41. Bundeswettbewerb Informatik

Aufgabe 1: Weniger krumme Touren

Florian Bange
Teilnahme-ID: 64810

17. April 2023

Inhaltsverzeichnis

0. Einleitung

Dieses Dokument beinhaltet meine Dokumentation der ersten Aufgabe[1] der zweiten Runde des 41. Bundeswettbewerbs Informatik[2] aus dem Jahr 2023.
Das gegebene Problem besteht darin, fuer eine Menge an Orten im 2-dimensionalen Raum eine Route zu finden, sodass

1. fuer alle Abbiegewinkel α stets gilt, dass $\alpha \leq 90°$, und
2. die Laenge der Strecke, die zurueckzulegen ist, moeglichst klein ist.

Im Folgenden wird die erste Bedingung als „Abbiegewinkelbedingung“ bezeichnet.
Offensichtlich hat dieses Problem starke Aehnlichkeit zum Traveling Salesman Problem (TSP)[3], sodass im Folgenden Loesungsvorschlaege gegeben werden, die ahenlich zu denen des TSPs sind. Auszerdem kann man deswegen vermuten, dass das gegebene Problem NP-schwer ist. Deswegen werden hier neben exakten Loesungsverfahren haubsaechtlich heuristische Verfahren vorgestellt.

Um das Problem zu loesen, wird es zunaechst mit Hilfe der Vektorraeume $\mathbb{R}^2$ und $\mathbb{R}_2$ im Kapitel 1 mathematisch modelliert und praezise definiert. Dabei werden Orte als Punkte im 2-dimensionalen Raum $\mathbb{R}^2$ dargestellt und Verbindungen zwischen ihnen als Richtungsvektoren der Menge $\mathbb{R}_2$. Danach wird eine Verallgemeinerung des Problems mit n-dimensionalen Punkten im Vektorraum $\mathbb{R}^n$ vorgestellt (Kapitel 2).
Anschlieszend wird in Kapitel 3 auf die Loesbarkeit des Problems eingegangen, wobei die Frage „Existiert immer eine Loesung?“ behandelt wird.
Daraufhin werden in Kapitel 4 zunaechst exakte Loesungsverfahren und anschlieszend die heuritischen Verfahren vorgestellt, die das verallgemeinerte Problem loesen koennen, sodass implizit auch das originale Problem geloest wird. Dabei werden (halb-)formale Korrektheitsbegruendungen fuer die Algorithmen gegeben. Diese Loesungsvorschlaege werden daraufhin in Kapitel 5 bezueglich ihrer Zeit- und Platzkomplexitaet analysiert und verglichen.
Im Kapitel 6 werden danach Erweiterungen des Problems vorgestellt. Dabei wird das Problem zunaechst weiter verallgemeinert, indem die Abbiegewinkelbedingung auf beliebige Winkel erweitert wird. Zudem werden zwei weitere Erweiterungen des Problems vorgestellt, wobei mehrere Reisende (oder Piloten) die Orte besuchen (oder ueberfliegen) muessen bzw. einer dies in bestimmten Zeitraeumen pro Ort tun muss. Auf diese Probleme wird anschlieszend kurz eingegange.

Weiter wird in Kapitel 7 erlaeutert, wie die zuvor beschriebenen Loesungsvorschlaege in ein Java-Programm umgesetzt wurden. Schlussendlich werden fuer das originale Problem zahlreiche Beispiele, bestehend aus denen der BwInf.-Website und eigenen, erlaeutert, die zeigen, dass das Programm tatsaechlich funktioniert (Kapitel 8). Zuletzt werden die wichtigsten Teile des in Kapitel 9 Java-Quellcodes erlaeutert.

[1] Fuer die Aufgaben siehe: `https://bwinf.de/fileadmin/bundeswettbewerb/41/aufgaben412.pdf`
[2] Siehe 41. BwInf. Runde 2: `https://bwinf.de/bundeswettbewerb/41/2/`
[3] Traveling Salesman Problem: `https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`

1. Modellierung und Definition des originalen Problems - Punkte im $\mathbb{R}^2$

Um das Problem der ersten Aufgabe theoretisch betrachten zu koennen, wird es nun mathematisch modelliert und formal definiert. Dabei werden Punkte im 2-dimensionalen Raum $\mathbb{R}^2$ genutzt, um Orte darzustellen. Im Folgenden soll zwischen diesen Punkten und Verbindungsvektoren zwischen ihnen unterschieden werden. Dazu werden zwei verschiedene Vektorraeume und Notationen eingefuehrt - Punkte und Vektoren (s. Kapitel 1.1). Anschlieszend wird auf den Winkel zwischen zwei Vektoren bzw. drei Punkten eingegangen (s. Kapitel 1.2). Zuletzt werden Abbiegewinkel definiert, das gestellte Problem praezise definiert und die Abbiegewinkelbedingung vereinfacht (s. Kapitel 1.3).

1.1. Punkte und Vektoren im 2-dimensionalen Raum

Definition 1: 2-dimensionaler Raum $\mathbb{R}^2$

Es sei

$$\mathbb{R}^2 \coloneqq \{(x,\ y)\ :\ x,\ y \in \mathbb{R}\}.$$

Dabei ist $(x,\ y) \in \mathbb{R}^2$ ein 2-Tupel.

Bemerkung 2: Komponenten I

Es sei

$$P \coloneqq (x,\ y) \in \mathbb{R}^2,$$

dann wird x erste Komponente, und y zweite Komponente von P genannt.

Bemerkung 3: Punkte des 2-dimensionaler Raums $\mathbb{R}^2$

Zwar ist $\mathbb{R}^2$ im eigentlichen Sinne ein Vektorraum, sofern man die Vektoraddition und Skalarmultiplikation sinnvoll definieren wuerde, sodass man Elemente dieser Menge auch Vektoren nennen koennte, dennoch wird im Folgenden ein Element

$$P \coloneqq (x,\ y) \in \mathbb{R}^2$$

als Punkt im 2-dimensionalen Raum aufgefasst und mit Groszbuchstaben bezeichnet.
Dabei entspricht die erste Komponente $x \in \mathbb{R}$ der x-Koordinate und die zweite Komponente der y-Koordinate in einem kartesischen Koordinatensystem.

Definition 4: Reeller Vektorraum $\mathbb{R}_2$

Es sei

$$\mathbb{R}_2 \coloneqq \left\{ \begin{pmatrix} x \\ y \end{pmatrix}\ :\ x,\ y \in \mathbb{R} \right\},$$

wobei

$$\begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}_2$$

ein Spaltenvektor ist.

Bemerkung 5: Komponenten II

Es sei

$$\vec{v} \coloneqq \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}_2$$

dann wird, analog zu $(x,\ y) \in \mathbb{R}^2$, x erste Komponente, und y zweite Komponente von $\vec{v}$ genannt.

Bemerkung 6: Richtungsvektoren

Sowie $\mathbb{R}^2$, ist auch ein $\mathbb{R}_2$ Vektorraum, sodass man Elemente dieser Menge Vektoren nennen kann. Im Folgenden wird

$$\vec{v} \coloneqq \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}_2$$

als Pfeil mit Laenge (Norm) und Richtung aufgefasst und mit der ueblichen Schreibweise $\vec{s}$, $\vec{v}$, ... etc. bezeichnet. Fuer ein $\vec{v} \in \mathbb{R}_2$ werden im Folgenden die Bezeichnungen Vektor, Richtungsvektor und Pfeil synonym genutzt.
Weiter kann man einen Richtungsvektor zwar als Verbindung zwischen einem Anfangspunkt und einem Endpunkt auffassen (s. Verbindungsvektor: Def. 13), dennoch hat ein beliebiger Vektor $\vec{v} \in \mathbb{R}_2$ keinen bestimmten Anfangspunkt.

Definition 7: Vektoraddition und Skalarmultiplikation fuer Vektoren des $\mathbb{R}_2$

Nun werden die Vektoraddition $\oplus$ und Skalarmultiplikation $\odot$ fuer Elemente des $\mathbb{R}_2$ sinnvoll definiert, sodass man $\mathbb{R}_2$ nach der Definition[4] als $\mathbb{R}$-Vektorraum auffassen kann:

$$\oplus \ : \ \mathbb{R}_2 \times \mathbb{R}_2 \ni (\vec{v},\ \vec{w}) = (\begin{pmatrix} x_1 \\ y_1 \end{pmatrix},\ \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}) \longmapsto \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix} \in \mathbb{R}_2$$

$$\odot \ : \ \mathbb{R} \times \mathbb{R}_2 \ni (\alpha,\ \vec{v}) = (\alpha,\ \begin{pmatrix} x \\ y \end{pmatrix}) \longmapsto \begin{pmatrix} \alpha \cdot x \\ \alpha \cdot y \end{pmatrix} \in \mathbb{R}_2$$

Beispiel 8: Vektoraddition und Skalarmultiplikation

Es seien $\vec{u},\ \vec{v} \in \mathbb{R}_2$ mit

$$\vec{u} = \begin{pmatrix} 5 \\ 2 \end{pmatrix},\ \vec{v} = \begin{pmatrix} 2 \\ 5 \end{pmatrix}.$$

Dann ist

$$\vec{u} \oplus \vec{v} = \begin{pmatrix} 5 \\ 2 \end{pmatrix} \oplus \begin{pmatrix} 2 \\ 5 \end{pmatrix} = \begin{pmatrix} 7 \\ 7 \end{pmatrix}.$$

Weiter sei $\vec{w} \in \mathbb{R}_2$ mit

$$\vec{w} = \begin{pmatrix} 2 \\ 2 \end{pmatrix},$$

dann ist

$$-\frac{3}{2} \odot \vec{w} = \begin{pmatrix} -3 \\ -3 \end{pmatrix}$$

Bemerkung 9: Nullvektor und das additiv inverse Element

Das Neutralelement bezgl. der Addition $\oplus$ der abelschen Gruppe[5] $(\mathbb{R}_2,\ \oplus)$ wird als

$$\vec{0} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

notiert und Nullvektor genannt.
Weiter sei $\vec{v} \in \mathbb{R}_2$ mit

$$\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix},$$

dann ist

$$-\vec{v} = (-1) \odot \vec{v} = \begin{pmatrix} -x \\ -y \end{pmatrix}$$

das inverse Element von $\vec{v}$, da

$$(-\vec{v}) \oplus \vec{v} = \begin{pmatrix} -x \\ -y \end{pmatrix} \oplus \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} (-x) + x \\ (-y) + y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \vec{0}$$

gilt.

[4] Siehe auch `https://de.wikipedia.org/wiki/Vektorraum` (Definition)
[5] Siehe `https://de.wikipedia.org/wiki/Abelsche_Gruppe`

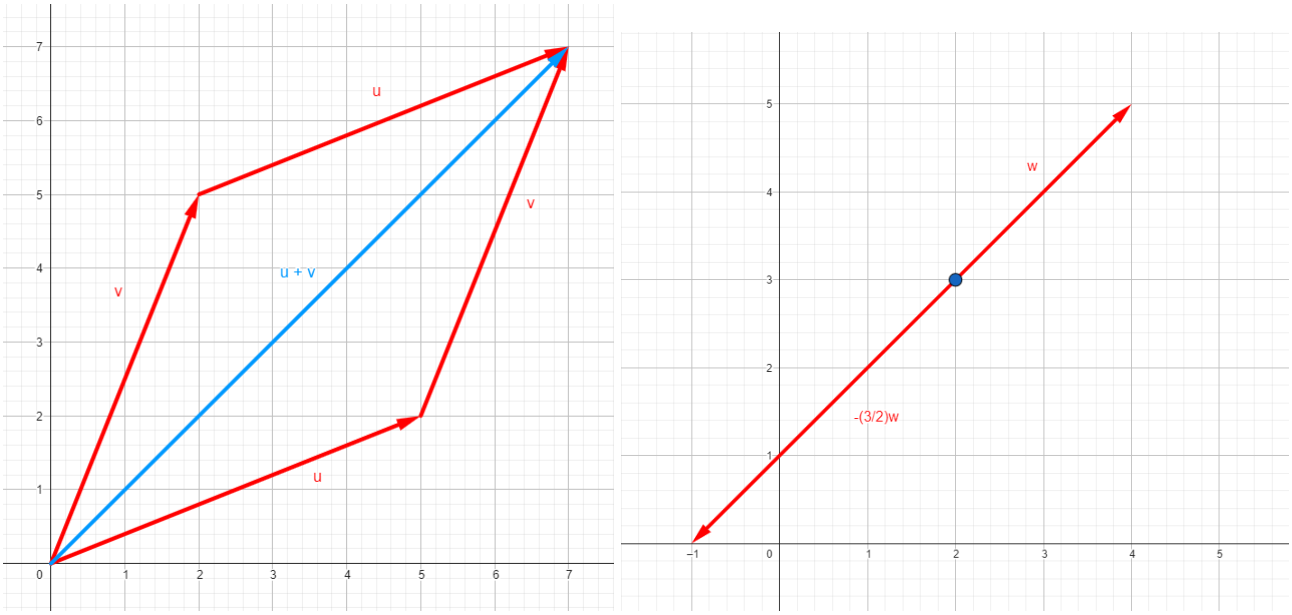

Abbildung 1: Klassische Darstellung der Vektoraddition und Skalarmultiplikation

Definition 10: Norm (Laenge eines Richtungsvektors)

Es sei

$$\vec{v} \coloneqq \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}_2,$$

dann wird die Laenge des Vektors $\vec{v}$ mit $\|\vec{v}\|$ bezeichnet und ist wie folgt definiert:

$$\|\vec{v}\| \coloneqq \sqrt{x^2 + y^2}$$

Bemerkung 11: Laenge eines Richtungsvektors

Es sei $\vec{v} \in \mathbb{R}_2$ mit

$$\vec{v} \coloneqq \begin{pmatrix} x \\ y \end{pmatrix},$$

dann laesst sich der Vektor so darstellen, dass ein rechtwinkliges Dreieck entsteht, bei dem die Katheten die Laengen x und y haben und $\vec{v}$ die Hypotenuse ist.
Somit ergibt sich durch den Satz des Pythagoras, dass fuer die Laenge $\|\vec{v}\|$ des Vektors $\vec{v}$ gilt, dass

$$\|\vec{v}\| = \sqrt{x^2 + y^2}$$

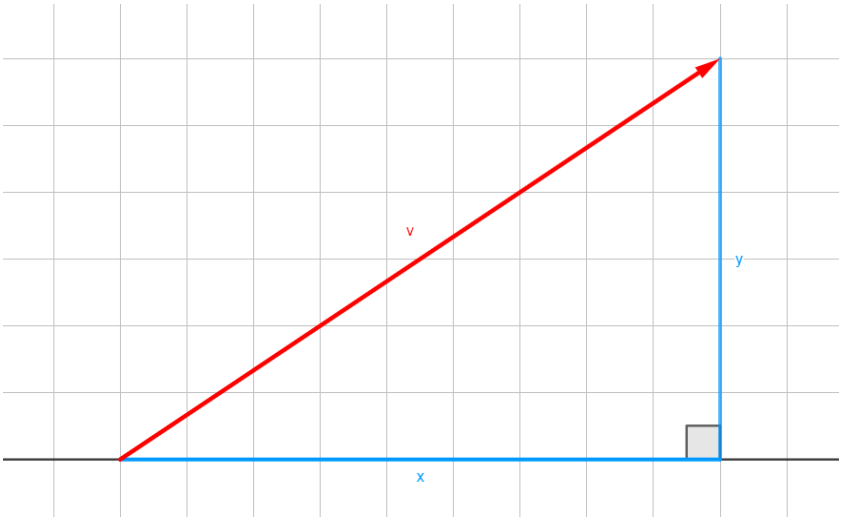

Abbildung 2: Laenge des Vektors $\vec{v}$ nach dem Satz des Pythagoras

Definition 12: Skalarprodukt von Richtungsvektoren

Es seien $\vec{v},\ \vec{w} \in \mathbb{R}_2$ mit

$$\vec{v} \coloneqq \begin{pmatrix} v_1 \\ v_2 \end{pmatrix},\ \vec{w} \coloneqq \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \in \mathbb{R}_2,$$

dann wird das Skalarprodukt von $\vec{v}$ und $\vec{w}$ mit $\vec{v} \circ \vec{w}$ notiert und ist wie folgt definiert:

$$\vec{v} \circ \vec{w} \coloneqq v_1 \cdot w_1 + v_2 \cdot w_2$$

Definition 13: Verbindungsvektor zwischen Punkten des $\mathbb{R}^2$

Es seien $P,\ Q \in \mathbb{R}^2$ Punkte der 2-dimensionalen Ebene mit

$$P = (p_1,\ p_2),\ Q = (q_1,\ q_2).$$

Nun wird der eindeutige Richtungsvektor definiert, der den Punkt P mit dem Punkt Q verbindet und bei P beginnt. Dieser Richtungsvektor wird auch als $\overrightarrow{PQ}$ notiert.

$$\overrightarrow{PQ} \coloneqq \begin{pmatrix} q_1 - p_1 \\ q_2 - p_2 \end{pmatrix} \in \mathbb{R}_2$$

Definition 14: Ortsvektoren

Es sei $O \coloneqq (0,\ 0)$ der Punkt, der den Ursprung des kartesischen Koordinatensystems darstellt, und

$$P = (p_1,\ p_2) \in \mathbb{R}^2$$

beliebig.
Dann nennt man den Verbindungsvektor $\overrightarrow{OP}$ auch Ortsvektor des Punktes P, dabei sind die Komponenten des Punktes P identisch mit denen des Ortvektors $\overrightarrow{OP}$.

$$\overrightarrow{OP} = \begin{pmatrix} p_1 - 0 \\ p_2 - 0 \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \in \mathbb{R}_2$$

Beispiel 15: Verbindungsvektor und Ortsvektor

Es seien $P,\ Q \in \mathbb{R}^2$ mit

$$P = (1,\ 3),\ Q = (7,\ 1).$$

Dann ist

$$\overrightarrow{QP} = \begin{pmatrix} 1 - 7 \\ 3 - 1 \end{pmatrix} = \begin{pmatrix} -6 \\ 2 \end{pmatrix}$$

der Verbindungsvektor von Q nach P.

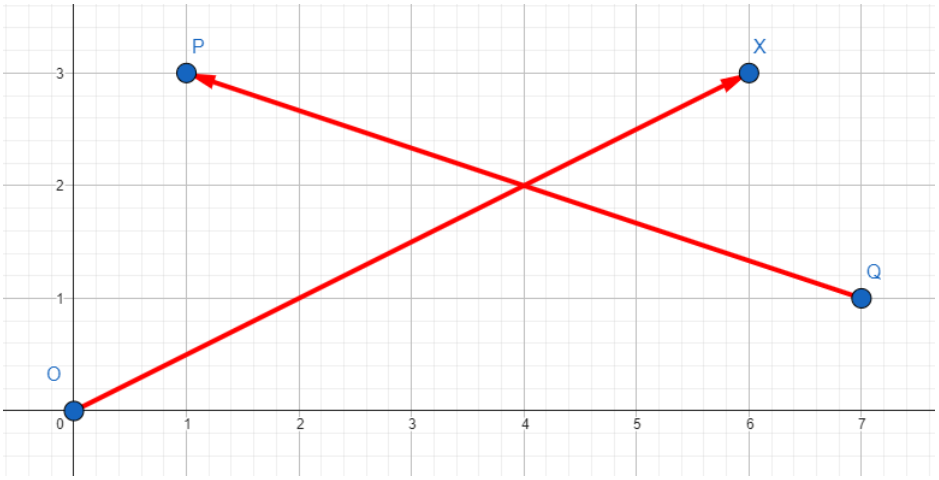

Abbildung 3: Verbindungsvektor $\overrightarrow{QP}$ und Ortsvektor $\overrightarrow{OX}$

Weiter sei $X = (6, 3) \in \mathbb{R}^2$.
Dann ist der zu X gehoerende Ortsvektor

$$\overrightarrow{OX} = \begin{pmatrix} 6 \\ 3 \end{pmatrix}.$$

Bemerkung 16: Exkurs - Affine Raeume

Eine genauere Moeglichkeit, diesen Zusammenhang zwischen Punkten und Vektoren zu definieren, sind Affine Raeume[6].
Dabei gibt es eine Menge A an Punkten (hier: $\mathbb{R}^2$), einen K-Vektorraum V (hier: $\mathbb{R}_2$), sowie eine Abbildung $\tau : A \times A \to V$, die zwei Punkten $P,\ Q \in A$ einen Verbindungsvektor $\overrightarrow{PQ}$ zuordnet, sodass bestimmte Bedingungen erfuellt sind.

1.2. Winkel zwischen Vektoren und Punkten im 2-dimensionalen Raum

Definition 17: Winkel zwischen Richtungsvektoren

Es seien $\vec{v},\ \vec{w} \in \mathbb{R}_2$ mit $\vec{v} \neq \vec{0}$ und $\vec{w} \neq \vec{0}$, wobei beide Vektoren denselben Anfangspunkt haben, so wird der Winkel, der zwischen den Vektoren $\vec{v}$ und $\vec{w}$ eingeschlossen ist, als

$$\measuredangle(\vec{v},\ \vec{w})$$

bezeichnet.
Dabei ist immer ein Winkel gemeint, fuer den gilt, dass

$$\begin{aligned} &\measuredangle(\vec{v},\ \vec{w}) \in [0°;\ 180°] = [0;\ \pi] \\ \iff &0° = 0 \leq \measuredangle(\vec{v},\ \vec{w}) \leq 180° = \pi \end{aligned}$$

Bemerkung 18: Winkel zwischen Richtungsvektoren

Es seien $\vec{v} \neq \vec{0},\ \vec{w} \neq \vec{0} \in \mathbb{R}_2$.
Waere einer der beiden Vektoren der Nullvektor $\vec{0}$, so waere $\measuredangle(\vec{v},\ \vec{w})$ nicht definiert.
Ansonsten gilt offensichtlich

$$\measuredangle(\vec{v},\ \vec{w}) = \measuredangle(\vec{w},\ \vec{v}),$$

da der Winkel, der zwischen $\vec{v}$ und $\vec{w}$ eingeschlossen ist, dem Winkel, der zwischen $\vec{w}$ und $\vec{v}$ eingeschlossen ist, entspricht.
Auszerdem gilt, dass der Winkel $\measuredangle(\vec{v},\ \vec{w})$ unabhaengig vom gemeinsamen Anfangspunkt der Vektoren ist.

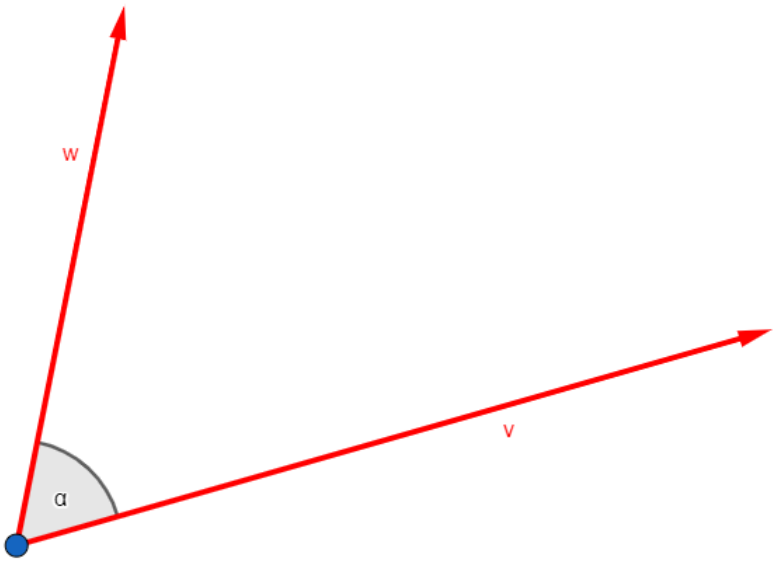

Abbildung 4: Winkel $\alpha = \measuredangle(\vec{w},\ \vec{v})$

[6] Siehe auch `https://de.wikipedia.org/wiki/Affiner_Raum`

Satz 19: Vektor-Winkel-Formel

Es seien $\vec{v} \neq \vec{0},\ \vec{w} \neq \vec{0} \in \mathbb{R}_2$ mit

$$\vec{v} := \begin{pmatrix} v_1 \\ v_2 \end{pmatrix},\ \vec{w} := \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \in \mathbb{R}_2,$$

und einem gemeinsamen Anfangspunkt.
Weiter sei $\alpha := \measuredangle(\vec{v},\ \vec{w})$ der Winkel, der zwischen den Vektoren $\vec{v}$ und $\vec{w}$ eingeschlossen ist, dann ist

$$\vec{v} \circ \vec{w} = \cos(\alpha) \cdot \|\vec{v}\| \cdot \|\vec{w}\|$$
$$\iff \cos(\alpha) = \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}$$

Nach Definition 10 und 12:

$$\cos(\alpha) = \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}}$$

Beweis:

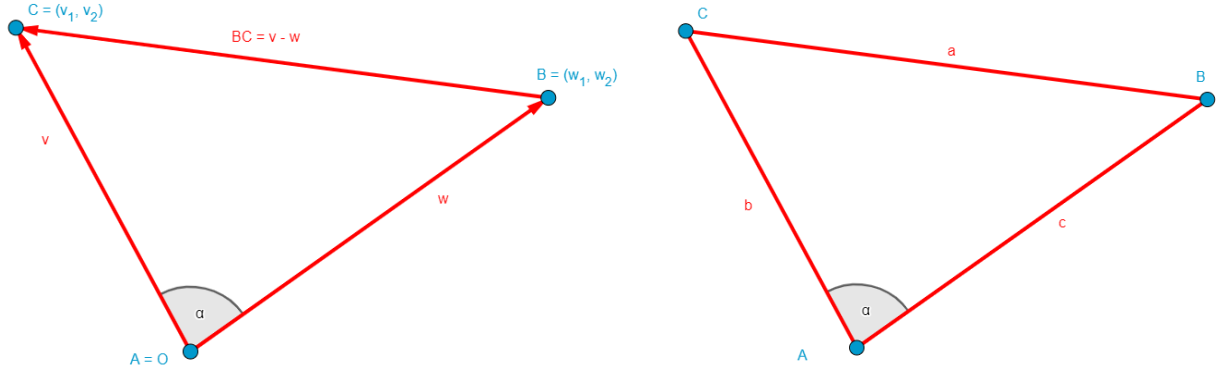

Abbildung 5: Grafik zum Beweis

Es sei o.B.d.A. der gemeinsame Anfangspunkt von $\vec{v}$ und $\vec{w}$ der Ursprung $O = (0,\ 0)$ (vergl. Bemerkung 18: der Winkel ist unabhaengig vom Anfangspunkt).
Somit ist der Endpunkt des Vektors $\vec{v}$ der Punkt $(v_1,\ v_2)$, sowie der Endpunkt des Vektors $\vec{w}$ der Punkt $(w_1,\ w_2)$.
Nun sei $\triangle ABC$ ein Dreieck, wobei $A := O$, $B := (w_1,\ w_2)$ und $C := (v_1,\ v_2)$. Dann gilt fuer die Laengen b und c der Seiten des Dreiecks:

$$b = \|\vec{v}\|,\ c = \|\vec{w}\|.$$

Weiter gilt fuer den Verbindungsvektor von B und C nach Definition 12:

$$\|\overrightarrow{BC}\| = \left\| \begin{pmatrix} v_1 - w_1 \\ v_2 - w_2 \end{pmatrix} \right\| = \|\vec{v} - \vec{w}\|$$

Somit ist

$$a = \|\vec{v} - \vec{w}\|$$

Weiter sei α der Winkel $\measuredangle BAC$ bzw. der Winkel zwischen den Vektoren $\vec{v}$ und $\vec{w}$:

$$\alpha = \measuredangle BAC = \measuredangle(\vec{v},\ \vec{w})$$

Nach dem Kosinussatz[7] gilt:

$$a^2 = b^2 + c^2 - 2 \cdot \cos(\alpha) \cdot b \cdot c$$

Daraus folgt, dass

$$\|\vec{v} - \vec{w}\|^2 = \|\vec{v}\|^2 + \|\vec{w}\|^2 - 2 \cdot \cos(\alpha) \cdot \|\vec{v}\| \cdot \|\vec{w}\|.$$

[7] Siehe auch `https://de.wikipedia.org/wiki/Kosinussatz`

Auszerdem gilt:

$$\begin{aligned}
&\|\vec{v}-\vec{w}\|^2 \\
&= \left(\sqrt{(v_1-w_1)^2+(v_2-w_2)^2}\right)^2 \\
&= |(v_1-w_1)^2+(v_2-w_2)^2| && \mid (v_1-w_1)^2+(v_2-w_2)^2 \geq 0 \\
&= (v_1^2+w_1^2-2\cdot v_1w_1)+(v_2^2+w_2^2-2\cdot v_2w_2) \\
&= (v_1^2+v_2^2)+(w_1^2+w_2^2)-2\cdot(v_1w_1+v_2w_2) && \mid \text{Def. 10, 12} \\
&= \|\vec{v}\|^2+\|\vec{w}\|^2-2\cdot\vec{v}\circ\vec{w}
\end{aligned}$$

Somit ist

$$\begin{aligned}
&\|\vec{v}\|^2+\|\vec{w}\|^2-2\cdot\cos(\alpha)\cdot\|\vec{v}\|\cdot\|\vec{w}\| = \|\vec{v}\|^2+\|\vec{w}\|^2-2\cdot\vec{v}\circ\vec{w} && \mid -\|\vec{v}\|^2-\|\vec{w}\|^2 \\
\iff &-2\cdot\cos(\alpha)\cdot\|\vec{v}\|\cdot\|\vec{w}\| = -2\cdot\vec{v}\circ\vec{w} && \mid :\ (-2) \\
\iff &\cos(\alpha)\cdot\|\vec{v}\|\cdot\|\vec{w}\| = \vec{v}\circ\vec{w}
\end{aligned}$$

□

Bemerkung 20: Kosinus und Arkuskosinus

Interpretiert man den Kosinus als Abbildung

$$\cos\ :\ [0;\ \pi] \to [-1;\ 1]$$

bzw.

$$\cos\ :\ [0°;\ 180°] \to [-1;\ 1]$$

dann ist cos auf dem angegebenen Definitionsbereich bijektiv und somit invertierbar. Auszerdem ist die Abbildung dann streng monoton fallend.
Die Umkehrfunktion zu cos ist arccos[8]:

$$\arccos\ :\ [-1;\ 1] \to [0;\ \pi]$$

bzw.

$$\arccos\ :\ [-1;\ 1] \to [0°;\ 180°]$$

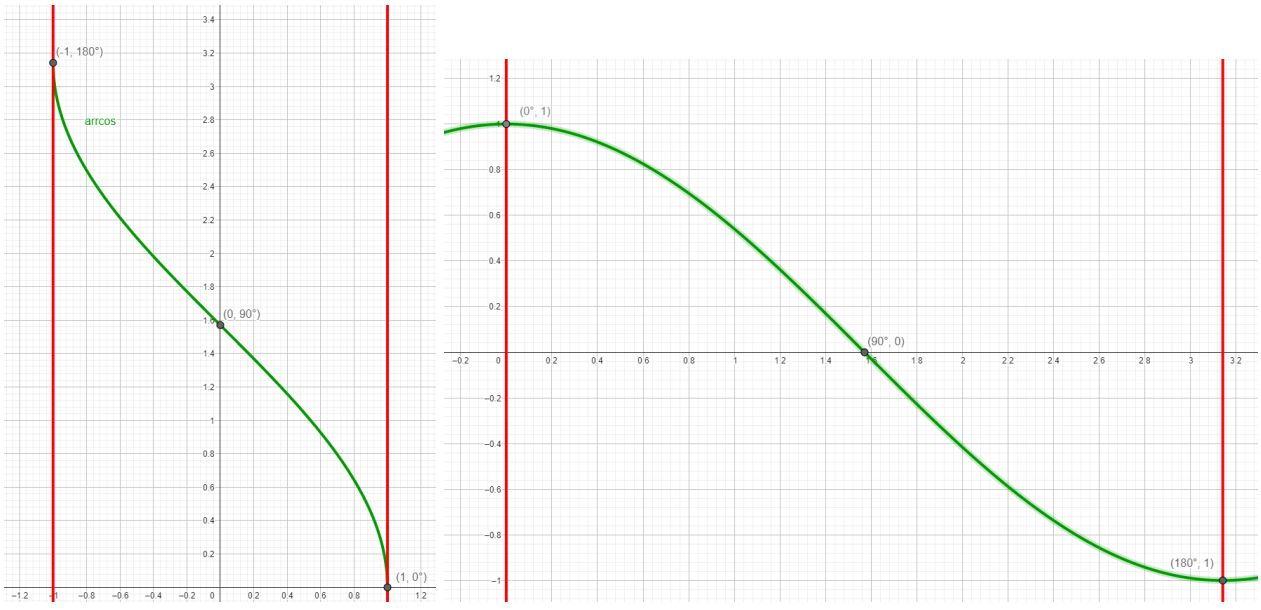

Abbildung 6: Graph des Arrkuskosinus (links) und Kosinus (rechts)

Somit gilt fuer $\cos(\alpha)$ mit $\alpha \in [0;\ \pi]$, dass

$$\begin{aligned}
&\cos(\alpha) \in [-1;\ 1] \\
\iff &-1 \leq \cos(\alpha) \leq 1
\end{aligned}$$

[8] Siehe auch `https://de.wikipedia.org/wiki/Sinus_und_Kosinus` (Umkehrfunktion)

Auch gilt fuer $\arccos(x)$ mit $x \in [-1;\ 1]$, dass

$$\begin{aligned} &\arccos(x) \in [0;\ \pi] \\ \iff &0 \leq \arccos(x) \leq \pi \end{aligned}$$

Wichtige Werte des Kosinus (und Arrkuskosinus durch Umkehrung) fuer Winkel $\alpha \in [0; \pi]$:

1. $\cos(0°) = \cos(0) = 1 \iff \arccos(1) = 0° = 0$
2. $\cos(45°) = \cos(\frac{\pi}{4}) = \frac{1}{\sqrt{2}} \iff \arccos(\frac{1}{\sqrt{2}}) = 45° = \frac{\pi}{4}$
3. $\cos(60°) = \cos(\frac{\pi}{3}) = \frac{1}{2} \iff \arccos(\frac{1}{2}) = 60° = \frac{\pi}{3}$
4. $\cos(90°) = \cos(\frac{\pi}{2}) = 0 \iff \arccos(0) = 90° = \frac{\pi}{2}$
5. $\cos(180°) = \cos(\pi) = -1 \iff \arccos(-1) = 180° = \pi$

Satz 21:

Es seien $\vec{v},\ \vec{w} \in \mathbb{R}_2$ mit $\vec{v} \neq \vec{0}$ und $\vec{w} \neq \vec{0}$ und

$$\vec{v} \coloneqq \begin{pmatrix} v_1 \\ v_2 \end{pmatrix},\ \vec{w} \coloneqq \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}.$$

Weiter sei $\alpha \coloneqq \measuredangle(\vec{v},\ \vec{w})$.
Damit Satz 19

$$\cos(\alpha) = \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} = \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}}$$

unter der soeben eingefuehrten Interpretation (bzw. Definiton) des Kosinus als Abbildung

$$\cos\ :\ [0;\ \pi] \to [-1;\ 1]$$

(vergl. Bemerkung 20) Sinn ergibt, muss gelten, dass

$$-1 \leq \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} = \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}} \leq 1$$

In der Tat ist dies der Fall.

Beweis:
Es seien $\vec{v},\ \vec{w} \in \mathbb{R}_2$ mit $\vec{v} \neq \vec{0}$ und $\vec{w} \neq \vec{0}$ und

$$\vec{v} \coloneqq \begin{pmatrix} v_1 \\ v_2 \end{pmatrix},\ \vec{w} \coloneqq \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}.$$

Dann ist

$$\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2} \neq 0$$

Weiter gilt im Allgemeinen

$$\sqrt{w_1^2 + w_2^2} \geq 0 \tag{1}$$

1. Fall: $v_1 \cdot w_1 + v_2 \cdot w_2 \leq 0$.
   Dann gilt wegen Ungleichung 1 auch

   $$\frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}} \leq 0 < 1$$

   Z.z. bleibt

   $$-1 \leq \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}}$$

   Trivialerweise gilt immer:

   $$(v_1 w_2 - v_2 w_1)^2 \geq 0$$

Daraus folgt der Rest des ersten Teils des Beweises:

$$\begin{aligned}
&(v_1w_2 - v_2w_1)^2 \geq 0 \\
\iff &(v_1w_2)^2 + (v_2w_1)^2 - 2 \cdot (v_1w_2) \cdot (v_2w_1) \geq 0 && | + (v_1w_1)^2 + (v_2w_2)^2 + 2 \cdot (v_1w_1) \cdot (v_2w_2) \\
\iff &(v_1w_1)^2 + (v_2w_2)^2 + (v_1w_2)^2 + (v_2w_1)^2 \geq (v_1w_1)^2 + (v_2w_2)^2 + 2 \cdot (v_1w_1) \cdot (v_2w_2) \\
\iff &(v_1^2 + v_2^2) \cdot (w_1^2 + w_2^2) \geq (v_1w_1 + v_2w_2)^2 && | \ \sqrt{\dots} \\
\iff &\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2} \geq \sqrt{(v_1w_1 + v_2w_2)^2} && | \ (v_1 \cdot w_1 + v_2 \cdot w_2) \leq 0 \\
\iff &\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2} \geq -(v_1w_1 + v_2w_2) && | \cdot (-1) \\
\iff &-\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2} \leq v_1w_1 + v_2w_2 && | : \left(\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}\right) \\
\iff &-1 \leq \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}}
\end{aligned}$$

2. Fall: $0 \leq v_1 \cdot w_1 + v_2 \cdot w_2$
   Dann gilt wegen Ungleichung 1 auch

$$-1 < 0 \leq \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}}$$

   Z.z. bleibt

$$\frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}} \leq 1$$

   Trivialerweise gilt immer:

$$0 \leq (v_1w_2 - v_2w_1)^2$$

   Daraus folgt der Rest des zweiten Teils des Beweises, analog zum ersten Teil:

$$\begin{aligned}
&0 \leq (v_1w_2 - v_2w_1)^2 \\
\iff &0 \leq (v_1w_2)^2 + (v_2w_1)^2 - 2 \cdot (v_1w_2) \cdot (v_2w_1) && | + (v_1w_1)^2 + (v_2w_2)^2 + 2 \cdot (v_1w_1) \cdot (v_2w_2) \\
\iff &(v_1w_1)^2 + (v_2w_2)^2 + 2 \cdot (v_1w_1) \cdot (v_2w_2) \leq (v_1w_1)^2 + (v_2w_2)^2 + (v_1w_2)^2 + (v_2w_1)^2 \\
\iff &(v_1w_1 + v_2w_2)^2 \leq (v_1^2 + v_2^2) \cdot (w_1^2 + w_2^2) \\
\iff &\sqrt{(v_1w_1 + v_2w_2)^2} \leq \sqrt{(v_1^2 + v_2^2)} \cdot \sqrt{(w_1^2 + w_2^2)} && | \ (v_1 \cdot w_1 + v_2 \cdot w_2) \leq 0 \\
\iff &v_1w_1 + v_2w_2 \leq \sqrt{(v_1^2 + v_2^2)} \cdot \sqrt{(w_1^2 + w_2^2)} && | : \left(\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}\right) \\
\iff &\frac{v_1w_1 + v_2w_2}{\sqrt{(v_1^2 + v_2^2)} \cdot \sqrt{(w_1^2 + w_2^2)}} \leq 1
\end{aligned}$$

Aus Fall 1 und Fall 2 folgt, dass fuer alle $\vec{v},\ \vec{w} \in \mathbb{R}_2$ mit $\vec{v} \neq \vec{0}$ und $\vec{w} \neq \vec{0}$ gilt, dass

$$-1 \leq \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} = \frac{v_1 \cdot w_1 + v_2 \cdot w_2}{\sqrt{v_1^2 + v_2^2} \cdot \sqrt{w_1^2 + w_2^2}} \leq 1$$

□

Folgerung 22: Berechnung des Winkels zwischen zwei Richtungsvektoren

Es seien $\vec{v} \neq \vec{0},\ \vec{w} \neq \vec{0} \in \mathbb{R}_2$ Richtungsvektoren mit einem gemeinsamen Anfangspunkt.
Weiter sei $\alpha \coloneqq \measuredangle(\vec{v},\ \vec{w})$ der Winkel, der zwischen $\vec{v}$ und $\vec{w}$ eingeschlossen ist.
Nach Satz 19 gilt

$$\begin{aligned}
&\vec{v} \circ \vec{w} = \cos(\alpha) \cdot \|\vec{v}\| \cdot \|\vec{w}\| \\
\iff &\cos(\alpha) = \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}
\end{aligned}$$

Weiter gilt nach Satz 21

$$\cos(\alpha) = \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} \in [-1;\ 1]$$

Somit laesst sich der zuvor definierte/erlaeuterte arccos verwenden:

$$\alpha = \arccos\left(\frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}\right)$$

Bemerkung 23:

Nach Bemerkung 20

$$0 \leq \arccos(x) \leq \pi,$$

ist Definition 17

$$0 \leq \measuredangle(\vec{v},\ \vec{w}) \leq \pi$$

tatsaechlich erfuellt.

Beispiel 24: Winkel zwischen Vektoren des $\mathbb{R}_2$

Es seien $\vec{s}$, $\vec{t}$, $\vec{u}$, $\vec{v} \in \mathbb{R}_2$ und $\vec{w} \in \mathbb{R}_2$ mit

$$\vec{s} = \begin{pmatrix} -5 \\ 5 \end{pmatrix},\ \vec{t} = \begin{pmatrix} -5 \\ 0 \end{pmatrix},\ \vec{u} = \begin{pmatrix} 5 \\ 5 \end{pmatrix},\ \vec{v} = \begin{pmatrix} 0 \\ 5 \end{pmatrix},\ \vec{w} = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$$

wobei alle Vektoren als Anfangspunkt $O = (0,\ 0)$ (d.h. den Ursprung) haben.

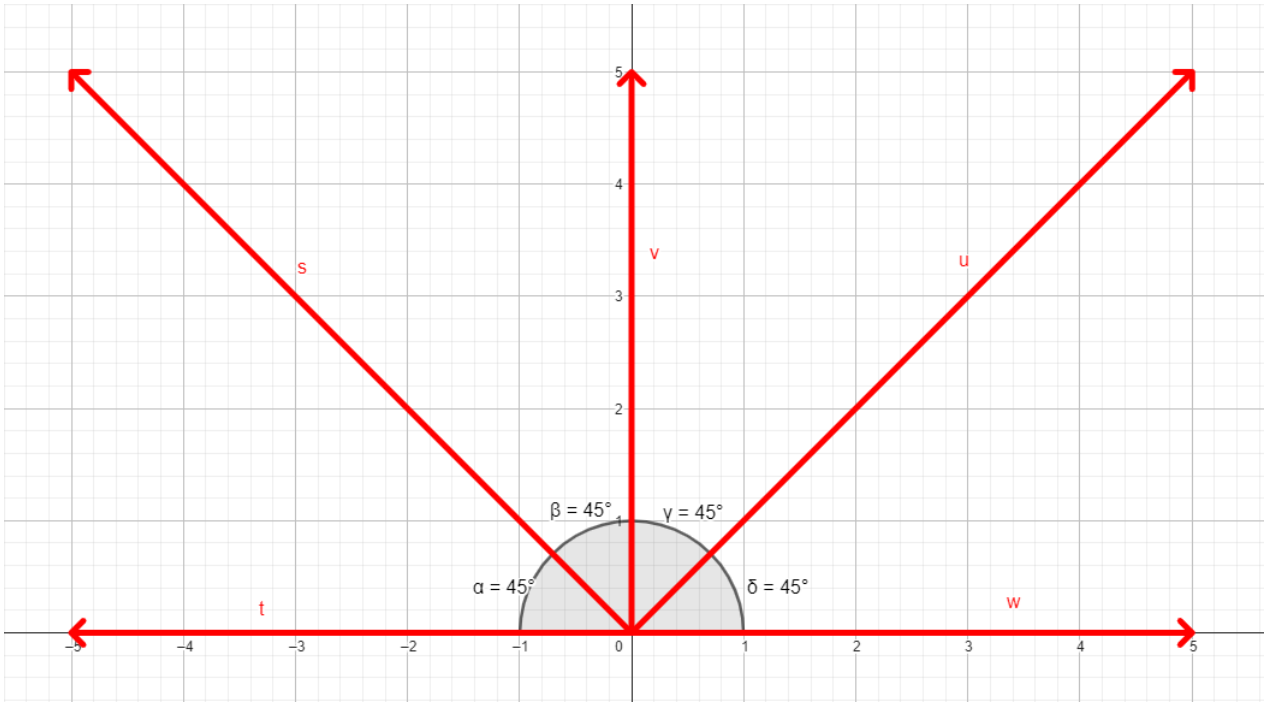

Abbildung 7: $\vec{s}$, $\vec{t}$, $\vec{u}$, $\vec{v}$ und $\vec{w}$ (s.o.)

Es sei θ_1 der Winkel zwischen $\vec{s}$ und $\vec{v}$. Dann gilt nach Satz 19:

$$\cos(\theta_1) = \frac{\vec{s} \circ \vec{v}}{\|\vec{s}\| \cdot \|\vec{v}\|} = \frac{\begin{pmatrix} -5 \\ 5 \end{pmatrix} \circ \begin{pmatrix} 0 \\ 5 \end{pmatrix}}{\left\| \begin{pmatrix} -5 \\ 5 \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} 0 \\ 5 \end{pmatrix} \right\|} = \frac{(-5) \cdot 0 + 5 \cdot 5}{\sqrt{(-5)^2 + 5^2} \cdot \sqrt{0^2 + 5^2}} = \frac{25}{\sqrt{2 \cdot 25} \cdot \sqrt{25}} = \frac{25}{\sqrt{2} \cdot \sqrt{25} \cdot \sqrt{25}} = \frac{1}{\sqrt{2}}$$

Nach 2. folgt:

$$\theta_1 = \arccos(\frac{1}{\sqrt{2}}) = 45° = \frac{\pi}{4}$$

Weiter sei θ_2 der Winkel zwischen $\vec{s}$ und $\vec{u}$. Dann ist

$$\cos(\theta_2) = \frac{\vec{s} \circ \vec{u}}{\|\vec{s}\| \cdot \|\vec{u}\|} = \frac{\begin{pmatrix} -5 \\ 5 \end{pmatrix} \circ \begin{pmatrix} 5 \\ 5 \end{pmatrix}}{\left\| \begin{pmatrix} -5 \\ 5 \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} 5 \\ 5 \end{pmatrix} \right\|} = \frac{(-5) \cdot 5 + 5 \cdot 5}{\sqrt{(-5)^2 + 5^2} \cdot \sqrt{5^2 + 5^2}} = \frac{-25 + 25}{\sqrt{2 \cdot 25} \cdot \sqrt{2 \cdot 25}} = \frac{0}{\sqrt{2} \cdot \sqrt{25} \cdot \sqrt{2} \cdot \sqrt{25}} = \frac{0}{10} = 0$$

Nach 4. folgt, dass

$$\theta_2 = \arccos(0) = 90° = \frac{\pi}{2}$$

Zuletzt sei θ_3 der Winkel zwischen $\vec{t}$ und $\vec{w}$. Dann ist

$$\cos(\theta_3) = \frac{\vec{t} \circ \vec{w}}{\|\vec{t}\| \cdot \|\vec{w}\|} = \frac{\begin{pmatrix} -5 \\ 0 \end{pmatrix} \circ \begin{pmatrix} 5 \\ 0 \end{pmatrix}}{\left\| \begin{pmatrix} -5 \\ 0 \end{pmatrix} \right\| \cdot \left\| \begin{pmatrix} 5 \\ 0 \end{pmatrix} \right\|} = \frac{(-5) \cdot 5 + 0 \cdot 0}{\sqrt{(-5)^2 + 0^2} \cdot \sqrt{5^2 + 0^2}} = \frac{-25}{\sqrt{25} \cdot \sqrt{25}} = \frac{-25}{25} = -1$$

Nach 5. folgt, dass

$$\theta_3 = \arccos(-1) = 180° = \pi$$

Definition 25: Winkel zwischen Punkten des $\mathbb{R}^2$

Es seien $P,\ Q,\ R \in \mathbb{R}^2$ Punkte der Ebene $\mathbb{R}^2$.
Dann ist der Winkel $\measuredangle(P,\ Q,\ R)$ zwischen den Punkten (beachte Reihenfolge!) definiert als der Winkel zwischen den Verbindungsvektoren $\overrightarrow{QP}$ und $\overrightarrow{QR}$:

$$\measuredangle(P,\ Q,\ R) \coloneqq \measuredangle\left(\overrightarrow{QP},\ \overrightarrow{QR}\right)$$

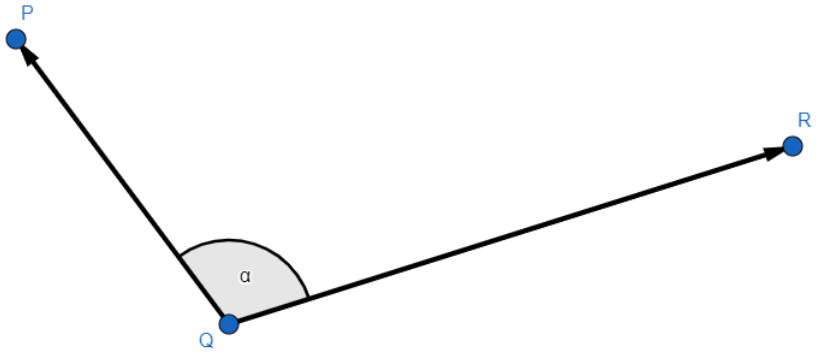

Abbildung 8: Winkel $\alpha = \measuredangle(P,\ Q,\ R)$ zwischen drei Punkten $P,\ Q,\ R \in \mathbb{R}^2$

Bemerkung 26: Winkel zwischen Punkten des $\mathbb{R}^2$

Es seien $P,\ Q,\ R \in \mathbb{R}^2$ Punkte der Ebene mit

$$P = (p_1,\ p_2),\ Q = (q_1,\ q_2),\ R = (r_1,\ r_2).$$

Dann ist der zuvor definierte Winkel $\measuredangle(P,\ Q,\ R)$ zwischen den Punkten abhaengig von der Reihenfolge der Punkte. Im Allgemeinen gilt beispielsweise

$$\measuredangle(P,\ Q,\ R) \neq \measuredangle(Q,\ P,\ R)$$

Allerdings gilt nach Bemerkung 18 im Allgemeinen

$$\measuredangle(P,\ Q,\ R) \coloneqq \measuredangle\left(\overrightarrow{QP},\ \overrightarrow{QR}\right) = \measuredangle\left(\overrightarrow{QR},\ \overrightarrow{QP}\right) =: \measuredangle(R,\ Q,\ P).$$

Auszerdem gilt auch hier der Satz 19 (Vektor-Winkel-Formel):
Es sei

$$\alpha \coloneqq \measuredangle(P,\ Q,\ R) \coloneqq \measuredangle\left(\overrightarrow{QP},\ \overrightarrow{QR}\right),$$

dann ist

$$\cos(\alpha) = \frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|}$$

und nach Folgerung 22 laesst sich $\alpha = \measuredangle(P, Q, R)$ bestimmen durch

$$\alpha = \measuredangle(P,\ Q,\ R) = \arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|}\right).$$

1.3. Abbiegewinkel im 2-dimensionalen Raum und Definition des Problems

Definition 27: Abbiegewinkel zwischen Punkten des $\mathbb{R}^2$

Es seien $P,\ Q,\ R \in \mathbb{R}^2$ Punkte der Ebene mit

$$P = (p_1,\ p_2),\ Q = (q_1,\ q_2),\ R = (r_1,\ r_2).$$

Nun wird der Abbiegewinkel, gemaesz der Aufgabenstellung, durch diese Punkte bestimmt.
Es sei, wie zuvor definiert,

$$\beta := \measuredangle(P,\ Q,\ R) := \measuredangle\left(\overrightarrow{QP},\ \overrightarrow{QR}\right)$$

der Winkel zwischen den Punkten P, Q und R.

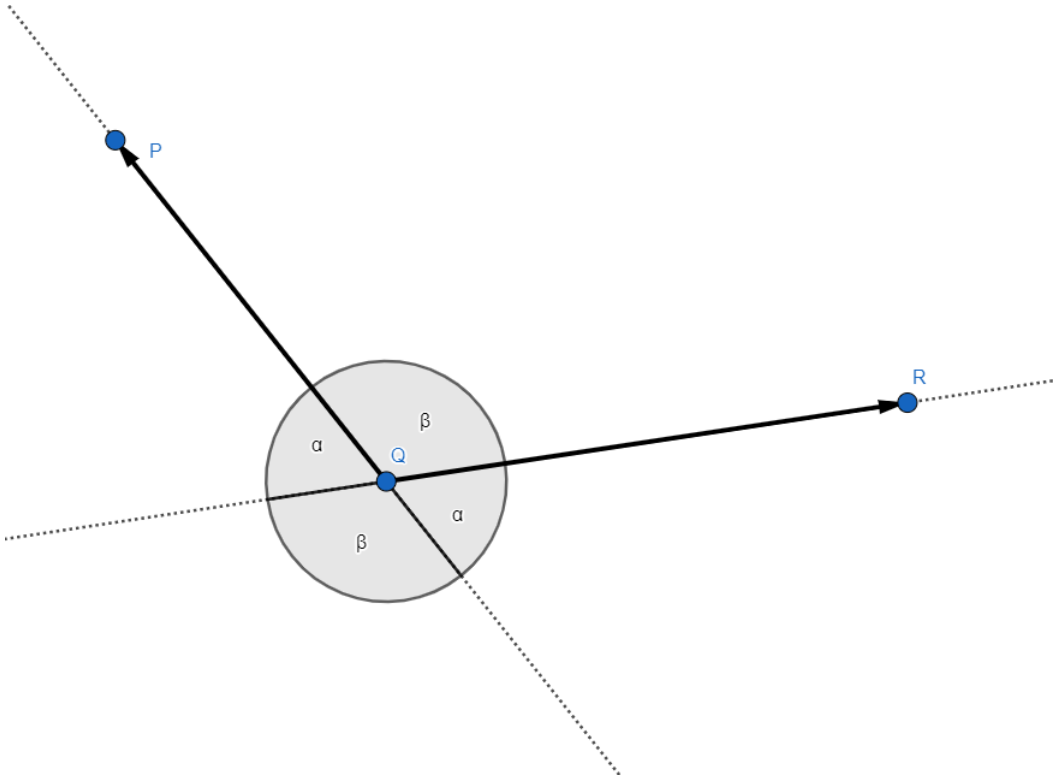

Abbildung 9: Abbiegewinkel

Dann ist der Abbiegewinkel α definiert, als der Nebenwinkel von β (also der Nebenwinkel des Winkels $\measuredangle(P,\ Q,\ R)$), sodass nach dem Nebenwinkelsatz[9] und der Bemerkung 26 gilt, dass

$$\begin{aligned}
&\alpha + \beta = 180^\circ \\
\iff &\alpha = 180^\circ - \beta \\
\iff &\alpha = 180^\circ - \measuredangle(P, Q, R) \\
\iff &\alpha = 180^\circ - \arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|}\right)
\end{aligned}$$

[9] Siehe auch `https://de.m.wikipedia.org/wiki/Winkel` (Nebenwinkel)

Definition 28: Formale Definition der Aufgabe

Gegeben seien $m \in \mathbb{N}$ Punkte $P_1,\ P_2,\ \dots\ ,\ P_m \in \mathbb{R}^2$ mit

$$P_1 = (x_1,\ y_1),\ P_2 = (x_2,\ y_2),\ \dots\ ,\ P_m = (x_m,\ y_m) \in \mathbb{R}^2,$$

Wobei $\forall i,\ j \in \{1,\ 2,\ \dots\ ,\ m\}$ mit $i \neq j$, gilt dass

$$P_i \neq P_j.$$

D.h., die Punkte sind paarweise verschieden.
Nun sei

$$P \coloneqq \{P_1,\ P_2,\ \dots\ ,\ P_m\},$$

mit $|P| = m$ die Menge aller m gegebenen Punkte.
Gesucht wird ein m-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m) \in P^m$ mit

$$P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}.$$

D.h., die Punkte $Q_1,\ Q_2,\ \dots\ ,\ Q_m \in \mathbb{R}^2$ entsprechen genau den m gegebenen Punkten $P_1,\ P_2,\ \dots\ ,\ P_m$.
Nun soll fuer das m-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m) \in P^m$, welches die Route des Piloten modelliert, folgendes gelten:

1. Die Abbiegewinkelbedingung:
   Fuer $i \in \{1,\ 2,\ \dots\ ,\ m-2\}$ gilt

   $$180° - \measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2}) \leq 90°$$

   D.h., fuer alle drei Punkte, die in der Route k aufeinanderfolgen gilt, dass der Abbiegewinkel (s. Definition 27)

   $$180° - \measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2})$$

   kleiner oder gleich 90° ist.

2. Die Strecke s soll minimiert werden. Dabei ist

   $$s \coloneqq \sum_{i=1}^{m-1} \left\| \overrightarrow{Q_i Q_{i+1}} \right\|$$

   die Strecke, die zurueckgelegt werden muss.

Bemerkung 29: Vereinfachung der Abbiegewinkelbedingung

In der Tat laesst sich die Abbiegewinkelbedingung fuer drei aufeinanderfolgende Punkte $P,\ Q,\ R \in \mathbb{R}^2$ stark vereinfachen.
Es seien

$$P = (p_1,\ p_2),\ Q = (q_1,\ q_2),\ R = (r_1,\ r_2)$$

Dann lautet die Abbiegewinkelbedingung

$$\begin{aligned}
&180° - \measuredangle(P,\ Q,\ R) \leq 90° && \mid -180° \\
\iff &- \measuredangle(P,\ Q,\ R) \leq -90° && \mid \cdot (-1) \\
\iff &\measuredangle(P,\ Q,\ R) \geq 90° && \mid \text{s. Bem. 26} \\
\iff &\arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|}\right) \geq 90°
\end{aligned}$$

Da cos die Umkehrfunktion von arrcos ist, und cos streng monoton fallend ist, folgt:

$$\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|} \leq \cos(90°) = 0$$

Da

$$\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\| \geq 0,$$

gilt weiter:

$$\begin{aligned}
&\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|} \leq 0 \\
\iff &\overrightarrow{QP} \circ \overrightarrow{QR} \leq 0 \\
\iff &\begin{pmatrix} p_1 - q_1 \\ p_2 - q_2 \end{pmatrix} \circ \begin{pmatrix} r_1 - q_1 \\ r_2 - q_2 \end{pmatrix} \leq 0 \\
\iff &(p_1 - q_1) \cdot (r_1 - q_1) + (p_2 - q_2) \cdot (r_2 - q_2) \leq 0
\end{aligned}$$

Somit laesst sich die Abbiegewinkelbedingung

$$\Big(\forall i \in \{1,\ 2,\ \ldots\ ,\ m-2\}\Big)\Big[180° - \measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2}) \leq 90°\Big]$$

wie folgt, auch ohne die Abbiegewinkel der Aufgabenstellung, formulieren:

$$\Big(\forall i \in \{1,\ 2,\ \ldots\ ,\ m-2\}\Big)\Big[\measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2}) \geq 90°\Big]$$

Und weiter:

$$\Big(\forall i \in \{1,\ 2,\ \ldots\ ,\ m-2\}\Big)\Big[\overrightarrow{Q_{i+1}Q_i} \circ \overrightarrow{Q_{i+1}Q_{i+2}} \leq 0\Big]$$

2. Verallgemeinerung des Problems - Punkte im $\mathbb{R}^n$

Nun wird das zuvor definierte Problem insofern erweitert, dass die Punkte im n-dimensionalen Raum $\mathbb{R}^n$ liegen, wobei $n \in \mathbb{N}$.

2.1. Punkte und Vektoren im n-dimensionalen Raum

Definition 30: Reeller Koordinatenraum $\mathbb{R}^n$

Es sei $n \in \mathbb{N}$. Nun wird der reelle Koordinatenraeum[10] $\mathbb{R}^n$ wie folgt definiert:

$$\mathbb{R}^n \coloneqq \{(x_1,\ x_2,\ \dots\ ,\ x_n) \mid x_1,\ x_2,\ \dots\ ,\ x_n \in \mathbb{R}\}.$$

D.h., R^n ist die Menge aller Tupel der Laenge n mit Elementen aus $\mathbb{R}$.

Definition 31: Reeler Vektorraum $\mathbb{R}_n$

Es sei $n \in \mathbb{N}$ und

$$\mathbb{R}_n \coloneqq \left\{ \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \mid x_1,\ x_2,\ \dots\ ,\ x_n \in \mathbb{R} \right\},$$

wobei

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

ein Spaltenvektor der Laenge n ist.

Bemerkung 32: Komponenten

Es sei $n \in \mathbb{N}$ und $P \in \mathbb{R}^n$, sowie $\vec{x} \in \mathbb{R}_n$ mit

$$P = (p_1,\ \dots\ ,\ p_n)$$

und

$$\vec{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$$

dann nennt man $p_1,\ \dots\ ,\ p_n$ bzw. $x_1,\ \dots\ ,\ x_n$ Komponenten von P bzw. $\vec{x}$.
Dabei ist p_1 bzw. x_1 die erste Komponente, p_2 bzw x_2 die zweite, usw.

Bemerkung 33: Interpretation von $\mathbb{R}^n$ und $\mathbb{R}_n$

Analog zum Fall $n = 2$, werden, fuer ein beliebiges $n \in \mathbb{N}$, die Elemente von $\mathbb{R}^n$ als Punkte im n-dimensionalen Raum aufgefasst und mit Groszbuchstaben bezeichnet.
Waehrend die Elemente von $\mathbb{R}_n$ als Verbindungsvektoren zwischen den Punkten, bzw. als Pfeile (auch: Vektor oder Richtungsvektor) aufgefasst werden. Richtungsvektoren werden wie ueblich mit $\vec{v}$, $\vec{u}$, $\vec{w}$, usw. notiert. Auszerdem haben Richtungsvektoren stehts eine Richtung und eine Laenge. Im Gegensatz zu den Punkten der Menge $\mathbb{R}^n$.

[10] Siehe auch `https://de.wikipedia.org/wiki/Koordinatenraum`

Definition 34:

Analog zu $\mathbb{R}_2$, kann man auch fuer $\mathbb{R}_n$ fuer ein beliebiges $n \in \mathbb{N}$ die Vektoraddition $\oplus$ und die Skalarmultiplikation $\odot$ so definieren, dass man den $\mathbb{R}_n$ als $\mathbb{R}$-Vektorraum bezeichnen kann:

$$\oplus \ : \ \mathbb{R}_n \times \mathbb{R}_n \ni (\vec{v},\ \vec{w}) = (\begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix},\ \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}) \longmapsto \begin{pmatrix} v_1 + w_1 \\ \vdots \\ v_n + w_n \end{pmatrix} \in \mathbb{R}_n$$

$$\odot \ : \ \mathbb{R} \times \mathbb{R}_n \ni (\alpha,\ \vec{v}) = (\alpha,\ \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}) \longmapsto \begin{pmatrix} \alpha \cdot v_1 \\ \vdots \\ \alpha \cdot v_n \end{pmatrix} \in \mathbb{R}_2$$

Bemerkung 35: Nullvektor und das additiv inverse Element

Erneut existiert das Neutralelement des Vektorraums $\mathbb{R}_n$ fuer $n \in \mathbb{N}$. Dieser wird Nullvektor genannt und wie folgt notiert:

$$\vec{0} \coloneqq \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix}$$

Auszerdem existiert fuer jedes $\vec{v} \in \mathbb{R}_n$ mit

$$\vec{v} \coloneqq \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

ein additiv inverses Element

$$-\vec{v} = (-1) \odot \vec{v} = \begin{pmatrix} -v_1 \\ \vdots \\ -v_n \end{pmatrix},$$

sodass

$$(-\vec{v}) \oplus \vec{v} = \begin{pmatrix} -v_1 \\ \vdots \\ -v_n \end{pmatrix} \oplus \begin{pmatrix} v_1 \\ \vdots \\ v_1 \end{pmatrix} = \begin{pmatrix} (-v_1) + v_1 \\ \vdots \\ (-v_n) + v_n \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 0 \end{pmatrix} = \vec{0}$$

Definition 36: Norm und Skalarprodukt von Vektoren des $\mathbb{R}_n$

Es sei $n \in \mathbb{N}$ und $\vec{v} \in \mathbb{R}_n$, dann ist die Norm bzw. Laenge $\|\vec{v}\|$ des Vektors $\vec{v}$ mit

$$\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix}$$

wie folgt definiert:

$$\|\vec{v}\| \coloneqq \sqrt{\sum_{i=1}^{n} v_i^2}$$

Weiter seien $\vec{u},\ \vec{w} \in \mathbb{R}^n$ mit

$$\vec{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_n \end{pmatrix},\ \vec{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix},$$

dann ist das Skalarprodukt[11] $\vec{u} \circ \vec{w}$ von $\vec{u}$ und $\vec{w}$ wie folgt definiert:

$$\vec{u} \circ \vec{w} \coloneqq \sum_{i=1}^{n} u_i \cdot w_i$$

Das heiszt, das Skalarprodukt entspricht der Summe der einzeln multiplizierten Komponenten der Vektoren.

[11] Bzw. Standartskalarprodukt: `https://de.wikipedia.org/wiki/Standardskalarprodukt`

Definition 37: Verbindungsvektor

Betrachtet man zwei Punkte $P,\ Q$ des $\mathbb{R}^n$ mit $n \in \mathbb{N}$ (also Punkte im n-dimensionalen Raum), so nennt man den Vektor, der die Punkte miteinander verbindet, Verbindungsvektor. Dies ist ein Richtungsvektor und dieser wird mit $\overrightarrow{PQ} \in \mathbb{R}_n$ notiert.
Es seien

$$P = (p_1,\ \dots\ ,\ p_n),\ Q = (q_1,\ \dots\ ,\ q_n).$$

Nun ist

$$\overrightarrow{PQ} \coloneqq \begin{pmatrix} q_1 - p_1 \\ \vdots \\ q_n - p_n \end{pmatrix}$$

der Verbindungsvektor zwischen P und Q. Dieser beginnt bei P und zeigt zu, bzw. endet bei Q.

Definition 38: Ortsvektor

Es sei $n \in \mathbb{N}$ und

$$O \coloneqq (0, 0,\ \dots\ ,\ 0) \in \mathbb{R}^n$$

der Ursprung bzw. der Nullpunkt.
Weiter sei $P \in \mathbb{R}^n$ beliebig mit

$$P = (p_1,\ p_2,\ \dots\ ,\ p_n).$$

Dann nennt man den Verbindungsvektor $\overrightarrow{OP}$ vom Nullpunkt zum Punk P auch Ortsvektor des Punkts P. Dabei sind die Komponenten des Punktes P identisch mit denen des Ortvektors:

$$\overrightarrow{OP} = \begin{pmatrix} p_1 - 0 \\ \vdots \\ p_2 - 0 \end{pmatrix} = \begin{pmatrix} p_1 \\ \vdots \\ p_2 \end{pmatrix}$$

2.2. Winkel zwischen Vektoren und Punkten im n-dimensionalen Raum

Definition 39: Der Winkel zwischen Vektoren des $\mathbb{R}_n$

Es seien $\vec{v},\ \vec{w} \in \mathbb{R}^n \setminus \{\vec{0}\}$ mit $n \in \mathbb{N}$, wobei

$$\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_n \end{pmatrix},\ \vec{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$$

dann wird der Winkel $\alpha \coloneqq \measuredangle(\vec{v},\ \vec{w})$ zwischen den zwei Vektoren $\vec{v}$ und $\vec{w}$ durch

$$\cos(\alpha) \coloneqq \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}$$

definiert.[12]

Bemerkung 40: Cauchy-Schwarz-Ungleichung

Nach der Cauchy-Schwarz-Ungleichung[13] gilt:

$$\begin{aligned}
& (\vec{v} \circ \vec{w})^2 \leq \|\vec{v}\|^2 \cdot \|\vec{w}\|^2 \\
\iff & \sqrt{(\vec{v} \circ \vec{w})^2} \leq \sqrt{\|\vec{v}\|^2 \cdot \|\vec{w}\|^2} && \mid\ \|\vec{v}\| \cdot \|\vec{w} \geq 0 \\
\iff & |\vec{v} \circ \vec{w}| \leq \|\vec{v}\| \cdot \|\vec{w}\| \\
\iff & \frac{|\vec{v} \circ \vec{w}|}{\|\vec{v}\| \cdot \|\vec{w}\|} \leq 1 \\
\iff & -1 \leq \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} \leq 1
\end{aligned}$$

[12] Siehe auch `https://de.wikipedia.org/wiki/Standardskalarprodukt` (Abgeleitete Begriffe)
[13] Siehe auch `https://de.wikipedia.org/wiki/Cauchy-Schwarzsche_Ungleichung`

Somit ist

$$\cos(\alpha) = \frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|} \in [-1;\ 1]$$

Daraus folgt, dass α mit

$$\alpha = \arccos\left(\frac{\vec{v} \circ \vec{w}}{\|\vec{v}\| \cdot \|\vec{w}\|}\right) \in [0;\ \pi]$$

der Winkel $\measuredangle(\vec{v},\ \vec{w})$, der zwischen den Vektoren $\vec{v}$ und $\vec{w}$ eingeschlossen ist, ist.

Definition 41: Winkel zwischen Punkten des $\mathbb{R}^n$

Es sei $n \in \mathbb{N}$ und $P,\ Q,\ R \in \mathbb{R}^n$ mit

$$P = (p_1,\ \dots\ ,\ p_n),\ Q = (q_1,\ \dots\ ,\ q_n),\ R = (r_1,\ \dots\ ,\ r_n).$$

Dann ist der Winkel $\measuredangle(P,\ Q,\ R)$ zwischen den Punkten P, Q und R definiert als der Winkel zwischen den Verbindungsvektoren $\overrightarrow{QP}$ und $\overrightarrow{QR}$ der Punkte:

$$\measuredangle(P,\ Q,\ R) \coloneqq \measuredangle(\overrightarrow{QP},\ \overrightarrow{QR})$$

Somit gilt, nach Bemerkung 40, fuer $\alpha \coloneqq \measuredangle(P,\ Q,\ R)$:

$$\cos(\alpha) = \frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\|\overrightarrow{QP}\| \cdot \|\overrightarrow{QR}\|} \in [-1;\ 1]$$

Daraus folgt:

$$\alpha = \arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\|\overrightarrow{QP}\| \cdot \|\overrightarrow{QR}\|}\right) \in [0;\ \pi]$$

Satz 42: Winkel zwischen Vektoren und Punkten

Aufgrund der Kummutativitaet des Skalarprodukts und der Multiplikation. D.h., wegen

$$\vec{v} \circ \vec{w} = \sum_{i=1}^{n} v_i \cdot w_i = \sum_{i=1}^{n} w_i \cdot v_i = \vec{w} \circ \vec{v}$$

und

$$|\vec{v}| \cdot |\vec{w}| = |\vec{w}| \cdot |\vec{v}|$$

fuer $\vec{v},\ \vec{w} \in \mathbb{R}_n$ fuer $n \in \mathbb{N}$ gilt, dass

$$\frac{\vec{v} \circ \vec{w}}{|\vec{v}| \cdot |\vec{w}|} = \frac{\vec{w} \circ \vec{v}}{|\vec{w}| \cdot |\vec{v}|},$$

sodass auch

$$\measuredangle(\vec{v},\ \vec{w}) \coloneqq \arccos\left(\frac{\vec{v} \circ \vec{w}}{|\vec{v}| \cdot |\vec{w}|}\right) = \arccos\left(\frac{\vec{w} \circ \vec{v}}{|\vec{w}| \cdot |\vec{v}|}\right) =: \measuredangle(\vec{w},\ \vec{v}).$$

Somit gilt auch fuer $P,\ Q,\ R \in \mathbb{R}^n$, dass

$$\begin{aligned}\measuredangle(P,\ Q,\ R) &= \arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\|\overrightarrow{QP}\| \cdot \|\overrightarrow{QR}\|}\right) \\ &= \arccos\left(\frac{\overrightarrow{QR} \circ \overrightarrow{QP}}{\|\overrightarrow{QR}\| \cdot \|\overrightarrow{QP}\|}\right) = \measuredangle(R,\ Q,\ P).\end{aligned}$$

2.3. Definition des Problems im n-dimensionalen Raum

Bemerkung 43: Abbiegewinkel

Wie in Bemerkung 29 gezeigt wurde, laesst sich das Problem der gegebenen Aufgabe auch ohne die Abbiegewinkel der Aufgabenstellung formulieren. Dementsprechend wird fuer den verallgemeinerten Fall im n-dimensionalen Raum auf die Definition des Abbiegewinkels verzichtet und das Problem wird im Folgenden nur bezueglich des Winkels zwischen drei Punkten definiert.

Definition 44: Verallgemeinerte Definition der Aufgabenstellung

Es seien $m \in \mathbb{N}$ Punkte $P_1,\ P_2,\ \dots\ ,\ P_m \in \mathbb{R}^n$ fuer $n \in \mathbb{N}$ gegeben, wobei

$$P_i = (p_{i,1},\ p_{i,2},\ \dots\ ,\ p_{i,n}) \in \mathbb{R}^n$$

fuer $i \in \{1,\ 2,\ \dots\ ,\ n\}$ gilt.
Dabei gilt (analog zum originalen Problem) $\forall i,\ j \in \{1,\ 2,\ \dots m\}$ mit $i \neq j$, dass

$$P_i \neq P_j$$

D.h. (wieder), die Punkte sind paarweise verschieden.
Weiter sei

$$P \coloneqq \{P_1,\ P_2,\ \dots\ ,\ P_m\}$$

mit $|P| = m$ die Menge der m gegebenen Punkte.
Nun wird ein m-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m) \in P^m$ gesucht, mit

$$P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}.$$

D.h., die Punkte $Q_1,\ Q_2,\ \dots\ ,\ Q_m$ entsprechen genau den m gegebenen Punkten $P_1,\ P_2,\ \dots\ ,\ P_m$.
Fuer dieses m-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m) \in P^m$ soll folgendes gelten:

1. Die Abbiegewinkelbedingung (bzw. eine Umformung dieser):
   Fuer alle $i \in \{1,\ 2, \dots\ ,\ m-2\}$ gilt

   $$\measuredangle\,(Q_i,\ Q_{i+1},\ Q_{i+2}) \geq 90°$$

   D.h., der Winkel zwischen je drei Punkten, die in k aufeinanderfolgen, ist stets mindestens 90°.

2. Diese Strecke s soll minimiert werden. Dabei ist

   $$s \coloneqq \sum_{i=1}^{m-1} \left\| \overrightarrow{Q_i Q_{i+1}} \right\|$$

   die Strecke, die zurueckgelegt werden muss.

Bemerkung 45: Vereinfachung der Abbiegewinkelbedingung

Analog zu Bemerkung 29 laesst sich die Abbiegewinkelbedingung

$$\measuredangle\,(Q_i,\ Q_{i+1},\ Q_{i+2}) \geq 90°$$

fuer $i \in \{1,\ 2, \dots\ ,\ m-2\}$ erneut vereinfachen.
Es seien $P,\ Q,\ R \in \mathbb{R}^n$ drei aufeinanderfolgende Punkte der Route k. Solaesst sich die Abbiegewinkelbedingung umformen:

$$\measuredangle(P,\ Q,\ R) \geq 90° \qquad \text{| Def. 41}$$

$$\iff \arccos\left(\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|} \right) \geq 90°$$

Da cos die Umkehrfunktion von arccos ist, und cos streng monoton fallend ist, folgt:

$$\frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|} \leq \cos(90°) = 0$$

Da

$$\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\| \geq 0,$$

gilt weiter:

$$\begin{aligned} & \frac{\overrightarrow{QP} \circ \overrightarrow{QR}}{\left\|\overrightarrow{QP}\right\| \cdot \left\|\overrightarrow{QR}\right\|} \leq 0 \\ \iff & \overrightarrow{QP} \circ \overrightarrow{QR} \leq 0 \end{aligned}$$

Somit laesst sich die Abbiegewinkelbedingung erneut wie folgt formulieren:
Fuer alle $i \in \{1,\ 2, \dots\ ,\ m-2\}$ gilt

$$\overrightarrow{P_{i+1}P_i} \circ \overrightarrow{P_{i+1}P_{i+2}} \leq 0$$

Satz 46: Symetrie des Problems

In der Tat ist das soeben definierte Problem symetrisch. Damit ist gemeint, dass eine Loesung des Problem sowohl „vorwaerts“ als auch „rueckwerts“ korrekt ist:
Es sei

$$k = (Q_1,\ Q_2,\ \dots\ ,\ Q_{m-1}\ ,\ Q_m) \in P^m$$

(siehe Def. 44) eine Loesung des Problems.
Dann ist auch

$$k' = (Q_m,\ Q_{m-1},\ \dots\ ,\ Q_2,\ Q_1)$$

eine Loesung des Problems.

Beweis:
Da

$$k = (Q_1,\ Q_2,\ \dots\ ,\ Q_{m-1}\ ,\ Q_m) \in P^m$$

eine Loesung des Problems ist, gilt nach Def. 44, dass fuer alle $j \in \{1,\ 2, \dots\ ,\ m-2\}$ gilt, dass

$$\measuredangle\left(Q_j,\ Q_{j+1},\ Q_{j+2}\right) \geq 90°$$

Daraus folgt nach Satz 42 folgt, dass

$$\measuredangle\left(Q_{j+2},\ Q_{j+1},\ Q_j\right) \geq 90° \tag{2}$$

$\forall j \in \{1,\ 2, \dots\ ,\ m-2\}$. Somit gilt fuer alle $i \in \{m,\ m-1, \dots\ ,\ 3\}$, dass

$$\measuredangle\left(Q_i,\ Q_{i-1},\ Q_{i-2}\right) \geq 90°,$$

indem man fuer $i \in \{m,\ m-1, \dots\ ,\ 3\}$ ein $j \coloneqq i-2 \in \{1,\ 2, \dots\ ,\ m-2\}$ waehlt und Ungleichung 2 anwendet.
D.h., alle drei Punkte, die in k' aufeinanderfolgen, haben eine Winkel, der groeszer oder gleich $90°$ ist.
Weiter ist

$$s \coloneqq \sum_{i=1}^{m-1} \left\|\overrightarrow{Q_iQ_{i+1}}\right\|$$

minimiert, sodass auch

$$\begin{aligned} s' &\coloneqq \sum_{i=1}^{m-1} \left\|\overrightarrow{Q_{m-i+1}Q_{m-i}}\right\| \\ &= \left\|\overrightarrow{Q_mQ_{m-1}}\right\| + \left\|\overrightarrow{Q_{m-1}Q_{m-2}}\right\| + \dots + \left\|\overrightarrow{Q_2Q_1}\right\| \\ &= \left\|\overrightarrow{Q_2Q_1}\right\| + \dots + \left\|\overrightarrow{Q_{m-1}Q_{m-2}}\right\| + \left\|\overrightarrow{Q_mQ_{m-1}}\right\| \\ &= \left\|\overrightarrow{Q_1Q_2}\right\| + \dots + \left\|\overrightarrow{Q_{m-2}Q_{m-1}}\right\| + \left\|\overrightarrow{Q_{m-1}Q_m}\right\| \\ &= \sum_{i=1}^{m-1} \left\|\overrightarrow{Q_iQ_{i+1}}\right\| =: s \end{aligned}$$

minimiert ist. □

3. Loesbarkeit und Komplexitaet des originalen ($\mathbb{R}^2$) und verallgemeinerten ($\mathbb{R}^n$) Problems

In diesem Kapitel wird kurz auf die Loesbarkeit des Problems eingegangen und dessen Komplexitaet behandelt.

In [2] von Sándor Fekete aus dem Jahrt 1997 wird erwaehnt, dass nicht alle alle Mengen an Punkten im euklidischen Raum eine Route (die alle Punkte besucht) haben, bei dem alle Winkel stumpf sind (d.h. der Winkel ist zwischen 90° und 180° grosz). Dabei wird folgendes Beispiel fuer beliebig viele Punkte gegeben. Dabei existiert ein gleichseitiges Dreieck und die restlichen Punkte sind in dessen Innerem. Dadurch muesste einer der Punkte des Dreiecks zwangslaeufig in der Mitte (d.h., nicht am Anfang oder Ende) des Weges vorkommen, sodass die Ecke zusammen mit den zwei Nachbarn des Punktes im Weg einen stumpfen Winkel bilden wuerden, aber dies ist unabhaengig von den Nachbarn nicht moeglich.

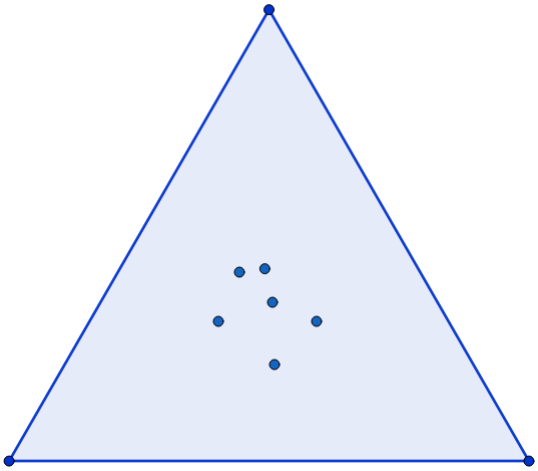

Abbildung 10

Eine solche Konstruktion von Mengen an Punkten laesst sich vermutlich in hoehre Dimensionen erweitern.
Weiter wird in [2] zwar auf die Komplexitaet einiger endlicher Mengen an erlaubten Winkeln eingegangen. Dennoch bleibt es offen, die Komplexitat der Berechnung von Wegen mit nur stumpfen (und nur spitzen) Winkeln zu bestimmen.
Im folgenden wird davon ausgegangen, dass es nicht in polynomieller Zeit moeglich ist, solche Wege zu bestimmen und somit das gegebene Problem (ohne die Komponente des kuerzesten Wegen) effizient zu loesen. So wie es auch NP-schwer ist, einen Hamilton-Weg zu finden[14].
Weitere Instanzen des Problems, die keine Loesung besitzen, lassen sich einfach durch ausprobieren finden.

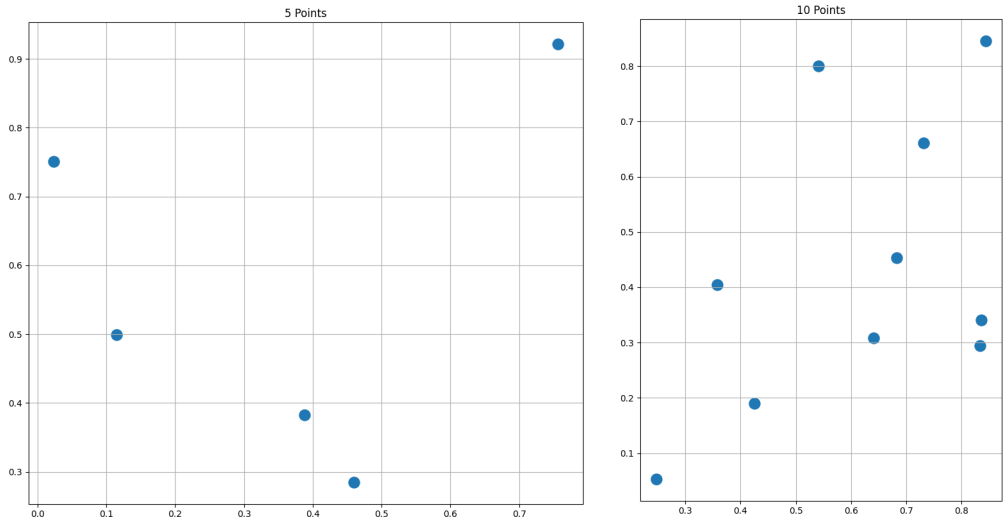

Abbildung 11: Nicht loesbare Instanzen

[14] Siehe auch `https://en.wikipedia.org/wiki/Hamiltonian_path_problem`

4. Loesungsvorschlaege I - Verallgemeinertes Problem: Punkte im $\mathbb{R}^n$

Im Folgenden wird auf Loesungsverfahren fur das verallgemeinerte Problem eingegangen. Denn dadurch wird das originale Problem (fuer $n = 2$) impliziert mitgeloest. Weiter wurde dieser Ansatz gewaehlt, da der allgemeine Fall des Problems im n-dimensionalen Raum ($n \in \mathbb{N}$) sich genau so (einfach) loesen laesst, wie das originale Problem.
Dabei werden zunaechst exakte Loesungsverfahren vorgestellt, welche auf Bruteforce[15] beruhen. Anschlieszend werden heuristischen Verfahren vorgestellt, die das Problem in effizienter Zeit aber dafuer nicht unbedingt perfekt loesen.
Fuer alle Loesungsverfahren wird in Kapite 5 eine Zeit- und Platzkomplexitaetsanalyse durchgefuehrt.

4.1. Exaktes Loesungverfahren I - Bruteforce I

Es seien $m,\ n \in \mathbb{N}$, wobei m der Anzahl an paarweise verschiedenen Punkten entspricht, die gegeben sind und n deren Dimension ist.

Algorithmus 1:
Nun wird das erste, exakte Loesungsverfahren vorgestellt. Bei diesem handelt es sich um ein Bruteforceverfahren, welches schlicht und einfach alle moeglichen Routen ausprobiert und anschliezend ueberprueft, ob es sich um eine moegliche Route handelt. D.h., ob die Abbiegewinkelbedingung nach Definition 44 und Bemerkung 45 erfuellt ist. Anschlieszend wird die Laenge der aktuellen Route berechnet und mit der bisher besten Route vergliechen, wobei die Laenge der besten Route mit Unendlich initialisiert wird. Danach wird ggf. die beste Route angepasst. Nachdem alle Permutationen probiert wurde, wird die beste Route oder ggf. $NULL$ zurueckgegeben (falls keine Loesung gefunden wurde).

Algorithmus 1 : Bruteforce I

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $k = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

$(k_{\text{best}},\ length_{\text{best}}) \leftarrow (NULL,\ \infty)$
foreach $k' = (k_1,\ k_2,\ \dots\ ,\ k_m)$ **in** alle Permutationen von $(P_1,\ P_2,\ \dots\ ,\ P_m)$ **do**
  // Abbiegewinkelbedingung pruefen
  if $\overrightarrow{k'_{i+1}k'_i} \circ \overrightarrow{k'_{i+1}k'_{i+2}} \leq 0\ \forall\ i \in \{1,\ 2,\ \dots\ ,\ m-2\} \leq 0$ **then**
    // Aktuelle Routenlaenge mit bester Routenlaenge vergleichen und ggf. anpassen
    $length_{\text{current}} \leftarrow \sum_{i=1}^{m-1} \left\|\overrightarrow{k'_i k'_{i+1}}\right\|$ // Laenge der Route k'
    if $length_{current} < length_{best}$ **then**
      $(k_{\text{best}},\ length_{\text{best}}) \leftarrow (k',\ length_{\text{current}})$ // k' Kopie!
    end if
  end if
end foreach
return k_{best}

Algorithmus[16] 2 und Algorithmus 3:
Um die Permutationen in der Praxiz effizient zu generieren, kann beispielsweise der Heap-Algorithmus (s. [1]) nach B.R. Heap aus dem Jahr 1963 benutzt werden. Dieser nimmt eine Liste P der Laenge N entgegen und gibt alle Permutationen der Liste aus. Weiter kann dieser auch nicht-rekursiv formuliert werden[17], sodass Algorithmus 1 einfach in den iterativen Heap-Algorithmus eingebettet werden kann (siehe Algorithmus 3).
Dazu wird Algorithmus 2 bengenutzt, um die Permutationen der Punkte $P_1,\ P_2,\ \dots\ ,\ P_m$ zu erzeugen, woraufhin (wie in Algorithmus 1) an den Stellen, an denen zuvor die Permutation ausgegeben wurden, die aktuelle Permutation (Route) auf die Abbiegewinkelbedingung ueberprueft wird und mit der bisher besten Route vergli-

[15] Also brutaler Kraft - lol. Siehe auch https://de.wikipedia.org/wiki/Brute-Force-Methode
[16] Pseudocode von https://de.wikipedia.org/wiki/Heap-Algorithmus
[17] Siehe auch https://de.wikipedia.org/wiki/Heap-Algorithmus und https://sedgewick.io/wp-content/uploads/2022/03/2002PermGeneration.pdf (Seite 16)

chen wird. Auch hier wird am Ende die bisher beste Route zurueck gegeben, oder ggf. $NULL$ (keine Loesung).

Algorithmus 2 : Iterativer Heap-Algorithmus

Input : Liste $P = (p_0,\ p_1,\ \dots\ ,\ p_{N-1})$ der Laenge $N \in \mathbb{N}$

PRINT(P) `// Permutation ausgeben`
$C = (c_0,\ c_1\ \dots\ ,\ c_{N-1}) \leftarrow (0,\ 0,\ \dots\ ,\ 0)$
$n \leftarrow 1$
while $n < N$ **do**
  if $c_n < n$ **then**
    if n *ist grade* **then**
      Tausche 0-tes und n-tes Element von P
    else
      Tausche c_n-tes und n-tes Element von P
    end if
    PRINT(P) `// Permutation ausgeben`
    $c_n \leftarrow c_n + 1$
    $n \leftarrow 1$
  else
    $c_n \leftarrow 0$
    $n \leftarrow n + 1$
  end if
end while

Algorithmus 3 : Bruteforce I mit Hilfe des iterativen Heap-Algorithmus

Input : $P = (P_0,\ P_1,\ \dots\ ,\ P_{m-1})$, wobei $\forall i \in \{0,\ 1,\ \dots\ ,\ m-1\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $k = (Q_1,\ Q_2,\ \ddots,\ Q_m)$, oder $NULL$ (keine Loesung)

$(k_{\text{best}},\ length_{\text{best}}) \leftarrow (NULL,\ \infty)$
$C = (c_0,\ c_1\ \dots\ ,\ c_{m-1}) \leftarrow (0,\ 0,\ \dots\ ,\ 0)$
$n \leftarrow 1$
`// P auf Abbiegewinkelbedingung pruefen und ggf. beste Routen anpassen`
$length_{\text{current}} \leftarrow$ Lange der Route P
$(k_{\text{best}},\ length_{\text{best}}) \leftarrow (P,\ length_{\text{current}})$, falls Abbiegewinkelbedingung fuer P gilt
while $n < m$ **do**
  if $c_n < n$ **then**
    if n *ist grade* **then**
      Tausche 0-tes und n-tes Element von P
    else
      Tausche c_n-tes und n-tes Element von P
    end if
    $c_n \leftarrow c_n + 1$
    $n \leftarrow 1$
    `// Aktuelle Routenlaenge mit bester Routenlaenge vergleichen und ggf. anpassen`
    $length_{\text{current}} \leftarrow$ Lange der Route P
    $(k_{\text{best}},\ length_{\text{best}}) \leftarrow (P,\ length_{\text{current}})$, falls Abbiegewinkelbedingung fuer P gilt und $length_{\text{current}} < length_{\text{best}}$ (als Kopie!)
  else
    $c_n \leftarrow 0$
    $n \leftarrow n + 1$
  end if
end while
return k_{best}

Algorithmus 4:
Neben dem iterativen Verfahren, um alle moeglichen Permutationen durchzugehen (Algorithmus 2), laesst sich einfach ein rekursiver Algorithmus definieren, der ebenfalls alle moeglichen Permutationen der Punkte liefert, und die beste (also kuerzeste) von ihnen findet (s. Algorithmus 4). Bei diesem wird - begonnen mit einer leeren Route - in jedem rekursiven Aufruf durch alle noch nicht genutzten Punkte iteriert. Der aktuelle Punkt wird anschlieszend zur Route hinzugefuegt, woraufhin der Algorithmus rekursiv aufgerufen wird. Daraufhin wird der aktuelle Punkt wieder von der Route entfernt. Dieses Prinzip hat starke Aehnlichtkeit zum Durchlaufen eines Grahpen via Depth-First-Search[18].
Sollte die Route bei einem Aufruf des Algorithmus aus allen Punkten bestehen, so wird die aktuelle Route auf die Abbiegewinkelbedingung ueberprueft und anschliezend mir dem bisherigen minmum verglichen, wobei das minimum wieder mit Unedlich initialisiert wurde. Das bedeutet, dass die kuerzeste Route ausserhalb des eigentlichen Algorithmus existiert.

Algorithmus 4 : Rekursives Bruteforce I

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $k = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  Q ← [ ] // Leere Liste
2  (Q_best, length_best) ← (NULL, ∞)
3  Foo(Q)
4  // Input fuer Foo: Aktuelle Route Q = [ Q_1, Q_2, ... , Q_k ], k ∈ ℕ_0 mit 0 ≤ k ≤ m
5  Function Foo(Q = [ Q_1, Q_2, ... , Q_k ]):
6      // Abbruchbedingung
7      if k = m then
8          // Abbiegewinkelbedingung
9          if ⟶(Q_{i+1}Q_i) ∘ ⟶(Q_{i+1}Q_{i+2}) ≤ 0 ∀ i ∈ {1, 2, ... , m − 2} ≤ 0 then
10             // Aktuelle Routenlaenge mit bester Routenlaenge vergleichen und ggf.
                   anpassen
11             length_current ← Σ_{i=1}^{m−1} ‖⟶(Q_iQ_{i+1})‖
12             if length_current < length_best then
13                 (Q_best, length_best) ← (Q, length_current) // Koppie!
14             end if
15         end if
16     else
17         for p ∈ P \ {Q_1, Q_2, ... , Q_k} do
18             Q.add(p)
19             Foo(Q)
20             Q.remove(p)
21         end for
22     end if
23 return Q_best
```

[18] Tiefensuche - Siehe auch https://de.wikipedia.org/wiki/Tiefensuche

4.2. Exaktes Loesungverfahren II - Bruteforce II

Algorithmus 5 : Iterativer Heap-Algorithmus II

Input : Liste $P = (p_0,\ p_1,\ \dots\ ,\ p_{n-1})$ der Laenge $N \in \mathbb{N}$

```
1  for i ← 0 to N − 1 by 1 do
2      for j ← i + 1 to N − 1 by 1 do
3          P′ ← P // Permutation kopieren
4          Tausche 0-tes und i-tes Element von P′
5          Tausche n − 1-tes und j-tes Element von P′
6          C = (c_0, c_1 ... , c_{n−1}) ← (1, 1, ... , 1)
7          PRINT(P′) // Permutation ausgeben
8          n ← 2
9          while n < N − 1 do
10             if c_n < n then
11                 if n ist ungrade then
12                     Tausche 1-tes und n-tes Element von P′
13                 else
14                     Tausche c_n-tes und n-tes Element von P′
15                 end if
16                 PRINT(P′) // Permutation ausgeben
17                 c_n ← c_n + 1
18                 n ← 2
19             else
20                 c_n ← 1
21                 n ← n + 1
22             end if
23         end while
24     end for
25 end for
```

Algorithmus 5:
Tatsaechlich laesst sich die Anzahl an Routen, die ausprobiert werden muessen noch reduzieren, da es sich bei dem Problem um ein symetrisches handelt (siehe Satz 46). Das heiszt, es koennen die Haelfte der Routen ausgelassen werden, da diese in der umgekehrten Reihenfolge bereits getestet wurden.
D.h., im Algorithmus 1 kann anstatt durch alle moeglichen Permutationen auch durch die Haelfte iteriert werden, sodass von allen Permutationen p und p' der Punkte, wobei p' die Umdrehung (also die umgedrehte/umgekehrte Permutation) von p ist, nur genau eine Permutation vorkommt.
Um durch all diese Permutationen zu iterieren laesst sich ein Algorithmus verwenden, bei dem mittels zwei geschachtelten Schleifen durch alle Paar $(a,\ b) \in \mathbb{N}^2$ iteriert wird, wobei $1 \leq a < b \leq m$. a und b sind nun die Indizes des ersten und des letzte Element der aktuellen Permutationen. Anschliezend werden basierend auf dem Heap-Algorithmus alle Permutationen der Elemente zwischen dem ersten und letzten Element erzeugt (s. Algorithmus 4). Analog zu Algorithmus 2 laesst laesst sich auch in Algorithmus 5 der zuvor beschriebene Algoritmus 1 einbetten. Dies kann analog zu Algorithmus 3 geschehen, indem zunaechst die beste Route und deren Laenge initialisiert werden und anschlieszend an den Stellen des Algorithmus, an denen die Permutationen ausgegeben werden, die aktuelle Permutation der Punkte bezgl. der Abbiegewinkelbedingung ueberprueft wird und die Laenge der aktuellen Route mit der der besten Route verglichen wird, woraufhin die beste Route ggf. angepasst wird.

4.3. Exaktes Loesungverfahren III - Branch-and-Bound

Basierend auf Algorithmus 4 wird nun ein rekursives Verfahren zur Loesung des Problems vorgestellt, welches Prinzipien des Branch-and-Bound[19] und Backtracking[20] nutzt, um trotz Bruteforce (d.h., trotz des Ausprobierens aller moeglichen Permutationen) in der Praxis eine deutlich bessere Laufzeit aufweisen kann, als die vorherigen Algorithmen.

Der folgende Algorithmus 6 funktioniert analog zum Algorithmus 4. Es werden also alle Permutationen an Punkten ausprobiert, indem diese rekursiv aufgebaut werden. Im Vergleich zu Algorithmus 4 hat dieser allerdings zwei ausschlaggebendene Unterschiede:

1. Ein Punkt wird nur zu aktuellen Permutation (Route) hinzugefuegt, wenn die Abbiegewinkelbedingung (bezgl. der letzten zwei und dem neuen Punkt) angehalten wird (s. Zeile 18).
   (Aehnlich wie beim Backtracking, wenn eine „schlechte“ Entscheidung getroffen wurde. E.g. subset sum problem[21]).
   Dies hat zur Folge, dass am Ende nur Routen betrachtet werden, die die Abbiegewinkelbedingung bereits einhalten.

2. Ist eine aktuelle Permutation (Route) beim Aufruf der rekursiven Methode bereits laenger als die bisher beste Route, so wird die Methode abgebrochen (aehnlich wie beim Branch-and-Bound Ansatz).
   Um dies zu vereinfachen, wird die aktuelle Laenge der Route beim Methoden-Aufruf mit uebergeben und beim hinzufugen von Punkten angepasst.

Algorithmus 6 : Rekursives Bruteforce II

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $Q = (Q_1,\ Q_2,\ \dot{},\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  Q ← [ ] // Leere Liste
2  (Q_best, length_best) ← (NULL, ∞)
3  Foo(Q, 0)
4  // Foos Input: Aktuelle Route Q = [ Q_1, Q_2, ... , Q_k ], (0 ≤ k ≤ m) und ihre Laenge l.
5  Function Foo(Q = [ Q_1, Q_2, ... , Q_k ], l):
6      if l ≥ length_best then
7          return
8      end if
9      // Abbruchbedingung
10     if k = m then
11         // Aktuelle Routenlaenge mit bester Routenlaenge vergleichen und ggf. anpassen
12         if l < cost_best then
13             (Q_best, length_best) ← (Q, l) // Kopie!
14         end if
15     else
16         for p ∈ P \ {Q_1, Q_2, ... , Q_k} do
17             // Abbiegewinkelbedingung
18             if k ≤ 1 ODER (Q_k Q_{k-1}) ∘ (Q_k p) ≤ 0 then
19                 Q.add(p)
20                 Foo(Q, l + ||Q_k p||)
21                 Q.remove(p)
22             end if
23         end for
24     end if
25 return Q_best
```

[19] Siehe auch https://de.wikipedia.org/wiki/Branch-and-Bound
[20] Siehe auch https://de.wikipedia.org/wiki/Backtracking
[21] Siehe auch https://en.wikipedia.org/wiki/Subset_sum_problem

4.4. Greedy Strategie - Nearest Neighbour Algorithmus

Von nun an werden anders als zuvor nicht mehr exakte Loesungsverfahren betrachtet, sonders heuristische Verfahren.

Die dabei benutzte Heursistik ist die naheliegende Nearest-Neighbour-Heuristik[22], das auch beim TSP zum Einsatz kommt. Dieses beruht darauf, von einem Startpunkt aus „greedy“[23] stets den (distanzmaeszig) naechst moeglichen Punkt zu waehlen, bis alle Punkte benutzt wurden. In dem folgenden Verfahren wird dabei natuerlich zusaetzlich die Abbiegewinkelbedingung beim auswaehlen des naechsten Punktes beachtet.
Moegliche Ansaetze sind hier moeglich, indem man verschieden viele Startpunkte oder Kombination von Punkten als Startpunkte waehlt und ggf. die beste entstandene Route zurueckgibt:

1. **Algorithmus 9:**
   Ein zufaelliger, bzw. der erste Punkt P_1 wird als Startpunkt Q_1 gewaehlt. Daraufhin wird der (nach der euklidischen Distanz) naechste Punkt Q_2 gesucht. Diese beiden Punkte bilden den Anfang der Route Q. Daraufhin wird stets der nach der Distanz naechste Punkt gewaehlt, der noch nicht genutzt wurde und die Abbiegewinkelbedingung einhaelt und wird zu Q hinzugefuegt. Sollte kein moeglicher Punkt mehr existieren, der als naechstes gewaehlt werden kann, bevor alle Punkte gewaehlt wurden, so wird $NULL$ zurueckgegeben, d.h., keine Loesung gefunden.

2. **Algorithmus 10:**
   Anstatt nur den einen Punkt als Anfangspunkt zu waehlen, wird jeder der m Punkte einmal als Anfangpunkt gewahelt. Daraufhin faehrt der Algorithmus fort, so wie Algorithmus 9. D.h., fuer jeden moeglichen Anfangspunkt wird anschlieszend der naechste Punkte gewaehlt und dann je der naechste Punkte, der die Abbiegewinkelbedingung einhaelt. Fuer jeden moeglichen ausprobierten Anfangspunkt wird dabei die Laenge der aktuellen Route (falls gefunden) mit der bisher besten Laenge verglichen, wobei diese wieder mit Unendlich initialisiert wird und die beste Route wird gegebenenfalls angepasst.

3. **Algorithmus 11 / 12:**
   Anstatt jeden Punkt als Anfangspunkt zu waehlen, koennen auch alle moeglichen Permutatioen an $k \in \mathbb{N}$ mit $k \geq 2$ Punkten als Anfang der Route Q gewaehlt werden. Um diese Permutationen zu erzeugen, kann fuer $k \geq 3$ muessen diese selbstverstaendlich bereits die Abbiegewinkelbedingung einhalten. Daraufhin sucht der Algorithmus fuer den aktuellen Routenanfang die weiteren Punkte mit Hilfe des Greedy-Verfahrens.

Um die Algorithmen 9, 10, 11 und 12 simpel darzustellen, wird zunaechst Algorithmus 8 definiert, welcher fuer den Anfang einer Route Q der Laenge $k \geq 2$ die restlichen Punkte greedy zur Route hinzufuegt.
Um Algorithmus 12 fuer $k \geq 3$ zu realisieren, wird auszerdem Algorihtmus 7 definiert. Dieser prueft fuer einen gegebenen Routenanfang, ob die Abbiegewinkelbedingung eingehalten wird, falls k groezser, oder gleich drei ist.

Algorithmus 7 : Hilfsalgorithmus - Ueberprueft die Abbiegewinkelbedingung fuer eine Route Q

Input : Route $Q = [\ Q_1,\ Q_2,\ \dots\ ,\ Q_k\ ]$ mit $k \geq 0$
Output : **WAHR**, falls die Abbiegewinkelbedingung eingehalten wird. Sonst **FALSCH**

```
1  Function checkRoute(Q = [ Q_1, Q_2, ... , Q_k ]):
2      if k ≤ 2 then
3          return WAHR
4      end if
5      for i ← 1 to k − 2 by 1 do
6          // Abbiegewinkelbedingung
7          if Q_{i+1}Q_i ∘ Q_{i+1}Q_{i+1} > 0 then
8              return FALSCH
9          end if
10     end for
11     return WAHR
```

[22] Siehe auch https://de.wikipedia.org/wiki/Nearest-Neighbor-Heuristik
[23] Siehe auch https://de.wikipedia.org/wiki/Greedy-Algorithmus

Bemerkung - Algorithmus 7:
Um fuer eine Route Q der Laenge $k \in \mathbb{N}_0$ zu bestimmen, ob die Abbiegewinkelbedingung eingehalten wird, wird zunaechst geprueft, ob die Route Q nur 0, 1, oder 2 Elemente hat. Denn in diesem Fall existiert kein einziger Abbiegewinkel. Deswegen wird in diesem Fall „WAHR“ zurueckgegeben. Ansonsten wird durch die Zahlen $i = 1,\ 2,\ \dots\ ,\ k-2$ iteriert und in jedem Durchgang ueberprueft, ob die Abbiegewinkelbedingung nicht eingehalten wird, indem fuer die Punkte Q_i, Q_{i+1} und Q_{i+2} getestet wird, ob

$$\overrightarrow{Q_{i+1}Q_i} \circ \overrightarrow{Q_{i+1}Q_{i+1}} > 0.$$

Ist dies der Fall, so wird sofort „FALSCH” zurueckgegeben. Nachdem die Schleife fertig durchgelaufen ist, sodass alle Abbiegewinkel ueberprueft wurden, wird „WAHR“ zurueckgegeben.

Algorithmus 8 : Hilfsalgorithmus - Berechnet greedy die restlichen Punkte fuer eine Route Q

Input : $P = \{P_1,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$, und Anfang der Route $Q = [\ Q_1,\ \dots\ ,\ Q_k\ ]$ mit $k \geq 2$ und $\{Q_1,\ \dots\ ,\ Q_k\} \subseteq P$
Output : $Q = [\ Q_1,\ Q_2,\ \dot{},\ Q_m\ ]$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  Function createRoute(Q = [ Q_1, Q_2, ... , Q_k ], P = {P_1, P_2, ... , P_m}):
2      // Restliche Punkte greedy hinzufuegen
3      while |Q| < m do
4          k ← |Q| // Laenge der Liste Q
5          Q_first ← Q[k − 1] // Vorletztes Element der Liste Q
6          Q_second ← Q[k] // Letztes Element der Liste Q
7          Q_next ← NULL // Naechstes Element der Liste Q; wird gesucht
8          // Durch die nicht benutzten Punkte iterieren
9          for p in P \ Q do
10             // Abbiegewinkelbedingung
11             if Q_second Q_first ∘ Q_second p ≤ 0 then
12                 if Q_next = NULL oder ||Q_second p|| < ||Q_second Q_next|| then
13                     Q_next ← p
14                 end if
15             end if
16         end for
17         // Pruefe, ob kein moeglicher Punkt gefunden wurde
18         if Q_next = NULL then
19             return NULL
20         else
21             Q.add(Q_next)
22         end if
23     end while
24     return Q
```

Bemerkung - Algorithmus 8:
Um fuer eine Route Q der Laenge[24] $k \in \mathbb{N}$ mit $k \geq 2$ basierend auf der Menge $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$ die restlichen Punkte der Route mit dem Greedy-Verfahren zu bestimmen, wird so lange, wie die Route noch weniger Elemente enthaelt als m (D.h., $|Q| < m$), folgendes getan:
Zunaechst werden die letzten beiden Punkte Q_{first} und Q_{second} der bisher berechneten Route definiert. Zudem wird auch der Punkt Q_{next} mit $NULL$ initialisiert. Dieser wird im Folgenden der Punkt sein, der als naechstes in die Route eingefuegt werden soll. Anschliesszend werden alle Punkte aus P durchgegangen, die noch nicht Teil der Route sind - d.h., die Menge $P \backslash Q$. Fuer jedes Element $p \in P \backslash Q$ dieser Menge wird anschliesszend ueberprueft, ob die Abbiegewinkelbedingung fuer die Punkte Q_{first}, Q_{second} und p erfuellt ist. Also ob $\overrightarrow{Q_{\text{second}}Q_{\text{first}}} \circ \overrightarrow{Q_{\text{second}}p} \leq 0$ gilt. Ist dies der Fall, so kann der Punkt Q_{next} angepasst werden (also auf p gesetzt werden), falls dieser noch keinen Wert hat (also den Zustand $NULL$ hat) oder der Abstand $\|\overrightarrow{Q_{\text{second}}p}\|$ vom letzten Punkt Q_{second} der Route zu p kleiner ist als der Abstand $\|\overrightarrow{Q_{\text{second}}Q_{\text{next}}}\|$ vom letzten Punkt Q_{second} zum aktuell besten Punkt Q_{next}. Nachdem alle Punkte der Menge $P \setminus Q$ durchgegangen wurden, wird geprueft, ob ein moeglicher Punkt gefunden, wurde. Also ob Q_{next} nicht $NULL$ ist. Ist dies doch der Fall, so wird die Funktion abgebrochen und es wird $NULL$ zurueckgegeben (es wurde keine Route gefunden). Ansonsten kann Q_{next} zur Route Q hinzugefuegt werden. Hat die Route nun die Laenge m, so kann diese zurueckgegeben werden.

[24] die Mindestlaenge zwei ist wichtig, damit die Abbiegewinkelbedingung ab dem ersten neuen Punkt berueckksichtigt werden kann

Algorithmus 9 : Nearest Neigbour I - Ein Anfangspunkt

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $Q = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  if m ≤ 2 then
2  |  return Liste besthend aus allen gegebenen Punkten
3  end if
4  Q ← [ ] // Leere Liste, einsbasierte Indizierung
5  // Anfangspunkt waehlen und zur Route hinzufuegen
6  Q_1 ← P_1
7  Q.add(Q_1)
8  // Distanzmaeszig naechsten Punkt (von Q_1 aus) zu Q hinzufuegen
9  Q_2 ← P_2
10 for p in P \ {P_1, P_2} do
11 |  if ||Q_1 p|| < ||Q_1 Q_2|| then
12 |  |  Q_2 ← p
13 |  end if
14 end for
15 Q.add(Q_2)
16 // Restliche Punkte greedy hinzufuegen
17 return createRoute(Q, P)
```

Algorithmus 10 : Nearest Neigbour II - Jeder moegliche Anfangspunkt

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $Q = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  if m ≤ 2 then
2  |  return Liste besthend aus allen gegebenen Punkten
3  end if
4  (Q_best, length_best) ← (NULL, ∞)
5  for Q_1 in P do
6  |  Q ← [ ] // Leere Liste, einsbasierte Indizierung
7  |  // Aktuellen Anfangspunkt zur Route hinzufuegen
8  |  Q.add(Q_1)
9  |  // Distanzmaeszig naechsten Punkt (von Q_1 aus) zu Q hinzufuegen
10 |  Q_2 ← P_2
11 |  for p in P \ {P_1, P_2} do
12 |  |  if ||Q_1 p|| < ||Q_1 Q_2|| then
13 |  |  |  Q_2 ← p
14 |  |  end if
15 |  end for
16 |  Q.add(Q_2)
17 |  // Restliche Punkte greedy hinzufuegen
18 |  Q ← createRoute(Q, P)
19 |  // Aktuelle Route Q mit bisher besten Route vergleichen und ggf. anpassen
20 |  if Q ≠ NULL then
21 |  |  length_current ← Σ_{i=1}^{m-1} ||Q[i]Q[i+1]||
22 |  |  if length_current < length_best then
23 |  |  |  (Q_best, length_best) ← (Q, length_current)
24 |  |  end if
25 |  end if
26 end for
27 return Q_best
```

Bemerkung - Algorithmus 9:
Wie bereits erklaert, soll dieser Algorithmus das Problem fuer m n-dimensionale Punkte loesen, indem das Greedy-Verfahren Nearest-Neigbour verwendet wird. Dazu wird die Route Q hier als Liste dargestellt, welche die Methoden add(...) bietet und Zugriff aud ihre Elemente mit Q[index] ermoeglicht. Nachdem der Sonderfall des Problems fuer weniger als 3 Pukte abgefragt wurde (in dem Fall besteht das Ergebnis aus diesen ein oder zwei Punkten), wird zunaechst zu dieser Liste Q (der Route) der erste Punkt P_1 hinzugefuegt. Anschlieszend, wird der Punkt der Menge $P \setminus \{P_1\}$ gesucht, der nach der Distanz am naechsten an P_1 dran ist. Dazu wird dieser Punkt Q_2 mit P_2 initialisiert und anschlieszend wird die Menge $P \setminus \{P_1,\ P_2\}$ durchgegangen. Falls dabei ein Punkt p gefunden wird der naeher an Q_1 ist, als der aktuelle Punkt Q_2 (d.h., wenn $\|\overrightarrow{Q_1 p}\| < \|\overrightarrow{Q_1 Q_2}\|$), wird Q_2 auf den neuen Punkt p gestzt. Nachdem alle Punkte durchgegangen wurden, sodass der Punkt gefunden wurde, der am naechsten an P_1 ist, kann dieser Punkt Q_2 zu Liste Q hinzugefuegt werden. Zu letzt wird mittels der Funktion *createRoute* (siehe Algorithmus 8) der Rest der Route mit Hilfe des Greedy-Verfahrens Nearest-Neigbour berechnet und sofort zurueckgegeben.

Bemerkung - Algorithmus 10:
Auch Algorithmus 10 soll das Problem loesen und funktioniert aehnlich wie Algorithmus 9. Allerdings werden nun alle moeglichen Startpunkte $Q_1 \in P$ ausprobiert, anstatt nur einen auszuprobieren. Zunaechst werden allerdings die beste Route Q_{best} und dessen Laenge $cost_{\text{best}}$ mit $NULL$ und ∞ initialisiert. Anschlieszend werden alle moeglichen Startpunkte ausprobiert. Fuer jeden Anfangspunkt $Q_1 \in P$ wird genau, wie in Algorithmus 9 die aktuelle Route Q als Liste dargestellt, der erste Punkt Q_1 zu dieser hinzugefugt und anschliezend der Punkt, der Q_1 am naechsten ist, ebenfalls zur Route hinzugefuegt, woraufhin der Rest der Route mittels der Funktion *createRoute* (siehe Algorithmus 8) berechnet wird. Dabei ist so beachten, dass die entstehende Liste Q auch $NULL$ seien kann. Dementspraechend wird dies abgefragt. Ist Q nicht $NULL$, so wird die Laenge $cost_{\text{current}}$ der aktuellen Route Q berechnet und mit der bisher besten Laenge $cost_{\text{best}}$ verglichen. Falls die aktuelle Route kuerzer ist als die bisher kuerzeste, so wird diese angepasst. Nachdem alle moeglichen Anfangspunkte probiert wurden, wird die insgesamt beste Route Q_{best} zurueckgegeben.

Algorithmus 11 : Nearest Neigbour III - Alle moeglichen Paare an Anfangspunkten ($k = 2$)

Input : $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$, wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$
Output : $Q = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

1 $(Q_{\text{best}},\ length_{\text{best}}) \leftarrow (NULL,\ \infty)$
2 **for** Q_1 ***in*** P **do**
3 **for** Q_2 ***in*** $P \setminus \{Q_1\}$ **do**
4 $Q \leftarrow [\,]$ `// Leere Liste, einsbasierte Indizierung`
5 `// Aktuelle Anfangspunkte zur Route hinzufuegen`
6 Q.add(Q_1)
7 Q.add(Q_2)
8 `// Restliche Punkte greedy hinzufuegen`
9 $Q \leftarrow$ `createRoute(`$Q,\ P$`)`
10 `// Aktuelle Route` Q `mit bisher besten Route vergleichen und ggf. anpassen`
11 **if** $Q \neq NULL$ **then**
12 $length_{\text{current}} \leftarrow \sum_{i=1}^{m-1} \left\| \overrightarrow{Q[i]Q[i+1]} \right\|$
13 **if** $length_{current} < length_{best}$ **then**
14 $(Q_{\text{best}},\ length_{\text{best}}) \leftarrow (Q,\ length_{\text{current}})$
15 **end if**
16 **end if**
17 **end for**
18 **end for**
19 **return** Q_{best}

Bemerkung - Algorithmus 11:
Algorithmus 11 stellt einen Sonderfall von Algorithmus 12 dar, wobei $k = 2$ ist. D.h., fuer den Anfang der Route Q werden alle moeglichen Permutationen der Groesze 2 ausprobiert. Bzw., alle Paare $(P,\ Q)$ mit $P \neq Q$. Dazu wird, wie bei Algorithmus 10, zunaechst die beste Route Q_{best} und dessen Laenge $cost_{\text{best}}$ mit $NULL$ und ∞ initialisiert. Anschhlieszend wird mit Hilfe von zwei Schleifen, die mit den Laufvariablen Q_1 bzw. Q_2 durch alle Elemente der Menge P bzw. $P \setminus \{Q_1\}$ durchgehen, durch alle Paare (wie zuvor erklaert) iterriert.
Anschlieszend wird wieder die Route Q als leere Liste initialisiert und die Punkte Q_1 und Q_2 werden hinzugefuegt. Daraufhin wird der Rest der aktuellen Route mit Hilfe der *createRoute* Funktion erzeugt. Dann wird, falls $Q \neq NULL$, die aktuelle Laenge $cost_{\text{current}}$ der Route berechnet und mit der Laenge der besten Route verglichen. Ist die aktuelle Route kuerzer als die bisher beste Route, so wird die beste Route und dessen Laenge auf Q und $cost_{\text{current}}$ gesetzt. Nachdem alle Paare durchgelaufen wurden, wird die insgesamt beste Route Q_{best} zurueckgegeben.

Algorithmus 12 : Nearest Neigbour III - Beliebig groszer Routenanfang ($k \in \mathbb{N}$ mit $k \geq 2$)

Input : $k \in \mathbb{N}$ mit $k \geq 2$ und $P = \{P_1,\ P_2,\ \dots\ ,\ P_m\}$,
wobei $\forall i \in \{1,\ 2,\ \dots\ ,\ m\}\ P_i \in \mathbb{R}^n$ und $|P| = m$ mit $k \leq m$ und $m \geq 2$
Output : $Q = (Q_1,\ Q_2,\ \dots,\ Q_m)$, wobei $P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$, oder $NULL$ (keine Loesung)

```
1  (Q_best, length_best) ← (NULL, ∞)
2  for Q in alle Permutationen von P der Laenge k do
3  |  // Permutation auf Abbiegewinkelueberpruefen
4  |  if not checkRoute(Q) then
5  |  |  continue for-loop
6  |  end if
7  |  // Restliche Punkte greedy hinzufuegen
8  |  Q ← createRoute(Q, P)
9  |  // Aktuelle Route Q mit bisher besten Route vergleichen und ggf. anpassen
10 |  if Q ≠ NULL then
11 |  |  length_current ← Σ_{i=1}^{m-1} ||Q[i]Q[i+1]→||
12 |  |  if length_current < length_best then
13 |  |  |  (Q_best, length_best) ← (Q, length_current)
14 |  |  end if
15 |  end if
16 end for
17 return Q_best
```

Bemerkung - Algorithmus 12:
Algorithmus 12 initiiert ebenfalls zuerst Q_{best} und dessen Laenge $cost_{\text{best}}$ mit $NULL$ und ∞. Daraufhin werden in einer Schleife alle Permutationen Q der Laenge $k \in \mathbb{N}$ mit $k \geq 2$ der Menge P durchgegangen. Jede dieser Permutationen stellt eine aller moeglichen Routenanfaenge der Laenge k dar. Damit sich die am Ende entstehende Route an die Abbiegewinkelbedingung haelt, wird fuer jedes Q zunaechst mit Hilfe der Funktion *checkRoute* (siehe Algorithmus 7) ueberprueft, ob der Routenanfang Q die Abbiegewinkelbedingung einhaelt. Ist dies nicht der Fall, so wird der aktuelle Durchlauf der Schleife abgebrochen und zur naechsten Permutation gesprungen. Anschlieszend wird der Rest der Route Q mit Hilfe der Funktion *createRoute* (siehe Algorithmus 8) berechnet. Da das Ergebnis auch $NULL$ sein kann, wird dieser Fall ueberprueft. Ist dies nicht der Fall, so wird die Laenge $cost_{\text{current}}$ der aktuellen Route berechnet und mit der bisher besten Routenlaenge Q_{best} verglichen. Ist $cost_{\text{current}} < cost_{\text{best}}$, so wird die beste Route und dessen Laenge auf Q und $cost_{\text{current}}$ gesetzt. Nachdem alle Permutation durchgegangen wurden, wird die insgesamt beste Route Q_{best} zurueckgegeben.

Um alle Permutationen der Groesze k der Menge P zu erzeugen, kann beispielsweise ein einfaches rekursives Programm (analog zu Algorithmus 4) genutzt werden, welches als Uebergabeparameter die aktuelle Permutation S (als Liste), P und k bekommt. Die Abbruchbedingung ist dann, ob die Permutation die Laenge k hat. Ist dies der Fall, so kann die aktuelle Permutation S behandelt werden. Sonst wird durch alle Elemente der Menge $P \setminus S$ iteriert, das aktuelle Element zur Liste S hinzugefuegt und die Funktion wird rekursiv aufgerufen, wobei S als Referenz uebergeben wird. Anschlieszend wird das aktuelle Element wieder von S entfernt.

5. Analyse der Zeit- und Platzkomplexitaet der Loesungsverfahren und Diskussion der verschiedenen Loesungsvorschlaege

Nachdem nun die exakten Loesungsverfahen mit dem Bruteforce Verfahren und die Heuristiken zum loesen des Problems vorgestellt wurden, werden diese nun bezgl. ihrer Zeit- und Platzkomplexitaet analysiert. Zuletzt werden die Algorithmen verglichen und es auf ihre Tauglichkeit in der Realitaet eingegangen.

5.1. Berechnen der Strecke einer Route und Pruefen der Abbiegewinkelbedingung

Da in den meisten der behandelten Algorithmen die Strecke einer Route berechnen und die Abbiegewinkelbedingung ueberpruefen, wird nun auf die Zeit-und Platzkomplexitaet eingegangen, die die Berechnung dieser beiden Dinge aufweist.

1. Berechnen der Strecke einer Route.
   Es sei $Q = (Q_1,\ Q_2,\ \dots\ ,\ Q_k)$ eine Route mit n-dimensionalen Punkten $Q_1,\ Q_2,\ \dots\ ,\ Q_k \in \mathbb{R}^n$ der Laenge $k \in \mathbb{N}$.
   Dann ist die Strecke s der Route definiert als

   $$s \coloneqq \sum_{i=1}^{n-1} \|\overrightarrow{Q_i Q_{i+1}}\|.$$

   Die Berechnung von

   $$\|\overrightarrow{AB}\| \coloneqq \sqrt{\sum_{i=1}^{n}(b_i - a_i)^2}$$

   fuer $A,\ B \in \mathbb{R}^n$ mit

   $$A = (a_1,\ a_2,\ \dots\ ,\ a_n),\ B = (b_1,\ b_2,\ \dots\ ,\ b_n)$$

   besitzt eine Zeitkomplexitaet von $\mathcal{O}(n)$, da die Summe der Differenzen der Komponenten der Punkte maximal lineare Zeit benoetigen.
   Die Zeitkomplexitaet $\mathcal{O}(n)$ ist korrekt unter der Annahme, dass die Grundrechenarten Subtraktion und Multiplikation (wegen $(b_i - a_i)^2$), sowie das Berechnen der Quadratwurzel eine Zeitkomplexitaet von $\mathcal{O}(1)$ (also eine konstante Laufzeit) aufweisen. Diese Annahme ist sinnvoll, da die Punkte in der Realitaet (bzw. in den Beispielsaufgaben) relativ kleine Komponenten haben, dessen Groesze nicht als Eingabeparameter angesehen werden wird. Weiter wird die Quadratwurzel in der Realitaet nicht exakt, sondern nur bis auf eine bestimmte Nachkommastelle berechnet.
   Daraus folgt, dass die Berechnung der Strecke s (s.o.) eine Laufzeit von $(k-1) \cdot \mathcal{O}(k) = \mathcal{O}(kn)$ hat, da die Berechnung der Strecke zwischen zwei Punkte mit einer Zeitkomplexitaet von $\mathcal{O}(n)$ $(k-1)$ mal ausgefuehrt wird. Die Zeitkomplexitaet der einzelnen Additionen der Strecken wird hier ebenfalls als $\mathcal{O}(1)$ ausgefasst.
   Bemerkung: Fuer den originalen Fall $n = 2$ ist die Zeitkomplexitaet $\mathcal{O}(2k) = \mathcal{O}(k)$

2. Pruefen der Abbiegewinkelbedingung.
   Es sei wieder $Q = (Q_1,\ Q_2,\ \dots\ ,\ Q_k)$ eine Route der Laenge $k \in \mathbb{N}$ mit n-dimensionalen Punkten $Q_1,\ Q_2,\ \dots\ ,\ Q_k \in \mathbb{R}^n$.
   Das Pruefen der Abbiegewinkelbedinung, also (fuer $k \geq 2$) die Bedingung

   $$(\forall i \in \{1,\ 2,\ \dots\ ,\ k-2\})\,[\overrightarrow{Q_{i+1}Q_i} \circ \overrightarrow{Q_{i+1}Q_{i+2}} \leq 0]$$

   weist ebenfalls eine Zeitkomplexitaet von $\mathcal{O}(mn)$ auf.
   Denn die Berechnung von

   $$\overrightarrow{BA} \circ \overrightarrow{BC} = \sum_{i=1}^{n}(a_i - b_i) \cdot (c_i - b_i)$$

   fuer $A,\ B,\ C \in \mathbb{R}^n$ mit

   $$A = (a_1,\ a_2,\ \dots\ ,\ a_n),\ B = (b_1,\ b_2,\ \dots\ ,\ b_n),\ C = (c_1,\ c_2,\ \dots\ ,\ c_n)$$

weist analog zu Punkt 1. (s.o.) eine Laufzeit von $\mathcal{O}(n)$ auf, sofern die Grundrechenarten in konstanter Zeit laufen.
Somit werden $k-2$ vergleiche mit einer Laufzeit von $\mathcal{O}(n)$ durchgefuehrt (unter der Annahme das man zwei Zahlen in $\mathcal{O}(1)$ vergleichen kann), sodass eine Laufzeit von $(k-2)\cdot\mathcal{O}(n) = \mathcal{O}(kn)$ zur Stande kommt.
Bemerkung: Fuer den originalen Fall $n = 2$ ist die Zeitkomplexitaet $\mathcal{O}(2k) = \mathcal{O}(k)$

5.2. Exakte Loesungsverfahren

5.2.1 Bruteforce I

Insbesondere wurden mit Algorithmus 3 (bzw. Algorithmus 1), und Algorithmus 4 verschiedene Algorithmen vorgestellt, die je alle moeglichen Permutationen der Elemente der Menge P ausprobieren, um die best moegliche Route zu finden. Dabei geht Algorithmus 3 basierend auf dem Heap-Algorithmus iterativ vor, wahrend Algorithmus 5 rekursiv formuliert ist.

Zeitkomplexitaetsanalyse des Algorithmus 3:
Algorithmus 3 (als konkretere Variante von Algorithmus 1) besteht maszgeblich daraus, dass jede moegliche Permutation berechnet wird und fuer jede Permutation die Strecke der aktuellen Route berechnet, und die Abbiegewinkelbedingung ueberprueft wird. Neben Zuweisungen werden auszerdem Elemente von Listen getauscht und Vergleiche getaetigt. Alle diese Dinge haben eine konstante Laufzeit ($\mathcal{O}(1)$).
Bekannter Weise, ist die Anzahl an Permutationen von $m \in \mathbb{N}$ Objekten (hier: Punkte)

$$m! \coloneqq \prod_{i=1}^{m} = m \cdot (m-1) \cdot \cdots \cdot 2 \cdot 1.$$

Da nun fuer jede Permutation (Route) der Laenge m die aktuelle Routenlaenge berechnet wird und anschlieszend die Abbiegewinkelbedingung ueberprueft wird, wird fuer jede Permutation eine Laufzeit von (s.o.)

$$\mathcal{O}(mn) + \mathcal{O}(mn) = \mathcal{O}(mn)$$

benoetigt.
Da dies fuer jede der $m!$ berechneten Permutation geschieht, entsteht insgesamt eine Laufzeit von

$$m! \cdot \mathcal{O}(mn) = \mathcal{O}(m! \cdot mn) = \mathcal{O}((m+1)! \cdot n) = \mathcal{O}(m! \cdot n)$$

Fuer den originalen Fall $n = 2$ ergibt sich

$$\mathcal{O}(m! \cdot 2) = \mathcal{O}(m!)$$

Platzkomplexitaetsanalyse des Algorithmus 3:
Da der Heap-Algorithmus ein Array (bzw. Liste) der Laenge m benutzt (C), die Route P die Laenge m hat, und die beste Route die Laenge m hat, wobei die Routen je aus n-dimensionalen Punkten bestehen, ist - unter der Annahme, dass die Komponenten der Punkte, und die Eintraegt der Liste C eine Platzkomplexitaet von $\mathcal{O}(1)$ auffweisen - die Platzkomplexitaet des gesamten Algorithmus

$$\mathcal{O}(m) + \mathcal{O}(mn) + \mathcal{O}(mn) = \mathcal{O}(mn).$$

Sodass sich fuer den originalen Fall $n = 2$

$$\mathcal{O}(2 \cdot m) = \mathcal{O}(m)$$

ergibt.

Zeitkomplexitaetsanalyse des Algorithmus 4:
Die Laufzeitkomplexitaet des rekursiven Algorithmus besteht ebenfalls haupsaechlich darin alle Permutationen der Route zu erzeugen und zu testen.
Fuer eine Route wird dabei ebenfalls (analog zu Algorithmus 3 einne Laufzeit von

$$\mathcal{O}(mn)$$

benoetigt.
Weiter wird in jedem rekursiven Schritt ein Durchgang weniger in der Schleife gemacht, wobei mit m Iterationen angefangen wird, sodass eine Laufzeit von

$$\mathcal{O}(mn) \cdot m \cdot (m-1) \cdot \dots \cdot 2 \cdot 1 = \mathcal{O}(mn) \cdot m! = \mathcal{O}(m! \cdot n)$$

(vergleiche Algorithmus 3) zu stande kommt, bzw. erneut fuer den originalen Fall $\mathcal{O}(m!)$.

Platzkomplexitaetsanalyse des Algorithmus 4:
Aufgrund der Rekursion muessen bei der Analyse der Platzklomplexitaet ebenfalls die Rueckspungaddressen beachtet werden. Hier kann dabei maximal eine Tiefe von m entstehen. Ebenfalls muss der Algorithmus eine Liste mit m Punkte mit je n Koordinaten speichern, sodass insgesamt ebenfalls eine Komplexitaet von

$$\mathcal{O}(m) + \mathcal{O}(mn) = \mathcal{O}(mn)$$

zur Stande kommt, sodass der originale Fall wieder eine Komplexitaet von $\mathcal{O}(m)$ erreicht.

5.2.2 Bruteforce II

Zeitkomplexitaetsanalyse des Algorithmus 5:
Da Algorithmus 5, bzw. eine konrete Implementierung ebendieses, $\frac{1}{2} \cdot m!$ Permutationen der Punkte testen wuerden und auch je eine Zeit von $\mathcal{O}(mn)$ pro Permutation benoetigt, erreicht auch dieser Algorithmus eine Zeitkomplexitaet von $\mathcal{O}(m! \cdot n)$, bzw. fuer $n = 2$ eine von $\mathcal{O}(m!)$, wenn gleich dieser Algorithmus in der Praxis doppelt schnell sein sollte, wie Algorithmus 3.

Platzkomplexitaetsanalyse des Algorithmus 5:
Auch dieser Algorithmus muss verschiedene Listen (bzw. Arrays) der Groesze n und der Groesze $\mathcal{O}(n \cdot m)$, sodass die Platzkomplexitaet bei $\mathcal{O}(mn)$ liegt (fuer $n = 2$, $\mathcal{O}(m)$).

5.2.3 Bruteforce III

Zeitkomplexitaetsanalyse des Algorithmus 6:
Zwar ist dieser Algorithmus in der Praxis deutlich besser als die vorherigen, trotzdem erreicht er keine bessere worst-case Komplexitaet, da auch dieser evtl. bei bestimmten Eingaben alle Permutationen probieren muss.
Somit ist die Laufzeit so, wie bei Algorithmus 4 $\mathcal{O}(m! \cdot n)$ und fuer $n = 2$ $\mathcal{O}(m!)$.

Platzkomplexitaetsanalyse des Algorithmus 6:
Auch die Platzkomplexitaetsanalyse veraendert sich im Vergleich zu Algorithmus 4 nicht und bleibt $\mathcal{O}(mn)$, bzw. fuer $n = 2$ $\mathcal{O}(m)$.

5.3. Nearest-Neighbour-Heuristik

5.3.1 Vorueberlegung

Um die heuristischen Algorithmen bezueglich ihrere Zeit- und Platzkomplexitaet zu analysieren, werden zunaechst die Algorithmen betrachtet, die benutzt werden, um die Abbiegewinkelbedinung (Algorithmus 7) und um gierig eine Route zu berechnen (Algorithmus 8).

Zeitkomplexitaetsanalyse des Algorithmus 7:
Wie bereits in der Analyse der Exakten Verfahren erwaehnt, ergibt sich dafuer eine Zeitkomplexitaet von $\mathcal{O}(nk)$ (wenn die Route die Laenge k hat), da durch $k-2$ Triple an Punkten iteriert werden muss und jeweils eine Berechnung des Aufwandes $\mathcal{O}(n)$ getaetigt werden muss.

Platzkomplexitaetsanalyse des Algorithmus 7:
Die Platzkomplexitaet fuer diesen Algorithmus ist hingegen konstant (d.h., $\mathcal{O}(1)$), da bis auf die Liste keine neuen Datenstrukturen oder andere nicht konstante Variablen genutzt werden.

Zeitkomplexitaetsanalyse des Algorithmus 8:
Die Zeitkomplexitaet fuer das Erstellen einer Route nach dem greedy Verfahen betraegt $\mathcal{O}\left(m^2 \cdot n\right)$ unter der Annahme, das $k = 2$, bzw. konstant ist, sodass der Route $\mathcal{O}(m)$ Punkte hinzugefuegt werden muessen. Da in jedem Durchlauf der while-Schleife entweder die Funktion abgebrochen wird oder ein Punkt zur Route hinzugefuegt wird, laeuft die Schleife $\mathcal{O}(m)$ mal. Dabei wird in jedem Durchgang durch $\mathcal{O}(m)$ Punkte iterriert und jeweils die Abbiegewinkelbedingung fuer 3 Punkte geprueft. Dies hat bekanntermaszen eine Zeitkomplexitaet von $\mathcal{O}(n)$, sodass insgesamt eine Zeitkomplexitaet von

$$\mathcal{O}(m) \cdot \mathcal{O}(m) \cdot \mathcal{O}(n) = \mathcal{O}\left(m^2 \cdot n\right)$$

entsteht. Daraus folgt, dass fuer das originale Problem ($n = 2$) eine Laufzeitkomplexitaet von $\mathcal{O}\left(m^2\right)$ errreicht wird.

Platzkomplexitaetsanalyse des Algorithmus 8:
Da in dem Algorithmus eine Liste genutzt wird (die Route Q), die eine Platzkomplexitaet von $\mathcal{O}(m \cdot n)$ hat und ansonsten nur Punkte und Zahlen in Variablen gespeichert werden, ist $\mathcal{O}(m \cdot n)$ auch die Platzkomplexitaet fuer diesen Algorithmus.

5.3.2 Nearest Neigbour I - Ein Anfangspunkt

Zeitkomplexitaetsanalyse des Algorithmus 9:
Der erste heuristische Algorithmus hat eine Zeitkomplexitaet $\mathcal{O}\left(m^2 \cdot n\right)$. Denn zunaechst werden $m-2$ mal eine Zeitkomplexitaet von $\mathcal{O}(n)$ verwendet, um den zweiten Punkt zu finden und anschlieszend wird der soeben analysierte Algorithmus 8 benutzt, welcher eine Zeitkomplexitaet von $\mathcal{O}\left(m^2 \cdot n\right)$ aufweist. Somit ist die Zeitkomplexitaet hier fuer den originalen Fall quadratisch in der Anzahl der Punkten ($\mathcal{O}\left(m^2\right)$).

Platzkomplexitaetsanalyse des Algorithmus 9:
Die Platzkomplexitaet folgt fuer diesen Algorithmus ebenfalls aus dem vorherigen und betragt somit fuer den verallgemeinerten Fall $\mathcal{O}(m \cdot n)$ und fuer den originalen Fall $\mathcal{O}(m)$.

5.3.3 Nearest Neigbour II - Jeder moegliche Anfangspunkt

Zeitkomplexitaetsanalyse des Algorithmus 10:
Da Algorithmus 10 im Grunde nichts anderes ist als eine m-Fache wiederholung (fuer jeden Anfangspuntk) des Algorithmus 9 mit der Zeitkomplexitaet $\mathcal{O}\left(m^2 \cdot n\right)$ ist, wobei jedes mal noch ggf. die Laenge der aktuellen Route in einer Zeit von $\mathcal{O}\left(mn\right)$ bestimmt wird, folgt, dass die Zeitkomplexitaet bei

$$m \cdot \left(\mathcal{O}\left(m^2 \cdot n\right) + \mathcal{O}\left(mn\right)\right) = \mathcal{O}\left(m^3 \cdot n\right) + \mathcal{O}\left(m^2 \cdot n\right) = \mathcal{O}\left(m^3 \cdot n\right)$$

liegt. Sodass der originale Fall kubisch in der Anzahl an Punkten, d.h. in $\mathcal{O}\left(m^3\right)$ ist.

Platzkomplexitaetsanalyse des Algorithmus 10:
Doch auch bei diesem Algorithmus aendert sich die Platzkomplexitaet im Vergleich zu den vorherigen zwei Algorithmen nicht, denn es wird immer noch maximal eine Route gleichzeitig betrachtet, sodass die Platzkomplexitaet weiterhin $\mathcal{O}\left(m \cdot n\right)$ fuer den verallgemeinerten Fall und $\mathcal{O}\left(m\right)$ fuer den originalen Fall lautet.

5.3.4 Nearest Neigbour III - Beliebig groszer Routenanfang

Zeitkomplexitaetsanalyse des Algorithmus 11:
Dieser Algorithmus benutzt zwei geschachtelte for-Schleifen, die je $\mathcal{O}\left(m\right)$ Durchlaeufe taetigen. Fuer jeden inneren Durchgang wird anschlieszend mit Hilfe des Algorithmus 8 eine Route berechnet und gff. ihre Laenge. Somit wird insgesamt eine Zeitkomplexitaet von

$$\mathcal{O}\left(m\right) \cdot \mathcal{O}\left(m\right) \cdot \Big(\mathcal{O}\left(m^2 \cdot n\right) + \mathcal{O}\left(m \cdot n\right)\Big) = \mathcal{O}\left(m^4 \cdot n\right) + \mathcal{O}\left(m^3 \cdot n\right) = \mathcal{O}\left(m^4 \cdot n\right)$$

erreicht. Sodass fuer den orignalen Fall eine Zeitkomplexitaet von $\mathcal{O}\left(m^4\right)$ vorliegt.

Platzkomplexitaetsanalyse des Algorithmus 11:
Wie bei den vorherigen Algorithmen aendert sich die Platzkomplexitaet auch hier nicht, da immer noch nur maximal eine Route gleichzeitig betrachtet wird, sodass die Platzkomplexitaet weiterhin $\mathcal{O}\left(m \cdot n\right)$ fuer den verallgemeinerten Fall und $\mathcal{O}\left(m\right)$ fuer den originalen Fall betraegt.

Zeitkomplexitaetsanalyse des Algorithmus 12:
Fuer den allgemeineren Fall, dass alle Permutationen der Punkte P der Groesze k probiert werden, ergibt sich die Laufzeit wie folgt.
Es gibt genau

$$m \cdot (m-1) \cdot \cdots \cdot (m-k+1) = \frac{m!}{(m-k)!} = \mathcal{O}\left(m^k\right)$$

Permutationen der Punkte P der Groesze k. Fuer jede dieser Permutationen wird zunaechst mit einer Laufzeit von $\mathcal{O}\left(m \cdot n\right)$ die Abbiegewinkelbedingung fuer die aktuelle Permutation geprueft. Anschlieszend wird mit einer Laufzeit von $\mathcal{O}\left(m^2 \cdot n\right)$ gerieg der Rest der Route gesucht und zuletzt wird mit einer Laufzeit von $\mathcal{O}\left(m \cdot n\right)$ ggf. die Laenge der Route bestimmt. Somit entsteht insgesamt eine Laufzeit von

$$\mathcal{O}\left(m^k\right) \cdot \Big(\mathcal{O}\left(m \cdot n\right) + \mathcal{O}\left(m^2 \cdot n\right) + \mathcal{O}\left(m \cdot n\right)\Big) = \mathcal{O}\left(m^k\right) \cdot \mathcal{O}\left(m^2 \cdot n\right) = \mathcal{O}\left(m^{k+2} \cdot n\right)$$

Platzkomplexitaetsanalyse des Algorithmus 12:
Doch selbst bei dieser Zeitkomplexitaet bleibt die Platzkomplexitaet aus dem selben Grund wie zuvor bei $\mathcal{O}\left(m \cdot n\right)$.

5.4. Vergleich der Loesungsverfahren

In der Tat lassen sich die exakten Algorithmen 3, 4 und 5 aufgrund ihrere extremen Laufzeit nur fuer 10 bis 11 Punkte verwenden (Siehe auch in den Beispielen). Dabei hat 4 tatsaechlich ungefaehr die Haelfte an Laufzeit wie 3. Im Gegensatz dazu hat der rekursive Algorithmus 5 in der Praxis eine deutlich schlechtere Laufzeit.
Der Algorithmus 6 hingegen, der sehr viele moegliche Routen durch Ausschlusskriterien garnichts erst ausprobiert, hat in der Praxis eine deutlich bessere Laufzeit als die anderen exakten Algorithmen und kann Routen fuer bis zu 20 Punkte exakt finden.
Das bedeutet: Existiert eine Loesung, so wird sie durch eines der exakten Loesungsverfahren eindeutig gefunden. Diese Algorithmen haben allerdings nur fuer $n \leq 20$ eine akzeptable Laufzeit.

Die Heuristischen Loesungsverfahren hingegen finden nicht immer eine Loesungen aber haben dafuer deutlich bessere, polynomielle Laufzeiten. So lassen sich in der Praxis am besten die Algorithmen 10 und 11 nutzen, die eine Laufzeit von $\mathcal{O}\left(m^3 \cdot n\right)$ und $\mathcal{O}\left(m^4 \cdot n\right)$ haben. Bzw. fuer den originalen Fall die Zeitkomplexitaeten $\mathcal{O}\left(m^3\right)$ und $\mathcal{O}\left(m^4\right)$. Fuer kleinere m (max. ca 25) laesst sich ggf. auch noch Algorithmus 12 fuer $k = 3$ benutzen, sodass eine Laufzeit von $\mathcal{O}\left(m^5 \cdot n\right)$ zu Stande kommt. Groesze k sind hingegen unrealistisch.

Die Platzkomplexitaet ist hingegen bei allen Loesungsalgorithmen unproblematisch da je nur eine Platzkomplexitaet von $\mathcal{O}\left(m \cdot n\right)$ erreicht wird.

6. Erweiterungen des Problems

Im Folgenden werden einige Erweiterungen des zuvor verallgemeinerten Problems vorgestellt. Loesungsideen zu diesen werden teilweise kurz erlaeutert.

6.1. Variabler Abbiegewinkel

Eine triviale Erweiterung des verallgemeinerten Problems im $\mathbb{R}^n$ fuer $n \in \mathbb{N}$ ist die folgende:
Gegeben sind erneut $m \in \mathbb{N}$ paarweise verschiedene Punkte $P_1,\ P_2,\ \dots\ ,\ P_m \in \mathbb{R}^n$ und das Problem ist identisch zu dem des verallgemeinerten Problems, bis auf die Abbiegewinkelbedingung.
Fuer

$$P \coloneqq \{P_1,\ P_2,\ \dots\ ,\ P_m\}$$

ist ein m-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m)$ gesucht, sodass

$$P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$$

Dabei soll wieder die Strecke s minimiert werden, wobei (wieder)

$$s \coloneqq \sum_{i=1}^{n-1} \left\| \overrightarrow{Q_i Q_{i+1}} \right\|$$

die Strecke zwischen allen Punkten ist.
Nun ist allerdings die Abbiegewinkelbedingung (bezgl. des eigentlichen Winkels zwischen drei Punkten und nicht dem der Aufgabenstellung) verallgemeinert:
Fuer alle $i \in \{1,\ 2,\ \dots n-2\}$ gilt:

$$\measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2}) \geq \theta$$

fuer einen Winkel $\theta \in [0°;\ 180°]$.

Dieses Problem kann genau geloest werden, wie das verallgemeinerte Problem (Fall $\theta = 90°$), indem statt der Vereinfachung der Abbiegewinkelbedinung ueberprueft wird, ob sich je drei Punkte an die Abbiegewinkelbedingung fur den gegebenen Winkel halten.

6.2. Mehrere Piloten

Wieder seien $m \in \mathbb{N}$ Punkte $P_1,\ P_2,\ \dots\ ,\ P_m \in \mathbb{R}^n$ mit $n \in \mathbb{N}$ gegeben.
Nun existieren weiter $t \in \mathbb{N}$ Piloten (oder Reisende) und der Winkel $\theta \in [0°;\ 180°]$.
Nun sind t Strecken $k_1,\ k_2, \dots k_t$ gesucht, wobei fuer $i \in \{1,\ \dots\ ,\ t\}$

$$k_i = (Q_{i,1},\ Q_{i,2},\ \dots\ ,\ Q_{i,t_i})$$

ist und fuer

$$P \coloneqq \{P_1,\ P_2,\ \dots\ ,\ P_m\}$$

gilt

$$P = \left\{ Q \in \mathbb{R}^n\ :\ Q \in \{Q_{i,1},\ Q_{1,2},\ \dots\ ,\ Q_{i,t_i}\} \text{ fuer } i \in \{1,\ \dots\ ,\ t\} \right\}$$

sowie

$$\sum_{i=1}^{t} t_i = n$$

D.h., jeder der m gegebenen Punkte $P_1,\ P_2,\ \dots\ ,\ P_m$ ist genau einmal in genau einer Strecke enthalten.
Auszerdem soll wieder die Abbiegebedingung bezgl. des Winkels θ gelten - nun fuer die einzelnen Strecken:
Fuer jede Strecke

$$k_i = (Q_{i,1},\ Q_{i,2},\ \dots\ ,\ Q_{i,t_i})$$

mit $i \in \{1,\ \dots\ ,\ t\}$ und $t_i \geq 3$ gilt fuer jedes $j \in \{1,\ \dots\ ,\ t_i - 2\}$, dass

$$\measuredangle(Q_{i,j},\ Q_{i,j+1},\ Q_{i,j+2}) \geq \theta$$

Auszerdem soll wieder die Strecke s, die zurueckgelegt werden muss, minimiert werden. Diese besteht hier aus der Summe der einzelnen Strecken:

$$s \coloneqq \sum_{i=1}^{t} \sum_{j=1}^{t_i - 1} \left\| \overrightarrow{Q_{i,j} Q_{i,j+1}} \right\|$$

Dieses Problem koennte beispielsweise geleost werden, indem man das Problem fuer einen Pilot loesst und die Route anschlieszend so in t Stuecke (Je eine Pro Pilot) aufteilt, sodass moeglichst lange Strecken (zwischen je zwei Punkten) entfernt werden.
Alternativ koennte man auch versuchen die gegebenen Punkte so in t Gruppen aufzuteilen, dass die Punkte in einer Gruppe moeglichst nah aneinander sind, woraufhin man fuer jede Gruppe einen Algorithmus fuer einen Piloten ausfuehrt. Diese Gruppierung koennte beispielsweise durch Methoden fuer das Clustering des Machine Learnings geschehen[25].

6.3. Besuchszeiten

Wie zuvor seien $m \in \mathbb{N}$ paarweise verschiedene Punkte $P_1,\ P_2,\ \dots\ ,\ P_m \in \mathbb{R}^n$ mit $n \in \mathbb{N}$ gegeben.
Die letzte Erweiterung des Problems beinhaltet nur einen Piloten. Dafuer gibt es allerdings fuer jeden Punkt P_i eine Zeitbeschraenkung x_i mit

$$x_i = (a_i,\ b_i) \in \mathbb{R}^2$$

D.h., der Pilot/der Reisende darf den Punkt nur in einem bestimmten Zeitraum besuchen. Der Einfachheit halber ist damit gemeint, dass die zuvor zurueckgelegte Strecke im n-dimensionalen Raum mindestens a_i und maximal b_i betraegt.
Gesucht ist also wieder ein n-Tupel $k = (Q_1,\ Q_2,\ \dots\ ,\ Q_m) \in P^m$ fuer

$$P \coloneqq \{P_1,\ P_2,\ \dots\ ,\ P_m\}.$$

Wobei

$$P = \{Q_1,\ Q_2,\ \dots\ ,\ Q_m\}$$

gilt und die Abbiegebedingung fuer einen Winkel $\theta \in [0°;\ 180°]$ eingehalten wird.
Fuer alle $i \in \{1,\ 2,\ \dots n-2\}$ gilt:

$$\measuredangle(Q_i,\ Q_{i+1},\ Q_{i+2}) \geq \theta$$

Zudem soll wieder die Strecke s mit

$$s \coloneqq \sum_{i=1}^{n-1} \left\| \overrightarrow{Q_i Q_{i+1}} \right\|$$

minimiert werden.
Die Zeitraumbeschraenkung ist nun wie folgt definiert:
Fuer alle $i \in \{1,\ 2,\ \dots n\}$ mit $Q_i = P_j$ fuer $j \in \{1,\ 2,\ \dots n\}$ gilt, dass fuer die Strecke, die bis zum Punkt Q_i zurueckgelegt wurde,

$$s_i \coloneqq \sum_{k=1}^{i-1} \left\| \overrightarrow{Q_k Q_{k+1}} \right\|$$

gilt, dass

$$s_i \in [a_j;\ b_j]$$
$$\iff a_j \leq s_i \leq b_j.$$

[25] Siehe auch `https://de.wikipedia.org/wiki/Clusteranalyse` und `https://de.wikipedia.org/wiki/N`

7. Implementierung

Nun wird auf die Implementierung der Loesungsideen eingegangen.
Die Loesungsvorschlaege wurden alle in einem Java 8 Programm implementiert. Dabei wurde groesztenteils objektorientiert vorgegangen. Deswegen gibt es u.a. eine abstrakte Klasse *Solver*, die als Eltern-Klasse fuer Klassen fungiert, die das Problem tatsaechlich loesen. Das heiszt eine Instanz des Problem (also eine Menge an Punkten) wird immer durch eine Instanz einer Kind-Klasse der Klasse *Solver* geloest. Dadurch ist das benutzen unterschiedlicher konkreter Implemtierungen einfacher zu handhaben und zu testen.

Auszerdem gibt es gibt zwei Klassen fuer Punkte und Vektoren, welche intern Arrays des Typen double (64 Bits) benutzen und zahlreiche Methoden bieten, um die benoetigten mathematischen Berechnungen durchzufuehren, um das Problem zu loesen. Beispielsweise besitzt die Klasse *Point* eine Methode, um den Richtungsvektor (Klasse *Vector*) vom Punkt zu einem weiteren Punkt zu berechnen. Die Klasse Vector bietet wiederum beispielsweise die Methode *length*, um die korrekte Laenge eines Vektoren im n-dimensionalen Raum zu berechnen. Weiter enthaelt die Klasse Vector eine Methode um das Skalarprodukt eines Vektors mit einem anderen zu berechnen und Point verfuegt ueber eine Methode, um die Distanz eines Punktes zu einem anderen zu bestimmen.

Zudem gibt es einige Funktionen, die von vielen Loesungsimplementierungen benoetigt werden. Diese wurden in eine Hilfsklasse *Utils* ausgelagert und sind sowohl oeffentlich als auch statisch. Darunter faellt zum Beispiel das Berechnen der Laenge einer Route (als Liste oder Array von Instanzen der Klasse Point) oder das pruefen der Abiegewinkelbedinung (Eng.: „Turning angle constraint“) fuer drei Punkte oder eine ganze Route.

Wie bereits erwaeht, werden die Routen als Arrays oder als Listen der Klasse Point umgesetzt. Die Punkte wiederum koennen eine beliebige Dimension haben, da die Koordinaten in double-Arrays gespeichert werden.

8. Beispiele

Nun wird zunaechst auf die Beispieldateien der BwInf Website und anschlieszend auf eigene Beispiele eingegangen, wobei auch auf die Laufzeit und unterschiedlichen Ergebnisse eingegangen wird.

Bsp.	m = Anzahl Punkte	Laenge Algo. 1 in km	Laufzeit Algo. 1	Laenge Algo. 2 in km	Laufzeit Algo. 2
1	84	847.434	52ms	847.434	1744ms
2	60	2183.662	23ms	2183.662	408ms
3	120	1962.277	107ms	1944.603	7027ms
4	25	1205.069	7ms	1205.069	21ms
5	60	4062.040	21ms	3854.277	425ms
6	80	k.L.	29ms	4052.955	875ms
7	100	k.L.	51ms	5035.685	2493ms

Tabelle 1: Uebersicht ueber die Ergebnisse der BwInf-Beispieleingaben

Wie der Tabelle zu entnehmen ist, wurden nur zwei der beschriebenen Algorithmen verwendet. Dies hat den Grund, das die Exakten Loesungsverfahren fur mehr als 20 Punke aufgrund ihrere Laufzeit nicht andwendbar sind.
Fuer die Beispieleingaben der BwInf-Website werden also im folgenden die Algorithmen 10 und 11 genutzt. Dabei wird Algorithmus 10 „Algorithmus 1“, bzw. „Erster Algorithmus“ genannt, waehrend Algorithmus 11 „Algorithmus 2“, bzw. „Zweiter Algorithmus“ genannt wird. Dadurch kann es vorkommen, dass die Algorithmen verschiedene Loesungen ermitteln. In diesem Fall wird das beste Ergebnis (d.h., die des Algorithmus 2) vorgestellt.
Die exakten Loesungsverfahen werden anschlieszend bei eigenen Beispielen angewendet und verglichen. Dabei wurde eine Route zwischen den 10 bzw. 20 einwohnerstaerksten Staedten Deutschland gesucht. Dafuer wurden die echten Koordinaten der Staedte verwendet, wodruch die Darstellung etwas verzehrt ist.
Alle Loesungen in Form von Routen, werden sowohl mathematisch als Tupel von 2-dimensionalen Punkten, als auch visuell in einem Koordinatensystem dargestellt.

8.1. Beispiel 1 - „wenigerkrumm1.txt “ (84 Punkte)

Das erste Beispiel (siehe Abbildung 12) hat die folgende Loesung (s. unten) mit der Laenge 847.434km:

$Q_1 = ($
(0.0, 0.0), (10.0, 0.0), (20.0, 0.0), (30.0, 0.0), (40.0, 0.0), (50.0, 0.0), (60.0, 0.0), (70.0, 0.0),
(80.0, 0.0), (90.0, 0.0), (100.0, 0.0), (110.0, 0.0), (120.0, 0.0), (130.0, 0.0), (140.0, 0.0), (150.0, 0.0),
(160.0, 0.0), (170.0, 0.0), (180.0, 0.0), (190.0, 0.0), (200.0, 0.0), (210.0, 0.0), (220.0, 0.0), (230.0, 0.0),
(240.0, 0.0), (250.0, 0.0), (260.0, 0.0), (270.0, 0.0), (280.0, 0.0), (290.0, 0.0), (300.0, 0.0), (310.0, 0.0),
(320.0, 0.0), (330.0, 0.0), (340.0, 0.0), (350.0, 0.0), (360.0, 0.0), (370.0, 0.0), (380.0, 0.0), (390.0, 0.0),
(400.0, 0.0), (405.0, 15.0), (400.0, 30.0), (390.0, 30.0), (380.0, 30.0), (370.0, 30.0), (360.0, 30.0), (350.0, 30.0),
(340.0, 30.0), (330.0, 30.0), (320.0, 30.0), (310.0, 30.0), (300.0, 30.0), (290.0, 30.0), (280.0, 30.0), (270.0, 30.0),
(260.0, 30.0), (250.0, 30.0), (240.0, 30.0), (230.0, 30.0), (220.0, 30.0), (210.0, 30.0), (200.0, 30.0), (190.0, 30.0),
(180.0, 30.0), (170.0, 30.0), (160.0, 30.0), (150.0, 30.0), (140.0, 30.0), (130.0, 30.0), (120.0, 30.0), (110.0, 30.0),
(100.0, 30.0), (90.0, 30.0), (80.0, 30.0), (70.0, 30.0), (60.0, 30.0), (50.0, 30.0), (40.0, 30.0), (30.0, 30.0),
(20.0, 30.0), (10.0, 30.0), (0.0, 30.0), (−5.0, 15.0)
)

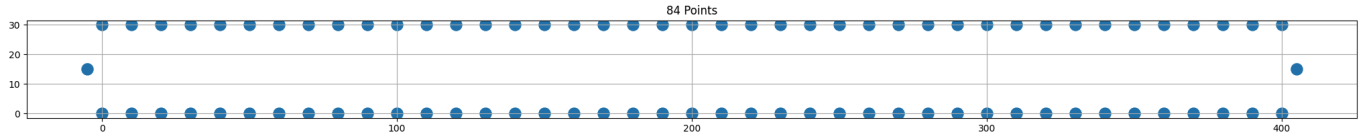

Abbildung 12: wenigerkrumm1.txt

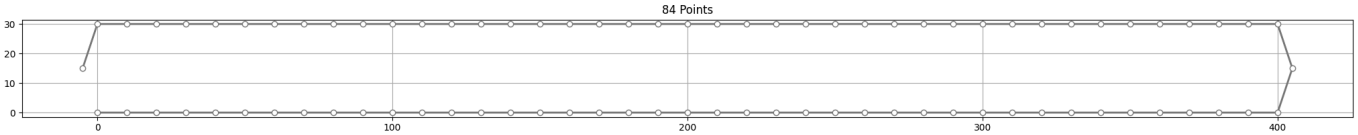

Abbildung 13: wenigerkrumm1.txt - Loesung

8.2. Beispiel 2 - „wenigerkrumm2.txt “ (84 Punkte)

Das zweite Beispiel der BwInf-Website, mit 60 Punkten, sieht wie folgt aus:

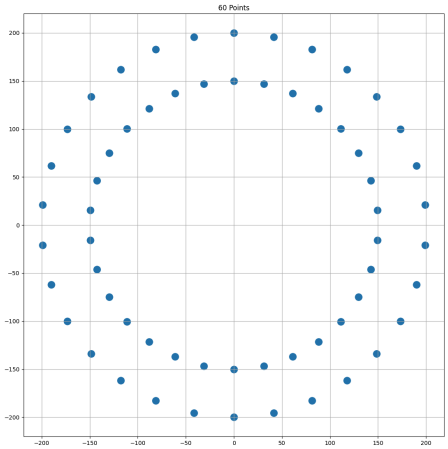

Abbildung 14: wenigerkrumm2.txt

Die beste gefundene Route hat die Laenge 2183.662km und sieht so aus:

$$Q_2 = \Big($$
$(-117.55705, -161.803399), (-81.347329, -182.709092), (-41.582338, -195.62952), (0.0, -200.0),$
$(41.582338, -195.62952), (81.347329, -182.709092), (117.55705, -161.803399), (148.628965, -133.826121),$
$(173.205081, -100.0), (190.211303, -61.803399), (198.904379, -20.905693), (198.904379, 20.905693),$
$(190.211303, 61.803399), (173.205081, 100.0), (148.628965, 133.826121), (117.55705, 161.803399),$
$(81.347329, 182.709092), (41.582338, 195.62952), (0.0, 200.0), (-41.582338, 195.62952),$
$(-81.347329, 182.709092), (-117.55705, 161.803399), (-148.628965, 133.826121), (-173.205081, 100.0),$
$(-190.211303, 61.803399), (-198.904379, 20.905693), (-198.904379, -20.905693), (-190.211303, -61.803399),$
$(-173.205081, -100.0), (-148.628965, -133.826121), (-88.167788, -121.352549), (-61.010496, -137.031819),$
$(-31.186754, -146.72214), (0.0, -150.0), (31.186754, -146.72214), (61.010496, -137.031819),$
$(88.167788, -121.352549), (111.471724, -100.369591), (129.903811, -75.0), (142.658477, -46.352549),$
$(149.178284, -15.679269), (149.178284, 15.679269), (142.658477, 46.352549), (129.903811, 75.0),$
$(111.471724, 100.369591), (88.167788, 121.352549), (61.010496, 137.031819), (31.186754, 146.72214),$
$(0.0, 150.0), (-31.186754, 146.72214), (-61.010496, 137.031819), (-88.167788, 121.352549),$
$(-111.471724, 100.369591), (-129.903811, 75.0), (-142.658477, 46.352549), (-149.178284, 15.679269),$
$(-149.178284, -15.679269), (-142.658477, -46.352549), (-129.903811, -75.0), (-111.471724, -100.369591)$
$\Big)$

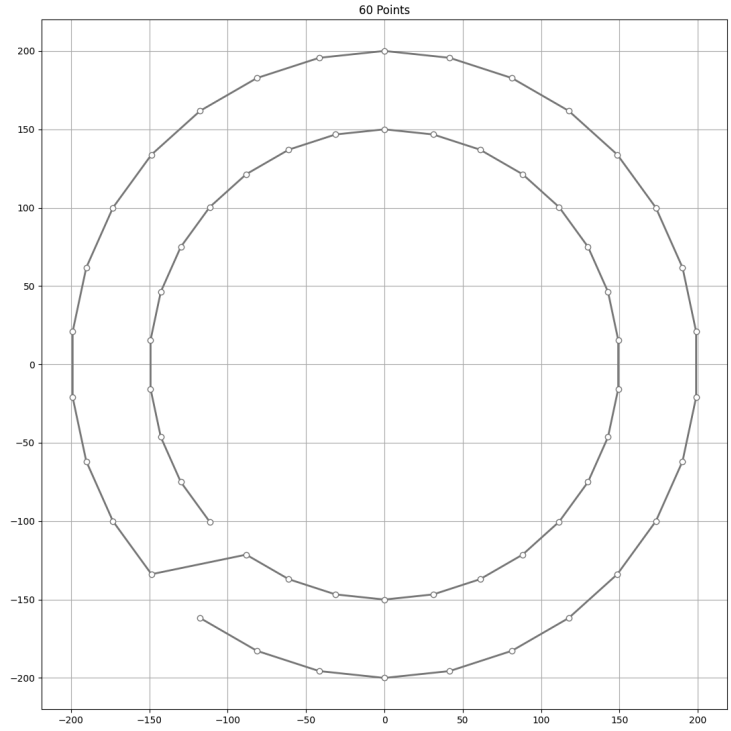

Abbildung 15: wenigerkrumm2.txt - Loesung

8.3. Beispiel 3 - „wenigerkrumm3.txt “ (120 Punkte)

Das dritte Beispiel besteht aus 120 Punkten und sieht wie folgt aus:

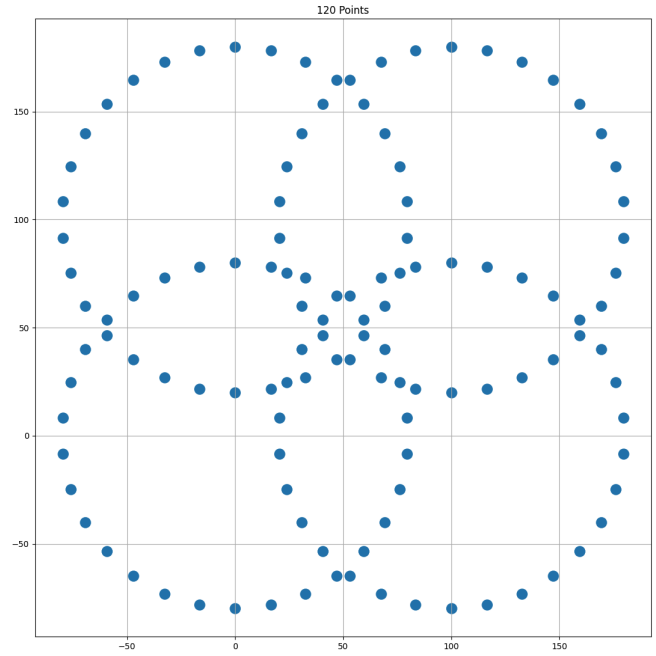

Abbildung 16: wenigerkrumm3.txt

Dieses Beispiel hat, anders als die vorherigen zwei Beispiele, je nach Algorithmus (1 oder 2) unterschiedliche Loesungen. Der ersten Algorithmus bietet nach 114ms eine Loesung der Laenge 1962.277km, waehrend der zweite die Laenge nach ca. 7s die Laenge 1944.603km erreicht. Aus Platzgruenden wird im Folgenden immer nur die beste Route gezeigt.

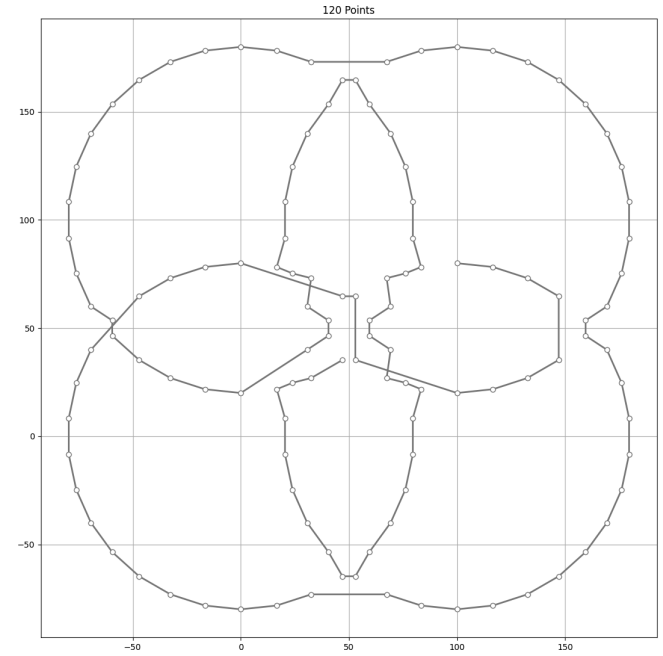

Abbildung 17: wenigerkrumm3.txt - Loesung

$Q_3 = \Big($
(47.02282, 35.27864), (32.538931, 26.916363), (23.915479, 24.72136), (16.632935, 21.748192),
(20.438248, 8.362277), (20.438248, −8.362277), (23.915479, −24.72136), (30.717968, −40.0),
(40.548414, −53.530449), (47.02282, −64.72136), (52.97718, −64.72136), (59.451586, −53.530449),
(69.282032, −40.0), (76.084521, −24.72136), (79.561752, −8.362277), (79.561752, 8.362277),
(83.367065, 21.748192), (76.084521, 24.72136), (67.461069, 26.916363), (69.282032, 40.0),
(59.451586, 46.469551), (59.451586, 53.530449), (69.282032, 60.0), (67.461069, 73.083637),
(76.084521, 75.27864), (83.367065, 78.251808), (79.561752, 91.637723), (79.561752, 108.362277),
(76.084521, 124.72136), (69.282032, 140.0), (59.451586, 153.530449), (52.97718, 164.72136),
(47.02282, 164.72136), (40.548414, 153.530449), (30.717968, 140.0), (23.915479, 124.72136),
(20.438248, 108.362277), (20.438248, 91.637723), (16.632935, 78.251808), (23.915479, 75.27864),
(32.538931, 73.083637), (30.717968, 60.0), (40.548414, 53.530449), (40.548414, 46.469551),
(30.717968, 40.0), (0.0, 20.0), (−16.632935, 21.748192), (−32.538931, 26.916363),
(−47.02282, 35.27864), (−59.451586, 46.469551), (−59.451586, 53.530449), (−69.282032, 60.0),
(−76.084521, 75.27864), (−79.561752, 91.637723), (−79.561752, 108.362277), (−76.084521, 124.72136),
(−69.282032, 140.0), (−59.451586, 153.530449), (−47.02282, 164.72136), (−32.538931, 173.083637),
(−16.632935, 178.251808), (0.0, 180.0), (16.632935, 178.251808), (32.538931, 173.083637),
(67.461069, 173.083637), (83.367065, 178.251808), (100.0, 180.0), (116.632935, 178.251808),
(132.538931, 173.083637), (147.02282, 164.72136), (159.451586, 153.530449), (169.282032, 140.0),
(176.084521, 124.72136), (179.561752, 108.362277), (179.561752, 91.637723), (176.084521, 75.27864),
(169.282032, 60.0), (159.451586, 53.530449), (159.451586, 46.469551), (169.282032, 40.0),
(176.084521, 24.72136), (179.561752, 8.362277), (179.561752, −8.362277), (176.084521, −24.72136),
(169.282032, −40.0), (159.451586, −53.530449), (147.02282, −64.72136), (132.538931, −73.083637),
(116.632935, −78.251808), (100.0, −80.0), (83.367065, −78.251808), (67.461069, −73.083637),
(32.538931, −73.083637), (16.632935, −78.251808), (0.0, −80.0), (−16.632935, −78.251808),
(−32.538931, −73.083637), (−47.02282, −64.72136), (−59.451586, −53.530449), (−69.282032, −40.0),
(−76.084521, −24.72136), (−79.561752, −8.362277), (−79.561752, 8.362277), (−76.084521, 24.72136),
(−69.282032, 40.0), (−47.02282, 64.72136), (−32.538931, 73.083637), (−16.632935, 78.251808),
(0.0, 80.0), (47.02282, 64.72136), (52.97718, 64.72136), (52.97718, 35.27864),
(100.0, 20.0), (116.632935, 21.748192), (132.538931, 26.916363), (147.02282, 35.27864),
(147.02282, 64.72136), (132.538931, 73.083637), (116.632935, 78.251808), (100.0, 80.0), $\Big)$

8.4. Beispiel 4 - „wenigerkrumm4.txt “ (25 Punkte)

Das vierte Beispiel besteht aus 25 Punkten und sieht wie folgt aus:

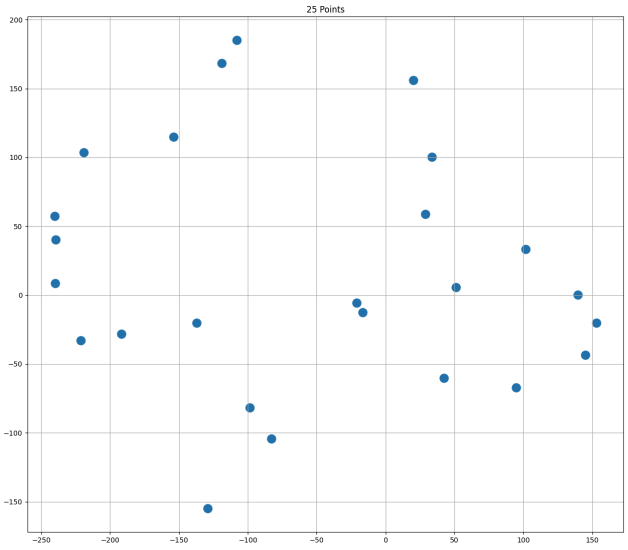

Abbildung 18: wenigerkrumm4.txt

Bei diesem Beispiel finden beide Algorithmen die selbe Loesung Q_4 der Laenge 1205.069km.

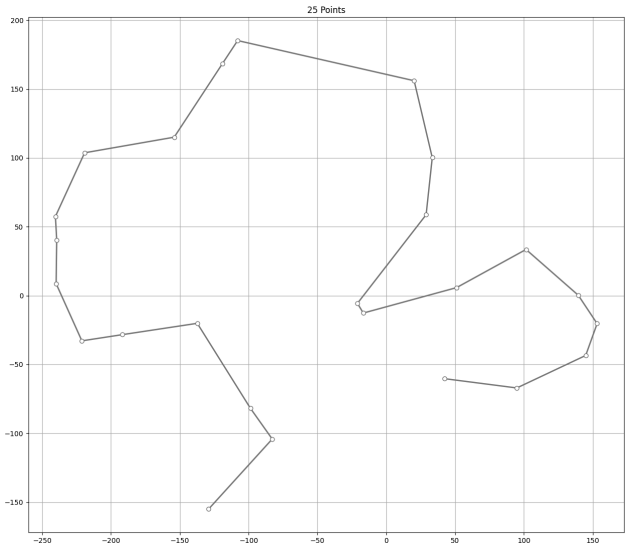

Abbildung 19: wenigerkrumm4.txt - Loesung

$Q_4 = \Big($
(42.137753, −60.319863), (94.789917, −67.087689), (144.832862, −43.476284), (153.130159, −20.36091), (139.446709, 0.233238), (101.498782, 33.484198), (51.00814, 5.769601), (−16.72313, −12.689542), (−20.971208, −5.637107), (28.913721, 58.69988), (33.379688, 100.161238), (20.212169, 156.013261), (−107.988514, 185.173669), (−119.026308, 168.453598), (−154.088455, 115.022553), (−219.148505, 103.685337), (−240.369194, 57.426131), (−239.414022, 40.427118), (−239.848226, 8.671399), (−221.149792, −32.862538), (−191.716829, −28.360492), (−137.317503, −20.146939), (−98.760442, −81.770618), (−82.864121, −104.1736), (−129.104485, −155.04164) $\Big)$

8.5. Beispiel 5 - „wenigerkrum5.txt “ (60 Punkte)

Das fuenfte Beispiel bestehend aus 60 Punkten (siehe Abb. 20) und hat wieder unterschiedliche Loesungen je nach Algorithmus (1 oder 2).
Der ersten Algorithmus bietet nach 21ms die Loesung Q_5 der Laenge 4062.040km und der zweite die Laenge 3854.277km nach 425ms. Erneut wird hier nur die bessere Route gezeigt.

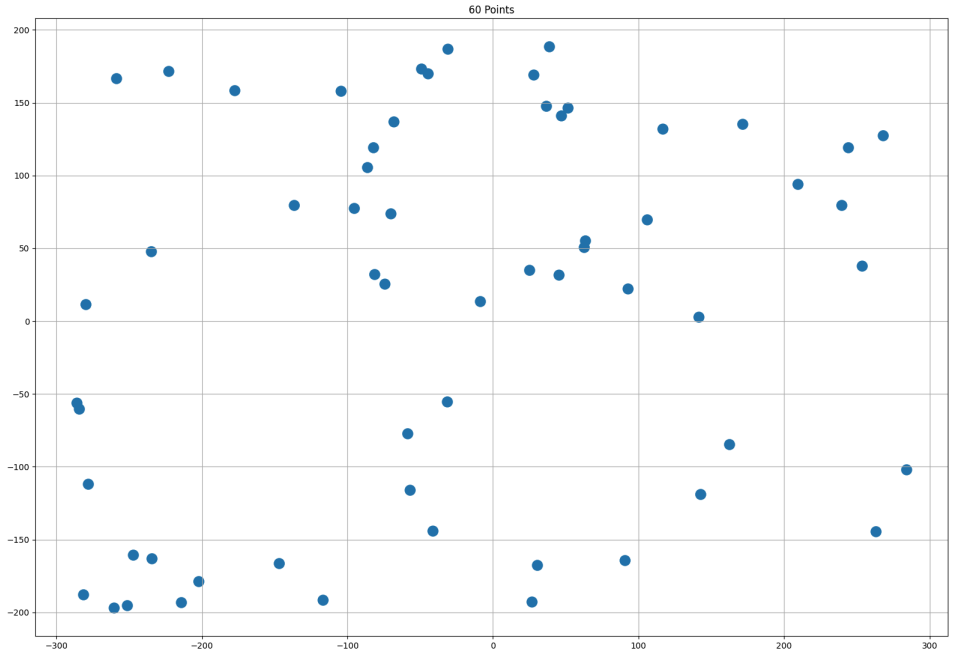

Abbildung 20: wenigerkrumm5.txt

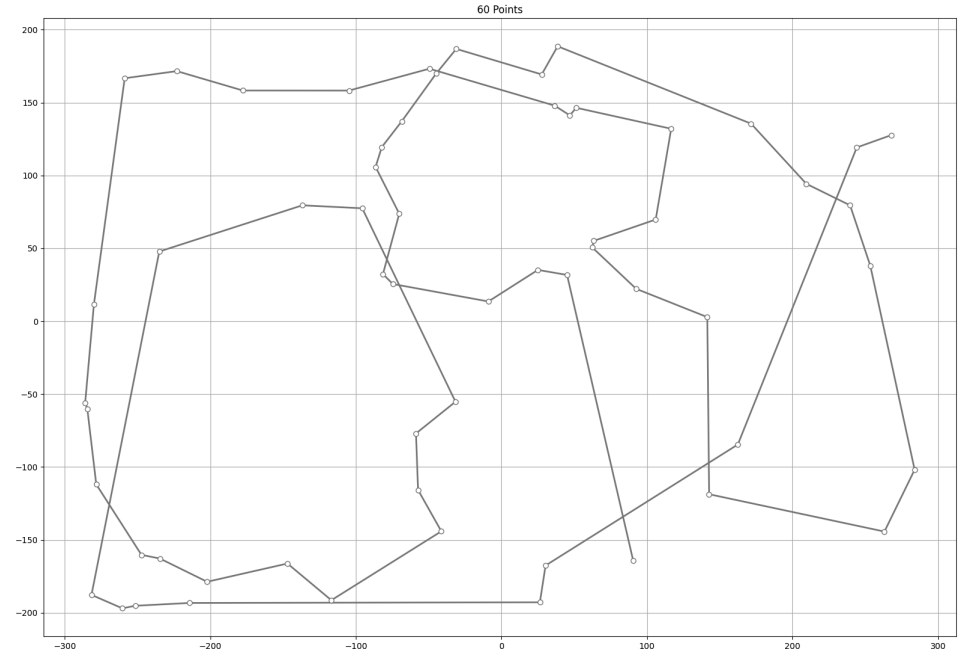

Abbildung 21: wenigerkrumm5.txt - Loesung

$Q_5 = \Big($
(90.584569, −164.218416), (45.123674, 31.740242), (25.098172, 35.205544), (−8.936916, 13.543851), (−74.639411, 25.542881), (−81.384378, 32.368323), (−70.183535, 73.738342), (−86.45758, 105.836348), (−82.17351, 119.465553), (−68.446198, 137.178953), (−44.669924, 170.088013), (−30.991436, 186.807059), (27.706327, 169.284192), (38.65473, 188.608557), (171.595574, 135.520994), (209.544977, 94.267052), (239.63955, 79.491132), (253.534863, 38.014987), (283.989938, −101.866465), (263.236651, −144.293091), (142.765554, −118.682439), (141.513053, 2.821137), (92.639946, 22.21603), (62.366656, 50.713913), (63.541591, 55.140221), (106.03343, 69.754891), (116.702667, 132.021991), (51.417675, 146.417721), (47.040512, 141.206562), (36.599805, 147.88535), (−49.447381, 173.210759), (−104.781549, 158.212048), (−177.685937, 158.265884), (−223.039999, 171.558368), (−258.868593, 166.669198), (−280.008136, 11.657786), (−286.024059, −55.955204), (−284.547616, −60.154961), (−278.409792, −111.914073), (−247.341131, −160.277639), (−234.711279, −162.774591), (−202.218443, −178.735864), (−147.023475, −166.22013), (−116.831788, −191.380552), (−41.263039, −144.118212), (−57.266232, −115.737582), (−58.684205, −76.988884), (−31.548604, −55.223912), (−95.621797, 77.468533), (−136.787038, 79.501703), (−235.099412, 47.810306), (−281.67899, −187.717923), (−260.477802, −196.955535), (−251.656688, −195.189953), (−214.362324, −193.26519), (26.451074, −192.813352), (30.366828, −167.573232), (162.493244, −84.574019), (244.228552, 119.192512), (267.845908, 127.627482) $\Big)$

8.6. Beispiel 6 - „wenigerkrum6.txt “ (80 Punkte)

Das sechste Beispiel bestehend aus 80 Punkten und ist in Abbildung 22 dargestellt. Bei diesem Beispiel findet Algorithmus 1 keine Loesung, der zweite Algorithmus hingegen schon. Nach 875ms erreicht dieser eine Loesung (Q_6) der Laenge 4052.955km (siehe Abbildung 23).

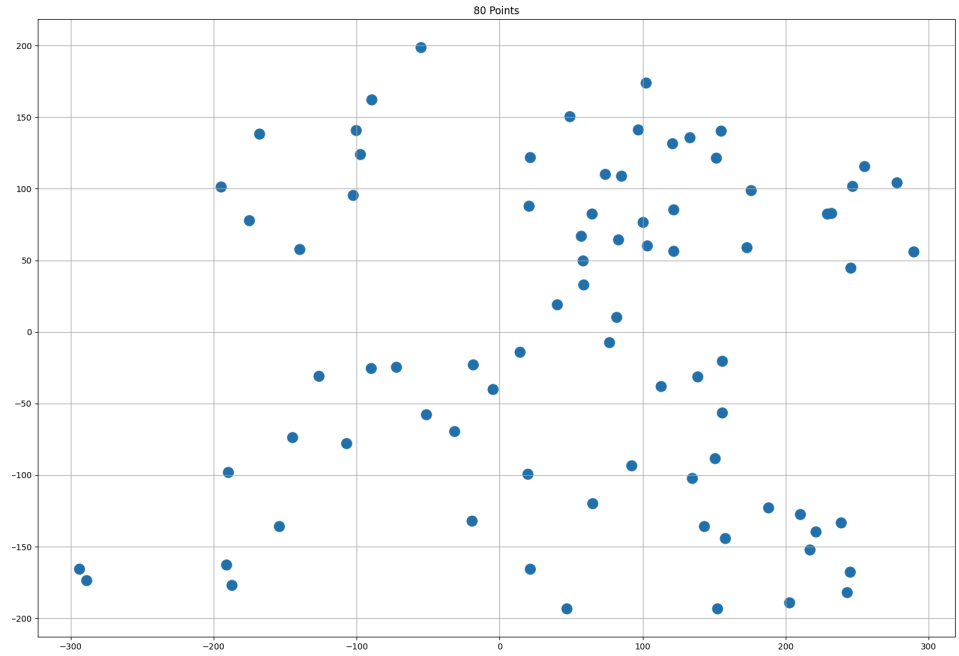

Abbildung 22: wenigerkrumm6.txt

$$Q_6 = \Big($$
(238.583388, −133.143524), (150.526118, −88.230057), (134.692592, −102.152826), (143.114152, −135.866707), (157.588994, −144.200765), (187.669263, −122.655771), (210.186432, −127.403563), (221.028639, −139.435079), (216.82592, −152.024123), (245.020791, −167.448848), (242.810288, −182.054289), (202.34698, −189.069699), (152.102728, −193.381252), (46.674278, −193.090008), (21.176627, −165.421555), (−19.310012, −131.810965), (−31.745416, −69.20796), (−51.343758, −57.654823), (−72.565291, −24.28182), (−90.16019, −25.200829), (−126.569816, −30.645224), (−144.887799, −73.49541), (−189.988471, −98.043874), (−191.216327, −162.689024), (−187.485329, −177.031237), (64.943433, −119.784474), (92.25582, −93.514104), (112.833346, −38.057607), (138.136997, −31.348808), (155.341949, −20.252779), (172.836936, 59.184233), (175.677917, 98.929343), (151.432196, 121.427337), (154.870684, 140.32766), (102.223372, 174.201904), (49.091876, 150.678826), (21.067444, 122.164599), (20.21829, 88.031173), (56.716498, 66.959624), (58.019653, 49.937906), (58.71662, 32.83593), (40.327635, 19.216022), (14.005617, −14.015334), (−4.590656, −40.067226), (19.765322, −99.2364), (155.405344, −56.437901), (121.392544, 56.694081), (102.909291, 60.107868), (100.006932, 76.579303), (121.661135, 85.267672), (120.906436, 131.79881), (132.794476, 135.681392), (228.929427, 82.624982), (231.944823, 82.961057), (246.621634, 101.705861), (255.134924, 115.594915), (277.821597, 104.262606), (289.298882, 56.051342), (245.415055, 44.794067), (83.005905, 64.646774), (64.559003, 82.567627), (73.689751, 110.224271), (85.04383, 108.946389), (96.781707, 141.370805), (−55.091518, 198.826966), (−89.453831, 162.237392), (−100.569041, 140.808607), (−97.391662, 124.120512), (−102.699992, 95.632069), (−139.74158, 57.93668), (−175.118239, 77.842636), (−194.986965, 101.363745), (−167.994506, 138.195365), (81.740403, 10.276251), (76.647124, −7.289705), (−18.507391, −22.90527), (−107.196865, −77.792599), (−154.225945, −135.522059), (−288.744132, −173.349893), (−293.833463, −165.440105) $\Big)$

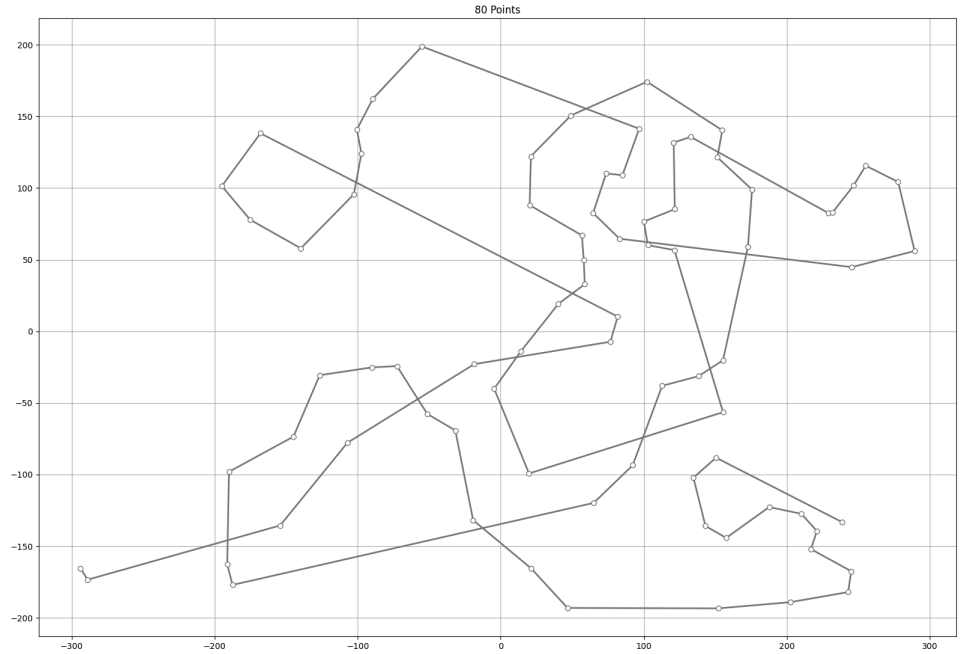

Abbildung 23: wenigerkrumm6.txt - Loesung

8.7. Beispiel 7 - „wenigerkrum7.txt “ (100 Punkte)

Das siebte und letzte Beispiel der BwInf-Website bestehend aus 100 Punkten und ist in Abbildung 24 dargestellt. Auch bei diesem Beispiel findet Algorithmus 1 keine Loesung, der zweite Algorithmus allerdings schon. Nach ca 2,5s erreicht dieser die Loesung Q_7 der Laenge 5035.6855km (siehe Abbildung 25).

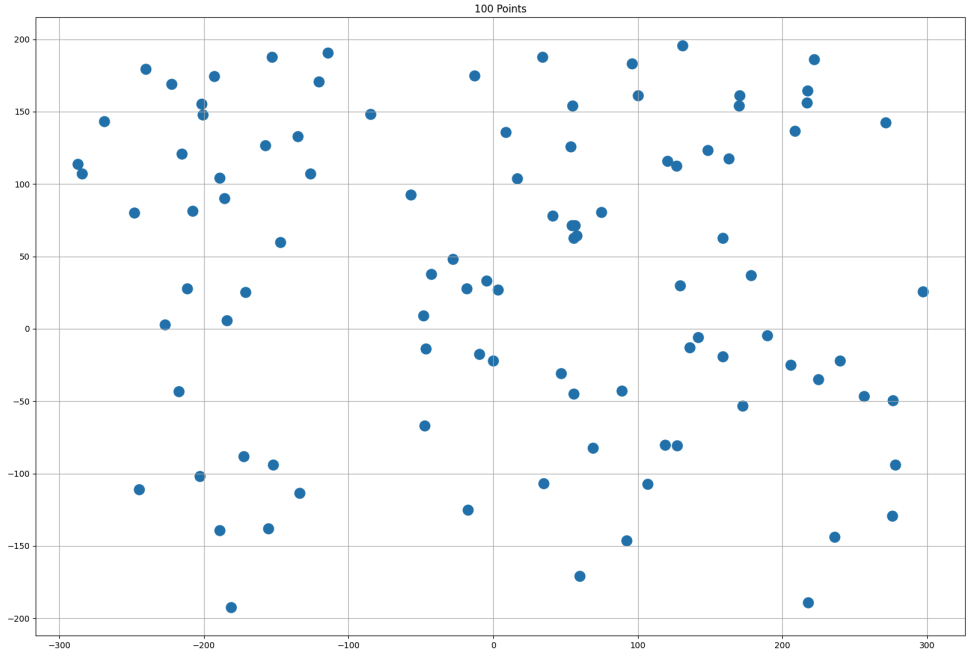

Abbildung 24: wenigerkrumm7.txt

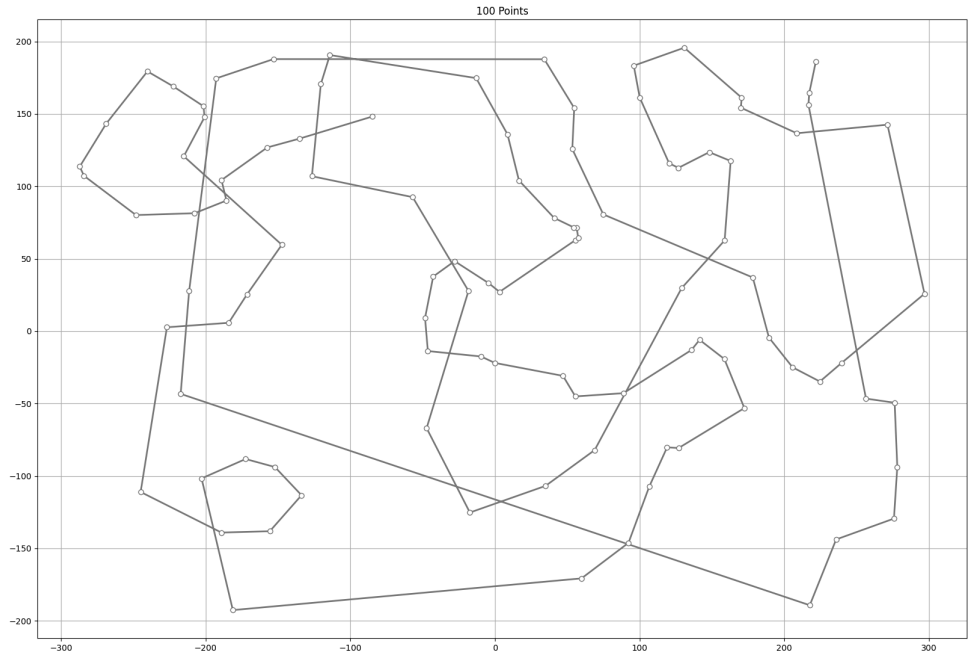

Abbildung 25: wenigerkrumm7.txt - Loesung

$Q_7 = \Big($
(−84.6269, 148.216494), (−134.985023, 132.944989), (−157.423365, 126.800331), (−189.062172, 104.285631),
(−185.649161, 90.144456), (−207.665172, 81.410371), (−248.169463, 80.132237), (−284.129027, 107.252583),
(−287.058297, 113.599823), (−268.739142, 143.276483), (−240.249363, 179.334919), (−222.492322, 169.033315),
(−201.485143, 155.27483), (−200.771246, 147.741054), (−215.113949, 120.740679), (−147.363185, 59.608175),
(−171.354954, 25.463068), (−184.0927, 5.737284), (−226.787625, 2.658862), (−244.959501, −111.046573),
(−189.135201, −139.078513), (−155.651746, −138.151811), (−133.730932, −113.306155), (−152.130365, −93.844349),
(−172.378071, −88.298187), (−202.828627, −101.70005), (−181.208895, −192.622935), (59.8272, −170.713714),
(92.29804, −146.169487), (106.599423, −107.433987), (118.989764, −80.203583), (126.904044, −80.733297),
(172.389228, −53.13327), (158.552316, −19.254407), (141.433472, −6.023095), (135.781192, −13.05344),
(88.818853, −42.834512), (55.550895, −45.089968), (47.011363, −30.88718), (−0.200936, −21.927663),
(−9.580869, −17.516639), (−46.403062, −13.755804), (−48.354421, 9.091412), (−42.704691, 37.679514),
(−27.911955, 48.326745), (−4.434919, 33.164884), (3.152113, 27.10389), (55.434792, 62.72916),
(57.555364, 64.417343), (56.389778, 71.618509), (54.551865, 71.567133), (40.897139, 78.152317),
(16.573231, 104.020979), (8.64366, 135.90743), (−13.030259, 174.698005), (−114.146166, 190.615321),
(−120.386562, 170.589454), (−126.568914, 106.964962), (−56.914543, 92.501249), (−18.316063, 27.75586),
(−47.266557, −66.984045), (−17.356579, −125.254131), (34.959132, −106.842499), (68.910854, −82.123346),
(129.024315, 29.701695), (158.742184, 62.618834), (162.923138, 117.465744), (148.108328, 123.558283),
(126.799911, 112.72728), (120.375033, 115.889661), (100.043893, 161.295125), (95.947213, 183.278211),
(130.854855, 195.695082), (170.514252, 161.16985), (169.990437, 154.260412), (208.592696, 136.61846),
(271.301094, 142.524086), (296.911892, 25.811569), (239.616628, −21.94416), (224.599361, −34.798485),
(205.717887, −24.976511), (189.387028, −4.465225), (178.19836, 37.031984), (74.8875, 80.586458),
(53.436521, 125.683201), (54.766523, 154.053847), (34.079032, 187.731112), (−153.13014, 187.817274),
(−192.681053, 174.522947), (−211.429137, 27.770425), (−217.28247, −43.316616), (217.599278, −189.258062),
(235.827007, −143.838844), (275.793495, −129.415477), (278.105364, −93.771765), (276.276517, −49.448662),
(256.475967, −46.591418), (216.691, 156.31437), (217.218893, 164.294928), (221.808162, 186.241012) $\Big)$

8.8. Eigenes Beispiel - Deutschlands 20 groeszte Staedte

Um nun die exakten Loesungsverfahren zu testen, wird im Weiteren eine Route zwischen den 10, bzw. 20 groeszten Staedten Deutschlands nach Einwohneranzahl[26] gesucht, die sich an die Abbiegewinkelbedingung halten soll.
Fuer die Punkte, die die Staedte darstellen, wurden dabei die echten Koordinaten[27] der Staete verwendet. Beispielsweise hat Berlin den Breitengrad 52.523 (Nord), und 13.413 (Ost), sodass diese Stadt durch den Punkt (13.413, 52.523) dargestellt wird.
Da die Erde nicht flach ist[28], sind die Laengengrade nicht so weit voneinander entfernt, wie die Breitengrade. Waehrend der Abstand von zwei Breitengraden stets genau 222km ist, sind die Laengengrade aufgrund der Kugelform der Erde nicht an allen Stellen gleichweit voneinander entfernt. So ist der zehnte Laengengrad in Bayern (48. Breitengrad) ca. 150km vom zwölften entfernt, waehrend dies im Norden (54. Breitengrad) nur noch ca. 130km sind und zwei Laengengrade am Aequator genau 224.18km voneinander entfernt sind.
Da es allerdings aufgrund der Berechnung der Distanz und der Abbiegewinkelbedingung wichtig ist, dass Breitengrade und Laengengrade immer gleichweit vonander und gegenseitig entfernt sind, wird so getan als ob dies der Fall waere und die Grafik ist dementspraechend verzert.

Im Folgenden werden die vier verschiedenen Bruteforce Algorithmen betrachet, die zuvor vorgestellt wurden:

1. Algorthmus 3 - Iteratives ausprobieren aller Permutationen mit Hilfe des Heap-Algorithmus.
2. Algorithmus 5 - Iteratives ausprobieren der Haelfte aller Permutationen.
3. Algorithmus 4 - Rekursives ausprobieren aller Permutationen.
4. Algorithmus 6 - Rekursives ausprobieren aller Permutationen mit Abbruchbedinungen (Branch-and-Bound, und Backtracking).

In der Tabelle 2 ist dargestellt, wie lange die vier unterschiedlichen Algorithmen brauchen um ein Route fuer m (5 bis 20) Punkte (Staedte) zu finden. Dabei wurde je aufgehoert, die Algorithmen zu benutzen, nachdem diese laenger als eine Minute gebraucht haben. Die dazugehoerigen Graphen findet mit der logarithmische Darstellung in Abbildung 26.

m = Anzahl Staedte	Algorthmus 3	Algorithmus 5	Algorithmus 4	Algorithmus 6
5	1ms	1ms	1ms	2ms
6	1ms	1ms	2ms	2ms
7	5ms	2ms	12ms	2ms
8	13ms	5ms	31ms	3ms
9	50ms	19ms	177ms	3ms
10	257ms	142ms	1162ms	7ms
11	2,266s	1,143s	11,876s	9ms
12	26,365s	13,228s	2,447min	21ms
13	5,722min	3,062min	?	39ms
14	?	?	?	114ms
15	?	?	?	408ms
16	?	?	?	631ms
17	?	?	?	1,756s
18	?	?	?	4,375s
19	?	?	?	18,887s
20	?	?	?	60,271s

Tabelle 2: Uebersicht ueber die Laufzeit der unterschiedlichen Bruteforce Algorithmen

[26] 2021, nach `https://de.wikipedia.org/wiki/Liste_der_GroÃ§stÃ¤dte_in_Deutschland`
[27] Nach `https://www.fwiegleb.de/geodat/geo-cdef/`
[28] fuer dieses Beispiel sehr unpraktisch

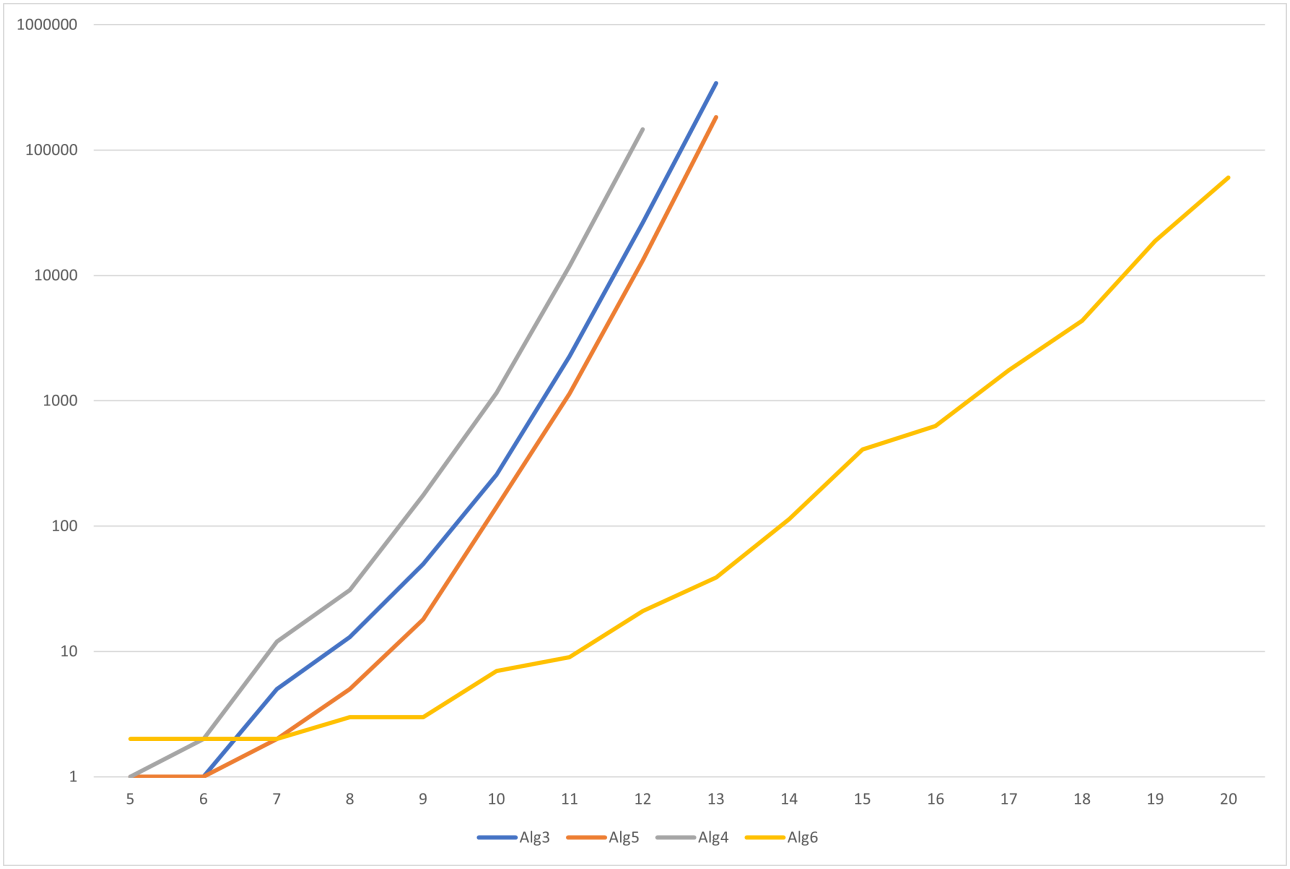

Abbildung 26: Vergleich der Laufzeiten der vier Algorithmen

Durch Abbildung 26 laesst sich erkennen, dass alle Algorithmen eine expontielle Laufzeit haben, wobei und die Laufzeit des rekursiven Algorithmus 4, der alle Permutationen ausprobiert am schnellsten wachst. Weiter ist logischer weise die Laufzeit des Algorithmus 5 knapp unter der des Algorithmus 3, da die Anzahl der getesteten Permutationen sich um einen Faktor 2 unterscheidet. Zuletetzt sieht man, dass die Laufzeit des Algorithmus 6 deutlich langsamer waechst als die anderen, da dieser sehr viele moegliche Routen garnicht erst ausprobiert.

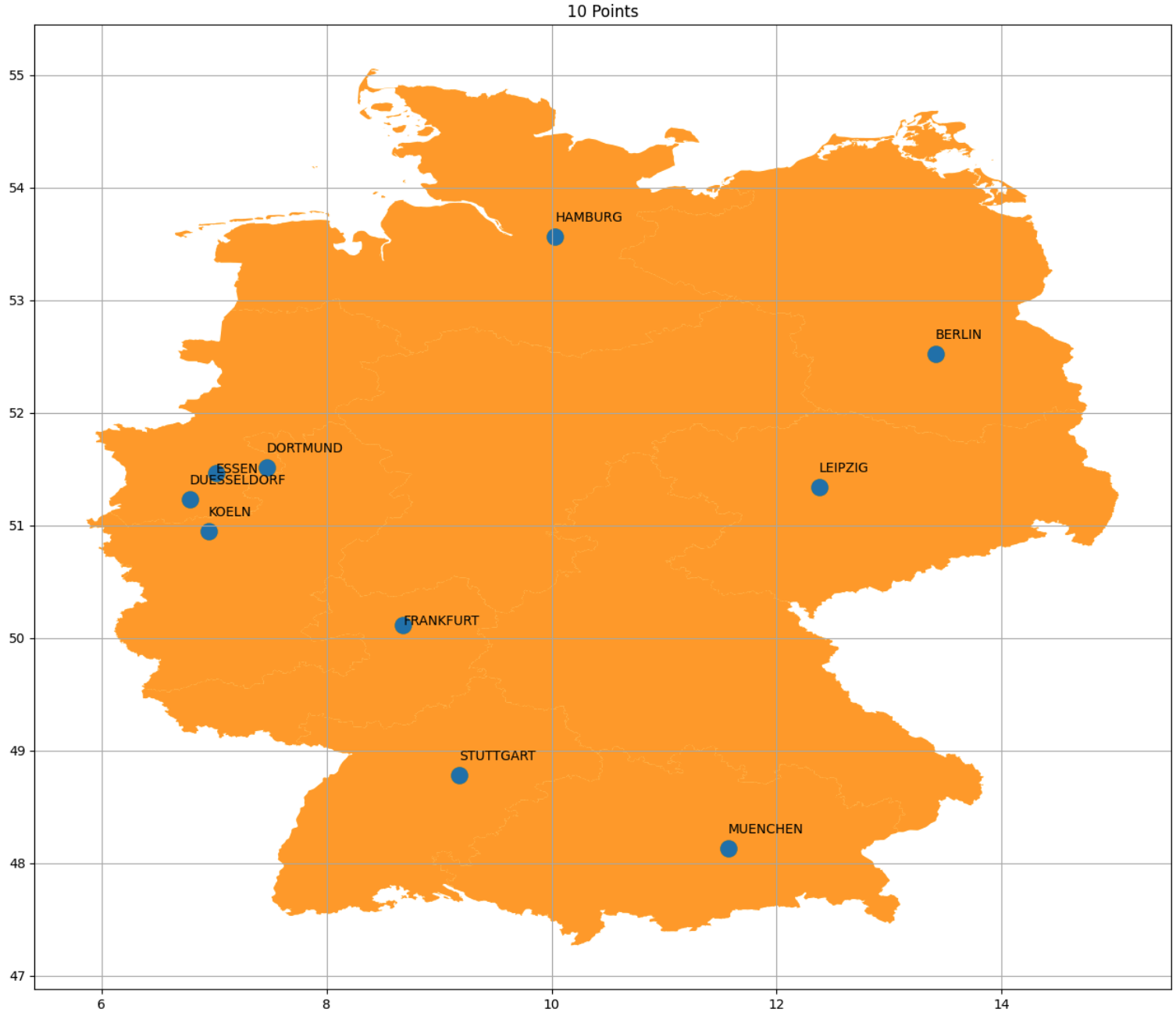

Abbildung 27: Top 10 deutsche Staedte nach Einwohnerzahl

Die 10 groeszten Staedte Deutschlands nach Einwohnerzahl im Jahr 2021 mit den genutzten Koordinaten:

1. Berlin, (13.413, 52.523)
2. Hamburg (10.033, 53.567)
3. Muenchen (11.567, 48.133)
4. Koeln (6.950, 50.950)
5. Frankwurt am Main (8.683, 50.117)
6. Stuttgart (9.183, 48.783)
7. Duesseldorf (6.783, 51.233)
8. Leipzig (12.377, 51.339)
9. Dortmund (7.467, 51.517)
10. Essen (7.017, 51.467)

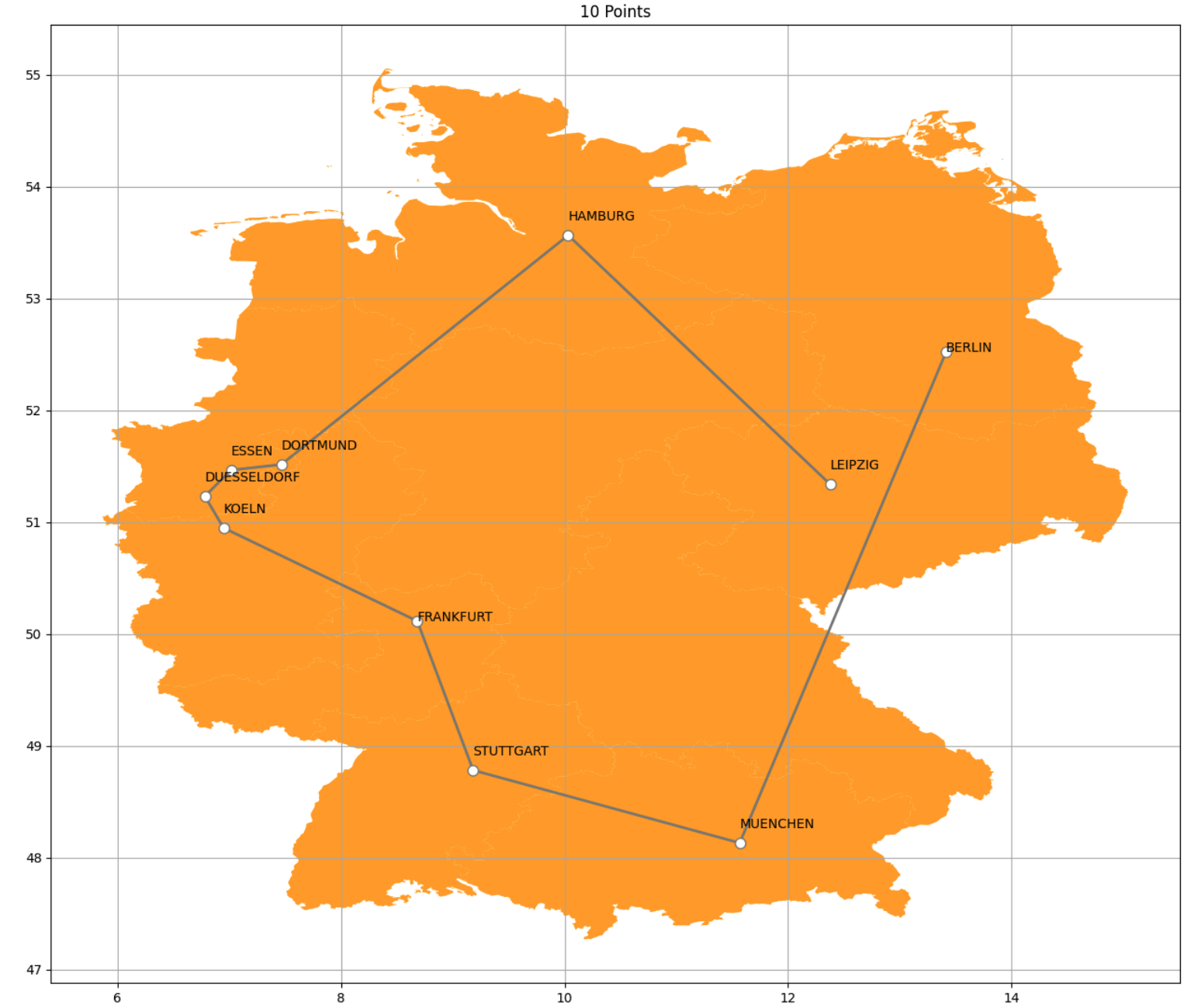

Abbildung 28: Route, die die Top 10 deutsche Staedte verbindet

Top 10 Staedte - Route:
Um die beste Route zu finden, die die 10 groeszten Staedte verbindet, laesst sich also jeder der vier Algorithmen problemlos verwenden (vergl. Tabelle 2). Diese Staedte sind in Abbildung 30 dargestellt.
Die best moegliche Route ist die folgende der Laenge 18.211 (keine sinnvolle Einheit):

$$Q = \Big((12.377,\ 51.339),\ (10.033,\ 53.567),\ (7.467,\ 51.517),\ (7.017,\ 51.467),\ (6.783,\ 51.233),$$
$$(6.95,\ 50.95),\ (8.683,\ 50.117),\ (9.183,\ 48.783),\ (11.567,\ 48.133),\ (13.413,\ 52.523) \Big)$$

Erklaerung:

$$Q = \Big(\text{LEIPZIG},\ \text{HAMBURG},\ \text{DORTMUND},\ \text{ESSEN},\ \text{DUESSELDORF},$$
$$\text{KOELN},\ \text{FRANKFURT AM MAIN},\ \text{STUTTGART},\ \text{MUENCHEN},\ \text{BERLIN} \Big)$$

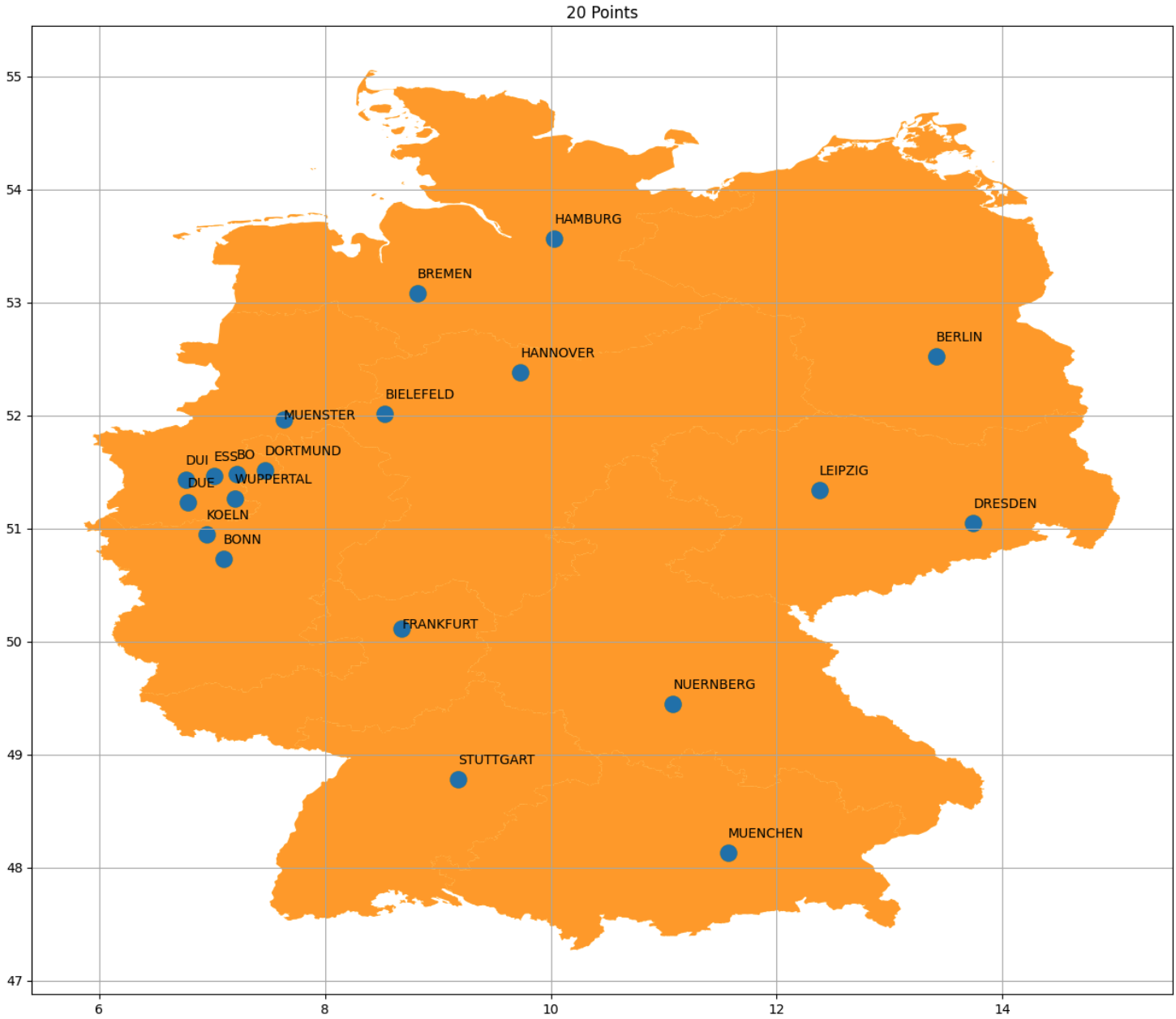

Abbildung 29: Top 20 deutsche Staedte nach Einwohnerzahl

Die 20 groeszten Staedte Deutschlands nach Einwohnerzahl im Jahr 2021 mit den genutzten Koordinaten und Abkuerzungen:

1. Berlin, (13.413, 52.523)
2. Hamburg (10.033, 53.567)
3. Muenchen (11.567, 48.133)
4. Koeln (6.950, 50.950)
5. Frankwurt am Main (8.683, 50.117)
6. Stuttgart (9.183, 48.783)
7. Duesseldorf (DUE) (6.783, 51.233)
8. Leipzig (12.377, 51.339)
9. Dortmund (7.467, 51.517)
10. Essen (ESS) (7.017, 51.467)
11. Bremen, (8.817, 53.083)
12. Dresden (13.739, 51.050)
13. Hannover (9.733, 52.383)
14. Nuernberg (11.083, 49.450)
15. Duisburg (DUI) (6.767, 51.433)
16. Buchum (BO) (7.217, 51.483)
17. Wuppertal (7.200, 51.267)
18. Bielefeld (8.533, 52.017)
19. Bonn (7.100, 50.733)
20. Muenster (7.633, 51.967)

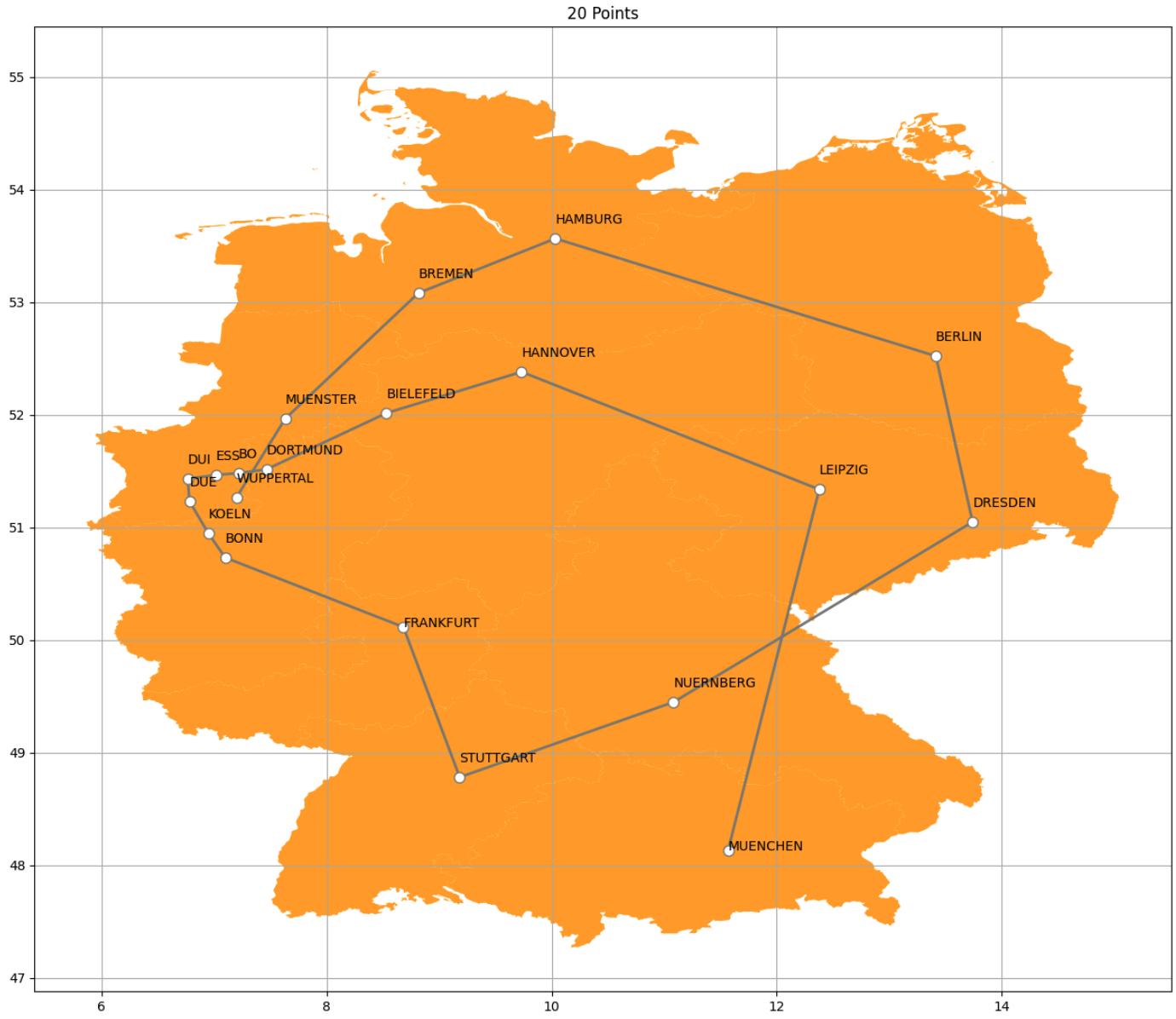

Abbildung 30: Route, die die Top 20 deutsche Staedte verbindet

Top 20 Staedte - Route:
Die Route durch die 20 groeszten Staedte hingegen, laesst sich exakt nur von Algorithmus 6 bestimmen. Dieser findet eine Route der Laenge 27.122 (Einheiten):

$$Q = \Big((7.2,\ 51.267),\ (7.633,\ 51.967),\ (8.817,\ 53.083),\ (10.033,\ 53.567),\ (13.413,\ 52.523),\ (13.739,\ 51.05),\ (11.083,\ 49.45),\ (9.183,\ 48.783),\ (8.683,\ 50.117),\ (7.1,\ 50.733),\ (6.95,\ 50.95),\ (6.783,\ 51.233),\ (6.767,\ 51.433),\ (7.017,\ 51.467),\ (7.217,\ 51.483),\ (7.467,\ 51.517),\ (8.533,\ 52.017),\ (9.733,\ 52.383),\ (12.377,\ 51.339),\ (11.567,\ 48.133) \Big)$$

Erklaerung:

$$Q = \Big(\text{WUPPERTAL},\ \text{MUENSTER},\ \text{BREMEN},\ \text{HAMBURG},\ \text{BERLIN},\ \text{DRESDEN},\ \text{NUERNBERG},\ \text{STUTTGART},\ \text{FRANKFURT AM MAIN},\ \text{BONN},\ \text{KOELN},\ \text{DUE - DUESSELDORF},\ \text{DUI - DUISBURG},\ \text{ESS - ESSEN},\ \text{BO - BOCHUM},\ \text{DORTMUND},\ \text{BIELEFELD},\ \text{HANNOVER},\ \text{LEIPZIG},\ \text{MUENCHEN} \Big)$$

9. Quellcode

Nun folgt der Quellcode der wichtigsten Teile der Loesungsvorschlaege.

9.1. Allgemein

Uebersicht ueber die Klasse Vector:
Im Folgende wird bei einigen Methoden der Rumpf und die Kommentierung ausgelassen.

```
1  public class Vector {
2
3      /**
4       * The point's dimension (>= 0)
5       */
6      private final int dimension;
7
8      /**
9       * The vector's components represented as double-array of length dimension (>= 0)
10      */
11     private final double[] components;
12
13     /**
14      * Constructor taking in the vector's components and if the given array should be cloned
15      *
16      * @param components The Vectors components
17      * @param cloneArray If the given array should be cloned or not
18      */
19     public Vector(double[] components, boolean cloneArray) {
20         this.dimension = components.length;
21         this.components = cloneArray ? components.clone() : components;
22     }
23
24     /**
25      * Public methode to compute the length of this vector to the power of 2.
26      * This method might be useful because it's easier to compute than the actual length of the vector.
27      *
28      * @return The length of this vector to the power of 2
29      */
30     public double lengthSquared() {
31         double length = 0D;
32
33         // Sum up the squares of the single components of the vector
34         for (int i = 0; i < this.dimension; i++) {
35             length += this.components[i] * this.components[i];
36         }
37
38         return length;
39     }
40
41     /**
42      * Public methode compute the dot product of this vector with another one.
43      *
44      * @param that The other vector
45      * @return The dot product of this vector with the other one
46      * @throws IllegalArgumentException If the vector don't have the same dimension
47      */
48     public double dotProduct(Vector that) {
49         // Making sure the vectors have the same dimension
50         if (that.dimension != this.dimension) throw new IllegalArgumentException();
51
52         double product = 0D;
53
54         for (int i = 0; i < this.dimension; i++) {
55             product += this.components[i] * that.components[i];
56         }
57
58         return product;
59     }
60
61     /**
62      * Public methode compute the cosine of the angle between this vector and another one.
63      *
```

```
     * @param that The other vector
     * @return The cosine of this vector and the other one
     * @throws IllegalArgumentException If the vectors don't have the same dimension
     */
    public double cos(Vector that) {
        return (this.dotProduct(that)) / (this.length() * that.length());
    }

    /**
     * Public method compute this vector's length using the functionMath#sqrt
     * and the lengthSquared-methode of the class.
     *
     * @return This vector's length
     */
    public double length() {
        return Math.sqrt(lengthSquared());
    }

    public String toString() { }

    public boolean equals(Object o) { }

    public int hashCode() { }
}
```

Uebersich ueber die Klasse Point:
Im Folgende wird erneut bei einigen Methoden der Rumpf und die Kommentierung ausgelassen.

```
public class Point {

    /**
     * The point's dimension (>= 0)
     */
    private final int dimension;

    /**
     * The point's coordinates represented as a double-array of length dimension (>= 0).
     */
    private final double[] coordinates;

    /**
     * Constructor taking in the point's coordinates and if the given array should be cloned.
     *
     * @param coordinates The point's coordinates
     * @param cloneArray If the given array should be cloned or not
     */
    public Point(double[] coordinates, boolean cloneArray) {
        this.dimension = coordinates.length;
        this.coordinates = cloneArray ? coordinates.clone() : coordinates;
    }

    /**
     * Constructor taking in the point's coordinates (always cloning the given array).
     *
     * @param coordinates The point's coordinates
     */
    public Point(double[] coordinates) {
        this(coordinates, true);
    }

    /**
     * Public method returning the connecting vector from this point to another one.
     *
     * @param that The other point
     * @return The connecting vector from this point to another one
     * @throws IllegalArgumentException If both points don't share the same dimension.
     */
    public Vector vectorToPoint(Point that) {
        // Check both points share the same dimension.
        if (that.getDimension() != this.getDimension()) throw new IllegalArgumentException();

        // Create a double-array that will contain the connecting vector's components
        // and fill it up with those components
        double[] components = new double[this.getDimension()];
```

```
47
            for (int i = 0; i < this.dimension; i++) {
49              components[i] = that.getCoordinate(i) - this.getCoordinate(i);
            }
51
            // Return a new vector with the computed components
53          return new Vector(components, false);
        }
55
        /**
57       * Public methode computing the distance from the point to another one
         * using the connecting vector.
59       *
         * @param that The other point
61       * @return The distance from this point to the other point
         * @throws IllegalArgumentException If the points do not share the same dimension
63       */
        public double distance(Point that) {
65          return this.vectorToPoint(that).length();
        }
67
        public int getDimension() { }
69
        public double getCoordinate(int i) { }
71
        public String toString() { }
73
        public boolean equals(Object o) { }
75
        public int hashCode() { }
77  }
```

Funktionen zum bestimmen der Laenge einer Route:
Die folgenden beiden Funktionen sind Teil der Klasse *Utils* und berechnen die Laenge einer, in Form einer Liste oder eines Arrays, gegebenen Route.

```
1       /**
         * Public static function computing the length of a given route given
3        *  as an array of points using the Point#distance methode
         *
5        * @param points The route given as an array of points
         * @return The route's length
7        */
        public static double length(Point[] points) {
9           double length = 0;

11          // Compute the route's length by adding up the length
            // from one point to the next.
13          for (int i = 0; i < points.length - 1; i++) {
                length += points[i].distance(points[i + 1]);
15          }

17          return length;
        }
19
        /**
21       * Public static function computing the length of a given route given a
         * list of points using the Point#distance methode
23       *
         * @param list The route given as a list of points
25       * @return The route's length
         */
27      public static double getLength(List<Point> list) {
            double length = 0D;
29
            for (int i = 0; i < list.size() - 1; i++) {
31              length += list.get(i).distance(list.get(i + 1));
            }
33
            return length;
35      }
```

Funktionen zum pruefen der Abbiegewinkelbedingung:
Die Folgenden Funktionen sind ebenfalls Teil der Klasse Utils und dienen dazu, die Abbiegewinkelbedingung (fuer 90°) fuer drei Punkte oder fuer eine Route (wieder in Form einer Liste oder eines Arrays) zu pruefen.

```
    /**
     * Public static function checking weather three points met the turning angle constraint.
     * E.i. the angle between them is at least 90 degrees.
     *
     * @param P The first point
     * @param Q The second point
     * @param R The last point
     * @return If the angle between QP and QR is at least 90 degrees
     */
    public static boolean turningAngleIsValid(Point P, Point Q, Point R) {
        Vector vec1 = Q.vectorToPoint(P), vec2 = Q.vectorToPoint(R);
        return vec1.dotProduct(vec2) <= 0;
    }

    /**
     * Public static function to check if a given route given as an array
     * of points mets the turning angle constraint.
     *
     * @param points The route given as an array of points
     * @return If the turning angle constraint is met by the given route
     */
    public static boolean turningAnglesAreValid(Point[] points) {
        // If the route only has 0, 1 or 2 points, the angle constraint is always met
        // since there are no angles at all
        if (points.length < 3) return true;

        for (int i = 0; i < points.length - 2; i++) {
            if (!turningAngleIsValid(points[i], points[i + 1], points[i + 2]))
                return false;
        }
        return true;
    }

    /**
     * Public static function to check if a given route given as a list
     * of points mets the turning angle constraint.
     *
     * @param points The route given as a list of points
     * @return If the turning angle constraint is met by the given route
     */
    public static boolean turningAnglesAreValid(List<Point> points) {
        for (int i = 0; i < points.size() - 2; i++) {
            Point P = points.get(i);
            Point Q = points.get(i + 1);
            Point R = points.get(i + 2);
            if (!Utils.turningAngleIsValid(P, Q, R)) return false;
        }

        return true;
    }
```

9.2. Exakte Loesungsverfahren

Implementierung des Algorithmus 3 (alle moeglichen Routen)

```
    /**
     * Implementation of the solve-methode solving the problem
     * for the given points by iterating through all possible permutations
     * of those points
     *
     * @return The best possible route (might be null)
     */
    @Override
    public Point[] solve() {
        // Initialize the best points (route), and it's length with null and -1.
        // Those variables will keep track of the best route found so far.
        Point[] bestPoints = null;
        double bestLength = -1;

        // Initialize the first permutation using given points from this#getPoints,
        // as well as the permutation's size N and the array c needed for Heap's Algorithm
        Point[] points = this.getPoints();
        int N = this.getSize();
        int[] c = new int[N];

        // For the first route (permutation), check if the turning angle constraint is met
        if (Utils.turningAnglesAreValid(points)) {
            bestPoints = points.clone();
            bestLength = Utils.length(bestPoints);
        }

        // Use the Heap-Algorithm to iterate through all
        // permutations of the points
        int n = 1;
        while (n < N) {
            if (c[n] < n) {
                Utils.swap(points, n % 2 == 0 ? 0 : c[n], n);
                c[n]++;
                n = 1;

                // For the current permutation of the points (route),
                // check if the turning angle constraint is met
                if (Utils.turningAnglesAreValid(points)) {
                    double length = Utils.length(points);

                    // If the turning angle constraint is met, check if the
                    // current route is shorter than the best route and
                    // update the best Route if so
                    if (bestLength < length) {
                        bestPoints = points.clone();
                        bestLength = length;
                    }
                }
            } else {
                c[n] = 0;
                n += 1;
            }
        }

        return bestPoints;
    }
```

Implementierung des Algorithmus 5 (haelfte aller moeglicher Routen):

```java
/**
 * Implementation of the solve-methode solving the problem
 * for the given points by iterating through half of all
 * possible permutations of those points
 * @return The best possible route (might be null)
 */
@Override
public Point[] solve() {
    // Initialize the best points (route), and it's length with null and -1.
    // Those variables will keep track of the best route found so far.
    Point[] bestPoints = null;
    double bestLength = -1;

    // Initialize the first permutation using given points from this#getPoints,
    // as well as the permutation's size N
    Point[] points = this.getPoints();
    int N = this.getSize();

    // Iterate through all pairs (a, b) \in \N such that 0 <= a < b <= N-1
    for (int a = 0; a < N; a++) {
        for (int b = a + 1; b < N; b++) {
            // For the current pair create a copy of the original points (route)
            // and set the a-th point at the start and the b-th point at the
            // end of the current permutation (route)
            Point[] currentPoints = points.clone();
            Utils.swap(currentPoints, 0, a);
            Utils.swap(currentPoints, N - 1, b);

            // For the first route (permutation), check if the turning angle constraint is met
            if (Utils.turningAnglesAreValid(currentPoints)) {
                double length = Utils.length(currentPoints);

                // If the turning angle constraint is met, check if the
                // route is shorter than the best route and
                // update the best Route if so
                if (bestLength < length) {
                    bestPoints = currentPoints.clone();
                    bestLength = length;
                }
            }
            // Use Heap's Algorithm to iterate through all possible permutation of the
            // current route (where the start and end points stay as they are)
            int n = 2;
            int[] c = new int[N];
            for (int k = 0; k < N; k++) c[k] = 1;

            while (n < N - 1) {
                if (c[n] < n) {
                    Utils.swap(currentPoints, n % 2 == 1 ? 1 : c[n], n);
                    c[n]++;
                    n = 2;
                    // For the current permutation of the points (route),
                    // check if the turning angle constraint is met
                    if (Utils.turningAnglesAreValid(currentPoints)) {
                        double length = Utils.length(currentPoints);
                        // If the turning angle constraint is met, check if the
                        // current route is shorter than the best route and
                        // update the best Route if so
                        if (bestLength < length) {
                            bestPoints = currentPoints.clone();
                            bestLength = length;
                        }
                    }
                } else {
                    c[n] = 1;
                    n += 1;
                }
            }
        }
    }
    return bestPoints;
}
```

Implementierung des Algorithmus 4 (alle moeglichen Routen):
Zunaechst werden zwei Felder (Klassenvariablen) initialisiert:

```
    /**
     * The best route found so far as a list of points
     */
    private List<Point> bestList = null;

    /**
     * The best route's length initialized with negative one
     */
    private double minLength = -1;
```

Weiter sieht die Implementierung wie folgt aus:

```
    /**
     * Implementation of the solve-methode solving the problem
     * for the given points by iterating through all the
     * possible permutations of those points recursively.
     *
     * @return The best possible route (might be null)
     */
    @Override
    public Point[] solve() {
        // Create a Set containing all points that haven't been used yet.
        // E.i. filling it with all given points. Using a CopyOnWriteArraySet such that
        // one can remove and add element to it while looping through its elements.
        // In fact, that is faster than copying the Set all the time.
        Set<Point> pointsLeft = new CopyOnWriteArraySet<>();
        for (int k = 0; k < this.getSize(); k++) pointsLeft.add(this.getPoint(k));

        // Initialize the current list (route) as an empty list.
        List<Point> currentList = new ArrayList<>();

        // Call the recursive checking of all possible routes.
        checkAllRecursively(pointsLeft, currentList);

        // Return null if the best list is null, or return the best list as an array.
        if (this.bestList == null) return null;
        return this.bestList.toArray(Point[]::new);
    }

    /**
     * Private methode that is used to recursively check all possible
     * permutations of all points.
     *
     * @param pointsLeft The points which are currently not in the route
     * @param currentList The current route as a list of points
     */
    private void checkAllRecursively(Set<Point> pointsLeft, List<Point> currentList) {
        // If there are no points left, check the current route.
        // E.i., check if it mets the angle constraint and compare it to the best
        // route so far.
        if (pointsLeft.isEmpty()) {
            check(currentList);
            return;
        }

        // Iterate through all unused points (e.i.,
        // points that are not in the route yet)
        for (Point point : pointsLeft) {
            // Set the current point at the end of the current route and remove it from the points left
            pointsLeft.remove(point);
            currentList.add(point);

            // Recursively check all routes starting with the current subroute
            checkAllRecursively(pointsLeft, currentList);

            // Remove the current point from the current route and add it to the points left again
            currentList.remove(point);
            pointsLeft.add(point);
        }
    }
```

```
59
    /**
61   * Private methode called to test a given permutation of the points.
     * Checking if the given route mets the turning angle constraint and updating
63   * the currently best list if the given list (route) is shorter than
     * the best one
65   *
     * @param list The current list of points (route)
67   */
    private void check(List<Point> list) {
69      if (!Utils.turningAnglesAreValid(list)) return;

71      double length = Utils.getLength(list);

73      if (this.minLength == -1 || length < this.minLength) {
            this.minLength = length;
75          this.bestList = Utils.copyList(list);
        }
77  }
```

Implementierung des Algorithmus 6 (alle moeglichen Routen mit Bound and Bound / Backtracking): Zunaechst werden wieder zwei Klassenvariablen initialisiert:

```
1   /**
     * The best route found so far as a list of points
3    */
    private List<Point> bestList = null;
5
    /**
7    * The best route's length initialized with negative one
     */
9   private double minLength = -1;
```

Der restliche Code, sieht aehnlich wie der vorherige, wie folgt aus:

```
1   /**
     * Implementation of the solve-methode solving the problem
3    * for the given points by iterating through all the
     * possible permutations of those points recursively
5    * skipping necessary branches.
     *
7    * @return The best possible route (might be null)
     */
9   @Override
    public Point[] solve() {
11      // Create a Set containing all points that haven't been used yet.
        // E.i. filling it with all given points. Using a CopyOnWriteArraySet such that
13      // one can remove and add element to it while looping through its elements.
        // In fact, that is faster than copying the Set all the time.
15      Set<Point> pointsLeft = new CopyOnWriteArraySet<>();
        for (int k = 0; k < this.getSize(); k++) pointsLeft.add(this.getPoint(k));
17
        // Initialize the current list (route) as an empty list.
19      List<Point> currentList = new ArrayList<>();

21      // Call the recursive checking of all possible routes.
        checkAllRecursively(pointsLeft, currentList, 0D);
23
        // Return null if the best list is null, or return the best list as an array.
25      if (this.bestList == null) return null;
        return this.bestList.toArray(Point[]::new);
27  }
```

```java
    /**
     * Private methode that is used to recursively check all possible
     * permutations of all points skipping necessary branches.
     *
     * @param pointsLeft The points which are currently not in the route
     * @param currentList The current route as a list of points
     * @param currentLength The current route's length
     */
    private void checkAllRecursively(Set<Point> pointsLeft, List<Point> currentList,
                            double currentLength) {
        // Skip the current branch if the current route's length is bigger
        // than the best route's length (if that is not null)
        if (this.minLength != -1 && this.minLength < currentLength) {
            return;
        }

        // If there are no points left, update the best list to the
        // current one by copying.
        // Note that the current route is smaller than the best one
        // because of the last if-statement
        if (pointsLeft.isEmpty()) {
            this.minLength = currentLength;
            this.bestList = Utils.copyList(currentList);
            return;
        }

        // Get the current route's size (#points)
        int size = currentList.size();

        // Iterate through all unused points (e.i.,
        // points that are not in the route yet)
        for (Point nextPoint : pointsLeft) {
            // If there is more than one point in the current list (route),
            // make sure if turning angle constraint is met by the last two
            // points in the route and nextPoint before recursively going on
            if (size >= 2) {
                Point P = currentList.get(size - 2);
                Point Q = currentList.get(size - 1);

                if (!Utils.turningAngleIsValid(P, Q, nextPoint)) continue;
            }

            // Compute the distance from the last point of the current list to nextPoint.
            // The distance will be 0 if the current list is empty.
            double dist = currentList.isEmpty() ? 0D : currentList.get(size - 1).distance(nextPoint);

            // Set the current point (nextPoint) at the end of the current route
            // and remove it from the points left
            pointsLeft.remove(nextPoint);
            currentList.add(nextPoint);

            // Recursively check all routes starting with the current subroute
            checkAllRecursively(pointsLeft, currentList, currentLength + dist);

            // Remove the current point from the current route and add it to the points left again
            currentList.remove(nextPoint);
            pointsLeft.add(nextPoint);
        }
    }
```

9.3. Heuristische Loesungsverfahren

Naechsten Punkt finden:
Um den (distanzmaeszig) naechsten Punkt zu einem bestimmten Punkt in einer Menge zu finden, wird folgender Code verwendet werden:

```
1     /**
       * Private method finding a point P that is the closest to a given point Q
3      * of the points in a given set.
       * @param points The set of points
5      * @param Q The point Q
       * @return The point of the given set that is the closest to Q
7      */
      private Point getNext(Set<Point> points, Point Q) {
9         // Initialize the closest point (to Q) and its length with null and -1
          Point bestPoint = null;
11        double bestDistance = -1;

13        // Iterate through all possible points
          for (Point point : points) {
15            // For the current point P get the distance from
              // P to the given point Q
17            double curDistance = Q.distance(point);

19            // If the current distance is smaller than the best distance
              // or the best distance is still -1, update the best point and
21            // the best distance
              if (bestDistance == -1 || curDistance < bestDistance) {
23                bestPoint = point;
                  bestDistance = curDistance;
25            }
          }
27        return bestPoint;
      }
```

Um weiter den naechten Punkt zu einem bestimmten Punkt in einer Menge zu finden, wobei ebenfalls die Abbiegewinkel (zu einem weiteren Punkt) eingehalten wird, wird folgender Code verwendet werden:

```
      /**
2      * Private methode finding the next point R from a given set of possible
       * points that is the closest to a given point Q and meets the angle
4      * constraint for P, Q and R, where P is another point.
       * @param points The set of possible points R
6      * @param P The point P
       * @param Q The point Q
8      * @return The point of the given set of points that is the closest to
       * the point Q and meets the angle constraint for P, Q and R (or null,
10     * if there's none)
       */
12    private Point getNext(Set<Point> points, Point P, Point Q) {
          // Initialize the closest point (to Q) and its length with null and -1
14        Point bestPoint = null;
          double bestDistance = -1;
16        // Iterate through all possible points
          for (Point point : points) {
18            // For the current point P, get its distance to Q
              // and if the angle constraint is met for P, Q and R
20            double curDistance = Q.distance(point);
              boolean valid = Utils.turningAngleIsValid(P, Q, point);
22            // Update the best point and its length if the angle constraint is met (valid)
              // and the best point still null or the current distance from P to Q is smaller
24            // than from the bestPoint to Q.
              if (valid && (bestDistance == -1 || curDistance < bestDistance)) {
26                bestPoint = point;
                  bestDistance = curDistance;
28            }
          }
30        // Return the best point found (might be null)
          return bestPoint;
32    }
```

Implementierung des Algorithmus 9 (ein Startpunkt):

```java
/**
 * Implementation of the solve-methode solving the problem
 * for the given points by starting at the first point given and then
 * always looking for the nearest point meeting the angle
 * constraint.
 *
 * @return The best route found (might be null if none could be found)
 */
@Override
public Point[] solve() {
    // If the amount of points given is smaller than 3,
    // the result is only the given points (since there won't
    // be any angles)
    if (this.getSize() <= 2) {
        return this.getPoints().clone();
    }

    // Initialize the route (points) as an array of points
    // and a Set containing all given points (will be the
    // set containing all points not used yet)
    Point[] points = new Point[this.getSize()];
    Set<Point> pointsLeft = Utils.toSet(this.getPoints());

    // Choose the first point of the route, add it to
    // the route and remove it from the points left
    Point firstPoint = this.getPoint(0);
    points[0] = firstPoint;
    pointsLeft.remove(firstPoint);

    // Get the second point to be the point that is the
    // closest to the first point
    Point secondPoint = getNext(pointsLeft, firstPoint);
    points[1] = secondPoint;
    pointsLeft.remove(secondPoint);

    // Use lastPoint and currentPoint to keep track of
    // the last two points in the current route.
    Point lastPoint = firstPoint;
    Point currentPoint = secondPoint;

    // Now, get the next elements for the route
    for (int k = 2; k < this.getSize(); k++) {
        // Get the next point using this#getNext
        Point next = getNext(pointsLeft, lastPoint, currentPoint);

        // If next is null (e.i. there is no point left that meets
        // the angle constraint), return null (e.i. no route was found)
        if (next == null) return null;

        // Otherwise, add the found point to the route and remove it from
        // the points left. Also, update the last and current point.
        points[k] = next;
        pointsLeft.remove(next);
        lastPoint = currentPoint;
        currentPoint = next;
    }

    // Finally, return the route
    return points;
}
```

Implementierung des Algorithmus 10 (jeden Startpunkt einmal):

```
/**
 * Implementation of the solve-methode solving the problem
 * for the given points by testing each possible point to be the route's
 * starting point. And for each one, add the left points greedily to the
 * route by always using the nearest point meeting the angle constraint.
 *
 * @return The best route found (might be null if none could be found)
 */
@Override
public Point[] solve() {
    // If the amount of points given is smaller than 3,
    // the result is only the given points (since there won't
    // be any angles)
    if (this.getSize() <= 2) {
        return this.getPoints().clone();
    }

    // Initializing the best route and its length found so far
    // with null and -1.
    Point[] bestRoute = null;
    double bestLength = -1;

    // Choose each possible point to be the starting point of
    // the route using its index. Hence, iterating through all indexes.
    for (int i = 0; i < this.getSize(); i++) {
        // Greedily get the current route with the starting point
        // given by the current index i.
        Point[] currentRoute = solve(i);

        // If currentRoute is null (e.i., no route was found),
        // continue with the next index.
        if (currentRoute == null) continue;

        // Get the current route's length and compare it to
        // the best one. If the current length is smaller,
        // update the best route.
        double curLength = Utils.length(currentRoute);

        if (bestLength == -1 || curLength < bestLength) {
            bestRoute = currentRoute;
            bestLength = curLength;
        }
    }

    // Finally, return the best route
    return bestRoute;
}

/**
 * Private methode for greedily finding a route given the index
 * of a starting point
 *
 * @param i The index of the starting point
 * @return The route or null if none was found
 */
private Point[] solve(int i) {
    // Initialize the route (points) as an array of points
    // and a Set containing all given points (will be the
    // set containing all points not used yet)
    Point[] points = new Point[this.getSize()];
    Set<Point> pointsLeft = Utils.toSet(this.getPoints());

    // Choose the first point of the route, add it to
    // the route and remove it from the points left
    Point firstPoint = this.getPoint(i);
    points[0] = firstPoint;
    pointsLeft.remove(firstPoint);

    // Get the second point to be the point that is the
    // closest to the first point
    Point secondPoint = getNext(pointsLeft, firstPoint);
    points[1] = secondPoint;
    pointsLeft.remove(secondPoint);

```

```
        // Use lastPoint and currentPoint to keep track of
        // the last two points in the current route.
        Point lastPoint = firstPoint;
        Point currentPoint = secondPoint;

        // Now, get the next elements for the route
        for (int k = 2; k < this.getSize(); k++) {
            // Get the next point using this#getNext
            Point next = getNext(pointsLeft, lastPoint, currentPoint);

            // If next is null (e.i. there is no point left that meets
            // the angle constraint), return null (e.i. no route was found)
            if (next == null) return null;

            // Otherwise, add the found point to the route and remove it from
            // the points left. Also, update the last and current point.
            points[k] = next;
            pointsLeft.remove(next);
            lastPoint = currentPoint;
            currentPoint = next;
        }

        // Finally, return the route
        return points;
    }
```

Implementierung des Algorithmus 11 (alle Paare an Startpunkten):

```
    /**
     * Implementation of the solve-methode solving the problem
     * for the given points by testing each possible pair of starting points
     * (first and second point in the route). For each pair it'll
     * look for the next points for the route by always using the
     * nearest point meeting the angle constraint.
     * @return The best route found (might be null if none could be found)
     */
    @Override
    public Point[] solve() {
        // If the amount of points given is smaller than 3,
        // the result is only the given points (since there won't
        // be any angles)
        if (this.getSize() <= 2) {
            return this.getPoints().clone();
        }

        // Initializing the best route and its length found so far
        // with null and -1.
        Point[] bestRoute = null;
        double bestLength = -1;
        // Iterate through all possible pairs of indexes (i!=j)
        // using a nested for-loop
        for (int i = 0; i < this.getSize(); i++) {
            for (int j = 0; j < this.getSize(); j++) {
                if (j == i) continue;
                // Greedily get the current route with the starting point
                // given by the current indexes i and j
                Point[] current = solve(i, j);
                // If currentRoute is null (e.i., no route was found),
                // continue with the next index.
                if (current == null) continue;
                // Get the current route's length and compare it to
                // the best one. If the current length is smaller,
                // update the best route.
                double curLength = Utils.length(current);

                if (bestLength == -1 || curLength < bestLength) {
                    bestRoute = current;
                    bestLength = curLength;
                }
            }
        }
        // Finally, return the best route
        return bestRoute;
    }
```

```
47
48      /**
49       * Private methode for greedily finding a route with two
50       * given starting points, given by their index.
51       * @param i The first point's index
52       * @param j The second point's index
53       * @return The route, or null if none was found
54       */
55      private Point[] solve(int i, int j) {
56          // Initialize the route (points) as an array of points
57          // and a Set containing all given points (will be the
58          // set containing all points not used yet)
59          Point[] points = new Point[this.getSize()];
60          Set<Point> pointsLeft = Utils.toSet(this.getPoints());
61
62          // Get the first and second points of the route
63          // given be the indexes, add them to the route
64          // and remove them from the points left
65          Point firstPoint = this.getPoint(i);
66          Point secondPoint = this.getPoint(j);
67
68          points[0] = firstPoint;
69          points[1] = secondPoint;
70
71          pointsLeft.remove(firstPoint);
72          pointsLeft.remove(secondPoint);
73
74          // Use lastPoint and currentPoint to keep track of
75          // the last two points in the current route.
76          Point lastPoint = firstPoint;
77          Point currentPoint = secondPoint;
78
79          for (int k = 2; k < this.getSize(); k++) {
80              // Get the next point using this#getNext
81              Point next = getNext(pointsLeft, lastPoint, currentPoint);
82
83              // If next is null (e.i. there is no point left that meets
84              // the angle constraint), return null (e.i. no route was found)
85              if (next == null) return null;
86
87              // Otherwise, add the found point to the route and remove it from
88              // the points left. Also, update the last and current point.
89              points[k] = next;
90              pointsLeft.remove(next);
91              lastPoint = currentPoint;
92              currentPoint = next;
93          }
94
95          // Finally, return the route
96          return points;
97      }
```

Implementierung des Algorithmus 12 (Alle Startrouten der Laenge k):

```
/**
 * Implementation of the solve-methode solving the problem
 * for the given points by testing each possible subroute of a given
 * size k to be the start of the route. For each startroute it'll
 * first  check if the turning angle constraint is met and then
 * look for the next points for the route by always using the
 * nearest point meeting the angle constraint.
 * @return The best route found (might be null if none could be found)
 */
@Override
public Point[] solve() {
    // Create a Set containing all points that haven't been used yet.
    // E.i. filling it with all given points. Using a CopyOnWriteArraySet such that
    // one can remove and add element to it while looping through its elements.
    // In fact, that is faster than copying the Set all the time.
    Set<Point> pointsLeft = new CopyOnWriteArraySet<>();
    for (int k = 0; k < this.getSize(); k++) pointsLeft.add(this.getPoint(k));

    // Initialize the current start subroute as an array of points
    // This array will be used for the creation of all starting routes.
    Point[] route = new Point[this.startSize];

    // Start iterating through all possible starting subroutes recursively
    this.checkAllRecursively(route, 0, pointsLeft);

    // Finally, return the best route found so far (might be null)
    return this.bestList;
}

/**
 * Private methode for to check each possible start subroute of the
 * given size recursively.
 * @param start The current route
 * @param index The current index the next point will have in the route
 * @param pointsLeft Set containing all points not used yet, e.i. all points not in the route yet
 */
public void checkAllRecursively(Point[] start, int index, Set<Point> pointsLeft) {
    // If the start subroute has the needed length, check it
    if (index == this.startSize) {
        // First, check weather the current starting route is valid,
        // e.i. the angle constraint is met. If this is not the case, return
        if (!Utils.turningAnglesAreValid(start)) {
            return;
        }
        // Greedily create the rest of the route
        Point[] route = getRoute(start, pointsLeft);

        // Check if route is null (e.i. the route was not found)
        if (route == null) return;

        // Get the current route's length and compare it to the best route
        // If the current one is smaller than the best one, update the best one
        double length = Utils.length(route);

        if (this.minLength == -1 || length < this.minLength) {
            this.minLength = length;
            this.bestList = route.clone();
        }
        return;
    }

    // Otherwise, iterate through all points not in the subroute yet
    for (Point p : pointsLeft) {
        // Add the current point p to the subroute and remove it from the points left
        start[index] = p;
        pointsLeft.remove(p);

        checkAllRecursively(start, index + 1, pointsLeft);

        // Add the current point to the points left again.
        // No need to remove it from the subroute, it'll just be replaces
        pointsLeft.add(p);
    }
}
```

```
    /**
77   * Methode for creating the whole route given a starting subroute
     * meeting the angle constraint.
79   *
     * @param startRoute the starting subroute
81   * @param set The points not used yet
     * @return The complete route created greedily
83   */
    private Point[] getRoute(Point[] startRoute, Set<Point> set) {
85      // First create the route as an array of points and copy
        // the given start subroute to the start of that array
87      Point[] route = new Point[this.getSize()];
        for (int i = 0; i < this.startSize; i++) route[i] = startRoute[i];
89
        // Create a copy of the points left
91      Set<Point> pointsLeft = new HashSet<>(set);

93      // Use lastPoint and currentPoint to keep track of
        // the last two points in the current route.
95      Point lastPoint = route[this.startSize - 2];
        Point currentPoint = route[this.startSize - 1];
97
        // Add the rest of the route greedily
99      for (int k = this.startSize; k < this.getSize(); k++) {
            // Get the next point using this#getNext
101         Point next = getNext(pointsLeft, lastPoint, currentPoint);

103         // If next is null (e.i. there is no point left that meets
            // the angle constraint), return null (e.i. no route was found)
105         if (next == null) return null;

107         // Otherwise, add the found point to the route and remove it from
            // the points left. Also, update the last and current point.
109         route[k] = next;
            pointsLeft.remove(next);
111         lastPoint = currentPoint;
            currentPoint = next;
113     }

115     // Finally, update the route found
        return route;
117 }
```

10. Literatur

[1] Heap, B. R. (Nov. 1963), «Permutations by Interchanges», The Computer Journal, Vol. 6, Issue 3, S. 293-298

[2] Fekete, Sándor P. (Sep. 1997), «Angle-Restricted Tours in the plane», Computational Geometry, Vol. 8, Issue 4