FACHARBEIT
Schuljahr 2023/2024

Betrachtungen zum winkelbeschränkten Hamiltonpfad

vorgelegt dem XXX
zum 04.2024

geschrieben im Leistungskurs Informatik
unter der Betreuung von XXX

von

Schmidt, Annika

1 Kurzfassung

Diese Arbeit befasst sich mit der Kombination von zwei Problemen der Graphentheorie, genauer mit der Suche nach einem kostenminimalen Hamiltonpfad unter Einhaltung einer Winkelbeschränkung. Erfolgt die Bewertung eines Pfades ausschließlich anhand der Summe aller Kantengewichte, so ist diese Fragestellung bereits gut untersucht [1], [2], [3].

Die im Folgenden betrachtete Erweiterung des winkelbeschränkten Hamiltonpfades besteht daher darin, dass die Kostenfunktion zusätzlich zum Kantengewicht auch den Winkel zwischen zwei Kanten des Pfades berücksichtigt. Für eine Winkelmenge $W \subseteq \{\theta \mid -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$, beschreibt das winkelbeschränkte Hamiltonpfadproblem die Entscheidung, ob eine Menge P von n Knoten in der euklidischen Ebene durch einen Hamiltonpfad, bestehend aus aneinandergereihte Geraden so verbunden werden kann, dass alle Winkel zwischen aufeinanderfolgenden Geraden aus der Menge W stammen.

Die Idee für diese Forschungsarbeit entspringt meiner Teilnahme am 41. Bundeswettbewerb Informatik [4], bei der ich mich intensiv mit der beschriebenen Thematik auseinandersetzte. Die Problemstellung wurde dort anhand eines Anwendungsbeispiels in der Luftfahrt erläutert und sollte eigenständig formalisiert und bearbeitet werden. Nichtsdestoweniger sind die in der folgenden Arbeit gewonnen Erkenntnisse unabhängig von der Wettbewerbsaufgabe mit dem Titel „Weniger krumme Touren“ zu verstehen.

Die Kombination vom minimalen, winkelbeschränkten Hamiltonpfad stellt eine faszinierende Herausforderung dar, die nicht nur theoretisches Interesse weckt, sondern auch in der realen Welt, insbesondere in der Robotik, von großer Bedeutung ist. Die Optimierung von Strecken für Roboter, um möglichst effizient verschiedene Punkte anzufahren, ist ein zentrales Anliegen in der Entwicklung autonomer Systeme. Eine Routenplanung mit einem möglichst geradlinigen Verlauf ermöglicht die Nutzung von lenkungsbeschränkten Robotern [5]. Durch geschickte Pfadplanung kann außerdem die Effizienz von Hochgeschwindigkeitsfahrzeugen wie bestimmten Autos, Zügen oder Flugzeugen gesteigert werden, da langwieriger Brems- und Beschleunigungsphasen in den Kurven vermieden werden können [6]. Neben der Leistung verbessert sich bei dieser schonenden Fahrweise mit seltenerem und geringerem Auftreten seitlicher Kräfte auch die Sicherheit und Lebensdauer der Roboter und Fahrzeuge erheblich.

In dieser Arbeit werden verschiedene Heuristiken zur Lösungsannäherung präsentiert. Des Weiteren erfolgt eine komplexitätstheoretische Einordnung des Problems. Basierend auf den Ergebnissen zu ausgewählten Beispielen ist eine Mischung aus einer Nearest Neighbour Heuristik und einer Nearest Insertion Heuristik das geeignetste der untersuchten Lösungsverfahren.

Inhaltsverzeichnis

2 Lösungsidee

Sei $G = (V, E, c)$ ein Graph. Dabei ist V die Menge der Knoten und $E \subseteq \{\{v, w\} : v, w \in V\}$ die Menge der Kanten. $c : E \to \mathbb{R}$ sei eine Gewichtungsfunktion. Des weiteren gelte $c(\{v, w\}) = c(\{w, v\})$, wir befinden uns also in einem ungerichteten Graphen. Unsere Punkte liegen in einem metrischen System, genauer der euklidischen Ebene. Es gelte die Dreiecksungleichung $c(\{u, w\}) <= c(\{u, v\}) + c(\{v, w\})$. Sei $W \subseteq \{\theta \mid -\frac{\pi}{2} \leq \theta \leq \frac{\pi}{2}\}$ die Menge der erlaubten Winkel zwischen aufeinanderfolgenden Liniensegmenten. Sei $\phi(\{v_i, v_{i+1}\})$ der Winkel zwischen den Liniensegmenten, die die Punkte v_i und v_{i+1} verbinden.

Gesucht ist eine Permutation ohne Wiederholung $P = (v_1, v_2, \ldots, v_n)$ der Koordinaten für die gilt:

$$P = \operatorname{argmin} \left\{ \sum_{i=1}^{n-1} c(\{v_i, v_{i+1}\}) : \phi(\{v_i, v_{i+1}\}) \in W \right\}$$

Damit ähnelt unser Problem auf den ersten Blick dem bekannten Problem des Handlungsreisenden (engl.: travelling salesmen problem, TSP) [2]. Der Unterschied besteht lediglich darin, dass wir einen Hamiltonpfad suchen, während das TSP wieder zum Ausgangspunkt zurückkehrt und somit einen Hamiltonkreis sucht (engl.: path travelling salesmen problem, P-TSP) [7]. Außerdem definieren wir als Kantengewicht den euklidischen Abstand zwischen zwei Punkten (engl.: euclidean travelling salesmen problem, E-TSP) [8]. Hinzu kommt bei uns außerdem die Winkelbeschränkung, die ebenfalls nicht Teil des originalen TSP ist, in manchen Variationen des solchen jedoch durchaus behandelt wird [9], [10]. Der Umgang und die Gewichtung der Winkelkomponente unterscheiden sich bisweilen allerdings enorm. So fokussieren sich S. Asaeedi und M. S. Shamaee [9] in ihrer Arbeit auf die Minimierung zusätzlich eingefügter Knoten (s.g. „stop points“ [9, S. 1]), während S. P. Fekete und G. J. Woeginger [10] das verwandte Entscheidungsproblem, ob eine winkelbeschränkte Route in einem gegebenen Graphen überhaupt existiert, zum Gegenstand ihrer Forschung gemacht haben. Die oben beschriebene Suche eines winkelbeschränkten Hamiltonpfads mit Minimierung der Routenlänge (engl.: angle restricted euclidean path travelling salesmen problem, AREP-TSP) gliedert sich somit in eine Reihe verwandter und kombinierbarer, nicht jedoch gleicher Probleme ein.

Grundsätzlich ist zu beachten, dass in unserer Aufgabenstellung der Startpunkt, ebenso wie der Endpunkt frei gewählt werden dürfen.

Das gestellte Problem ist NP-schwer.[1] Wir werden in dieser Arbeit daher auf Heuristiken setzen, die sich dem Optimum möglichst annähert, aber in einer polynomiellen Laufzeit zu einem Ergebnis kommen.

Das Problem lässt sich in zwei Teilproblemen skizzieren:

1. einen Hamiltonpfad mit Winkelbeschränkung finden
2. einen möglichst kurzen Hamiltonpfad finden

Es gibt mehrere Gründe dafür, diese beiden Punkten in genannter Reihenfolge abzuhandeln:

- Den kürzesten Hamiltonpfad durch einen vollständigen Graphen gibt es immer, einen Hamiltonpfad mit Winkelbeschränkung nicht. Ein Beispiel dafür wären die Eckpunkte eines gleichseitigen Dreiecks. Falls es einen solchen Pfad nicht gibt/der verwendete Algorithmus zum Schluss kommt, dass es keinen gibt, soll nicht unnötig Rechenzeit auf eine vorherige Streckenoptimierung verschwendet worden sein.
- Einen möglichst kurzen Hamiltonpfad zu finden stellt ein Optimierungsproblem da. Es bräuchte also eine Abbruchbedingung, z.B. in Form einer bestimmten Güte der Lösung, während die Suche nach einem winkelbschränkten Hamiltonpfad zumindest in einer Richtung eine definierte Grenze besitzt, nämlich sobald ein solcher Pfad gefunden ist.

Womöglich gäbe es auch Algorithmen, die den Anspruch stellen, beide Probleme parallel lösen zu können, um ein Ergebnis möglichst nah am globalen Optimum zu finden. Wir werden uns ihrer in dieser Arbeit allerdings nacheinander annehmen. Der Fokus liegt dabei auf dem Finden des winkelbeschränkten Hamiltonpfades, anschließend werden wir einen nach-optimierenden Algorithmus anwenden, um die Pfadlänge zu verkürzen.

[1] Beweisführung in Kapitel 2.7

2.1 Lösung 1: Nearest Neighbour Heuristik

Die Nearest Neighbour Heuristik ist einer der einfachsten Ansätze einen winkelbeschränkten Hamiltonpfad zu ermitteln. Diese Heuristik basiert darauf, vom aktuellen Knoten aus immer zum nächstgelegenen zu gehen. Es werden also in einem Greedy-Verfahren immer die kürzeste Kante ausgewählt. Um die Winkelbeschränkung einzuhalten, werden außerdem nur solche Kanten betrachtet, die zur vorherigen Kante in einem flachen Winkel stehen.

Eine wichtige Erkenntnis ist, dass dieser Winkel (im Gegensatz zu einer Distanz) keine feste Eigenschaft einer Kante oder eines Knotens ist, sondern durch den vorher gewählten Weg beeinflusst wird. Während der nächste Nachbar ungeachtet der Winkel für einen beliebigen Knoten zur jedem Zeitpunkt immer der selbe Knoten darstellt, kann der nächste Nachbar unter Berücksichtigung des Winkels bei einem beliebigen Punkt zu verschiedenen Zeitpunkten variieren. Darum sollte in unseren Algorithmen jeder Punkt einmal als Startpunkt angenommen werden.

1. Wähle einen Knoten aus dem Graphen als Startknoten aus und füge ihn zum Pfad hinzu.
2. Wiederhole, bis alle Knoten im Pfad enthalten sind:
   a) Finde unter den Knoten, die noch nicht im Pfad enthalten sind, den Knoten, der dem letzten Pfadglied am nächsten liegt und der die Winkelbeschränkung erfüllt. Füge ihn zum Pfad hinzu.

2.2 Lösung 2: Nearest Insertion Heuristik und Farthest Insertion Heuristik

Die Idee der Nearest Insertion Heuristik bzw. der Farthes Insertion Heuristik ist es, inkrementell einen Pfad durch Hinzugabe von Knoten bis hin zu einem Hamiltonpfad zu erweitern.

Konkret bedeutet das, dass ein Knoten nach bestimmten Gesichtspunkten ausgewählt wird (besonders hohe oder niedrige Entfernung, gemessen entweder zum gesamten bisherigen Pfad oder aber zu einem einzigen Punkt in diesem Pfad).

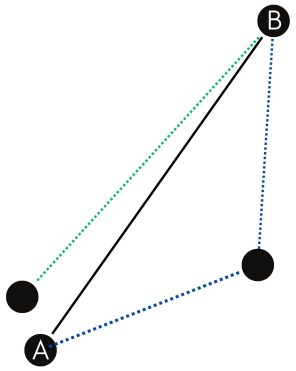

Abbildung 1: Beispielhafte Darstellung für den Unterschied zwischen gesamtem Maximum (blau) zu lokalem Maximum (grün) in der Farthes Insertion Heuristik.

Anschließend wird dieser Punkt nach bestimmten Gesichtspunkten in den bisherigen Graphen eingefügt (besonders geringe Zunahme der Streckenlänge und unter Einhaltung der Winkelbeschränkung). Der entscheidende Unterschied dieser Heuristik zu der zuvor genannten Nearest Neighbour Heuristik liegt somit darin, dass nicht an den Pfad angehängt, sondern in den Pfad eingefügt (engl.: inserted) wird.

1. Wähle einen Knoten aus dem Graphen als Startknoten aus und füge ihn zum Pfad hinzu.
2. Wiederhole, bis alle Knoten im Pfad enthalten sind:
   a) Suche unter den Knoten außerhalb des Pfades den Knoten mit der geringsten/größten Entfernung zu allen/dem nächstgelegenen Knoten im Pfad.
   b) Finde das Knotenpaar (bzw. den Start- oder Endknoten) im Pfad, bei dem beim Einfügen des Knotens der Pfad minimal länger wird und die Winkelbeschränkung erfüllt ist. Füge den neuen Knoten zwischen diesen beiden Knoten (bzw. am Start oder am Ende) in den Pfad ein.

2.3 Lösung 3: Random Insertion Heuristik

Die Random Insertion Heuristik ähnelt in ihrer Vorgehensweise den zuvor genannten Insertion Heuristiken. Hier findet die Auswahl des Knotens allerdings rein zufällig statt, um ihn anschließend nach wie vor möglichst günstig und unter Einhaltung der Winkelbeschränkung einzufügen.

1. Wähle einen Knoten aus dem Graphen als Startknoten aus und füge ihn zum Pfad hinzu.
2. Wiederhole, bis alle Knoten im Pfad enthalten sind:
   a) Wähle unter den Knoten außerhalb des Pfades einen zufälligen Knoten.
   b) Finde das Knotenpaar (bzw. den Start- oder Endknoten) im Pfad, bei dem beim Einfügen des Knotens der Pfad minimal länger wird und die Winkelbeschränkung erfüllt ist. Füge den neuen Knoten zwischen diesen beiden Knoten (bzw. am Start oder am Ende) in den Pfad ein.

2.4 Lösung 4: Nearest Neighbour mit Insertion

Hierbei handelt es sich um eine Mischung des 2.1er und des 2.2er Algorithmus'. Der Ansatz unterscheidet sich insofern von der Nearest Insertion Heuristik, als das die Auswahl eines Punktes in Schritt a) direkt unter Berücksichtigung seines späteren Platzes im Pfad erfolgt. In der Praxis bedeutet das, dass man die Punkte, die noch nicht Teil des Pfades sind an jeder beliebigen Stelle im Pfad einsetzt, prüft ob die Winkelbedingung erfüllt ist und falls dem so ist, angibt, wie sehr sich die Pfadlänge durch Einfügen des Punktes an dieser Stelle verlängert.

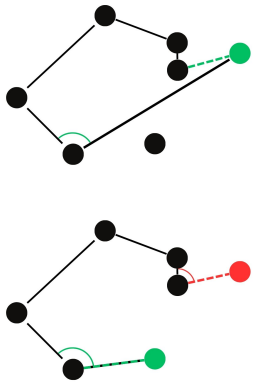

Abbildung 2: Beispielhafte Darstellung für den Unterschied von Nearest Insertion mit lokalem Minimum (oben) und Nearest Neighbour mit Insertion (unten). Im oberen Fall wird zuerst der näheste Punkt gesucht, dann davon unabhängig eine Möglichkeit ihn in den Graphen einzubauen, im unteren Fall wird der näheste Punkt unter Berücksichtigung der Winkelbedingung gesucht.

Anschließend nimmt man den Punkt, der an einer beliebigen Stelle mit geringster Streckenzunahme und eingehaltener Winkelbeschränkung eingesetzt werden kann und fügt ihn an dieser Stelle in den Pfad ein.

1. Wähle einen Knoten aus dem Graphen als Startknoten aus und füge ihn zum Pfad hinzu.
2. Wiederhole, bis alle Knoten im Pfad enthalten sind:
   a) Gehe alle Kanten im Pfad durch (auch Start und Ende):
      i. Gehe alle Knoten, die nicht bereits im Pfad sind durch:
         A. Falls beim Einfügen dieses Knoten in diese Kante die Winkelbedingung erfüllt ist, merke dir den Knoten, die Kante und die Zunahme der Wegstrecke.
   b) Nimm den Knoten mit der geringsten Zunahme der Wegstrecke und füge ihn an die gemerkte Kante ein.

2.5 Lösung 5: Backtracking-Algorithmus

Keiner der bisher entwickelten Algorithmen wird auf beliebigen Probleminstanzen ein optimales Ergebnis liefern, denn es ist möglich, dass sie keinen winkelbeschränkten Hamiltonpfad findet, obwohl ein solcher existiert.

Natürlich gäbe es auch Möglichkeiten, die Optimallösung zum gegebenen Teilproblem (der Suche nach dem winkelbeschränkten Hamiltonpfad) zu ermitteln. Mit einem Backtracking-Algorithmus kann der winkelbeschränkte Hamiltonpfad entweder gefunden oder aber definitiv ausgeschlossen werden, dass dieser existiert. Die Idee basiert darauf, so lange einen Pfad aufzubauen, bis ein Knoten hinzugefügt wird, von dem aus die Teillösung nicht zu einer gültigen Gesamtlösung ausgebaut werden kann. Dann wird der letzte Schritt zurückgenommen und stattdessen ein alternativer Weg ausprobiert.

Allerdings besitzt der Backtracking-Algorithmus im schlechtesten Fall eine exponentielle Laufzeit, die wir durch unsere Heuristiken vermeiden wollten. Somit ist er für größere Probleminstanzen ungeeignet und soll an dieser Stelle nicht näher betrachtet werden. Er kann höchstens dann sinnvoll zum Einsatz kommen, wenn die Heuristiken keinen winkelbeschränkten Hamiltonpfad finden und Unsicherheit besteht, ob es sich um eine Schwäche der Heuristik oder tatsächlich um eine Porbleminstanz handelt, auf der kein winkelbeschränkter Hamiltonpfad gefunden werden kann.

2.6 Nach-Optimierung durch 2-opt-Algorithmus

Dieser Algorithmus dient dazu, den Pfad, der durch eine der Lösungen 2.1 bis 2.4 gefunden wurde, zu verkürzen. Dazu werden zwei Kanten aus dem Graphen entfernt und gekreuzt wieder eingesetzt. Sofern der entstandene Pfad kürzer und die Winkelbeschränkung erfüllt ist, wird der gefundene Graph als neue Lösung festgelegt.

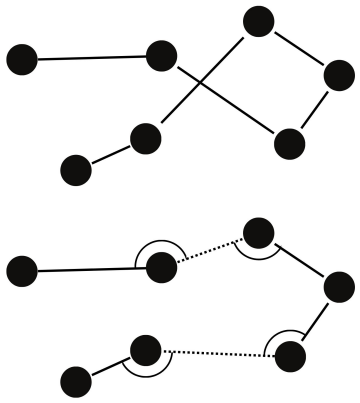

Abbildung 3: In der Anwendung des 2-opt-Algroithmus' werden Kanten dann vertauscht, wenn sich die Route dadurch verkürzt und alle betroffenen Winkel über 90° liegen.

1. Wiederhole, bis keine Verbesserung mehr stattfindet:
   a) Wähle zwei Kanten aus dem Pfad und verbinde sie über Kreuz zu einem neuen Pfad .
   b) Falls der neue Pfad kürzer als der alte Pfad ist und die Winkelbeschränkungen erfüllt sind:
      i. Nimm den neuen Pfad als gegebenen Pfad an.

2.7 Beweis der NP-Schwere

Das Problem, den kürzesten Hamiltonpfad mit Winkelbeschränkung in einem vollständigen Graphen zu finden, ist NP-schwer. Wir beweisen dies mit Hilfe einer Polynomialzeitreduktion: Dabei wird das NP-schwere Problem des euklidischen path traveler salesman problem (EP-TSP) [11] auf das Problem des kürzesten Hamiltonpfads mit Winkelbeschränkung (AREP-TSP) reduziert. EP-TSP beschreibt die Suche

nach der Permutation einer Reihe von Koordinaten $P \subset \mathbb{R}^2$, so dass die Summe der Länge der euklidischen Distanz zwischen aufeinanderfolgenden Koordinaten minimal ist.

Wir wenden effiziente Konstruktion an. Denn EP-TSP kann auf AREP-TSP reduziert werden, indem jeder Knoten in G durch eine Gitter aus 16 Knoten an dessen Stelle ersetzt wird (Abbildung 4). Diese Gitter hebeln das Problem der Winkelbeschränkung aus (Abbildung 5), denn von jeder beliebigen Himmelsrichtung kann nun unter Einhaltung der Winkelbedingung in eine andere Himmelsrichtung gereist werden.

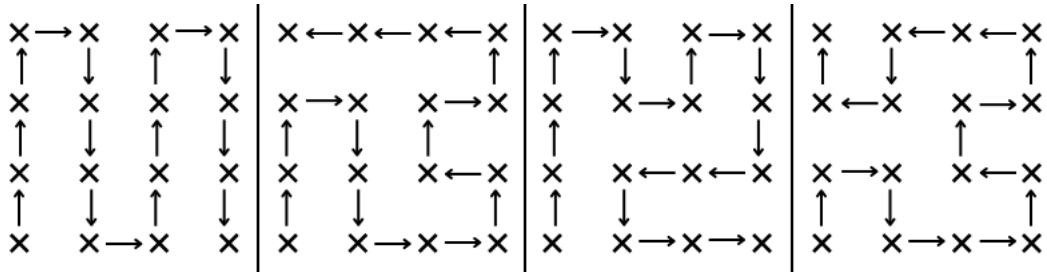

Abbildung 4: Durch das Gitterkomplex kann ein Knoten der unter dem Komplex liegt mit einem Knoten aus jeder anderen Richtung verbunden werden, ohne dass die Winkelbedingung verletzt wird. Durch entsprechende Drehung können somit alle Permutationen von Gittern als gültige Lösung für das AREP-TSP betrachtet werden.

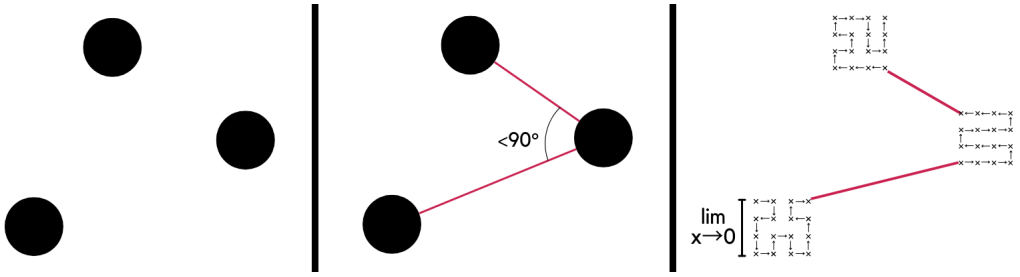

Abbildung 5: Eine Probleminstanz G (links) besitzt eine optimale Lösung für das EP-TSP (mitte), die durch die effiziente Konstruktion in G' (rechts) gleich der optimalen Lösung für das AREP-TSP ist. Die beiden roten Strecken gleichen sich durch $\lim_{x \to 0}$ einander an, während sich die schwarzen Strecken in den Gittern 0 angleichen.

Wo vorher also eine Permutation der Punkte gesucht war, ist nun eine Permutation der Gitterkomplexe gefragt. Indem wir die Größe der Gitter selbst gegen 0 gehen lassen, erhalten wir mit der Länge einer exakten Lösung von G' auch die Länge der exakte Lösung für G, sofern die 16 Knoten eines Gitterkomplexes direkt nacheinander abgehandelt werden.

Durch die Dreiecksungleichung ist gesichert, dass direkt nacheinander alle 16 Knoten eines Gitters abgehandelt werden. Umgekehrt lässt sich beweisen, dass eine Lösung, die einen Gitterkomplex mehrmals besucht, also nicht alle 16 Knoten direkt nacheinander abhandelt, nicht optimal sein kann. Denn durch das Weglassen des zweiten Besuches und stattdessen einem direkten Sprung zum nächsten Gitterkomplex verringert oder erhält sich nach $|a + b| \leq |a| + |b|$ die Streckenlänge.

Nun können wir das Problem des kürzesten Hamiltonpfads mit Winkelbeschränkung im konstruierten Graphen lösen. Wenn wir eine Lösung für das Problem des kürzesten Hamiltonpfads mit Winkelbeschränkung im konstruierten Graphen finden, haben wir auch eine Lösung für das euklidische path traveler salesman problem im ursprünglichen Graphen gefunden.

Die beschriebene Konstruktion kann für jede Probleminstanz in polynomieller Zeit durchgeführt werden, da wir den Graph lediglich um die 15 Knoten je Knoten in G erweitern. Sie versichert, dass für jede Probleminstanz G von EP-TSP, auch eine Probleminstanz G' von AREP-TSP existiert und, dass

die optimale Lösung beider Instanzen gleich ist.

Da EP-TSP NP-schwer ist, ist auch das Problem des kürzesten Hamiltonpfads mit Winkelbeschränkung NP-schwer. Demnach ist

$$\text{AREP-TSP} \geq_{\mathcal{P}} \text{EP-TSP}$$

Sofern $\mathcal{P} \neq \mathcal{NP}$, lässt sich AREP-TSP daher nicht in polynomieller Zeit optimal lösen.

3 Umsetzung

Um die Algorithmen anwenden zu können, benötigen wir einen einfachen Zugang zu den, zu verarbeitenden Daten. In diesem Fall sind das die Koordinaten und die Strecken, die zwischen diesen Koordinaten liegen. Wir speichern die Daten in Form einer Adjazenzmatrix.

3.1 Abstände, Winkel und Einfügen von Knoten

Im Folgenden werden allgemeine Funktionen erklärt, die zur Anwendung aller Heuristiken notwendig sind.

Abstände berechnen

Die Funktion `kantenBerechnen()` (siehe Anhang A, Zeile 19-36) berechnet die Kantenlängen aller Koordinaten zueinander unter Verwendung des Satzes von Pythagoras. Das Ergebnis wird in einer geschachtelten Liste gespeichert, wobei jede der inneren Liste die Kantenlängen eines Knotens zu allen anderen Knoten enthält.

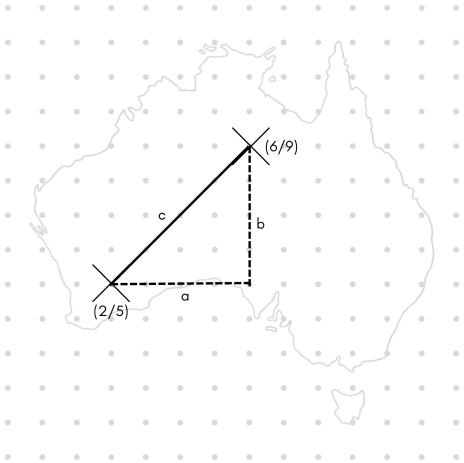

Abbildung 6: Der Abstand zwischen zwei Punkten in der euklidischen Ebene kann durch den Pythagorassatz berechnet werden

Die Funktion verwendet eine globale Variable `kanten_liste`, um die Kantenlängen zu speichern. Gerundet wird auf sechs Nachkommastellen. Dabei wird sich an den Beispieleingaben der Bundeswettbewerbsaufgabe (siehe Anhang B) orientiert.

Wenn die berechnete Kantenlänge gleich Null und `j` ungleich `i` ist, bedeutet das, eine Koordinate kommt mehrfach vor. Der Knoten und seine zugehörige Kantenliste wird aus `knoten_liste` und `kanten_liste` gelöscht.

Winkel berechnen

Die `winkelBerechnen()` (siehe Anhang A, Zeile 65-76) Funktion berechnet den Winkel zwischen zwei Kanten. Die Funktion arbeitet dafür mit den Parametern a, b und c, die die Indexe der Koordinatenpunkte darstellen. Sie überprüft zuerst, ob die drei Punkte kollinear sind, also auf einer Geraden liegen. Wenn dem so ist, gibt die Funktion entweder 0° zurück, wenn die Summe der Kanten ab und bc nicht gleich der Kante ac ist. Andernfalls wird 180° zurückgegeben.

Wenn sie nicht kollinear sind, berechnet die Funktion den Winkel beta zwischen den Kanten mit dem Sinussatz und gibt den Winkel in Grad zurück.

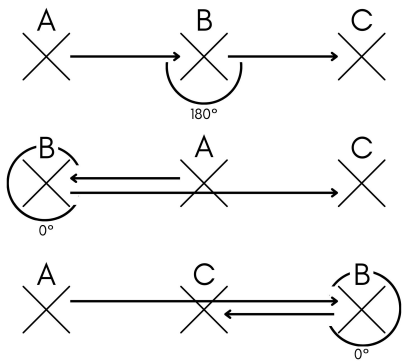

Abbildung 7: Fallunterscheidung bei kollinearen Koordinaten. Im oberen Fall ist die Winkelbedingung erfüllt, in den unteren Fällen allerdings nicht.

optimales Einfügen von Knoten in einen bestehenden Graphen
Die Funktion `moeglichstGuenstigEinfuegen()` (siehe Anhang A, Zeile 139-177) fügt einen Knoten möglichst optimal in einen Pfad ein. Optimal bedeutet hier, dass die Winkelbedingung erfüllt ist und die Länge des Pfades möglichst wenig ansteigt.

Die Funktion gibt eine Tupel aus vier Elementen zurück, bestehend aus:

- Einem booleschen Wert, der angibt, ob eine Änderung am Pfad vorgenommen wurde oder nicht
- Der aktualisierten Liste von Indexen der Knoten, die den Pfad bilden
- Der aktualisierten Liste von Koordinaten der Knoten, die den Pfad bilden
- Der aktualisierten Liste von booleschen Werten, die angibt, welche Kanten noch nicht im Pfad verwendet wurden

Die Funktion überprüft zunächst, ob der gegebene Knoten an den Anfang oder das Ende des Pfads angefügt werden kann. Ist dem so, wird die Verlängerung des Pfades, die das Anfügen zur Folge hätte, in der Variable `kuerzestes_einfuegen` gespeichert.

Falls der Knoten nicht am Anfang oder am Ende des Pfads eingefügt werden kann, wird überprüft, ob er zwischen zwei Knoten im Pfad eingefügt werden kann. Dabei müssen maximal drei verschiedene Winkel geprüft werden.

Wenn eine Einfügen möglich ist und die Pfadverlängerung geringer als die bisher als am besten angenommene Einfügung ist, werden die Werte überschrieben.

Nachdem der Algorithmus einmal über alle Elemente aus dem bisher bestehenden Pfad iteriert ist, wird geprüft, ob eine Einfügemöglichkeit gefunden wurde, die die Winkelbeschränkung erfüllt. In diesem Fall wird der Knoten dort eingefügt und True zurückgegeben. Falls keine Änderung am Pfad vorgenommen wurde, gibt die Funktion False zurück.

3.2 Nearest Neighbour

Das Implementieren der Nearest Neighbour Heuristik (siehe Anhang A, Zeile 187-212) ist vergleichsweise simpel. Da nur an das letzte Element der Route Koordinaten angehängt werden, muss nur ein Winkel geprüft werden.

Der Algorithmus ist in zwei Schleifen aufgebaut. Zunächst wird eine Schleife initialisiert, die durchläuft, bis alle Knoten besucht wurden oder festgestellt wird, dass kein Pfad existiert. Der Zähler `zaehler` gibt dabei an, welcher Knoten als Startknoten ausgewählt ist.

In jedem Schleifendurchlauf wird zuerst der aktuelle Knoten `index` und der Startknoten `start` gesetzt. Dann wird ein Besuchsarray `besucht` initialisiert, der Startknoten wird dabei als besucht markiert.

Es wird eine Liste `pfad_koordinaten` initialisiert, die die Koordinaten der Knoten auf dem Pfad enthält, analog dazu eine Liste `pfad_indexe`, gefüllt mit den Indizes der Knoten auf dem Pfad. Der erste Eintrag in diesen Listen ist der Startknoten. Die Variable `stelle` gibt den Index des aktuellen Knotens im Pfad an.

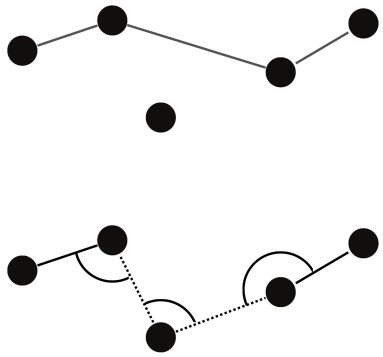

Abbildung 8: In den meisten Fällen müssen beim Einfügen eines Punktes in den Graphen drei verschiedene Winkel betrachtet werden. Ausnahmen sind die beiden Knoten am Anfang und am Ende des Pfades.

In einer inneren Schleife wird nun solange der nächstgelegene, unbesuchte Knoten gesucht, bis alle Knoten besucht wurden oder kein weiterer Knoten gefunden werden kann. Der nächste Knoten wird dabei mit der Funktion `naechsterKnoten` bestimmt.

Wenn ein nächster Knoten gefunden ist, wird dieser als besucht markiert und in die Listen `pfad_koordinaten` und `pfad_indexe` eingetragen. Der Index des aktuellen Knotens im Pfad wird erhöht und der nächste Knoten wird zum aktuellen Knoten. Es wird geprüft, ob alle Knoten besucht sind. Wenn dies der Fall ist, wird der Pfad als Ergebnis zurückgegeben.

Wenn kein nächster Knoten gefunden werden kann, geht der Algorithmus zurück zur äußeren Schleife und fährt mit einem anderen Knoten als Startknoten fort. Wenn alle Knoten als Startknoten ausprobiert und kein Pfad gefunden wurde, terminiert der Algorithmus und gibt das Ergebnis zurückgeben, das besagt, dass kein winkelbeschränkter Hamiltonpfad existiert.

Laufzeitbetrachtung

Die worst-case-Laufzeit von diesem Algorithmus wird wie folgt berechnet:

Die äußere Schleife läuft einmal für jeden Knoten im Graphen, daher wird sie im schlimmsten Fall n-mal durchlaufen, wobei n die Anzahl der Koordinaten ist.

Die innere Schleife ruft die Funktion `naechsterKnoten()` auf. Sie durchläuft alle Knoten im Graphen, um den nächsten Knoten auf dem Pfad zu finden. Außerdem ruft sie die Funktion `mogelichstGuenstigEinfuegen` auf, die selbst eine Laufzeit von n hat. Da sie im schlimmsten Fall n-mal aufgerufen wird, ergibt sich eine vereinfachte Laufzeit von $O(2n^2)$ für die innere Schleife.

Zusätzlich zu den Schleifen enthält der Code einige Konstantenoperationen, die im Folgenden nicht näher betrachten werden sollen.

Daher hat der Algorithmus vereinfacht eine Laufzeit von $O(n^3)$, da die äußere Schleife im schlimmsten Fall $O(n)$ Mal und die innere Schleife $O(2n^2)$ Mal durchlaufen wird.

$$O(n * (n * (n + n))) = O(n^3)$$

Der worst-case dieser Heuristik wäre eine Menge an Koordinaten, die immer bis zum vorletzten Knoten eine Route findet, dann aber abbrechen muss. Dieser Fall ist möglich (z.B. bei einem Dreieck), häufig wird aber bereits zu einem früheren Zeitpunkt kein Knoten gefunden, der unter Einhaltung der Winkelbedingung in den Graphen eingefügt werden kann. Das verbessert die average-Laufzeit.

3.3 Nearest Insertion Heuristik und Farthest Insertion Heuristik

Die Insertion Heuristiken (siehe Anhang A, Zeile 265-406) sind in vier Schleifen aufgebaut.

Ähnlich zur Nearest Neighbour Heuristik wird in der ersten Schleife jeder Knoten einmal als Startknoten angenommen. Gleichzeitig wird hier bereits ein zweiter Knoten in den Pfad hinzugefügt, der je nach

Heuristik der näheste oder der am weitesten entfernte Knoten ist. Die beiden Knoten werden hierbei als `links_unten` und `rechts_oben` bezeichnet, um eine Visualisierung des Vorgehens zu ermöglichen. Die Punkte entsprechen in der Realität nicht unbedingt dieser räumlichen Anordnung.

Dann folgt eine dreifach geschachtelte Schleife. In der tiefsten Ebene wird unter allen Knoten `uebrig`, die noch nicht in der Route berücksichtigt werden, der nächste Knoten ausgesucht. Dieser Knoten wird in den vier Heuristiken nach einem anderen Kriterium gewählt. Entweder der Knoten, der von allen Knoten im Pfad am weitesten oder nähesten liegt (globales Maximum/Minimum) oder aber der Knoten, der zu einem beliebigen Knoten auf dem Pfad am weitesten oder nähesten liegt (lokales Maximum/Minimum).

Diese Schleife wird durch die nächsthöher liegende Schleife solange aufgerufen, bis ein Knoten gefunden ist, der unter Einhaltung der Winkelbedingung in den Graphen eingefügt werden kann. Eingefügt wird durch die oben beschriebene Funktion `moeglichstGuenstigEinfuegen()`.

In der höchsten Ebene iteriert eine while-Schleife, bis jeder Knoten in den Graphen eingefügt wurde oder keiner der Knoten aus `uebrig` unter Einhaltung der Winkelbedingung in die Route eingefügt werden kann und so abgebrochen wird.

Laufzeit

Die worst-case-Laufzeit von diesem Algorithmus wird wie folgt berechnet:
Die äußere Schleife läuft einmal für jeden Knoten im Graphen, daher wird sie im schlimmsten Fall n-mal durchlaufen, wobei n die Anzahl der Koordinaten ist.

Die nächste Schleife wird im schlimmsten Fall zwei minus n-mal aufgerufen, bis alle Punkte in den Graphen hinzugefügt sind.

Auch die Schleife auf zweiter Ebene wird im schlimmsten Fall zwei minus n-mal aufgerufen, bis festgestellt ist, dass keiner der Punkte unter der Winkelbedingung in den Graphen hinzugefügt werden kann.

Die Schleife auf dritten Ebene läuft in jedem Fall n-mal durch, um den nächsten Knoten zu finden, dabei ruft sie die Funktion `laenge()` auf. Diese durchläuft im schlimmsten Fall n Knoten. Insgesamt ergibt sich also eine Laufzeit von $O(n^2)$ für die innerste Schleife.

Außerdem wird auf dritter Ebene die Funktion `moeglichstGuenstigEinfuegen()` aufgerufen. Diese Funktion hat eine Laufzeit von n.

Daher hat der Algorithmus eine Laufzeit von $O(n^4)$, da die äußere Schleife im schlimmsten Fall $O(n)$ besitzt, die zweite ebenfalls n-mal und die innere Schleife in jedem Fall $O(n^2 + n)$ Mal durchlaufen wird.

$$O(n * (n * (n * n + n))) = O(n^4)$$

Es wäre möglich, die Laufzeit der Insertion Heuristiken durch eine Halbierung der Maximalen Schleifendurchläufe der ersten Schleife zu verbessern. Dies wäre möglich, indem man pro Durchlauf nicht nur einen sondern zwei Punkte als Startpunkt begreift. Denn der erste Punkt `recht_oben`, nennen wir ihn B der zum Pfad hinzugefügt wird, müsste eigentlich später nicht ebenfalls noch einmal als Startpunkt gewählt werden. Dieser Punkt B wird nämlich als zweiten Punkt den bereits bekannten Startpunkt A wählen. Damit entsteht eine Liste $[B, A]$, die bereits in Form von $[A, B]$ existierte und später zu dem selben Pfad führen wird. Da die Laufzeit bei den gegebenen Beispiel allerdings keine Problematik darstellte, wurde diese Verbesserung in der Implementierung nicht berücksichtigt.

3.4 Random Insertion Heuristik

Die Random Insertion Heurstik (siehe Anhang A, Zeile 409-451) unterscheidet sich von seinen Vorgängern nur durch die Auswahl des nächsten Knotens. Denn hier wird kein bestimmtes Kriterium verfolgt, stattdessen wird ein zufälliger Knoten aus der Liste gewählt. In der Implementierung wurde auf einen Zufallsgenerator verzichtet, da sich die Koordinaten (siehe Anhang B) bereits in zufälliger Reihenfolge befinden.

Laufzeit

Entsprechend dazu verhält sich die Laufzeit der Random Insertion Heuristik nur geringfügig anders, als die übrigen Insertion Heuristiken.

Die einzige Veränderung findet man in der innersten Schleife. Zwar wird hier nach wie vor die Funktion `moeglichstGuenstigEinfuegen()` aufgerufen, doch zum Finden des nächsten Knoten brauchen wir nur noch eine einfache Schleife, die auch nur im schlimmsten, nicht aber in jedem Fall n-mal durchläuft.

Es ergibt sich nach wie vor eine Laufzeit von $O(n^4)$.

$$O(n * (n * (n + n))) = O(n^4)$$

3.5 Nearest Neighbour mit Insertion

Der Algorithmus `mischungNearestInsertionHeuristikUndNearestNeighbour()` (siehe Anhang A, Zeile 454-534) ist in drei großen while-Schleifen aufgebaut.

In der äußeren while-Schleife wird, wie bereits bekannt, der Startknoten und der Knoten mit dem geringsten Abstand zum Startknoten in den Pfad hinzugefügt.

In der nächsten while-Schleife wird geprüft, ob ein Knoten an den Anfang oder das Ende der Route angefügt werden kann. Falls dem so ist, wird sich der Punkt gemerkt. Außerdem wird in dieser Schleife die Variable `max_abstand` definiert. Diese Variable steht für den Abstand, den ein Punkt, der eingefügt werden soll, maximal zu dem Punkt haben kann, hinter dem er in die Route eingefügt werden soll. Alle Knoten mit länger Kante zum entsprechenden Knoten, erfüllen per Definition nicht die Winkelbedingung.

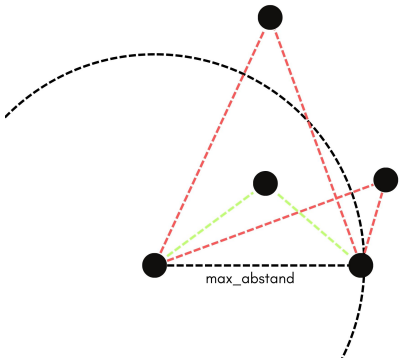

Abbildung 9: Alle Punkte, die nicht im Radius von `max_abstand` liegen, erfüllen die Winkelbedingung nicht

Dann wird eine weitere while-Schleife geöffnet, die nun für den Punkt aus der Route, der in der for-Schleife ausgesucht wurde, den nähesten, die Winkelbedingung erfüllenden, Knoten sucht.

Nachdem die for-Schleife vollständig durchgelaufen ist, ist nun der insgesamt am günstigsten einzufügende Punkt, direkt in Anbetracht seines späteren Platzes im Graphen gefunden und kann eingefügt werden.

Wenn irgendwann kein solcher Punkt mehr gefunden werden kann und alle möglichen Startpunkte einmal gewählt wurden, die Route aber trotzdem noch nicht vollständig ist, wird eine leere Liste zurückgegeben.

Laufzeit

Die Laufzeit dieser Heuristik berechnet sich wie folgt:

Die äußere Schleife kann im schlimmsten Fall n-mal durchlaufen werden, da jeder Punkt einmal als Startpunkt festgelegt wird.

Die nächste while-Schleife durchläuft im schlimmsten Fall $(n-2)$ Durchläufe. Genauso verhält es sich mit der ersten for-Schleife, die prüft, ob ein Punkt an den Anfang oder das Ende des Graphen hinzugefügt werden kann.

Die zweite for-Schleife durchläuft im schlimmsten Fall $(n-2)$ Durchläufe. Allerdings ist zu beachten, dass diese beiden for-Schleifen niemals beide n-mal durchlaufen werden. Tatsächlich kommen sie gemeinsam stets auf n, da die eine Schleife alle Knoten in der Route durchgeht und die andere Schleife alle übrigen Knoten abarbeitet.

Die zweite for-Schleife ruft nun selbst die while-Schleife auf, die n-mal durchlaufen kann. Diese wiederum ruft eine for-Schleife auf, sie kann ebenfalls n-mal durchlaufen.

Für die worst-case Laufzeit ergibt sich also $O(n^5)$.

$$O(n * (n * (n * (n * n)))) = O(n^5)$$

Allerdings ist zu beachten, dass die worst-case-Laufzeit niemals wirklich so eintreten kann. Dies liegt daran, dass das n in der dritten Klammer eigentlich eine Summe $uebrig + pfad\_indexe$ ist. Hier kann `pfad_indexe` nur einmal knapp n annehmen, wenn dann kein Punkt eingefügt werden kann, wird abgebrochen und das n in der zweiten Klammer würde nicht bis zum Ende durchlaufen. Es wird darum darauf hingewiesen, dass es sich bei O nur um eine Oberschranke handelt.

3.6 2-opt-Algorithmus

Die Funktion `twoOpt` (siehe Anhang A, Zeile 568-591) beginnt mit einer Schleife, die solange läuft, bis keine Verbesserungen mehr möglich sind.

Innerhalb der Schleife werden alle möglichen Paare von Kanten untersucht, die durch Tausch der Endpunkte möglicherweise zu einer kürzeren Route führen können. Wenn eine solche Verbesserung gefunden wird und die Winkelbedingung erfüllt ist, wird der Tausch durchgeführt und die Schleife beginnt von vorne.

Da wir immer nur verbessernde Schritte akzeptieren, wird irgendwann eine Lösung entstehen, die durch 2-opt-Schritte nicht mehr verbessert werden kann, das sogenannte lokale Minimum. Sobald keine weiteren Verbesserungen möglich sind, gibt die Funktion die optimierten Koordinaten und Indexe der Route zurück. Das bedeutet jedoch nicht, dass damit bereits die insgesamt beste Lösung — das globale Minimum — erreicht ist.

Laufzeit

Die äußere Schleife des Algorithmus ist eine while-Schleife, die so lange ausgeführt wird, bis keine Verbesserungen mehr möglich sind. Da der Algorithmus in jedem Durchlauf mindestens eine Verbesserung vornimmt, wird diese Schleife höchstens n^2-mal ausgeführt, wobei n in diesem Fall für die Menge an Kanten des Graphen steht.

Die innere Schleife hat eine Laufzeit von $O(n^2)$, da sie von i bis $n-2$ und für jedes i von $j = i+1$ bis n läuft.

Die Operationen innerhalb der Schleifen, ebenso wie die Funktion `winkelPruefen()` haben alle eine konstante Laufzeit.

Insgesamt hat der Algorithmus also eine worst-case Laufzeit von $O(n^4)$, wobei n die Menge der Kanten im Graph ist.

Die average-Laufzeit liegt jedoch auch hier stark darunter, da es unwahrscheinlich ist, dass jede Kante tatsächlich von einer Überkreuzung mit einer anderen Kante profitiert.

4 Diskussion

Um die Qualität der entwickelten Algorithmen zu evaluieren, wurden sie nun auf die sieben Beispieleingaben aus dem Bundeswettbewerb Informatik angewendet, die im Rahmen der Aufgabe „Weniger krumme Touren“ vorlagen (siehe Anhang B). Alle generierten Lösungen werden bereits unter Anwendung der 2-opt-Nachoptimierung ausgegeben (siehe Anhang C).

Bevor die verschiedenen Algorithmen verglichen werden können, muss erst sichergestellt werden, dass es sich bei den erzeugten Pfaden tatsächlich um gültige Lösungen des zugrundeliegenden Problems handelt. Genauer bedeutet das, dass es sich bei den Pfaden um einen Hamiltonpfad handeln muss, also alle Knoten des Graphen genau einmal besucht wurden und, dass die Winkelbedingung erfüllt ist, der Pfad also keine spitzen Winkel zwischen Pfadsegmenten aufweist. Anhand der Abbildungen in Anhang C kann durch optische Kontrolle sichergestellt und nachvollzogen werden, dass es sich bei den entwickelten Lösungen tatsächlich um winkelbeschränkte Hamiltonpfade handelt. Die Visualisierung der Pfade durch die turtle-Bibliothek hat sich somit als wertvolle Hilfe erwiesen.

Nun können die Längen der Pfade verglichen werden. Kürzere Pfadlängen entsprechen hierbei einer Lösung näher am Optimum und somit einer objektiv besseren Lösung.

Das Beispiel 1 (siehe Anhang B.1) wird von der Nearest Neighbour Heuristik genauso gut gelöst, wie von der gemischten Variante, der Nearest Neighbour mit Insertion Heuristik (siehe Anhang C.1).

Die Beispiele 2 und 3 (siehe Anhang B.2 und B.3) werden von der Nearest Neighbour Heuristik minimal besser gelöst, als von der Mischung aus Nearest Insertion und Nearest Neighbour (siehe Anhang C.2 und C.3).

Die Beispiele 4, 5 und 7 (siehe Anhang B.4, B.5 und B.7) werden von der Mischung aus Nearest Insertion und Nearest Neighbour am optimalsten gelöst (siehe Anhang C.4, C.5 und C.7).

Es fällt somit auf, dass die Mischung aus dem Nearest Insertion und dem Nearest Neighbour Algorithmus fast immer sehr gut oder sogar am besten abschneidet.

Eine überraschende Ausnahme bildet hierbei Beispiel 6 (siehe Anhang B.6), das von der Random Insertion Heuristik mit Abstand am besten gelöst werden konnte (siehe Anhang C.6).

Wichtig zu beachten ist, dass der randomisierte Algorithmus in diesem Fall deterministisch ist, da die Randomisierung lediglich durch die zufällige Reihenfolge der Eingabekoordinaten (siehe Anhang B.6) zustande kommt. Der Algorithmus wird also mit dieser spezifischen Eingabe stets zum gleichen und zugleich besten Ergebnis kommen. In einem anderen Anwendungskontext, in dem eine Zufallsvariable verwendet wird, kann die Qualität der Lösungen für ein und die selbe Eingabe hingegen stark variieren.

Die Interpretation der Ergebnisse deutet darauf hin, dass eine Mischung von Heuristiken zu konsistenteren Lösungen führt, da verschiedene Stärken kombiniert werden. So konnte die gemischte Heuristik stets akzeptable, manchmal sogar die besten Ergebnisse unter den getesteten Heuristiken erzielen.

Dennoch haben zum aktuellen Forschungsstand alle der sieben Algorithmen ihre Berechtigung. Zwar schneidet die Mischung aus der Nearest Neighbour Heuristik mit Insertion in vier der sieben Fällen am besten und sonst immer zumindest überdurchschnittlich ab, in vier der sieben Probleminstanzen können aber auch andere Algorithmen genauso gute oder bessere Lösungen liefern.

Weitere Forschung zum Problem des winkelbeschränkten, minimalen Hamiltonpfades sollte sich darum darauf konzentrieren, die entwickelten Algorithmen auf eine größere Anzahl von Beispielen anzuwenden, um empirische Daten zu sammeln und Effektivität der Algorithmen weiter zu bewerten. Insbesondere sollte die Frage beantwortet werden, ob sich für jeden Algorithmus eine Probleminstanz konstruieren lässt, auf der derselbige am besten arbeitet. Durch die Anwendung auf weitere Beispiele können außerdem mögliche Muster oder Trends identifiziert und die Algorithmen weiter optimiert werden, um eine noch bessere Leistung, auch und insbesondere in bestimmten Anwendungskontexten mit spezifischen Problemtypen zu erzielen.

4.1 Vergleich zum Optimum

Bisher haben wir die Lösungsqualität nur relativ zu den anderen Algorithmen bestimmt. Da wir zur Lösung unseres Problems Heuristiken verwendet haben, ist es aber auch sinnvoll einzuordnen, wie gut die gefundenen Lösungen an der Optimallösung liegen.

Da diese aber nur sehr schwer gefunden werden kann[2], suchen wir stattdessen nach einer unteren Schranke, einer Pfadlänge, die die Optimallösung mindestens haben muss. Eine geeignete Schranke wäre in unserem Fall die Länge des minimum spanning trees (MST) [12, S. 345 f.]. Der MST kann in polynomiller Laufzeit berechnet werden und gibt die minimale Kantenlänge an, die benötigt wird, um alle Knoten eines Graphen zu verbinden. In einer simplen Probleminstanz von nur zwei Koordinaten ist unser MST gleich der Optimallösung für unser winkelbeschränktes Hamiltonpfadproblem. Algorithmen um den MST zu berechnen sind der Prim-Algorithmus (siehe Anhang A, Zeile 536-559) oder der Kruskal-Algorithmus [12, S. 347 ff.].

[2] siehe Beweis der NP-Schwere in 2.7

Probleminstanz	Anh. B.1	Anh. B.2	Anh. B.3	Anh. B.4	Anh. B.5	Anh. B.6	Anh. B.7
MST	847.4	2171.9	1750.2	1156.8	2364.0	2552.0	3149.3
AREP-TSP	853.2	2183.7	1968.2	1205.1	3700.2	4400.4	4782.2
prozentualer Anteil	100.7%	100.5%	112.5%	104.2%	156.5%	172.4%	127.5%

Tabelle 1: Für jede der sieben Probleminstanzen aus Anhang B die Länge des MST und die kürzeste generierte Lösung aus Anhang C, sowie der prozentuale Anteil der generierten Lösung vom MST, um eine Vergleichbarkeit zu schaffen.

Die untere Schranke erlaubt es, unsere Lösungen einzuordnen. Sollte unsere Lösung sehr nah an der unteren Schranke oder ihr sogar gleich sein, wissen wir, dass nur noch wenig oder kein Optimierungsbedarf besteht. Umgekehrt lässt sich allerdings keine Aussage treffen, da die Möglichkeit besteht, dass die untere Schranke weit von der optimalen Lösung entfernt liegt. Eine gefundene Lösung, die unter der Schranke liegt (prozentualer Anteil $< 100\%$) kann jedoch mit Sicherheit als falsch angenommen werden.

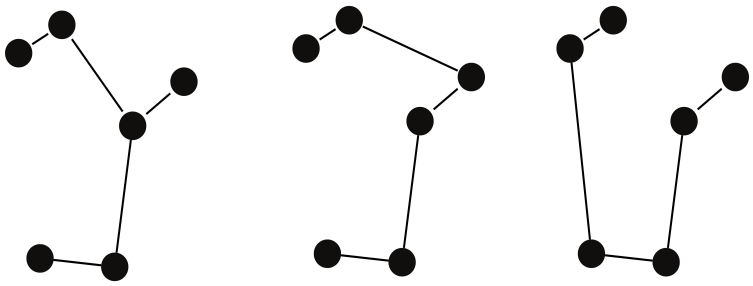

Abbildung 10: Abweichung der Pfadlänge vom minimum spanning tree (links) im Vergleich zum kürzesten Hamiltonpfad (mitte) und dem kürzesten winkelbeschränkten Hamiltonpfad (rechts)

Literaturverzeichnis

[1] S. Sabba und S. Chikhi, „Integrating the best 2-opt method to enhance the genetic algorithm execution time in solving the traveler salesman problem“, in Complex Systems and Dependability, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 195–208.

[2] D. Applegate, R. Bixby, und W. Cook, „On the solution of traveling salesman problems“, 1998.

[3] D. J. Rosenkrantz, R. E. Stearns, und P. M. Lewis II, „An analysis of several heuristics for the traveling salesman problem“, SIAM J. Comput., Bd. 6, Nr. 3, S. 563–581, 1977.

[4] „2. Runde“, Bwinf.de. [Online]. Verfügbar unter: https://bwinf.de/bundeswettbewerb/41/2/. [Zugegriffen: 12-Feb-2024].

[5] P. K. Agarwal, P. Raghavan, und H. Tamaki, „Motion planning for a steering-constrained robot through moderate obstacles“, in Proceedings of the twenty-seventh annual ACM symposium on Theory of computing - STOC '95, 1995.

[6] J. C. Clements, „Minimum-time turn trajectories to fly-to points“, Optim. Control Appl. Methods, Bd. 11, Nr. 1, S. 39–50, 1990.

[7] S. Arora, „Polynomial time approximation schemes for Euclidean TSP and other geometric problems“, in Proceedings of 37th Conference on Foundations of Computer Science, 2002, S. 2–11.

[8] V. Traub, J. Vygen, und R. Zenklusen, „Reducing path TSP to TSP“, in Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, 2020.

[9] S. Asaeedi und M. S. Shamaee, „Angle constrained Paths“, Research Square, 2023.

[10] S. P. Fekete und G. J. Woeginger, „Angle-restricted tours in the plane“, Comput. Geom., Bd. 8, Nr. 4, S. 195–218, 1997.

[11] C. H. Papadimitriou, „The Euclidean travelling salesman problem is NP-complete“, Theor. Comput. Sci., Bd. 4, Nr. 3, S. 237–244, 1977.

[12] A. Gogol-Döring und T. Letschert, Algorithmen und Datenstrukturen für Dummies. Berlin: Blackwell Verlag, 2019.

Anhang

Anhang A: Quellcode

```
from math import sqrt, degrees, acos
from sys import maxsize
from turtle import screensize, speed, setup, up, down, goto, dot, write, pencolor,
    ↪ hideturtle, clear

# Einlesen der Koordinaten aus der .txt-Datei in eine geschachtelte Liste
def koordinatenEinlesen(datei):
    global knoten_liste
    koordinaten = open(datei, mode="r", encoding="utf-8")
    # Die Koordinaten in eine einfache Liste einlesen [x1 y1, x2 y2, ...]
    knoten_liste = koordinaten.read().splitlines()
    # Die Koordinaten in einer geschachtelten Liste in x und y trennen [[x1, y1], [x2, y2
    ↪ ], ...]
    for i in range(len(knoten_liste)):
        knoten_liste[i] = knoten_liste[i].split(" ")
        for j in range(2):
            knoten_liste[i][j] = float(knoten_liste[i][j])
    return

```

```
# Berechnen der Kantenlaenge nach Pythagoras
def kantenBerechnen():
    global kanten_liste
    for i in range(len(knoten_liste)):
        # Erstellen einer leeren Liste fuer die Streckenlaengen von dem Knoten i zu den anderen Knoten
        kanten_i = []
        for j in range(len(knoten_liste)):
            # Berechnen der Kantenlaenge zwischen den Koordinaten i und j nach Pythagoras
            strecke = sqrt((knoten_liste[j][0] - knoten_liste[i][0])**2 + (knoten_liste[j][1] - knoten_liste[i][1])**2)
            strecke = round(strecke, 6)
            # Keine Knoten sollten doppelt vorkommen
            if strecke == 0.0 and j != i:
                del knoten_liste[i]
                del kanten_liste[i]
            # Hinzufuegen der Kantenlaenge zur Liste der Kantenlaengen von i zu jedem anderen Koordinaten
            kanten_i.append(strecke)
        # Hinzufuegen der Liste der Kantenlaengen von i zu den anderen Knoten zur Liste der Kantenlaenge
        kanten_liste.append(kanten_i)
    return

# Bei bestimmten Algorithmen ist es wichtig, dass in der Adjazenzliste beim Abstand zu sich selbst nicht 0 sondern ein besonders hoher Wert angegeben wird
def kantenMitMaxsizeZuSichSelbst(kanten, knoten):
    for i in range(len(knoten)):
        for j in range(len(knoten)):
            if j == i:
                kanten[i][j] = maxsize
    return kanten

# Bei anderen Algorithmen ist es wichtig, dass in der Adjazenzliste beim Abstand zu sich selbst nicht maxsize sondern 0 angegeben wird
def kantenMitNullZuSichSelbst(kanten, knoten):
    for i in range(len(knoten)):
        for j in range(len(knoten)):
            if j == i:
                kanten[i][j] = 0
    return kanten

# Gibt zurueck, ob der Winkel zwischen pc und pj im Gegenuhrzeigersinn oder im Uhrzeigersinn oder kollinear ist, das wird manchmal zum Berechnen der Winkel benoetigt
def orientierung(p, c, j):
    wert = (c[1] - p[1]) * (j[0] - c[0]) - (c[0] - p[0]) * (j[1] - c[1])
    if wert == 0:
        return "kollinear"
    elif wert > 0:
        return "mit"
    else:
        return "gegen"

# Diese Funktion berechnet den Winkel zwischen den Kanten ab und bc, wobei a, b und c Indexe der Koordinatenpunkte sind
def winkelBerechnen(punkt_a, punkt_b, punkt_c):
    if orientierung(knoten_liste[punkt_a], knoten_liste[punkt_b], knoten_liste[punkt_c]) != "kollinear":
        a = kanten_liste[punkt_c][punkt_b]
        b = kanten_liste[punkt_a][punkt_c]
        c = kanten_liste[punkt_b][punkt_a]
        beta = degrees(acos(round((b**2 - (a**2 + c**2)) / (-2*a*c), 6)))
        return beta
    else:
        if punkt_a == punkt_b or punkt_a == punkt_c or punkt_b == punkt_c or (kanten_liste[punkt_a][punkt_b] + kanten_liste[punkt_b][punkt_c]) != kanten_liste[punkt_a][punkt_c]:
            return 0
        else:
            return 180

# Zur Visualisierung der Pfade reicht eine simple Bibliothek wie turtle vollkommen aus
```

```
79 def turtle(liste, boolesche, farbe):
       clear()
81     hideturtle()
       up()
83     win_width, win_height, bg_color = 2000, 2000, 'white'

85     pencolor(farbe)
       setup()
87     screensize(win_width, win_height, bg_color)
       speed(10)
89
       for i in range(len(liste)):
91         if boolesche == False:
               up()
93         else:
               down
95         goto(liste[i][0], liste[i][1])
           down()
97         dot(3)
       return
99
   # Die Vorarbeit besteht darin, die Koordinaten aus den .txt-Datein in eine geschachtelte
       ↪ Liste einzulesen und die Kantenlaenge zwischen diesen Koordinaten zu berechnen
101 def vorarbeit(datei):
       koordinatenEinlesen(datei)
103    kantenBerechnen()
       #turtle(knoten_liste, False, "black")
105    return

107 # Diese Funktionen werden bei Farthes und Nearest Insertion benoetigt
   def abstand(pfad, kanten, knoten_aus_uebrig):
109    abstand = 0
       for i in pfad:
111        abstand += kanten[i][knoten_aus_uebrig]
       return abstand
113
   def minAbstandLokal(pfad, kanten, knoten_aus_uebrig):
115    abstand = 0
       for i in pfad:
117        if kanten[i][knoten_aus_uebrig] < abstand:
               abstand = kanten[i][knoten_aus_uebrig] > abstand
119    return abstand

121 def maxAbstandLokal(pfad, kanten, knoten_aus_uebrig):
       abstand = 0
123    for i in pfad:
           if kanten[i][knoten_aus_uebrig] > abstand:
125            abstand = kanten[i][knoten_aus_uebrig] > abstand
       return abstand
127
   # Diese Funktion wird beim Nearest Neighbour Algorithmus benoetigt
129 def naechsterKnoten(kanten_an_parent, parent, pfad_indexe, besucht, verboten):
       naechster_punkt = None
131    kuerzeste_kante = maxsize
       for j in range(len(kanten_an_parent)):
133        if besucht[j] != True and j not in verboten and kanten_an_parent[j] <
       ↪ kuerzeste_kante and (len(pfad_indexe) == 1 or winkelBerechnen(pfad_indexe[parent
       ↪ -1], pfad_indexe[parent], j) >= 90):
               kuerzeste_kante = kanten_an_parent[j]
135            naechster_punkt = j
       return naechster_punkt
137
   # Diese Funktion wird fuer alle Insertion-Algorithmen benoetigt
139 def moeglichstGuenstigEinfuegen(knoten, kanten, gewaehlte_kante_index, pfad_indexe,
       ↪ pfad_koordinaten, uebrig):
       aenderung = False
141    kuerzestes_einfuegen = maxsize
       an = None
143
       if winkelBerechnen(gewaehlte_kante_index, pfad_indexe[0], pfad_indexe[1]) >= 90 and
       ↪ kanten[pfad_indexe[0]][gewaehlte_kante_index] < kuerzestes_einfuegen:
145        kuerzestes_einfuegen = kanten[pfad_indexe[0]][gewaehlte_kante_index]
           an = "anfang"
```

```
147         aenderung = True

149     if winkelBerechnen(pfad_indexe[len(pfad_indexe)-2], pfad_indexe[len(pfad_indexe)-1],
        ↪ gewaehlte_kante_index) >= 90 and kanten[pfad_indexe[len(pfad_indexe)-1]][
        ↪ gewaehlte_kante_index] < kuerzestes_einfuegen:
            kuerzestes_einfuegen = kanten[pfad_indexe[len(pfad_indexe)-1]][
        ↪ gewaehlte_kante_index]
151         an = "len(pfad_indexe)-1"
            aenderung = True
153
        for l in range(len(pfad_indexe)-1):
155         if (kanten[l][gewaehlte_kante_index] + kanten[l+1][gewaehlte_kante_index]) -
        ↪ kanten[pfad_indexe[l]][l+1] < kuerzestes_einfuegen and winkelBerechnen(pfad_indexe
        ↪ [l], gewaehlte_kante_index, pfad_indexe[l+1]) >= 90 and ((l == 0 and (len(
        ↪ pfad_indexe) == 2 or winkelBerechnen(gewaehlte_kante_index, pfad_indexe[1],
        ↪ pfad_indexe[2]) >= 90)) or (l != 0 and ((l == len(pfad_indexe)-2 and
        ↪ winkelBerechnen(pfad_indexe[l-1], pfad_indexe[l], gewaehlte_kante_index) >= 90) or
        ↪  (winkelBerechnen(pfad_indexe[l-1], pfad_indexe[l], gewaehlte_kante_index) >= 90
        ↪ and winkelBerechnen(gewaehlte_kante_index, pfad_indexe[l+1], pfad_indexe[l+2]) >=
        ↪ 90)))):
                kuerzestes_einfuegen = (kanten[l][gewaehlte_kante_index] + kanten[l+1][
        ↪ gewaehlte_kante_index]) - kanten[pfad_indexe[l]][l+1]
157             an = l
                aenderung = True
159
        if aenderung != False:
161         if an == "anfang":
                pfad_indexe.insert(0, gewaehlte_kante_index)
163             pfad_koordinaten.insert(0, knoten[gewaehlte_kante_index])
                uebrig[gewaehlte_kante_index] = False
165             return(True, pfad_indexe, pfad_koordinaten, uebrig)
            elif an == "len(pfad_indexe)-1":
167             pfad_indexe.append(gewaehlte_kante_index)
                pfad_koordinaten.append(knoten[gewaehlte_kante_index])
169             uebrig[gewaehlte_kante_index] = False
                return(True, pfad_indexe, pfad_koordinaten, uebrig)
171         else:
                pfad_indexe.insert(an+1, gewaehlte_kante_index)
173             pfad_koordinaten.insert(an+1, knoten[gewaehlte_kante_index])
                uebrig[gewaehlte_kante_index] = False
175             return(True, pfad_indexe, pfad_koordinaten, uebrig)
        else:
177         return(False, pfad_indexe, pfad_koordinaten, uebrig)

179 # Diese Funktion wird am Ende zum Berechnen der Pfadlaenge benoetigt
    def laenge(kanten, pfad):
181     laenge = 0
        for i in range(len(pfad)-1):
183         laenge += kanten[pfad[i]][pfad[i+1]]
        return laenge
185
    # Hinzufuegen des naehesten Knotens, gemessen am letzten Pfadglied
187 def nearestNeighbour(kanten, knoten):
        zaehler = 0
189     while True:
            zaehler += 1
191         if zaehler == len(knoten):
                return (False, [], [])
193         start = knoten[zaehler]
            index = zaehler
195         besucht = len(knoten) * [False]
            besucht[index] = True
197         pfad_koordinaten = [start]
            pfad_indexe = [index]
199         stelle = 0
            while True:
201             naechster_knoten = naechsterKnoten(kanten[index], stelle, pfad_indexe,
        ↪ besucht, [])
                if naechster_knoten == None:
203                 break
                else:
205                 besucht[naechster_knoten] = True
                    pfad_koordinaten.append(knoten[naechster_knoten])
```

```
207                 pfad_indexe.append(naechster_knoten)
                    stelle += 1
209                 index = naechster_knoten
                    if len(pfad_koordinaten) == len(knoten):
211                     #turtle(pfad_koordinaten, True, "red")
                        return (True, pfad_koordinaten, pfad_indexe)
213
# Suchen nach dem naehesten Punkt im Vergleich zu allen Punkten des Pfades und
    ↪ anschliessend moeglichst guenstiges Einfuegen in den Pfad
215 def nearestInsertionHeuristik(kanten, knoten):
    kanten = kantenMitMaxsizeZuSichSelbst(kanten, knoten)
217 zaehler = 0
    while True:
219     zaehler += 1
        if zaehler == len(knoten):
221         return (False, [], [])
        links_unten = knoten[zaehler]
223     index_links_unten = zaehler

225     kuerzste_strecke = min(kanten[index_links_unten])
        index_rechts_oben = kanten[index_links_unten].index(kuerzste_strecke)
227     rechts_oben = knoten[index_rechts_oben]

229     uebrig = len(knoten) * [True]
        uebrig[index_rechts_oben] = False
231     uebrig[index_links_unten] = False

233     pfad_koordinaten = [links_unten, rechts_oben]
        pfad_indexe = [index_links_unten, index_rechts_oben]
235     abbruch = False
        while True:
237         if abbruch == True:
                break
239         if len(pfad_koordinaten) == len(knoten):
                #turtle(pfad_koordinaten, True, "red")
241             return (True, pfad_koordinaten, pfad_indexe)

243         zu_pruefen = uebrig.copy()
            while True:
245
                kuerzeste_kante = maxsize
247             kuerzeste_kante_index = None

249             for k in range(len(uebrig)):
                    if zu_pruefen[k] != False and abstand(pfad_indexe, kanten, k) <
    ↪ kuerzeste_kante:
251                     kuerzeste_kante = abstand(pfad_indexe, kanten, k)
                        kuerzeste_kante_index = k
253
                if kuerzeste_kante_index == None:
255                 abbruch = True
                    break
257             else:
                    zu_pruefen[kuerzeste_kante_index] = False
259
                aenderung, pfad_indexe, pfad_koordinaten, uebrig =
    ↪ moeglichstGuenstigEinfuegen(knoten, kanten, kuerzeste_kante_index, pfad_indexe,
    ↪ pfad_koordinaten, uebrig)
261             if aenderung == True:
                    break
263
# Suchen nach dem weitesten Punkt im Vergleich zu allen Punkten des Pfades und
    ↪ anschliessend moeglichst guenstiges Einfuegen in den Pfad
265 def farthestInsertionHeuristik(kanten, knoten):
    kanten = kantenMitNullZuSichSelbst(kanten, knoten)
267 zaehler = 0
    while True:
269     zaehler += 1
        if zaehler == len(knoten):
271         return (False, [], [])
        links_unten = knoten[zaehler]
273     index_links_unten = zaehler
```

```
275         laengste_strecke = max(kanten[index_links_unten])
            index_rechts_oben = kanten[index_links_unten].index(laengste_strecke)
277         rechts_oben = knoten[index_rechts_oben]

279         uebrig = len(knoten) * [True]
            uebrig[index_rechts_oben] = False
281         uebrig[index_links_unten] = False

283         pfad_koordinaten = [links_unten, rechts_oben]
            pfad_indexe = [index_links_unten, index_rechts_oben]
285         abbruch = False
            while True:
287             if abbruch == True:
                    break
289             if len(pfad_koordinaten) == len(knoten):
                    #turtle(pfad_koordinaten, True, "red")
291                 return (True, pfad_koordinaten, pfad_indexe)
                zu_pruefen = uebrig.copy()
293             while True:
                    laengste_kante = 0
295                 laengste_kante_index = None

297                 for k in range(len(zu_pruefen)):
                        if zu_pruefen[k] != False and abstand(pfad_indexe, kanten, k) >
    ↪ laengste_kante:
299                         laengste_kante = abstand(pfad_indexe, kanten, k)
                            laengste_kante_index = k
301
                    if laengste_kante_index == None:
303                     abbruch = True
                        break
305                 else:
                        zu_pruefen[laengste_kante_index] = False
307
                    aenderung, pfad_indexe, pfad_koordinaten, uebrig =
    ↪ moeglichstGuenstigEinfuegen(knoten, kanten, laengste_kante_index, pfad_indexe,
    ↪ pfad_koordinaten, uebrig)
309                 if aenderung == True:
                        break
311
# Suchen nach dem naehesten Punkt im Vergleich zu einem bestimmten Punkt des Pfades und
    ↪ anschliessend moeglichst guenstiges Einfuegen in den Pfad
313 def nearestInsertionHeuristikMitLokalemMinimum(kanten, knoten):
    kanten = kantenMitMaxsizeZuSichSelbst(kanten, knoten)
315 zaehler = 0
    while True:
317     zaehler += 1
        if zaehler == len(knoten):
319         return (False, [], [])
        links_unten = knoten[zaehler]
321     index_links_unten = zaehler

323     kuerzste_strecke = min(kanten[index_links_unten])
        index_rechts_oben = kanten[index_links_unten].index(kuerzste_strecke)
325     rechts_oben = knoten[index_rechts_oben]

327     uebrig = len(knoten) * [True]
        uebrig[index_rechts_oben] = False
329     uebrig[index_links_unten] = False

331     pfad_koordinaten = [links_unten, rechts_oben]
        pfad_indexe = [index_links_unten, index_rechts_oben]
333     abbruch = False
        while True:
335         if abbruch == True:
                break
337         if len(pfad_koordinaten) == len(knoten):
                #turtle(pfad_koordinaten, True, "red")
339             return (True, pfad_koordinaten, pfad_indexe)
            zu_pruefen = uebrig.copy()
341         while True:
                kuerzeste_kante = maxsize
343             kuerzeste_kante_index = None
```

```
345                 for k in range(len(uebrig)):
                        if zu_pruefen[k] != False and minAbstandLokal(pfad_indexe, kanten, k)
    ↪  < kuerzeste_kante:
347                         kuerzeste_kante = abstand(pfad_indexe, kanten, k)
                            kuerzeste_kante_index = k
349
                    if kuerzeste_kante_index == None:
351                     abbruch = True
                        break
353                 else:
                        zu_pruefen[kuerzeste_kante_index] = False
355
                    aenderung, pfad_indexe, pfad_koordinaten, uebrig =
    ↪ moeglichstGuenstigEinfuegen(knoten, kanten, kuerzeste_kante_index, pfad_indexe,
    ↪ pfad_koordinaten, uebrig)
357                 if aenderung == True:
                        break
359
# Suchen nach dem weitesten Punkt im Vergleich zu einem bestimmten Punkt des Pfades und
    ↪ anschliessend moeglichst guenstiges Einfuegen in den Pfad
361 def farthestInsertionHeuristikMitLokalemMaximum(kanten, knoten):
    kanten = kantenMitNullZuSichSelbst(kanten, knoten)
363     zaehler = 0
    while True:
365         zaehler += 1
        if zaehler == len(knoten):
367             return (False, [], [])
        links_unten = knoten[zaehler]
369         index_links_unten = zaehler

371         laengste_strecke = max(kanten[index_links_unten])
        index_rechts_oben = kanten[index_links_unten].index(laengste_strecke)
373         rechts_oben = knoten[index_rechts_oben]

375         uebrig = len(knoten) * [True]
        uebrig[index_rechts_oben] = False
377         uebrig[index_links_unten] = False

379         pfad_koordinaten = [links_unten, rechts_oben]
        pfad_indexe = [index_links_unten, index_rechts_oben]
381         abbruch = False
        while True:
383             if abbruch == True:
                    break
385             if len(pfad_koordinaten) == len(knoten):
                    #turtle(pfad_koordinaten, True, "red")
387                 return (True, pfad_koordinaten, pfad_indexe)
            zu_pruefen = uebrig.copy()
389             while True:
                    laengste_kante = 0
391                 laengste_kante_index = None

393                 for k in range(len(uebrig)):
                        if zu_pruefen[k] != False and maxAbstandLokal(pfad_indexe, kanten, k)
    ↪  > laengste_kante:
395                         laengste_kante = abstand(pfad_indexe, kanten, k)
                            laengste_kante_index = k
397
                    if laengste_kante_index == None:
399                     abbruch = True
                        break
401                 else:
                        zu_pruefen[laengste_kante_index] = False
403
                    aenderung, pfad_indexe, pfad_koordinaten, uebrig =
    ↪ moeglichstGuenstigEinfuegen(knoten, kanten, laengste_kante_index, pfad_indexe,
    ↪ pfad_koordinaten, uebrig)
405                 if aenderung == True:
                        break
407
# Suchen nach dem weitesten Punkt im Vergleich zu einem bestimmten Punkt des Pfades und
    ↪ anschliessend moeglichst guenstiges Einfuegen in den Pfad
```

```
409 def randomInsertionHeuristik(kanten, knoten):
    zaehler = 0
411 while True:
        zaehler += 1
413     if zaehler == len(knoten)-1:
            return (False, [], [])
415     links_unten = knoten[zaehler]
        index_links_unten = zaehler
417
        index_rechts_oben = zaehler+1
419     rechts_oben = knoten[index_rechts_oben]

421     uebrig = len(knoten) * [True]
        uebrig[index_rechts_oben] = False
423     uebrig[index_links_unten] = False

425     pfad_koordinaten = [links_unten, rechts_oben]
        pfad_indexe = [index_links_unten, index_rechts_oben]
427     abbruch = False
        while True:
429         if abbruch == True:
                break
431         if len(pfad_koordinaten) == len(knoten):
                #turtle(pfad_koordinaten, True, "red")
433             return (True, pfad_koordinaten, pfad_indexe)
            zu_pruefen = uebrig.copy()
435         while True:
                zufaellige_kante_index = None
437
                for k in range(len(uebrig)):
439                 if zu_pruefen[k] != False:
                        zufaellige_kante_index = k
441                     break

443             if zufaellige_kante_index == None:
                    abbruch = True
445                 break
                else:
447                 zu_pruefen[zufaellige_kante_index] = False

449             aenderung, pfad_indexe, pfad_koordinaten, uebrig =
    ↪ moeglichstGuenstigEinfuegen(knoten, kanten, zufaellige_kante_index, pfad_indexe,
    ↪ pfad_koordinaten, uebrig)
                if aenderung == True:
451                 break

453 # Dieser Algorithmus verbindet Nearest Neighbour und Nearest Insertion miteinander, indem
    ↪  die guenstigste Kante (direkt mit Beruecksichtigung der Winkelbedingung) gesucht
    ↪ und in den Pfad eingefuegt wird
  def mischungNearestInsertionHeuristikUndNearestNeighbour(kanten, knoten):
455     kanten = kantenMitMaxsizeZuSichSelbst(kanten, knoten)
        zaehler = 0
457     while True:
            zaehler += 1
459         if zaehler == len(knoten):
                return (False, [], [])
461         links_unten = knoten[zaehler]
            index_links_unten = zaehler
463
            kuerzste_strecke = min(kanten[index_links_unten])
465         index_rechts_oben = kanten[index_links_unten].index(kuerzste_strecke)
            rechts_oben = knoten[index_rechts_oben]
467
            uebrig = len(knoten) * [True]
469         uebrig[index_rechts_oben] = False
            uebrig[index_links_unten] = False
471
            pfad_koordinaten = [links_unten, rechts_oben]
473         pfad_indexe = [index_links_unten, index_rechts_oben]
            while True:
475             aenderung = False
                kuerzeste_kante = maxsize
477             kuerzeste_kante_index = None
```

```
            an = None

            for k in range(len(uebrig)):
                if uebrig[k] != False and winkelBerechnen(k, pfad_indexe[0], pfad_indexe
    ↪ [1]) >= 90 and kanten[pfad_indexe[0]][k] < kuerzeste_kante:
                    kuerzeste_kante = kanten[pfad_indexe[0]][k]
                    kuerzeste_kante_index = k
                    an = "anfang"
                    aenderung = True

                if uebrig[k] != False and winkelBerechnen(pfad_indexe[len(pfad_indexe)
    ↪ -2], pfad_indexe[len(pfad_indexe)-1], k) >= 90 and kanten[pfad_indexe[len(
    ↪ pfad_indexe)-1]][k] < kuerzeste_kante:
                    kuerzeste_kante = kanten[pfad_indexe[len(pfad_indexe)-1]][k]
                    kuerzeste_kante_index = k
                    an = len(pfad_indexe)-1
                    aenderung = True

            for l in range(len(pfad_indexe)-1):
                max_abstand = kanten[pfad_indexe[l]][pfad_indexe[l+1]]

                zu_pruefen = uebrig.copy()
                while True:

                    if len(pfad_koordinaten) == len(knoten):
                        #turtle(pfad_koordinaten, True, "red")
                        return (True, pfad_koordinaten, pfad_indexe)

                    naehester_punkt_index = None
                    for n in range(len(zu_pruefen)):
                        if zu_pruefen[n] != False and (kanten[l][n] + kanten[l+1][n]) -
    ↪ kanten[pfad_indexe[l]][l+1] < kuerzeste_kante and (naehester_punkt_index == None
    ↪ or (kanten[pfad_indexe[l]][n] + kanten[pfad_indexe[l+1]][n]) < (kanten[pfad_indexe
    ↪ [l]][naehester_punkt_index] + kanten[pfad_indexe[l+1]][naehester_punkt_index]))
    ↪ and kanten[pfad_indexe[l]][n] < max_abstand:
                            naehester_punkt_index = n

                    if naehester_punkt_index != None:
                        zu_pruefen[naehester_punkt_index] = False

                        if winkelBerechnen(pfad_indexe[l], naehester_punkt_index,
    ↪ pfad_indexe[l+1]) >= 90 and ((l == 0 and (len(pfad_indexe) == 2 or winkelBerechnen
    ↪ (naehester_punkt_index, pfad_indexe[1], pfad_indexe[2]) >= 90)) or (l != 0 and ((l
    ↪  == len(pfad_indexe)-2 and winkelBerechnen(pfad_indexe[l-1], pfad_indexe[l],
    ↪ naehester_punkt_index) >= 90) or (winkelBerechnen(pfad_indexe[l-1], pfad_indexe[l
    ↪ ], naehester_punkt_index) >= 90 and winkelBerechnen(naehester_punkt_index,
    ↪ pfad_indexe[l+1], pfad_indexe[l+2]) >= 90)))):
                            kuerzeste_kante = (kanten[l][naehester_punkt_index] + kanten[
    ↪ l+1][naehester_punkt_index]) - kanten[pfad_indexe[l]][l+1]
                            kuerzeste_kante_index = naehester_punkt_index
                            an = l
                            aenderung = True
                    else:
                        break

            if aenderung != False:
                if an == "anfang":
                    pfad_indexe.insert(0, kuerzeste_kante_index)
                    pfad_koordinaten.insert(0, knoten[kuerzeste_kante_index])
                    uebrig[kuerzeste_kante_index] = False
                elif an == len(pfad_indexe)-1:
                    pfad_indexe.append(kuerzeste_kante_index)
                    pfad_koordinaten.append(knoten[kuerzeste_kante_index])
                    uebrig[kuerzeste_kante_index] = False
                else:
                    pfad_indexe.insert(an+1, kuerzeste_kante_index)
                    pfad_koordinaten.insert(an+1, knoten[kuerzeste_kante_index])
                    uebrig[kuerzeste_kante_index] = False

            if aenderung == False:
                break

def primsAlgorithmus(kanten, knoten):
```

```
537     mst_laenge =  0
        uebrig = len(knoten) * [True]
539     uebrig[0] = False
        mst_enthalten = [0]
541
        while True:
543
            if len(mst_enthalten) == len(knoten):
545             return mst_laenge

547         naechster_knoten = None
            kuerzeste_entfernung = maxsize
549
            for j in range(len(uebrig)):
551             if uebrig[j] == False:
                    continue
553             for k in mst_enthalten:
                    if kanten[j][k] < kuerzeste_entfernung:
555                     naechster_knoten = j
                        kuerzeste_entfernung = kanten[naechster_knoten][k]
557         mst_laenge += kuerzeste_entfernung
            uebrig[naechster_knoten] = False
559         mst_enthalten.append(naechster_knoten)


561
    def winkelPruefen(pfad, i, j):
563     if (i == 1 or winkelBerechnen(pfad[i-2], pfad[i-1], pfad[i]) >= 90) and
        ↪ winkelBerechnen(pfad[i-1], pfad[i], pfad[i+1]) >= 90 and winkelBerechnen(pfad[j
        ↪ -2], pfad[j-1], pfad[j]) >= 90 and (j == len(pfad)-1 or winkelBerechnen(pfad[j-1],
        ↪  pfad[j], pfad[j+1]) >= 90):
            return True
565     else:
            return False
567
    def twoOpt(pfad_indexe, pfad_koordinaten, kanten, boolsche):
569     while True:
            verbessert = False
571         for i in range(1, len(pfad_indexe)-2):
                if verbessert == True:
573                 break
                for j in range(i+1, len(pfad_indexe)):
575                 if j-i == 1:
                        continue
577                 else:
                        neuer_pfad_indexe = pfad_indexe[:]
579                     neuer_pfad_koordinaten = pfad_koordinaten[:]
                        neuer_pfad_indexe[i:j] = pfad_indexe[j-1:i-1:-1]
581                     neuer_pfad_koordinaten[i:j] = pfad_koordinaten[j-1:i-1:-1]
                        if winkelPruefen(neuer_pfad_indexe, i, j) == True and ((kanten[
        ↪ pfad_indexe[i]][pfad_indexe[i-1]] + kanten[pfad_indexe[j]][pfad_indexe[j-1]]) > (
        ↪ kanten[neuer_pfad_indexe[i]][neuer_pfad_indexe[i-1]] + kanten[neuer_pfad_indexe[j
        ↪ ]][neuer_pfad_indexe[j-1]])):
583                         pfad_indexe = neuer_pfad_indexe
                            pfad_koordinaten = neuer_pfad_koordinaten
585                         verbessert = True
                            break
587         if verbessert == False:
                break
589     if boolsche == "ja":
            turtle(pfad_koordinaten, True, "green")
591     return (pfad_koordinaten, pfad_indexe)

593 knoten_liste = []
    kanten_liste = []
595


597
    welches_verfahren = input("Mit welchem Verfahren (Auswahl: nn, ni, fi, nilm, film, ri,
        ↪ nni, mst): ")
599 welche_datei = input("Welche Datei soll ausgelesen werden (z.B. wenigerkrumm1.txt): ")
    if welches_verfahren != "mst":
601     mit_oder_ohne_turtle = input("Soll die Route durch Australien visualisiert werden? (
        ↪ ja/nein) ")
```

```
    vorarbeit(welche_datei)
603
    if welches_verfahren == "nn":
605     fertig, pfad_koordinaten, pfad_indexe = nearestNeighbour(kanten_liste, knoten_liste)
        if fertig == True:
607         pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
            print("Die Laenge des Pfades bei Nearest Neighbour:", laenge(kanten_liste,
        ↪ pfad_indexe))
609         print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)

611 elif welches_verfahren == "ni":
        fertig, pfad_koordinaten, pfad_indexe = nearestInsertionHeuristik(kanten_liste,
        ↪ knoten_liste)
613     if fertig == True:
            pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
615         print("Die Laenge des Pfades bei Nearest Insertion:", laenge(kanten_liste,
        ↪ pfad_indexe))
            print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)
617
    elif welches_verfahren == "fi":
619     fertig, pfad_koordinaten, pfad_indexe = farthestInsertionHeuristik(kanten_liste,
        ↪ knoten_liste)
        if fertig == True:
621         pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
            print("Die Laenge des Pfades bei Farthes Insertion:", laenge(kanten_liste,
        ↪ pfad_indexe))
623         print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)

625 elif welches_verfahren == "nilm":
        fertig, pfad_koordinaten, pfad_indexe = nearestInsertionHeuristikMitLokalemMinimum(
        ↪ kanten_liste, knoten_liste)
627     if fertig == True:
            pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
629         print("Die Laenge des Pfades bei Nearest Insertion mit lokalem Minimum:", laenge(
        ↪ kanten_liste, pfad_indexe))
            print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)
631
    elif welches_verfahren == "film":
633     fertig, pfad_koordinaten, pfad_indexe = farthestInsertionHeuristikMitLokalemMaximum(
        ↪ kanten_liste, knoten_liste)
        if fertig == True:
635         pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
            print("Die Laenge des Pfades bei Farthes Insertion mit lokalem Maximum:", laenge(
        ↪ kanten_liste, pfad_indexe))
637         print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)

639 elif welches_verfahren == "ri":
        fertig, pfad_koordinaten, pfad_indexe = randomInsertionHeuristik(kanten_liste,
        ↪ knoten_liste)
641     if fertig == True:
            pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
643         print("Die Laenge des Pfades bei Random Insertion:", laenge(kanten_liste,
        ↪ pfad_indexe))
            print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
        ↪ pfad_koordinaten)
645
    elif welches_verfahren == "nni":
647     fertig, pfad_koordinaten, pfad_indexe =
        ↪ mischungNearestInsertionHeuristikUndNearestNeighbour(kanten_liste, knoten_liste)
        if fertig == True:
649         pfad_koordinaten, pfad_indexe = twoOpt(pfad_indexe, pfad_koordinaten,
        ↪ kanten_liste, mit_oder_ohne_turtle)
```

```
        print("Die Laenge des Pfades bei der Mischung aus Nearest Insertion und Nearest
    ↪ Neighbour:", laenge(kanten_liste, pfad_indexe))
651         print("Die Koordinaten werden dabei in folgenden Reihenfolge abgeflogen:",
    ↪ pfad_koordinaten)

653 elif welches_verfahren == "mst":
    print("Die Laenge des minimum spanning tree:", primsAlgorithmus(kanten_liste,
    ↪ knoten_liste))
```

Anhang B: beispielhafte Probleminstanzen

Jede Probleminstanz beschreibt eine Menge von Punkten in der euklidischen Ebene und enthält in jeder Zeile zwei Zahlen x und y : die Positionskoordinaten eines Ortes, mit km als Maßeinheit [4].

Anhang B.1: Beispieleingabe wenigerkrumm1

```
200.000000 0.000000
150.000000 0.000000
210.000000 30.000000
80.000000 30.000000
190.000000 0.000000
360.000000 0.000000
100.000000 0.000000
40.000000 30.000000
370.000000 30.000000
340.000000 0.000000
120.000000 0.000000
200.000000 30.000000
110.000000 30.000000
170.000000 0.000000
40.000000 0.000000
270.000000 0.000000
50.000000 30.000000
260.000000 30.000000
390.000000 0.000000
10.000000 30.000000
0.000000 0.000000
140.000000 0.000000
220.000000 30.000000
270.000000 30.000000
100.000000 30.000000
180.000000 0.000000
400.000000 0.000000
80.000000 0.000000
260.000000 0.000000
230.000000 30.000000
90.000000 30.000000
0.000000 30.000000
240.000000 30.000000
380.000000 0.000000
330.000000 0.000000
240.000000 0.000000
30.000000 0.000000
210.000000 0.000000
220.000000 0.000000
290.000000 0.000000
90.000000 0.000000
150.000000 30.000000
20.000000 30.000000
60.000000 0.000000
60.000000 30.000000
140.000000 30.000000
70.000000 30.000000
-5.000000 15.000000
290.000000 30.000000
250.000000 30.000000
405.000000 15.000000
250.000000 0.000000
400.000000 30.000000
300.000000 0.000000
280.000000 0.000000
310.000000 0.000000
350.000000 0.000000
280.000000 30.000000
180.000000 30.000000
390.000000 30.000000
310.000000 30.000000
360.000000 30.000000
20.000000 0.000000
320.000000 0.000000
130.000000 30.000000
120.000000 30.000000
160.000000 0.000000
190.000000 30.000000
330.000000 30.000000
350.000000 30.000000
370.000000 0.000000
380.000000 30.000000
70.000000 0.000000
30.000000 30.000000
110.000000 0.000000
10.000000 0.000000
170.000000 30.000000
300.000000 30.000000
160.000000 30.000000
230.000000 0.000000
320.000000 30.000000
50.000000 0.000000
340.000000 30.000000
130.000000 0.000000
```

Anhang B.2: Beispieleingabe wenigerkrumm2

```
81.347329 182.709092
81.347329 -182.709092
88.167788 -121.352549
-117.557050 -161.803399
-41.582338 -195.629520
149.178284 -15.679269
-111.471724 -100.369591
-173.205081 -100.000000
-129.903811 -75.000000
190.211303 61.803399
148.628965 133.826121
-41.582338 195.629520
-117.557050 161.803399
0.000000 -200.000000
-111.471724 100.369591
198.904379 20.905693
0.000000 150.000000
-31.186754 146.722140
129.903811 75.000000
-148.628965 133.826121
-149.178284 15.679269
-148.628965 -133.826121
-81.347329 182.709092
-129.903811 75.000000
173.205081 -100.000000
-198.904379 -20.905693
-88.167788 -121.352549
-142.658477 46.352549
111.471724 -100.369591
142.658477 46.352549
```

```
61.010496 137.031819        142.658477 -46.352549       -61.010496 -137.031819
31.186754 146.722140        117.557050 161.803399       -190.211303 -61.803399
61.010496 -137.031819       -149.178284 -15.679269      -88.167788 121.352549
31.186754 -146.722140       -61.010496 137.031819       -31.186754 -146.722140
88.167788 121.352549        41.582338 -195.629520       -198.904379 20.905693
-81.347329 -182.709092      -173.205081 100.000000      190.211303 -61.803399
149.178284 15.679269        173.205081 100.000000       129.903811 -75.000000
0.000000 200.000000         148.628965 -133.826121      41.582338 195.629520
111.471724 100.369591       -190.211303 61.803399       0.000000 -150.000000
-142.658477 -46.352549      117.557050 -161.803399      198.904379 -20.905693
```

Anhang B.3: Beispieleingabe wenigerkrumm3

```
169.282032 140.000000       20.438248 -8.362277         -69.282032 140.000000
179.561752 -8.362277        59.451586 -53.530449        -16.632935 78.251808
16.632935 -78.251808        169.282032 40.000000        179.561752 91.637723
176.084521 124.721360       32.538931 73.083637         69.282032 60.000000
-32.538931 73.083637        59.451586 53.530449         59.451586 153.530449
76.084521 75.278640         -32.538931 -73.083637       179.561752 108.362277
32.538931 173.083637        159.451586 -53.530449       47.022820 35.278640
-79.561752 108.362277       52.977180 164.721360        83.367065 178.251808
132.538931 73.083637        -32.538931 26.916363        69.282032 140.000000
79.561752 8.362277          -69.282032 -40.000000       -76.084521 124.721360
59.451586 46.469551         67.461069 173.083637        -16.632935 178.251808
40.548414 53.530449         83.367065 21.748192         52.977180 -64.721360
23.915479 -24.721360        30.717968 60.000000         -47.022820 64.721360
83.367065 78.251808         169.282032 -40.000000       -69.282032 60.000000
176.084521 24.721360        52.977180 64.721360         16.632935 78.251808
116.632935 21.748192        20.438248 108.362277        100.000000 80.000000
23.915479 24.721360         100.000000 180.000000       -47.022820 -64.721360
-59.451586 53.530449        147.022820 164.721360       176.084521 -24.721360
0.000000 180.000000         147.022820 -64.721360       -76.084521 75.278640
-47.022820 35.278640        132.538931 -73.083637       40.548414 -53.530449
-79.561752 8.362277         67.461069 26.916363         47.022820 64.721360
69.282032 40.000000         67.461069 73.083637         -79.561752 91.637723
83.367065 -78.251808        23.915479 124.721360        159.451586 46.469551
16.632935 21.748192         32.538931 -73.083637        30.717968 40.000000
116.632935 178.251808       116.632935 78.251808        -32.538931 173.083637
-79.561752 -8.362277        79.561752 -8.362277         -69.282032 40.000000
76.084521 24.721360         -47.022820 164.721360       16.632935 178.251808
159.451586 53.530449        100.000000 20.000000        100.000000 -80.000000
79.561752 91.637723         20.438248 8.362277          116.632935 -78.251808
20.438248 91.637723         40.548414 153.530449        -76.084521 24.721360
-59.451586 153.530449       147.022820 64.721360        179.561752 8.362277
0.000000 80.000000          40.548414 46.469551         47.022820 164.721360
147.022820 35.278640        -16.632935 21.748192        79.561752 108.362277
0.000000 -80.000000         32.538931 26.916363         132.538931 26.916363
159.451586 153.530449       -76.084521 -24.721360       176.084521 75.278640
-59.451586 46.469551        69.282032 -40.000000        76.084521 -24.721360
0.000000 20.000000          23.915479 75.278640         76.084521 124.721360
-16.632935 -78.251808       30.717968 140.000000        -59.451586 -53.530449
52.977180 35.278640         169.282032 60.000000        132.538931 173.083637
67.461069 -73.083637        47.022820 -64.721360        30.717968 -40.000000
```

Anhang B.4: Beispieleingabe wenigerkrumm4

```
20.212169 156.013261        144.832862 -43.476284       -98.760442 -81.770618
-82.864121 -104.173600      -20.971208 -5.637107        -119.026308 168.453598
153.130159 -20.360910       -239.414022 40.427118       51.008140 5.769601
-129.104485 -155.041640     -191.716829 -28.360492      42.137753 -60.319863
101.498782 33.484198        28.913721 58.699880         -240.369194 57.426131
-16.723130 -12.689542       -137.317503 -20.146939      -221.149792 -32.862538
-239.848226 8.671399        94.789917 -67.087689        33.379688 100.161238
-154.088455 115.022553      -107.988514 185.173669
139.446709 0.233238         -219.148505 103.685337
```

Anhang B.5: Beispieleingabe wenigerkrumm5

```
-281.678990 -187.717923     -278.409792 -111.914073     -57.266232 -115.737582
239.639550 79.491132        141.513053 2.821137         -286.024059 -55.955204
-258.868593 166.669198      -49.447381 173.210759       -214.362324 -193.265190
-81.384378 32.368323        -68.446198 137.178953       92.639946 22.216030
-280.008136 11.657786       -82.173510 119.465553       -116.831788 -191.380552
-284.547616 -60.154961      -136.787038 79.501703       -251.656688 -195.189953
-31.548604 -55.223912       63.541591 55.140221         -235.099412 47.810306
62.366656 50.713913         106.033430 69.754891        -177.685937 158.265884
-95.621797 77.468533        47.040512 141.206562        -260.477802 -196.955535
162.493244 -84.574019       30.366828 -167.573232       253.534863 38.014987
-147.023475 -166.220130     -234.711279 -162.774591     -44.669924 170.088013
209.544977 94.267052        25.098172 35.205544         26.451074 -192.813352
-58.684205 -76.988884       116.702667 132.021991       45.123674 31.740242
-74.639411 25.542881        -30.991436 186.807059       38.654730 188.608557
142.765554 -118.682439      -70.183535 73.738342        244.228552 119.192512
51.417675 146.417721        -202.218443 -178.735864     283.989938 -101.866465
263.236651 -144.293091      -223.039999 171.558368      -247.341131 -160.277639
-8.936916 13.543851         -104.781549 158.212048      267.845908 127.627482
90.584569 -164.218416       27.706327 169.284192        36.599805 147.885350
-86.457580 105.836348       171.595574 135.520994       -41.263039 -144.118212
```

Anhang B.6: Beispieleingabe wenigerkrumm6

```
102.909291 60.107868        -194.986965 101.363745      -4.590656 -40.067226
-89.453831 162.237392       152.102728 -193.381252      -191.216327 -162.689024
64.943433 -119.784474       238.583388 -133.143524      210.186432 -127.403563
121.392544 56.694081        151.432196 121.427337       -51.343758 -57.654823
-107.196865 -77.792599      221.028639 -139.435079      187.669263 -122.655771
20.218290 88.031173         -139.741580 57.936680       121.661135 85.267672
202.346980 -189.069699      -72.565291 -24.281820       46.674278 -193.090008
143.114152 -135.866707      155.405344 -56.437901       -189.988471 -98.043874
-144.887799 -73.495410      58.019653 49.937906         -175.118239 77.842636
92.255820 -93.514104        277.821597 104.262606       -187.485329 -177.031237
-55.091518 198.826966       19.765322 -99.236400        56.716498 66.959624
228.929427 82.624982        246.621634 101.705861       -18.507391 -22.905270
96.781707 141.370805        289.298882 56.051342        -167.994506 138.195365
154.870684 140.327660       172.836936 59.184233        81.740403 10.276251
112.833346 -38.057607       132.794476 135.681392       -19.310012 -131.810965
14.005617 -14.015334        155.341949 -20.252779       157.588994 -144.200765
138.136997 -31.348808       134.692592 -102.152826      40.327635 19.216022
73.689751 110.224271        -97.391662 124.120512       -126.569816 -30.645224
100.006932 76.579303        245.415055 44.794067        150.526118 -88.230057
120.906436 131.798810       255.134924 115.594915       76.647124 -7.289705
21.067444 122.164599        83.005905 64.646774         231.944823 82.961057
49.091876 150.678826        245.020791 -167.448848      58.716620 32.835930
85.043830 108.946389        -102.699992 95.632069       -288.744132 -173.349893
```

```
-293.833463 -165.440105     21.176627 -165.421555     -154.225945 -135.522059
-31.745416 -69.207960       -100.569041 140.808607    102.223372 174.201904
175.677917 98.929343        -90.160190 -25.200829     64.559003 82.567627
216.825920 -152.024123      242.810288 -182.054289
```

Anhang B.7: Beispieleingabe wenigerkrumm7

```
-47.266557 -66.984045       172.389228 -53.133270     275.793495 -129.415477
-185.649161 90.144456       -248.169463 80.132237     106.599423 -107.433987
34.079032 187.731112        -153.130140 187.817274    55.550895 -45.089968
170.514252 161.169850       278.105364 -93.771765     -189.062172 104.285631
56.389778 71.618509         217.218893 164.294928     -157.423365 126.800331
-56.914543 92.501249        120.375033 115.889661     -192.681053 174.522947
-18.316063 27.755860        -133.730932 -113.306155   189.387028 -4.465225
118.989764 -80.203583       -147.363185 59.608175     -48.354421 9.091412
271.301094 142.524086       -171.354954 25.463068     -211.429137 27.770425
-222.492322 169.033315      54.766523 154.053847      -201.485143 155.274830
169.990437 154.260412       8.643660 135.907430       -155.651746 -138.151811
235.827007 -143.838844      -114.146166 190.615321    53.436521 125.683201
55.434792 62.729160         276.276517 -49.448662     3.152113 27.103890
221.808162 186.241012       239.616628 -21.944160     296.911892 25.811569
-9.580869 -17.516639        148.108328 123.558283     -215.113949 120.740679
126.799911 112.727280       -17.356579 -125.254131    205.717887 -24.976511
100.043893 161.295125       54.551865 71.567133       -134.985023 132.944989
158.552316 -19.254407       -184.092700 5.737284      -126.568914 106.964962
-172.378071 -88.298187      16.573231 104.020979      -189.135201 -139.078513
-202.828627 -101.700050     129.024315 29.701695      256.475967 -46.591418
-200.771246 147.741054      217.599278 -189.258062    -284.129027 107.252583
-244.959501 -111.046573     92.298040 -146.169487     -240.249363 179.334919
34.959132 -106.842499       88.818853 -42.834512      224.599361 -34.798485
40.897139 78.152317         -42.704691 37.679514      130.854855 195.695082
178.198360 37.031984        -217.282470 -43.316616    68.910854 -82.123346
-120.386562 170.589454      -226.787625 2.658862      -4.434919 33.164884
-84.626900 148.216494       158.742184 62.618834      -27.911955 48.326745
162.923138 117.465744       -46.403062 -13.755804     135.781192 -13.053440
47.011363 -30.887180        -181.208895 -192.622935   126.904044 -80.733297
95.947213 183.278211        -152.130365 -93.844349    57.555364 64.417343
59.827200 -170.713714       74.887500 80.586458       141.433472 -6.023095
216.691000 156.314370       -13.030259 174.698005     -268.739142 143.276483
-287.058297 113.599823      -207.665172 81.410371
-0.200936 -21.927663        208.592696 136.618460
```

Anhang C: Lösungen beispielhafter Probleminstanzen

Anhang C.1: Beispielausgabe wenigerkrumm1

Die Länge des Pfades bei Nearest Neighbour: 853.2455519999999
Die Länge des Pfades bei Nearest Insertion: 847.434164
Die Länge des Pfades bei Farthes Insertion: 1304.868329
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 1279.2199150000001
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 1195.51303
Die Länge des Pfades bei Random Insertion: 1584.528913
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 847.434164

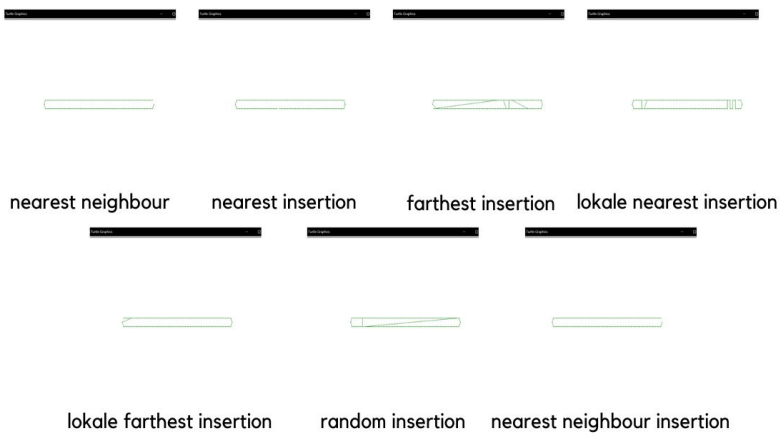

Abbildung 11: Visualisierung der Routen

Anhang C.2: Beispielausgabe wenigerkrumm2
Die Länge des Pfades bei Nearest Neighbour: 2183.662267
Die Länge des Pfades bei Nearest Insertion: 2674.965059999999
Die Länge des Pfades bei Farthes Insertion: 4041.215346999998
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 4348.126345999997
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 4095.5894619999963
Die Länge des Pfades bei Random Insertion: 3777.6229539999986
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 2183.6622669999997

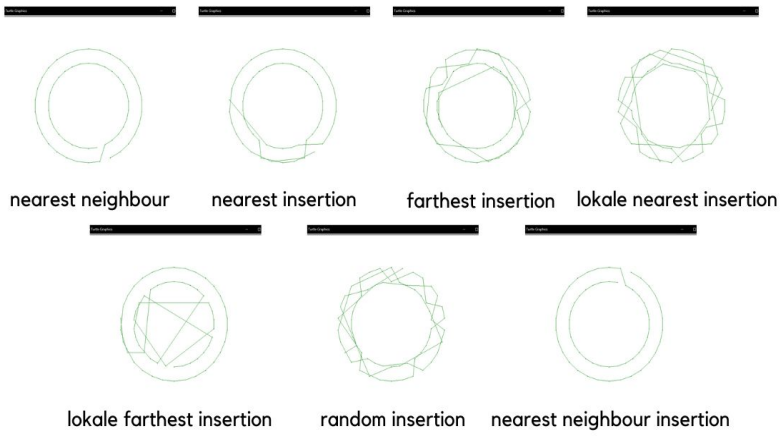

Abbildung 12: Visualisierung der Routen

Anhang C.3: Beispielausgabe wenigerkrumm3
Die Länge des Pfades bei Nearest Neighbour: 1968.1565759999964
Die Länge des Pfades bei Nearest Insertion: 3041.565679999997
Die Länge des Pfades bei Farthes Insertion: -
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 3307.295460999998
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 3010.037686999995
Die Länge des Pfades bei Random Insertion: 3465.613758999994
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 1998.5807149999978

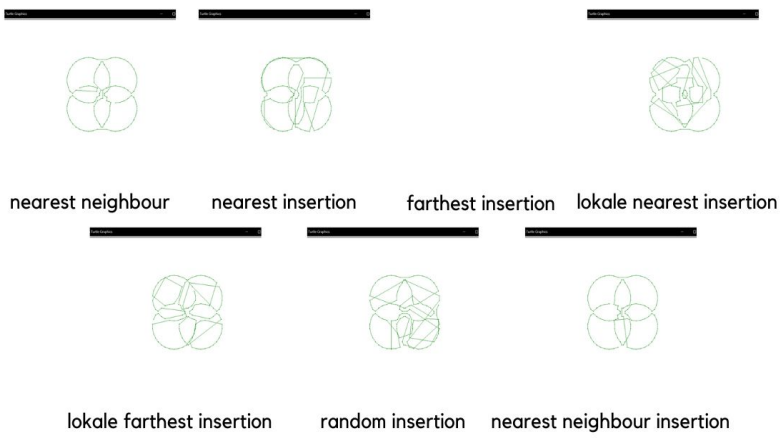

Abbildung 13: Visualisierung der Routen

Anhang C.4: Beispielausgabe wenigerkrumm4
Die Länge des Pfades bei Nearest Neighbour: 1243.4096730000003
Die Länge des Pfades bei Nearest Insertion: 1499.4978319999998
Die Länge des Pfades bei Farthes Insertion: 1243.409673
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 2121.351468
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 2415.3189849999994
Die Länge des Pfades bei Random Insertion: 1507.535052
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 1205.0685549999998

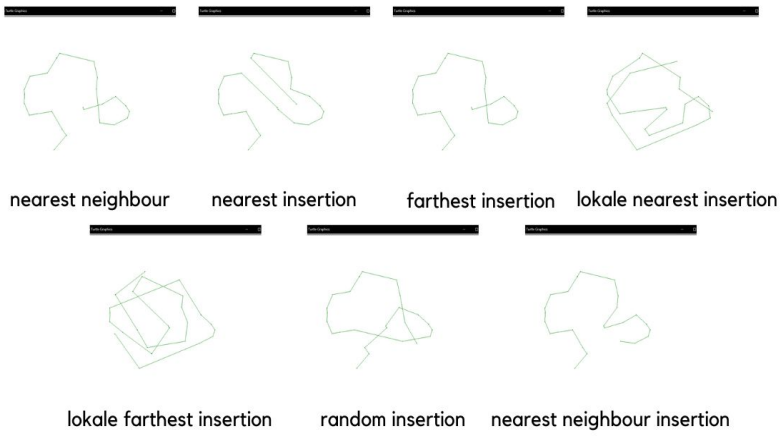

Abbildung 14: Visualisierung der Routen

Anhang C.5: Beispielausgabe wenigerkrumm5
Die Länge des Pfades bei Nearest Neighbour: 3838.6755900000003
Die Länge des Pfades bei Nearest Insertion: 4494.625033
Die Länge des Pfades bei Farthes Insertion: 4658.475449
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 4937.011791000002
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 3922.802687000001
Die Länge des Pfades bei Random Insertion: 4523.775915000001
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 3700.1659120000004

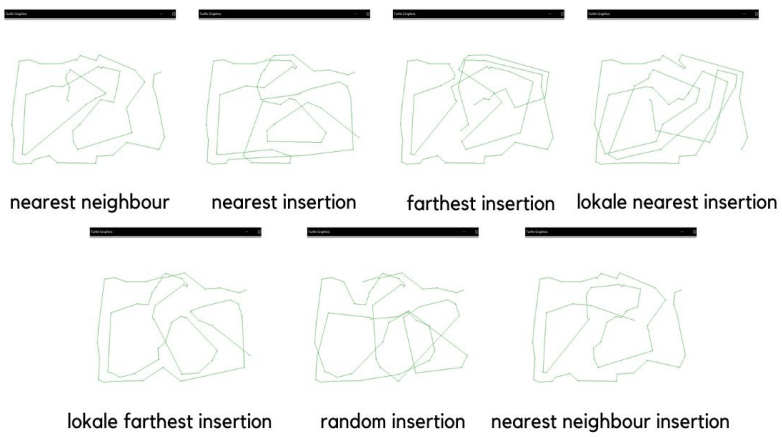

Abbildung 15: Visualisierung der Routen

Anhang C.6: Beispielausgabe wenigerkrumm6
Die Länge des Pfades bei Nearest Neighbour: -
Die Länge des Pfades bei Nearest Insertion: 4408.986957
Die Länge des Pfades bei Farthes Insertion: 4796.751525
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 4264.031631000002
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 4716.2226329999985
Die Länge des Pfades bei Random Insertion: 4273.251918000001
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 4400.3511340000005

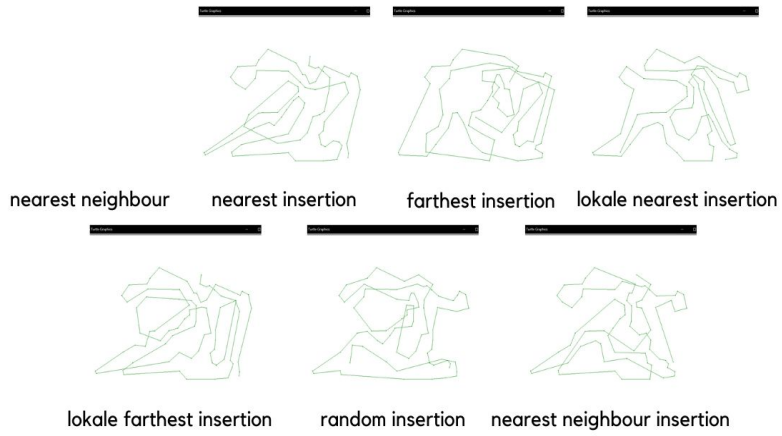

Abbildung 16: Visualisierung der Routen

Anhang C.7: Beispielausgabe wenigerkrumm7
Die Länge des Pfades bei Nearest Neighbour: -
Die Länge des Pfades bei Nearest Insertion: 6027.920091999997
Die Länge des Pfades bei Farthes Insertion: 5829.869839000001
Die Länge des Pfades bei Nearest Insertion mit lokalem Minimum: 6211.509133999997
Die Länge des Pfades bei Farthes Insertion mit lokalem Maximum: 6731.997896999999
Die Länge des Pfades bei Random Insertion: 5610.655477999999
Die Länge des Pfades bei der Mischung aus Nearest Insertion und Nearest Neighbour: 4782.190101999999

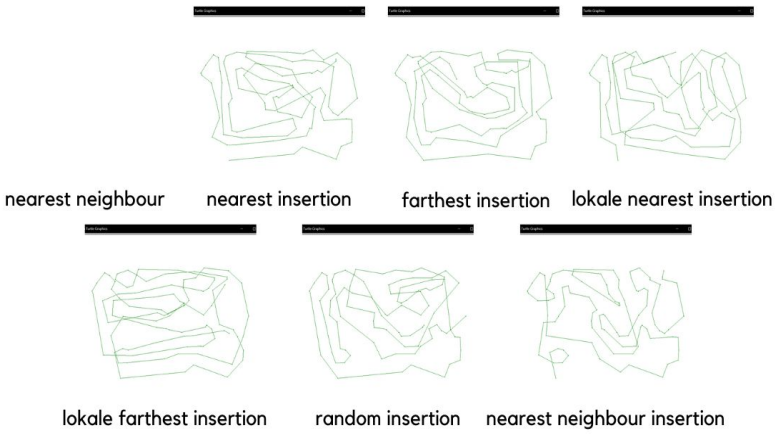

Abbildung 17: Visualisierung der Routen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel

selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und die den verwendeten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort, Datum: _____

Unterschrift: _____