Bearbeitung: Christoph Barth
Team-ID: : 00461
Team-Name: CB

Bundeswettbewerb Informatik
Runde 2

Aufgabe 1 - Weniger krumme Touren

Inhaltsverzeichnis

1 Lösungsidee **2**
1.1 Das TSP_{90} Problem . . . 2
1.2 Algorithmisierung . . . 2
1.3 Distanz . . . 3
1.4 Komplexität . . . 3
1.5 Beweis der **NP**-Vollständigkeit . . . 4

2 Umsetzung **6**

3 Beispiele **7**

4 Optimierung **7**
4.1 Laufzeit . . . 7
4.1.1 Memoization . . . 8
4.1.2 Hamiltonkreismethode . . . 8
4.1.3 Variierender Startpunkt . . . 9
4.1.4 Cheapest-Insertion Heuristik . . . 10
4.2 Distanz . . . 10

5 Entscheidungsproblem **11**
5.1 Konvexe-Hülle . . . 12
5.2 Generalisierung . . . 13

6 Anhang **14**

7 Code **14**

1 Lösungsidee

1.1 Das TSP_{90} Problem

Die Aufgabe verlangt, einen Pfad H durch jeden Punkt P_i einer Punktemenge P zu finden. H ist also eine Permutation von P, wobei der Punkt H_i an der i-ten Stelle des Pfades steht. Hierbei darf H keine Abbiegewinkel $\alpha > 90°$ enthalten. Der Abbiegewinkel α ist hierbei der Außenwinkel von jedem Tripel aus in H aufeinanderfolgenden Punkten. Die Berechnung von α ist über Vektoren, die die Punkte des Tripels verbinden, möglich:

$$\alpha = arccos(\frac{\overrightarrow{P_iP_{i+1}} * \overrightarrow{P_{i+1}P_{i+2}}}{|\overrightarrow{P_iP_{i+1}}| * |\overrightarrow{P_{i+1}P_{i+2}}|}) \leq 90° \quad (1)$$

Weiterhin soll der Pfad möglichst kurz sein, muss aber nicht der kürzeste sein. Wäre dies der Fall, wäre die Lösung eine Lösung des Travelling-Salesman-Problems und somit unter der Annahme $P \neq NP$ nur durch das Ausprobieren jedes möglichen Pfades möglich. Für n Punkte gibt es $(n-1)!$ verschieden Pfade. Diese Zahl übersteigt die Zahl an Teilchen im Universum bereits für $n = 61$ Die Beschränkung des Winkels wurde hier außer Acht gelassen, da dies die Größenordnung der Komplexität des Problems nicht ändert (Beweis in Kapitel 1.5). Die Lösung des Problems wäre somit bereits für relativ kleine n praktisch unmöglich.

Das Problem, den beschriebenen Pfad zu finden, besteht also daraus, **1.** einen Pfad durch alle Punkte einer Punktemenge P zu finden und dabei alle $\alpha > 90°$ zu vermeiden und **2.** diesen Pfad möglichst kurz zu gestalten. Im Folgenden wird das Problem TSP_{90} genannt.

1.2 Algorithmisierung

Ein geeigneter erster Ansatz für die Suche nach algorithmischen Lösungen sind bereits etablierte Algorithmen. Da hier in einem Graphen gesucht wird, bieten die Breiten (BFS)- und Tiefensuche (DFS) einen guten Startpunkt.

BFS beginnt die Suche bei einem gegebenen Startknoten und untersucht dann alle unbesuchten Nachbarn des Startknotens in einer Breite-zuerst-Reihenfolge, bevor es sich tiefer in den Graphen begibt. Um so einen Pfad durch alle Punkte aus P zu finden, müsste also jeder mögliche Pfad betrachtet werden, was, wie bereits erläutert, ungeeignet ist.

DFS erkundet hingegen jeden Suchzweig so lange wie möglich. Führt dieser in eine Sackgasse, verzweigt die Suche am letzten Entscheidungspunkt und wird von dort in gleicher Weise fortgesetzt. Dieses Umentscheiden nennt man Backtracking. Da jeder Pfad so zunächst vollständig erkundet wird, bevor andere Pfade betrachtet werden, ermöglicht dieses Verfahren das schnelle Finden einer Lösung.

Um DFS auf das TSP_{90} Problem anzuwenden, wird zunächst ein zufälliger Startpunkt H_0 gewählt, dieser aus P entfernt und dem Pfad H hinzugefügt. Jetzt wird ein Folgepunkt aus P gewählt, ebenfalls entfernt und H zugefügt. Beim weiteren Bestimmen von Folgepunkten muss darauf geachtet werden, dass der entstehende Abbiegewinkel in H 90° nicht überschreitet. Es werden also nur die Punkte als potentielle Folgepunkte angesehen, für die diese Bedingung gilt. Alle weiteren Folgepunkte werden mit demselben Verfahren bestimmt. Ist es an einem Punkt nicht möglich, weitere

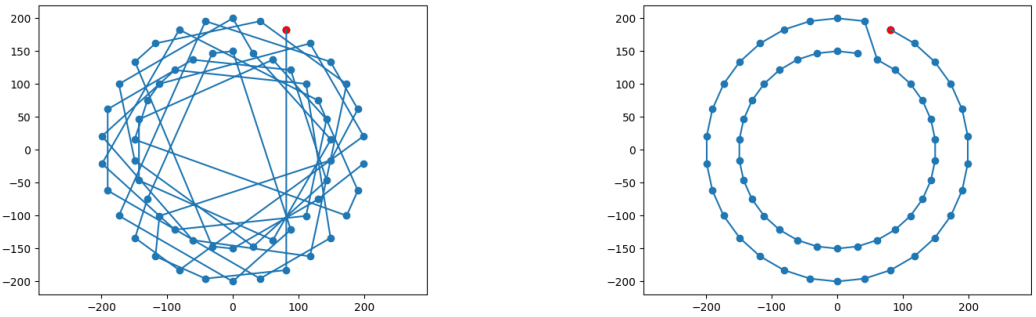


Abbildung 1: Pfade mit 1b und ohne 1a Nearest-Neighbour Heuristik

Folgepunkte zu bestimmen, obwohl P noch nicht leer ist, muss der Algorithmus backtracken. Hierfür wird der H zuletzt hinzugefügte Punkt H_{-1} wieder entfernt und ein anderer Folgepunkt für H_{-2} ausgewählt. Gelingt es, mit diesem Verfahren einen Pfad H zu konstruieren, sodass P keine Punkte mehr enthält, ist H eine Lösung des TSP_{90}-Problems. Für den Fall, dass keine weitere Möglichkeit der Umentscheidung mehr besteht, wurden alle möglichen Pfade H ausprobiert, ohne, dass eine Lösung gefunden wurde. Das TSP_{90} Problem ist in diesem Fall also nicht lösbar.

1.3 Distanz

Die so konstruierten Pfade H sind zwar technisch gesehen korrekte Lösungen des TSP_{90} Problems, erfüllen aber noch nicht die Anforderung der möglichst geringen Distanz. Um auch dieser Anforderung gerecht zu werden, kann die Distanz aller möglichen Folgepunkte als Heuristik verwendet werden, um näher gelegene Punkte zu bevorzugen.
Das bedeutet, dass der Algorithmus immer den Folgepunkt wählt, der die geringste Distanz zum letzten Punkt aus H hat. Die so entstandenen Pfade sind in der Regel kürzer, garantieren aber kein globales Optimum, weil der Algorithmus sich immer nur am lokalen Optimum orientiert und die Distanz zwischen Start- und Endpunkt nicht beachtet. Trotzdem sorgt die Nearest-Neighbour Heuristik bereits für signifikante Verbesserungen (Abb. 1).

Weitere Heuristiken werden im Kapitel **??** betrachtet.

1.4 Komplexität

In einem best-case Szenario ist der entworfene Algorithmus in der Lage, den Pfad immer mit dem ersten Folgepunkt zu. konstruieren. In diesem Fall beträgt die Laufzeit $\mathcal{O}(n * n) = (n^2)$, da für n Punkte in $\mathcal{O}(n)$ der nächste Folgepunkt bestimmt wird (Abb. 9).
Im schlimmsten Fall existiert kein Pfad H und es wird jeder mögliche Pfad ausprobiert. Hierfür gibt es in der Größenordnung von $\mathcal{O}(n!)$ Möglichkeiten. Da vom Worst-Case ausgegangen werden sollte, verhält sich die Laufzeit also fakultativ zur Eingabegröße n.
Dies ist sehr ineffizient und eigentlich inakzeptabel, kann bei manchen Problemen aber nicht vermieden werden. Dies ist der Fall, wenn ein Problem $\in$ **NP** oder **NP**-schwer ist. Ein Problem heißt **NP**-vollständig, wenn es sowohl in **NP** liegt und **NP**-schwer ist. Im Folgenden wird gezeigt, dass das TSP_{90} Problem **NP**-vollständig ist.

1.5 Beweis der NP-Vollständigkeit

Für den Beweis der **NP**-Vollständigkeit ist zunächst eine formale Definition des TSP_{90} Problems notwendig:

Um in einer Menge von Punkten nach einem Pfad zu Suchen, bietet sich ein Graph als bestehende mathematische Struktur an. Ein Graph $G = (V, E)$ besteht aus Knoten V und Kanten E, wobei eine Kante eine Verbindung zwischen zwei Knoten ist.
Die Punkte aus P werden als Knoten aufgefasst. Da es theoretisch möglich ist, von jedem Punkt in P zu jedem anderen Punkt in P zu kommen, exisitiert zwischen jedem Paar unterschiedlicher Knoten eine Kante. G heißt dann vollständiger Graph. Ein Pfad in einem Graphen, der jeden Knoten genau einmal besucht, nennt man Hamilton-Pfad.

Die Kodierung des Graphen durch Knoten und Kanten ist in diesem Zusammenhang allerdings ungeeignet, da sich die Winkelbedingung hier schlecht integrieren lässt. Geeigneter ist die Kodierung des Graphen durch eine Adjazentenmatrix A. Um A zu bilden, werden alle Knoten in G von 1 bis n durchnummeriert. Dann ist A eine $n \times n$ Matrix, wobei jeder Eintrag $A_{ij} \in \{0; 1\}$ dafür kodiert, ob eine Kante zwischen Knoten i und j existiert.
Erweitert man A um eine Dimenstion auf eine $n \times n \times n$ Matrix, kodiert jeder Eintrag M_{ijk} der Matrix dafür, ob eine Kante zwischen Knoten j und k existiert, wenn der Pfad vorher in Knoten i war.
A wird konstruiert mit

$$A_{ijk} = \begin{cases} 1 & \text{wenn} \quad \angle P_i P_j P_k \leq 90° \\ 0 & \text{sonst} \end{cases} \tag{2}$$

So wird jedem Tripel von Punkten $(P_i P_j P_k)$, welches einen Abbiegewinkel $\alpha \leq 90°$ hat, in A der Wert 1 zugeordnet. Ein Pfad H, der jeden Punkt der Eingabliste P einmal besucht ist dann eine Permutation der Zahlen $[1 : n]$. Die Länge des Pfades ist

$$|H| = \sum_{i=1}^{n-2} A_{i,i+1,i+2} \tag{3}$$

Gilt $|H| = n - 2$, so verwendet der Pfad, der durch H kodiert wird, nur Tripel deren Abbiegewinkel $\alpha \leq 90°$ ist.

Das Problem kann somit definiert werden als $TSP_{90} := \{A = \{0; 1\}^{n \times n \times n} | \text{Permutation } H \text{ mit } |H| = n - 2\}$

- $TSP_{90} \in$ **NP** gilt, wenn es möglich ist, eine Lösung des Problems zu raten und diese in polynomieller Zeit zu überprüfen. Hierzu kann eine zufällige Permutation H von P geraten werden. Anschließend kann $|H|$ mit der Matrix A gebildet werden. Gilt $|H| = 0$, so ist die Lösung gültig.
  $|H|$ kann durch Iteration über die Einträge $[3 : n]$ von H in linearer Zeit bestimmt werden. A kann in kubischer Zeit besimmt werden. Das Überprüfen ist also in polynomieller Zeit möglich.

- Um die **NP**-Schwere des TSP_{90}-Problems zu zeigen, kann man eine Reduktion von einem bekannten **NP**-schweren Problem auf TSP_{90} konstruieren, indem man eine Funktion definiert, die eine Instanz des **NP**-schweren Problems in polynomieller Zeit auf eine Instanz des TSP_{90}-Problems abbildet. Ein effizientes Lösungsverfahren für TSP_{90} könnte dann genutzt werden,

um die Abbildung des **NP**-schweren Problems effizient zu Lösen. Da die effiziente Lösung des NP-schweren Problems unter der Annahme $P \neq NP$ nicht möglich ist, kann es dann auch für TSP_{90} kein effizientes Lösungsverfahren geben.

Aufgrund der hohen Ähnlichkeit der Probleme TSP_{90} und **HAMILTON** wird die **NP**-Schwere durch die Reduktion von **HAMILTON** auf TSP_{90} gezeigt.
Das Problem **HAMILTON** nimmt das Encoding $G = (V, E)$ eines Graphen als Eingabe und fragt nach einem Hamilton-Pfad. Anders als beim TSP_{90} Problem muss G hier kein vollständiger Graph sein. In diesem Fall wäre die Lösung trivial. In **??** wird die NP-Schwere des Problems bewiesen.

Zunächst muss eine Reduktionsfunktion f bestimmt werden, die eine Eingabe $G = (V, E)$ des **HAMILTON**-Problems in eine Eingabe A des TSP_{90}-Problems umwandelt. Nach dem Durchnummerieren der Knoten in G wird A gebildet mit:

$$A_{ijk} = \begin{cases} 1 & \text{wenn } \{V_i V_j\} \in E \\ 0 & \text{sonst} \end{cases} \tag{4}$$

Also sind die Kosten, um von Knoten i nach Knoten j zu kommen 1, wenn die Kante $\{V_i V_j\}$ in G existiert und 0, wenn nicht. k hat keine Bedeutung.

Noch zu beweisen ist, dass $G \in$ **HAMILTON** $\Leftrightarrow f(G) = A \in TSP_{90}$. Hierzu müssen die Fälle $G \in$ oder $G \notin$ **HAMILTON**, $G \neq (V, E)$ und $n < 3$ betrachtet werden.

- Für den Fall $G \in$ **HAMILTON**, gibt es einen Hamilton-Pfad, der nur Kanten aus G nutzt. Analog dazu gibt es für TSP_{90} auch einen Pfad H mit $|H| = 0$, da nur Kanten aus E genutzt werden, deren Kosten $n - 2$ sind.
- Für den Fall $G \notin$ **HAMILTON**, gibt es keinen Hamilton-Pfad, der nur Kanten aus G nutzt. Analog dazu gibt es in TSP_{90} auch keine Permutation H der Länge $n - 2$.
- Für den Fall $G \neq (E, V)$, also dass G keinen Graphen kodiert, gibt es keinen Hamilton-Pfad. Analog dazu konstruiert die Reduktionsfunktion f auch keine Instanz eines TSP_{90}-Problems. Somit ist $f(G) \notin TSP_{90}$.
- Für den Fall $n = 2$ gibt **HAMILTON** einen Pfad durch beide Knoten zurück, wenn diese verbunden sind. TSP_{90} würde dies allerdings auch tun, wenn diese nicht verbunden sind, da die Länge jeder Permutation zweier Knoten immer 0 sein würde, da sie sich aus einer Summe $\sum_b^a$ mit $a < b$ berechnet. Um **HAMILTON** auch in diesem Fall noch auf TSP_{90} abzubilden, wird ein Sonderfall für $n = 2$ eingeführt, in dem gilt $|H| = A_{H_1,H_2,1} - 1$. Die Länge hängt für $n = 2$ also nur noch vom Bestehen einer Kante zwischen den Punkten H_1 und H_2 ab. Wenn sie existiert gilt $|H| = 1 - 1 = n - 2$

- Für $n = 1$ muss ein weiterer Sonderfall definiert werden, da in einem Graphen mit nur einem Knoten immer ein Hamilton-Pfad existiert und dieser auch nicht schärfer als 90° abbiegen kann. Damit H eine Lösung von TSP_{90} ist, muss aber immernoch gelten $|H| = n - 2 = -1$. Der Sonderfall kann also formuliert werden als $|H| = -1$ wenn $n = 1$.

Die Reduktion konstruiert mit der Reduktionsfunktion f eine $n \times n \times n$ Matrix A, wobei n die Anzahl von Knoten in G ist. A kann in kubischer Zeit ($\mathcal{O}(n^3)$) berechnet werden. f ist also in polynimaler Zeit berechenbar.

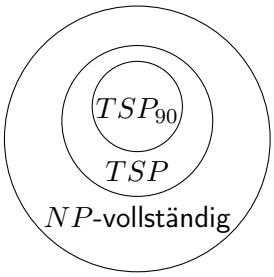


Abbildung 2: TSP_{90} ist NP-vollständig

Es gilt also $TSP_{90} \in$ **NP** und TSP_{90} ist **NP**-schwer. Kombiniert heißt das, dass TSP_{90} **NP**-vollständig ist und es unter der Annahme $P \neq NP$ nicht möglich ist, einen Algorithmus zu finden, der diese Aufgabe exakt in polynomieller Zeit löst.

2 Umsetzung

Bevor der entworfene Algorithmus gestartet werden kann, müssen die Punkte aus den Beispieldateien eingelesen werden. Die `get_data(n)`-Methode liest die Punkte als (x, y) Tupel in eine Liste ein. Sie ist zusammen mit anderen Hilfsmethoden in der Datei utils.py implementiert und wird hier nicht weiter erläutert.

Der Algorithmus selber ist in der `find_route_1`-Methode in algo.py implementiert und wird mit der Punkteliste als Parameter `points` aufgerufen.
Damit Backtracking möglich ist, muss der Algorithmus Informationen über frühere Entscheidungen des Algorithmus speichern, um sich dann umentscheiden zu können. Dies geschieht in der `backtracking_list`, die als erstes vom Algorithmus definiert wird.
Um den zufälligen Startpunkt der Tour auszuwählen, wird in einer `for`-Schleife über die gesamte Punkteliste `points` iteriert und jeder der Pukte als möglicher Startpunkt verwendet.
Innerhalb der `for`-Schleife wird zunächst die Tour `route` als Liste initialisiert, die anfangs zweimal den Startpunkt enthält und der Startpunkt aus `points` entfernt. Die Notwendigkeit des doppelten Speicherns des Startpunkts wird im weiteren Verlauf erläutert. Dann wird der `start_idx` auf 0 gesetzt. Er gibt an, ab welchem Index alle möglichen Folgepunkt betrachtet werden sollen und ist Teil des backtracking-Mechanismus.

Hierauf folgt die Konstruktion der `route` in der Endlosschleife. Diese bestimmt mit der `get_next_points`-Methode alle möglichen Folgepunkte und speichert sie in `next_points`.
`get_next_points` nimmt den letzten und den vorletzten Punkt der `route` als Parameter und gibt alle Punkte zurück, mit denen der neu entstehende Abbiegewinkel die 90° nicht übersteigt. Da die letzten zwei Punkte der `route` als Parameter verwendet werden, würde eine `route` der Länge 1 hier für einen Fehler sorgen, weshalb der Startpunkt am Anfang einfach zweimal hinzugefügt wurde. Der Nullvektor, der in der `get_next_rects` Methode so entsteht, wird durch eine `if`-Abfrage abgefangen und einfach alle Punkte als mögliche Folgepunkte zurückgegeben. So kann in der Hauptmethode `find_route_1` saubererer Code geschrieben werden.

Wenn in `next_points` mehr Punkte enthalten sind als der `start_idx` vorgibt, dann wird der

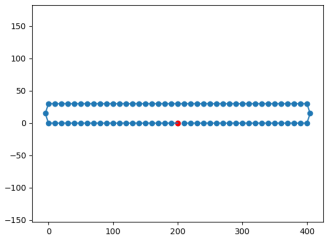
`route` der Punkt an der Stelle `start_idx` in `next_points` angefügt. Weiterhin wird dieser Punkt aus `points` entfernt und `start_idx+1` an die `backtracking_list` angehängt.

Sind in `next_points` nicht mehr Punkte enthalten als der `start_idx` vorgibt, gibt es keine Folgepunkte, die noch nicht ausprobiert wurden. Sind zusätzlich auch keine Punkte in `points` vorhanden, wurde eine Route zusammengesetzt und `route` als Lösung zurückgegeben.

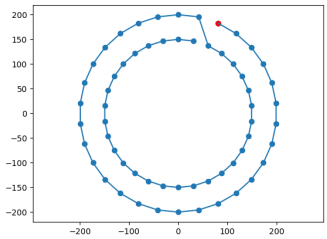
Ist `points` noch nicht leer, muss der Algorithmus backtracken. Hierzu wird das letzte Element aus `route` entfernt und `points` wieder angefügt. Der neue `start_idx` wird mit der $\mathcal{O}(pop()$-Methode aus der `backtracking_list` gelesen.

3 Beispiele

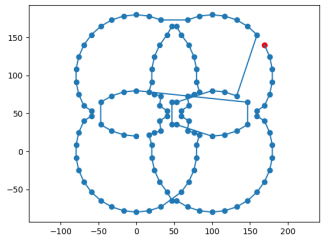
Der entworfene Algorithmus ist in der Lage die gesuchten Routen in den zur Verfügung gestellten Beispieldateien zu finden:



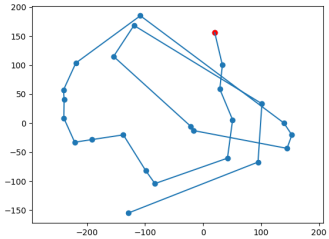
(a) wenigerkrumm1.txt



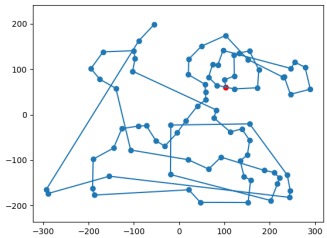
(b) wenigerkrumm2.txt



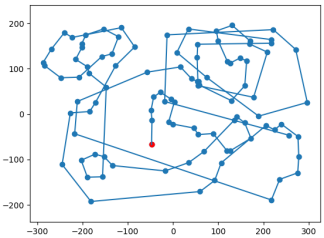
(c) wenigerkrumm3.txt



(d) wenigerkrumm4.txt



(e) wenigerkrumm6.txt



(f) wenigerkrumm7.txt

Eine Lösung für Beispiel wenigerkrumm5.txt war aufgrund einer zu hohen Laufzeit nicht berechenbar. Die Berechnung der Route wurde über mehrere Stunden bis zum 10^9ten Backtracking laufen gelassen und dann abgebrochen. Das Backtracking hat den Startpunkt nicht einmal ansatzweise erreicht, wehalb die gesamte Zeit derselbe Startpunkt verwendet wurde. Eine Extrapolation der Geschwindigkeit, in der der Algorithmus die verschiedenen Pfade berechnet hat, ergibt eine Restlaufzeit des Algorithmus von $\approx 10^{17}$ Jahren.

4 Optimierung

4.1 Laufzeit

Im Folgenden werden Techniken verwendet, um die Laufzeit des Algorithmus zu verbessern:

4.1.1 Memoization

Eine weit verbreitete Technik zur Optimierung von Algorithmen ist Memoization. Memoization speichert die Ergebnisse von Funktionen, um diese in der Zukunft nutzen zu können, ohne sie erneut berechnen zu müssen. Sie bietet sich hier besonders an, da es sehr wahrscheinlich ist, dass der Algorithmus den Winkel eines Tripels von Punkten sehr oft berechnet. Da sich dieser nicht ändert, ist er geeignet.

Für die Implementierung wird ein dreidimensionales Array `memoization_table` geschaffen, in dem mit einem `boolean` gespeichert wird, ob der Abbiegewinkel jedes Tripels von Punkten, für den jeder Eintrag des `memoization_tables` kodiert $> 90°$ ist. Der `memoization_table` ist äquivalent zur dreidimensionalen Adjazentenmatrix A, die in Kapitel 1.5 definiert wurde.

Die Methode `get_next_points`, in der zuvor die Winkelberechnung gestartet wurde, erhält nun den `memoization_table` als Parameter und prüft vor jeder Winkelberechnung, ob bereits ein Ergebnis vorhanden ist. Wenn ja, wird dieses verwendet. Wenn nicht, wird der Winkel berechnet und dem `memoization_table` hinzugefügt.

Dasselbe Verfahren wird auch auf die Distanz zweier Punkte angewendet, die ebenfalls mehrfach berechnet wird.
Durch diese Optimierung konnte die Laufzeit zum Beispiel für wenigerkrumm7.txt von 175ms auf 71ms verbessert werden. wenigerkrumm5.txt kann aber immernoch nicht gelöst werden. Der Algorithmus ist in der Methode `find_route_2` implementiert.

4.1.2 Hamiltonkreismethode

Ein weiterer Ansatz der Optimierung ist die Wahl des Startpunktes. Da dieser nicht festgelegt ist, werden aktuell alle Punkte als mögliche Startpunkte verwendet. Wenn der Algorithmus einen Pfad H durch alle Punkte aus P gefunden hat, kann der so entstandene Hamilton-Pfad durch Hinzufügen einer Kante, die den letzten und ersten Punkt aus H verbindet, zu einem Hamilton-Kreis erweitert werden. Hierbei können die entstehenden zwei Abbiegewinkel außer Acht gelassen werden. Durch das Entfernen einer beliebigen Kante kann H wieder zum Hamilton-Pfad und somit zu einer Lösung des TSP_{90} Problems werden. Durch das Entfernen der Kante werden auch zwei Abbiegewinkel entfernt, die gerne die Winkelbeschränkung verletzen können.
Mit diesem Wissen kann der Algorithmus dahingehend angepasst werden, dass er einen bzw. auch zwei aufeinanderfolgende Abbiegewinkel $> 90°$ bei der Konstruktion des Pfades erlaubt. Dann darf durch das Hinzufügen der letzten Kante aber kein scharfer Abbiegewinkel mehr entstehen.

Da ein Kreis weder Anfang noch Ende hat, befindet sich H nach dem Ergänzen zu einem Hamilton-Kreis in einer Art Superposition, in der er mit jedem seiner Punkten beginnen kann. Erst das Entfernen einer Kante im Nachhinein legt den Start- und Endpunkt des Pfades fest. So reicht es aus, am Anfang nur einen Startpunkt zu verwenden.

So können $n - 1$ Durchläufe des Algorithmus mit anderen Startpunkten gespart werden, wodurch sich die Laufzeit auf $\mathcal{O}(\frac{n!}{n}) = \mathcal{O}((n-1)!)$ verbessert. Dies mag zwar unbedeutend wirken, sorgt für $n = 60$ Laufzeit auf $\frac{1}{60}$ reduziert.
Da diese Verbesserung die Größenordnung des Problems nicht ändert, wird sich hiermit wenigerkrumm5.txt immernoch nicht lösen lassen, weshalb diese Verbesserung nicht implementiert wird.

4.1.3 Variierender Startpunkt

Während der Analyse von Zwischenergebnissen von `find_route_2` fällt auf, dass es Punkte gibt, die in nur sehr wenigen Pfaden integriert werden können (Abb. 3). Eine Lösung für dieses Problem ist, diese schwer integrierbaren Punkte zu bestimmen und den Pfad bei diesen starten oder enden zu lassen. Eine weitere Beobachtung ist, dass die Pfade mit der Zeit immer unkonntrollierter werden, was daran liegt, dass die NN-Heuristik mit der Zeit weiter entfernte Punkt miteinander verbindet. Es ist also gar nicht sinnvoll, den Algorithmus so lange laufen zu lassen, bis er alle Pfade ausprobiert hat.

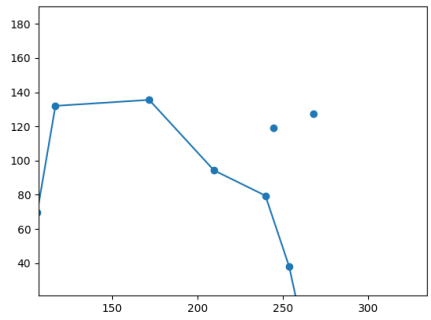


Abbildung 3: Schwer integrierbare Punkte

Der Algorithmus wird dahingehend verändert, dass er nach dem 100.000ten Backtracking den Startpunkt [1] wechselt. Diese Änderung ist zusammen mit dem Ergebnis von Kapitel 5.2 in `find_route_3` implementiert.
Mit dieser Veränderung ist der Algorithmus in der Lage, eine Lösung für `wenigerkrumm5.txt` innerhalb von 9.5s mit dem zweiten Startpunkt zu bestimmen.

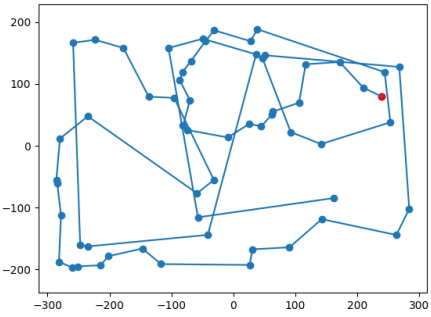


Abbildung 4: Lösung für `wenigerkrumm5.txt`

1. Hier werden alle Punkte als Startpunkte verwendet, ohne schwer integrierbare Punkte zu priorisieren, da das Kriterium für die Integrierbarkeit von Punkten erst später entwickelt wird.

4.1.4 Cheapest-Insertion Heuristik

Sollte sich mit dem Ansatz aus dem vorherigen Abschnitt keine Route bilden lasse, kann versucht werden die schwer integrierbaren Punkte in die bereits bestehende Route einzufügen. Manche dieser Punkte bieten sich dafür gut an, da sie in der Nähe von bereits bestehenden Kanten liegen:

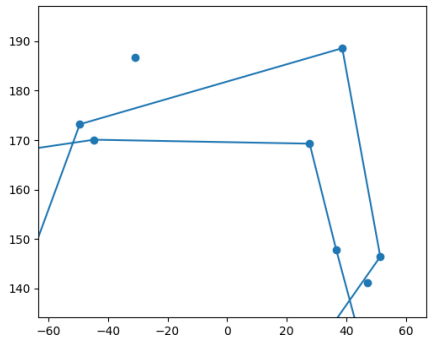


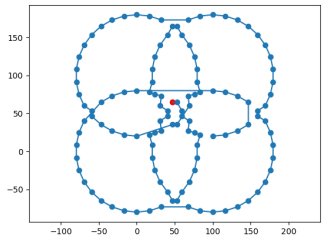
Abbildung 5: Szenario für Cheapest Insertion

Eine Technik, auch Heuristik genannt, um dies zu tun, ist Cheapest-Insertion Heuristik (CIH). Die Cheapest Insertion Heuristik bestimmt immer wieder den Punkt mit dem geringsten Abstand (Kosten) zu dem bereits vorhandenen Pfad. Dieser Knoten wird als "Kandidatenknoten" bezeichnet. Der Kandidatenknoten wird dann in die Kante der Tour mit der geringsten Distanz eingefügt.
Eigentlich wird die Cheapest-Insertion Heuristik genutzt, um einen Pfad für ein TSP-Problem von Anfang an aufzubauen, was hier aber nicht notwendig ist, da die NN-Heuristik bereits sehr gute Zwischenergebnisse generiert. Das Ziel ist, diese sehr guten Zwischenergebnisse zu identifizieren und CIH auf die letzten noch fehlenden Punkte anzuwenden.
Ein Ansatz, diese sehr guten Zwischenergebnisse zu identifizieren ist, der Anteil von Punkten aus P bereits in H enthalten sind. Fängt der Algorithmus an zu backtracken, also wieder Punkte aus H zu entfernen, obwohl H bereits z.B. 90% der Punkte enthalten sind, kann der Algorithmus die Technik wechseln und versuchen, die verbleibenden 10% der Punkte mit der CIH einzufügen.
Um an die Methode aus dem vorherigen Kapitel anzuknüpfen, kann für jeden Startpunkt auch ein bester/längster Pfad ausgegeben werden, auf den dann CIH angewendet wird.

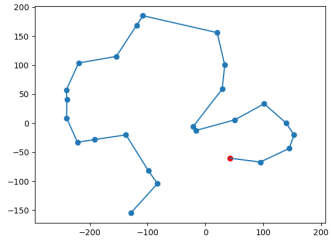
4.2 Distanz

Ein weiterer Ansatz der Optimierung ist neben der Laufzeit auch die Länge der entstehenden Lösungen, da die Ergebnisse der NN- oder CIH zwar in der Regel gute, aber nur selten optimale Lösungen liefern. Zusätzlich gibt es keine Schranken, die angeben, wie schlecht eine Lösung relativ zu der optimalen Lösung maximal sein kann. Hierzu gibt es viele andere Techniken, die hier aber nicht weiter betrachtet werden, da nur wenige dieser Techniken für beschränkte TSP anwendbar sind.

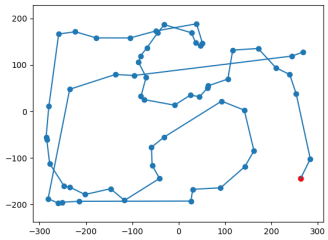
Ein erster Ansatz der Optimierung, der sich leicht in die aktuell bestehenden Algorithmen integrieren lässt, folgt aus Kapitel 4.1.3. Anstatt den ersten Pfad als Lösung zu verwenden, können Lösungen mit allen Startpunkten generiert werden und aus diesen die kürzeste ausgewählt werden. Hierzu wurde die Methode `get_length_of_route` implementiert, die die Gesamtlänge einer Route berechnet. Mit diesem Ansatz konnten die Ergebnisse für Beispieldateien $4 - 7$ verbessert werden:



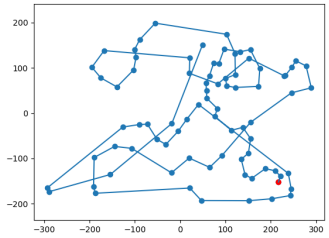
(a) wenigerkrumm3.txt Länge = 1962km; vorher 2103km -> Verbesserung: 7%



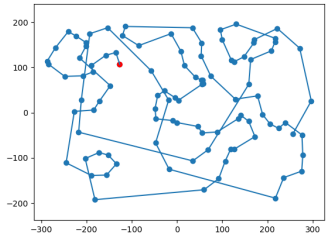
(b) wenigerkrumm4.txt Länge = 1205km; vorher 2198km -> Verbesserung: 82%



(c) wenigerkrumm5.txt Länge = 3507km; vorher 4738km -> Verbesserung: 35%



(d) wenigerkrumm6.txt Länge = 4222km; vorher 4830km -> Verbesserung: 14%



(e) wenigerkrumm6.txt Länge = 4943km; vorher 6218km -> Verbesserung: 26%

5 Entscheidungsproblem

Ein weiterer Teil der Aufgabe beschäftigt sich mit der Frage, ob jede Instanz des TSP_{90} lösbar ist. Die Antwort hierauf ist ein schlichtes Nein, wie zum Beispiel Abb. 6 zeigt.
Es stellt sich die Frage, welche Eigenschaften sich lösbaren und nicht-lösbaren Instanzen des TSP_{90} Problems zuordnen lassen und ob daraus Bedingungen für die Lösbarkeit formuliert werden können. Diese Frage wird dem Entscheidungsproblem von TSP_{90} zugeordnet, bei dem es nur darum geht, zu entscheiden, ob eine Probleminstanz lösbar ist oder nicht. Aus der **NP**-Vollständigkeit des TSP_{90}-Problems lässt sich leicht zeigen, dass diese für das zugehörige Entscheidungsproblem ebenfalls gilt. Hieraus folgt, dass es keine hinreichenden Bedingungen geben kann, die sich in polynomialer Zeit überprüfen lassen. Die Suche hiernach ist somit vergebens.

Neben hinreichenden Bedingungen gibt es notwendige Bedingungen, die erfüllt sein müssen, damit eine Konsequenz zutrifft. Da aus einer wahren notwendigen Bedingung nicht direkt die Wahrheit der Konsequenz folgt, weil diese von weiteren notwendigen Bedingungen abhängen kann, stellt eine notwendige Bedingung, die sich effizient berechnen lässt, keinen Konflikt mit der **NP**-Vollständigkeit des Entscheidungsproblems dar. Es folgt, dass notwendige Bedingungen, die sich in polynomialer Zeit berechnen lassen, existieren können.

Ein Ansatz, der zum Finden einer solchen Bedingung verwendet werden kann, ist die Idee aus Kapitel 4.1.2, nach der ein Pfad H, der Lösung des TSP_{90}-Problems ist, über einen Hamilton-Kreis mit maximal einem Abbiegewinkel $\alpha > 90°$ bestimmt werden kann. Für jeden Punkt P_i aus P kann ein Vor- und ein Nachfolgepunkt P_jP_k bestimmt werden, sodass der Abbiegewinkel des Tripels $P_jP_iP_k$ minimal ist. Für jeden Punkt wird also ein minimaler Abbiegewinkel bestimmt. Gibt es mehr als zwei Punkte in P, deren minimaler Abbiegewinkel $> 90°$ ist, so ist es nicht möglich einen Hamilton-Kreis

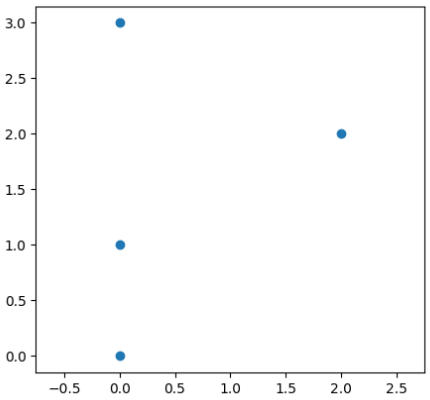


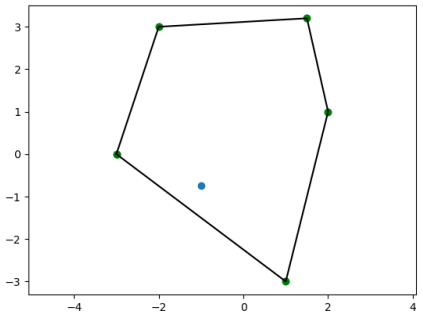
Abbildung 6: Nicht lösbare Instanz - $(0,0)(0,1)(2,2)(0,3)$

mit maximal einem Abbiegewinkel $> 90°$ zu konstruieren. Das TSP_{90}-Problem ist dann nicht lösbar.

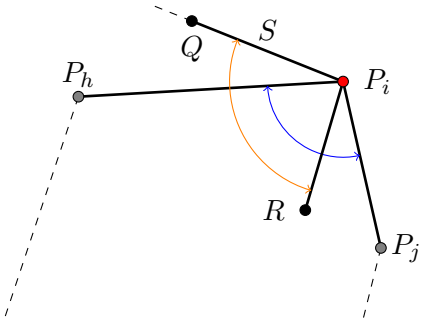
5.1 Konvexe-Hülle

Beim händischen Bestimmen der minimalen Abbiegewinkel α fällt auf, dass für jeden Punkt P_i, der Teil der konvexen Hülle K von P ist, die zwei Punkte, mit denen der minimale Abbiegewinkel gebildet wird, die Nachbarn von P_i in K sind. Die konvexe Hülle einer zweidimensionalen Punktemenge P ist das kleinste Polygon, das alle Punkte aus P und alle Verbindungsstrecken jedes Punktepaares aus P enthält (Abb. 7a). Hier kann eine Regelmäßigkeit vermutet werden. Sollte diese Vermutung wahr sein, kann der minimale Abbiegewinkel für manche Punkte schnell berechnet werden, da sich eine konvexe Hülle beispielsweise mit dem Quickhull-Algorithmus in $\mathcal{O}(n\,log\,n)$DAVID P. DOBKIN, "The Quickhull Algorithm for Convex Hulls", 1996, https://dl.acm.org/doi/pdf/10.1145/235815.235821 bestimmen lässt. So kann die zuvor formulierte Bedingung, dass eine Punktemenge maximal einen minimalen Abbiegewinkel $\alpha > 90°$ haben darf, verwendet werden um einen Teil von P effizient zu überprüfen.

Für den Beweis dieser Vermutung wird zunächst der den Abbiegewinkel α ergänzende Winkel $\alpha' = 180° - \alpha$ von dem Punkt P_i, der auf der konvexen Hülle K der Punktemenge P liegt konstruiert. Dieser wird mit den 2 benachbarten P_hP_j von auf K eingeschlossen (siehe Abb. 7b; blauer Winkel). Angenommen es existiert ein anderes Paar von Punkten R und $Q \in P$ und $\notin K$, mit denen P_i einen größeren Winkel α' (siehe Abb. 7b; orangener Winkel), also einen kleineren Abbiegewinkel α, einschließt, müssen beide Punkte innerhalb von K liegen, da K jeden Punkt aus P einschließt. Exemplarisch wird der Punkt R willkürlich in K gesetzt und dann eine Strecke $\overline{RP_i}$ konstruiert. Im Punkt P_i kann nun ein Winkel mit der Strecke $\overline{RP_i}$ aufgespannt werden, der größer ist als α', wodurch sich der angedeutete Strahl S ergibt, auf dem der zugehörige Punkt Q liegen muss. Da S vollständig außerhalb von K liegt und K jeden Punkt von P einschließt, muss $Q \notin P$ gelten, was im Widerspruch zur Annahme steht. Selbiges gilt mit $R = P_j$.



(a) Konvexe Hülle K für eine Punktemenge



(b) Konstruktion zum Beweis der Behauptung; Grafik kann als Zoom in die obere-rechte Ecke aus Abb. 7a aufgefasst werden.

Abbildung 7: Konvexe Hüllen zur Bestimmung von minimalen Abbiegewinkeln

5.2 Generalisierung

Um die Bedingung aus Kapitel 5.1 in einem größeren Maßstab anwenden zu können, muss der minimale Abbiegewinkel für jeden Punkt aus P gefunden werden können. Die einzige Möglichkeit hierfür ist das Ausprobieren jeder möglichen Kombination von Punkten, wofür es ungefähr $\binom{n}{2} \approx n^2$ mögliche Kombinationen gibt. Kodiert man die Ergebnisse mit einer 0 für einen Abbiegewinkel $\alpha \leq 90°$ und 1 für $\alpha > 90°$ und speichert diese in einem dreidimensionalen Array, so erhält man die in Kapitel **??** als formale Eingabe des TSP_{90}-Problems definierte dreidimensionale Adjazentenmatrix A.

Da beim Bestimmen des minimalen Abbiegewinkels für den Punkt P_i nur der Vorgänger und Nachfolger von P_i variabel sind, liegen alle Punktekombinationen in A in der von-nach-Ebene mit über$= i$ (siehe Abb. 8). Ist die Summe S aller Einträge der Ebene A_{jik} mit$j, k \in [1; n]$ gleich 0, ist der minimale Abbiegewinkel von $P_i > 90°$. Es gilt

$$S = \sum_{k=1}^{n} \sum_{j=1}^{n} A_{kij} \tag{5}$$

Mit $n_{S=0}$ als die Zahl der Punkte mit minimalem Abbiegewinkel $> 90°$ kann die Bedingung formuliert werden als $n_{S=0} \leq 2$. Anders als im vorherigen Kapitel lässt sich diese Bedingung nicht so schnell überprüfen, da die gesamte Matrix A mit n^3 Einträgen bestimmt werden muss. Andererseits kann durch das Überprüfen der Bedingung eine fakultative Laufzeit vermieden werden, falls keine Lösung existiert. Außerdem würde der Algorithmus an einem bestimmten Punkt vermutlich sowieso den Winkel für alle möglichen Tripel von Punkten berechnet haben, weshalb es nicht schadet, dies vor der Konstruktion einer Route zu tun.

Dieser Ansatz ist in `find_route_3` zusammen mit der Methode aus Kapitel 4.1.3 implementiert und stellt die finale Version meiner Abgabe dar.

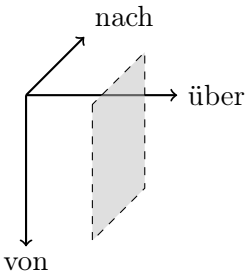


Abbildung 8: Dreidimensionale Adjazentenmatrix; Hervorgehoben sind die Werte, die zu einem festen Folgepunkt führen

6 Anhang

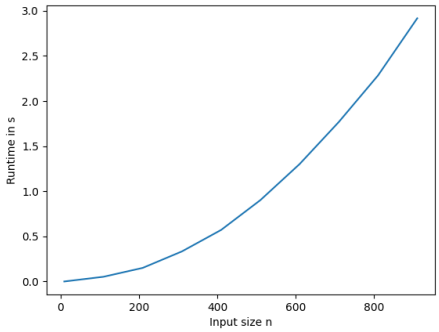


Abbildung 9: Best-Case Laufzeit

7 Code