer Science, 2008.

[45] Atlas F. Cook und Carola Wenk. Geodesic Fréchet distance inside a simple polygon. *ACM Trans. Algorithms*, 7(1):9:1–9:19, 2010. ISSN 1549-6325. doi: 10.1145/1868237.1868247.

[46] Atlas F. Cook, Jessica Sherette und Carola Wenk. Computing the Fréchet distance between polyhedral surfaces with acyclic dual graphs. In *19th Fall Workshop on Computational Geometry: 75-76, Tufts University, Medford, MA*, 2009.

[47] Atlas F. Cook, Anne Driemel, Sariel Har-Peled, Jessica Sherette und Carola Wenk. Computing the Fréchet distance between folded polygonsIn Frank Dehne, John Iacono und Jörg-Rüdiger Sack, Hrsg., *Algorithms and Data Structures*, S. 267–278. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-22300-6.

[48] Don Coppersmith und Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251 – 280, 1990. ISSN 0747-7171. doi: https://doi.org/10.1016/S0747-7171(08)80013-2.

[49] Evan Cordell. *Fréchet Distance for Simple Polygons*. Honors Thesis, Tulane University, 2013. http://www.cs.tulane.edu/~carola/research/evanCordellhonorsThesis.pdf.

[50] Mark de Berg und Atlas F. Cook. Go with the flow: The direction-based Fréchet distance of polygonal curves. In Alberto Marchetti-Spaccamela und Michael Segal, Hrsg., *Theory and Practice of Algorithms in (Computer) Systems*, S. 81–91. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19754-3.

[51] Mark de Berg, A. Frank van der Stappen, Jules Vleugels und Matthew J. Katz. Realistic input models for geometric algorithms. *Algorithmica*, 34(1):81–97, 2002. doi: 10.1007/s00453-002-0961-x.

[52] Denise Demirel. *Effizientes Lösen linearer Gleichungssysteme über GF(2) mit GPUs*. Diplomarbeit, Technische Universität Darmstadt, 2010.

[53] Rodney G. Downey und Michael R. Fellows. *Fundamentals of Parameterized Complexity*. Springer Publishing Company, Incorporated, 2013. ISBN 1447155580, 9781447155584.

[54] Anne Driemel. Two decades of algorithms for the Fréchet distance, 2016. `http://www.win.tue.nl/~adriemel/shonan2016.pdf`.

[55] Anne Driemel, Sariel Har-Peled und Carola Wenk. Approximating the Fréchet distance for realistic curves in near linear time. *CoRR*, abs/1003.0460, 2010.

[56] Thomas Eiter und Heikki Mannila. Computing discrete Fréchet distance. Technical Report CD-TR 94/64, Information Systems Department, Technical University of Vienna., Mar 1994.

[57] George M. Ewing. *Calculus of Variations with Applications*. Dover Publications, New York, 1985.

[58] Maurice Fréchet. Sur quelques points du calcul fonctionnel. *Rendiconti del Circolo Matematico di Palermo*, 22(1):1–72, 1906. ISSN 0009-725X. doi: 10.1007/BF03018603.

[59] Maurice Fréchet. Sur la distance de deux surfaces. *Annales de la Société Polonaise de Mathématique*, 3:4–19, 1924.

[60] Michael Godau. *On the complexity of measuring the similarity between geometric objects in higher dimensions*. Doktorarbeit, Freie Universität Berlin, 1999. `https://refubium.fu-berlin.de/handle/fub188/3580`.

[61] Daniel H. Greene. The decomposition of polygons into convex parts. In Franco P. Preparata, Hrsg., *Advances in Computing Research – Computational Geometry*, S. 235–259. JAI Press, 1983.

[62] Joachim Gudmundsson und Nacho Valladares. A GPU approach to subtrajectory clustering using the Fréchet distance. *IEEE Transactions on Parallel and Distributed Systems*, 26(4): 924–937, April 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2317713.

[63] Joachim Gudmundsson, Majid Mirzanezhad, Ali Mohades und Carola Wenk. Fast Fréchet distance between curves with long edges. In *3rd International Workshop on Interactive and Spatial Computing*, S. 52–58, New York, USA, 2018. ACM. ISBN 978-1-4503-5439-4. doi: 10.1145/3191801.3191811.

[64] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir und Robert E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1):209–233, Nov 1987. ISSN 1432-0541. doi: 10.1007/BF01840360.

[65] Leonidas J. Guibas und John Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126 – 152, 1989. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(89)90041-X.

[66] Dan Gusfield. The four-russians speedup. In *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[67] Sariel Har-Peled, Amir Nayyeri, Mohammad R. Salavatipour und Anastasios Sidiropoulos. How to walk your dog in the mountains with no magic leash. *CoRR*, abs/1401.7042, 2014.

[68] John Hershberger und Jack Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry*, 4(2):63 – 97, 1994. ISSN 0925-7721. doi: https://doi.org/10.1016/0925-7721(94)90010-8.

[69] Stefan Hertel und Kurt Mehlhorn. Fast triangulation of simple polygons. In Marek Karpinski, Hrsg., *Foundations of Computation Theory*, S. 207–218. Springer Berlin Heidelberg, 1983. ISBN 978-3-540-38682-7.

[70] Intel Corp. *Intel Math Kernel Library*, 2019. `https://software.intel.com/en-us/mkl`.

[71] Intel Corp. *Threading Building Blocks*, 2019. `https://www.threadingbuildingblocks.org`.

[72] Haitao Jiang, Binhai Zhu, Daming Zhu und Hong Zhu. Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23(4):519–527, May 2012. ISSN 1573-2886. doi: 10.1007/s10878-010-9370-2.

[73] Minghui Jiang, Ying Xu und Binhai Zhu. Protein structure-structure alignment with discrete Fréchet distance. In *5th Asia-Pacific Bioinformatics Conference, Hong Kong, China*, S. 131–141, 2007. `http://www.comp.nus.edu.sg/%7Ewongls/psZ/apbc2007/apbc162a.pdf`.

[74] J. Mark Keil und Jack Snoeyink. On the time bound for convex decomposition of simple polygons. *Int. J. Comput. Geometry Appl.*, 12:181–192, 1998.

[75] Khronos OpenCL Working Group. *The OpenCL Specification*, 1.2 edition, 2012. `https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf`.

[76] Steffen Mecke und Dorothea Wagner. Solving geometric covering problems by data reduction. In Susanne Albers und Tomasz Radzik, Hrsg., *Algorithms – ESA 2004*, S. 760–771. Springer Berlin Heidelberg, 2004.

[77] Steffen Mecke, Anita Schöbel und Dorothea Wagner. Station Location - Complexity and Approximation. In Leo G. Kroon und Rolf H. Möhring, Hrsg., *5th Workshop on Algorithmic Methods and Models for Optimization of Railways*, volume 2 of *OpenAccess Series in Informatics (OASIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. ISBN 978-3-939897-00-2. doi: 10.4230/OASIcs.ATMOS.2005.661.

[78] Steffen Mecke, Dorothea Wagner und Michael Dom. Set cover with almost consecutive ones property. In *Encyclopedia of Algorithms*, S. 832–834. Springer, 2008.

[79] Nimrod Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, October 1983. ISSN 0004-5411. doi: 10.1145/2157.322410.

[80] Amir Nayyeri und Anastasios Sidiropoulos. Computing the Fréchet distance between polygons with holes.In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi und Bettina Speckmann, Hrsg., *Automata, Languages, and Programming*, S. 997–1009. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-47672-7.

[81] Amir Nayyeri und Hanzhong Xu. On computing the Fréchet distance between surfaces. In *32nd International Symposium on Computational Geometry, Boston, MA, USA*, S. 55:1–55:15, 2016. doi: 10.4230/LIPIcs.SoCG.2016.55.

[82] Amir Nayyeri und Hanzhong Xu. On the decidability of the Fréchet distance between surfaces. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, New Orleans, LA, USA*, S. 1109–1120, 2018. doi: 10.1137/1.9781611975031.72.

[83] Eike Neumann. On the computability of the Fréchet distance of surfaces in the bit-model of real computation. *CoRR*, abs/1711.02161, 2017.

[84] Cedric Nugteren. *Tuned OpenCL BLAS*. https://github.com/CNugteren/CLBlast.

[85] NVIDIA Corp. *OpenCL Best Practices Guide*, 2009. https://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf.

[86] NVIDIA Corp. *OpenCL Programming Guide for the CUDA Architecture*, 2009. https://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingguide.pdf".

[87] Chansu Park, Ji-won Park, Sewon Park, Dongseong Seon und Martin Ziegler. Computable operations on compact subsets of metric spaces with applications to Fréchet distance and shape optimization. *CoRR*, abs/1701.08402, 2017.

[88] Qt. *Qt Cross-platform software development*, 5.10.1 edition. https://www.qt.io/.

[89] Günter Rote. Computing the Fréchet distance between piecewise smooth curves. *Computational Geometry*, 37(3):162 – 174, 2007. ISSN 0925-7721. doi: https://doi.org/10.1016/j.comgeo.2005.01.004. Special Issue on the 20th European Workshop on Computational Geometry.

[90] Nikolaus Ruf und Anita Schöbel. Set covering with almost consecutive ones property. *Discrete Optimization*, 1(2):215 – 228, 2004. ISSN 1572-5286. doi: https://doi.org/10.1016/j.disopt.2004.07.002.

[91] Marcus Schaefer. Complexity of some geometric and topological problems. In David Eppstein und Emden R. Gansner, Hrsg., *Graph Drawing*, S. 334–344. Springer Berlin Heidelberg, 2010.

[92] R. Sriraghavendra, K. Karthik und C. Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In *Ninth International Conference on Document Analysis and Recognition*, volume 1, S. 461–465, Sept 2007. doi: 10.1109/ICDAR.2007.4378752.

[93] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969. http://eudml.org/doc/131927.

[94] Andreas Weise. *Parallel Computation of the Fréchet Distance of Polygonal Curves on the GPU*. Masterarbeit, Freie Universität Berlin, 2012.

[95] Tim Wylie und Binhai Zhu. Protein chain pair simplification under the discrete Fréchet distance. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 10(6):1372–1383, Nov 2013. ISSN 1545-5963. doi: 10.1109/TCBB.2013.17.