Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 64903

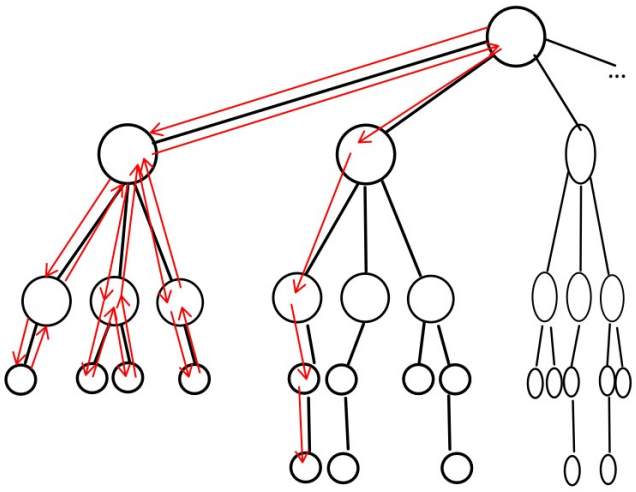
Bearbeiter/-in dieser Aufgabe:
Alessio Caputo

12. April 2023

Inhaltsverzeichnis

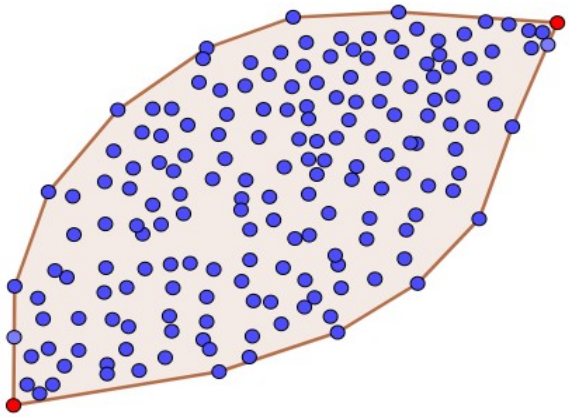
Lösungsidee

Das angeführte Problem hat ohne Zweifel einige Ähnlichkeit mit dem TSP(=Traveling Salesman Problem). Bei diesem müssen ebenfalls eine Reihe an Punkten zu einer möglichst kurzen Route verbunden werden. Es gibt jedoch zwei grundlegende Unterschiede. Im TSP wird nach einer in sich geschlossenen Route gesucht und nicht nach einem Weg, der von einem beliebigen Startpunkt zu einem Beliebigen Endpunkt führt. Auch gibt es keine Limitierung auf einen Innenwinkel von $\alpha \geq 90\,°$, was das Problem deutlich erschwert.
Der hier verfolgte Lösungsansatz verfolgt das „nearest-neighbour“-Prinzip, nach welchem der nächste Punkt einer Route der ist, der den kürzesten weg hat. Dieser doch recht Primitive Ansatz zeigt erstaunlicherweise recht gute Lösungen. Es kann mit einem Baumdiagramm recht simpel Veranschaulicht werden. Links stehen in den Folgeoptionen immer die Punkte mit dem geringsten Abstand. Rechts folglich die mit dem höchsten Abstand. Nun wird nach dem Suchprinzip des „Binary-Tree-Sort“-Algorithmus der erste Weg mit der gesuchten Länge n (n = Anzahl an Punkten der Eingabe) gesucht und ausgegeben. Dies ist aufgrund der Winkellimitierung notwendig, da der nächste Punkt nicht Zwangsläufig auch die Winkelbedingung erfüllt.



Das Startpunktproblem

Damit der Algorithmus ausgeführt werden kann muss zunächst ein Startpunkt aus der gegeben Punktmenge n berechnet werden. Dazu wird ein Punkt aus der Konvexen Hülle der Punkte gewählt, da bei einer relativ schwierigen Eingabe mit einem Punkt aus dieser Menge begonnen bzw. aufgehört werden muss. Die Konvexe hülle sind alle Punkte, deren Fläche alle weiteren Punkte einschließt. Dabei sind alle Innenwinkel $\alpha < 180°$. Gilt für einen Punkt $\alpha < 90°$, dann kann dieser Punkt von keinen anderen Punkten in mitten der Strecke „angeflogen“ werden. Dieses Problem kann umgangen werden, indem er zu einem Startpunkt/Endpunkt gemacht wird, da an dieser Stelle kein Punkt vorangestellt ist und der Winkel dadurch unerheblich ist. Nochmal anders sieht es jedoch aus, wenn für zwei Winkel innerhalb der Konvexen Hülle $\alpha < 90°$ gilt. Hier muss einer der Beiden Winkel ein Startpunkt und der andere Winkel ein Endpunkt sein, da ansonsten die Winkel nicht eingehalten werden können. Dies bedeutet nicht, dass garantiert eine Lösung für eine Eingabe mit 2 unmöglichen Winkeln auf der konvexen Hülle gefunden werden kann. Die Lösbarkeit hängt ebenfalls vor allem von den Punkten innerhalb der konvexen Hülle ab, da innerhalb dieser ebenfalls Punkte mit unmöglichen Winkeln existieren können. Für 3 unlösbare Winkel (Maximum an unlösbaren Winkeln) ist keine Lösung zu finden, da der Startpunkt und der Endpunkt nur zwei unlösbare Winkel abdecken kann.



Dieses Verfahren kann auch zur frühzeitigen Eliminierung von Verzweigungen genutzt werden, da es sich bei diesen letztlich auch nur um eine Punktmenge, wie die der Ausgangspunktmenge handelt. Der grundlegende Unterschied besteht hier nur in dem vorgegeben Startpunkt durch den bisherigen Weg.

Mit dieser Heuristik sollte das Lösen der Beispieleingaben und auch größerer Eingaben ohne Probleme möglich sein.

NP-scheere des Problems

Zunächst kann das Problem als Entscheidungsproblem formuliert werden: „Gibt es eine geschlossene Route, die alle Punkte verbindet und keine spitzen Winkel enthält?“ Um zu zeigen, dass das Problem NP-schwer ist, kann es auf ein bekanntes NP-schweres Problem reduziert werden. Hier wird das Hamilton-Kreis-Problem, bei dem es darum geht, herauszufinden, ob es in einem Graphen einen Kreis gibt, der alle Knoten enthält, gewählt. Es kann nun einen Grafen mit n Punkten erstellt werden, wobei jeder Punkt einem des Datensatzes entspricht. Wenn wir nun das ursprüngliche Problem lösen können, dh eine Route finden, die alle Punkte verbindet und keine spitzen Winkel enthält, haben wir gleichzeitig einen Hamilton-Kreis im Graphen gefunden. Da das Hamilton-Kreis-Problem NP-schwer ist, folgt daraus, dass das Problem der Aufgabestellung auch NP-schwer ist, da es auf das Hamilton-Kreis-Problem reduzierbar ist.

Implementierung

Die Lösungsidee wird in Python implementiert. (rückwirkend wäre JavaScript vermutlich aufgrund der später beschriebenen Erweiterung besser geeignet gewesen)

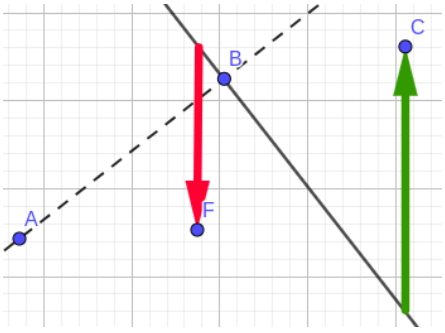
Einlesen und Darstellung der Punkte

Zunächst wird die Datei eingelesen und jeder Punkt wird in den Konstruktor der Klasse „P“(=Punkt/Point) überführt. Ein Objekt dieser Klasse stellt einen Punkt dar. Sie verfügt über die x bzw. y – Koordinaten des Punktes und eine Funktion „lenghtTo“, die die Distanz zu einem anderen Punkt abruft und zurückgibt. Da die Menge an Punkten von Beginn an fest steht, kann prophylaktisch die Distanz zu jedem anderen Punkt vorberechnet werden. Die „lenghtTo“ Funktion ruft lediglich einen Dictonary-Eintrag mit der Distanz zu diesem Punkt ab, die bei der Initialisierung der Punkte berechnet wurde. Dabei verfügt jeder Punkt über sein eigenes Dictonary mit allen weiteren Punkten. Dieses Vorgehen wurde aufgrund optimierter Zugriffszeiten gewählt. Dadurch wird jedoch auch jede Distanz zwei mal abgespeichert, was den Speicherbedarf des Programmes erhöht.

Valide Winkel

Recht schnell kommt es bei der Implementierung auf das Problem der Winkelüberprüfung des Programms. Es müssen etwa $\frac{l(n)!}{2(n-1)}$ (Zusammenhang wird in der Analyse erläutert) Winkelüberprüfungen während der Laufzeit des Programms durchgeführt werden, was bei hohen Eingaben den Algorithmus deutlich verlangsamen kann, wenn eine klassische Winkelberechnung mit trigonometrischen Funktionen verwendet wird. Bei dieser würde mit dem $\arccos()$ (oder einer anderen trigonometrischen Funktionen) der Winkel berechnet werden und anschließend überprüft, ob $\alpha \geq 90°$. Zwar Funktioniert dieses verfahren, jedoch gibt es eine effizientere Variante mit den selben Resultaten. In dieser Lösung werden lineare Funktionen verwendet, um den Winkel zu überprüfen.

Zunächst wird die orthogonale Funktion der Strecke AB durch B gebildet. Anschließend wird für den zu überprüfenden Punkt festgestellt, ob dieser ober oder unter der Funktion liegt. Welche Position der Punkt zur Funktion haben muss um Valide oder Invalide zu sein hängt von den Positionen von A und B ab. Diese werden einfach im Code alle mit *if*/*elif*-Abfragen abgefangen. Wenn $A_x = B_x$ gilt, dann kann keine Funktion gebildet werden, da diese eine Orthogonale zur X-Achse sein würde. Stattdessen wird nur die x-Koordinate des Punktes überprüft. Um Rechenzeit im Falle von $A_y = B_y$ zu sparen, wird nach dem gleichen Prinzip diese Überprüfung ebenfalls für die y-Koordinaten durchgeführt.



Die letzte Optimierung dieses Algorithmus bezieht sich nicht auf diesen selbst, sondern auf die Art, wie dieser Aufgerufen wird. Bei jedem Knotenpunkt der verschiedenen Abzweigung müssen eine Reihe von diesen Berechnungen durchgeführt werden. Dabei ist bei jeder Berechnung A und B identisch und somit auch die Funktion, mit der der Punkt überprüft werden muss. Dies ist ebenfalls

der Hauptgrund für die Überlegenheit dieses Algorithmus verglichen mit den trigonometrischen Berechnungsweisen.

Der einzige Nachteil, den dieser Algorithmus hat, ist das verlieren der Erweiterungsidee von anderen Winkeln. Jedoch kann in diesem Falle einfach ein anderer Winkelüberprüfungsmechanismus eingebaut werden.

Bestimmung der Startpunkte

Wie in der Lösungsidee beschrieben wird die konvexe Hülle der Punkte berechnet, um die nicht zu erreichenden Punkte dieser zu erhalten. Dafür wird der Graham-Scan Algorithmus genutzt, der eine schnelle Berechnung der konvexen Hülle mit einer Laufzeit von $O(n \log(n))$ ermöglicht. Bei diesem wird von einem Startpunkt aus (üblicherweise niedrigste y-Koordinate) der Punkt mit dem größten Winkel entlang der Route „angesteuert“. Dabei wird nicht jeder Winkel überprüft, sondern mit einem rekursiven Verhalten überprüft, ob eine Links oder Rechts Kurve vorliegt. Wenn die Hülle in die rechte Richtung gebildet wird, dürfen keine Rechtskurven innerhalb der Konvexen Hülle sein. Der Graham-Scan Algorithmus macht sich dies zu nutze, indem er im Falle einer Rechtskurve den Punkt der Kurve wieder aus der Route entfernt. Übrig bleibt eine Liste an Punkten, die die Konvexe Hülle in einer geordneten Rheinfolge bilden. Nun wird für jeden Punkt dieser Hülle überprüft, ob der Winkel valide ist. Anschließend sind drei Stadien möglich. Im falle keines Unerreichbaren Punktes wird der Punkt, der am weitesten von dem Zentrum entfernt ist, da dieser so nur einmal Angeflogen werden muss, zurückgegeben. Im Falle von einem Unerreichbaren Punkt wird dieser Zurückgegeben, um eine mögliche Route zu erhalten. Im falle von Zwei unerreichbaren Punkten wird einer der Beiden Zurückgegeben. Bei drei unerreichbaren Punkten ist keine Lösung möglich, deshalb beendet das Programm sich in diesem Fall selber. Es kann nur zu diesen Drei verschiedenen Stadien kommen, da ein Dreieck das größtmögliche gleichmäßige Geometrische Objekt ist, bei dem die alle Punkte einen Winkel <90 haben.

Berechnung der Route

Für die Berechnung der Route wird eine rekursive Funktion „step“ definiert. Diese Funktion wird für jeden Knotenpunkt des Baumdiagramms aufgerufen, und sie ruft sich selber auf, um die weiteren Zweige zu berechnen. In die Funktion werden der bisherige Weg und die übrigen Punkte als Parameter weitergeben. Zunächst wird überprüft, ob der bisherige Weg die gewünscht Länge hat, wenn dies der Fall ist, wird die Route ausgegeben und das Programm beendet. Sollte dies nicht der Fall sein, müssen alle weiteren möglichen Routen der Länge nach abgefragt werden. Dazu wird eine Abstandskarte zu allen Punkten mit Validen Winkeln erstellt. Sollte die Länge dieser Null sein, wird -1 als Zeichen für keine gefundene Route zurückgegeben. Wenn dies nicht der Fall ist, wird die Liste nach der Länge für eine Iteration sortiert. In dieser Iteration wird jeder Punkt der Route angehängt und aus den überigen Punkten entfernt, bevor die Funktion „step“ erneut aufgerufen wird. Wenn diese Funktion -1 zurückgibt wird das nächste Element der Iteration versucht. Wenn alle -1 Zurückgeben wird ebenfalls -1 zurückgegeben. Auf diese weise kann der gesamte Routenbaum speichereffizient berechnet werden.

Subrouteneliminierung

Wie in der Lösungsidee beschrieben kann nach dem Prinzip der Bestimmung der Startpunkte auch eine Subrouteneliminierung durchgeführt werden. Dies wurde zwar implementiert, jedoch wird der Algorithmus dadurch enorm verlangsamt. In Tests hat diese „Optimierung“ die Ausführzeit von

0.33s auf etwa 5 min gepusht. Offenbar ist diese Form der Subroutineliminierung nur in Theoretischer Sicht oder auf sehr hohen Ebenen des Baumes zu gebrauchen. Die „Optimierung“ wurde wieder aus dem Algorithmus entfernt.

Suchtiefe

So wie der Algorithmus aktuell arbeitet kann jede Beispieleingabe mit Ausnahme von *wenigerkrumm*5.*txt* innerhalb kurzer Zeit berechnet werden. Der Grund dafür ist die Lage der verschiedenen Punkte. Diese sind innerhalb *wenigerkrumm*5.*txt* (vermutlich eher zufällig) Großteils kreisförmig angeordnet. Dadurch wird es nahezu unmöglich eine Lösung innerhalb kurzer Zeit zu finden, da sobald der zweit nächste Punkt angesteuert wird der erst nächste Punkt schwieriger zu erreichen ist.

Eine Lösung für dieses Problem ist die Limitierung der Subrouten, die gewählt werden können. Diese Limitierung ist die „Suchtiefe“ (*depth* im code). Durch diese Limitierung kann das genannte Problem mit *wenigerkrumm*5.*txt* gelöst werden. Zudem wird verhindert einen Weg mit unnötig langer Länge zu wählen. Tests ergaben, dass eine Suchtiefe von 2-5 eine gute Balance zwischen Zeit und Ergebnisqualität (zumindest bei den Beispieleingaben) haben.

Sichtbarkeit von Punkten (Sichtbarkeitsgrapf)

Eine weitere Optimierung des Algorithmus ist die Modifikation der Sichtbarkeit verschiedener Punkte zur Optimierung der Routenlänge. Wenn eine maximale Distanz definiert wird, kann das die Qualität der Ergebnisse deutlich verbessern, da die Endlänge viel geringer ausfällt. Bei dieser Optimierung existieren jedoch zwei Probleme. Zunächst muss von dem Programm eine Maximallänge definiert werden. Dies ist an sich kein Problem, da eine Art Abstand aus den Punkten berechnet werden kann. Jedoch kann dies für verschiedene Eingaben, deren durchschnittliche Abstände starke Unterschiede aufweisen sehr Problematisch werden. Das zweite Problem besteht wieder in der Ausführzeit des Algorithmus. In Tests kam es nur bei *wenigerkrumm*4.*txt* zu einem besseren Ergebnis als ohne diese Optimierung. Alle anderen eingaben waren entweder Unverändert oder mussten aufgrund einer zu honen Ausführzeit abgebrochen werden. *Auch diese Optimierung wurde also nicht weiter Implementiert bzw. Verfolgt*

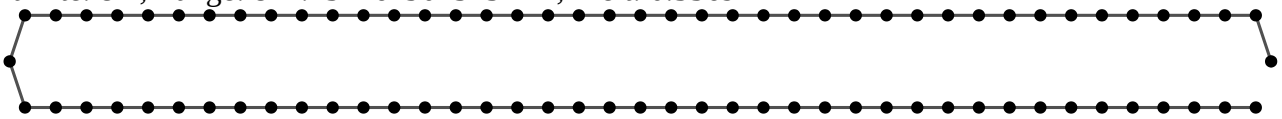
Ausgabe / Visualisierung

Da es schlicht keinen Sinn macht einen Datensatz an Punkten als Ausgabe (vor allem für debug-Zwecke) zu betrachten wird eine Ausgabe der Route als Visualisierung implementiert. Diese Basiert auf dem SVG(=Scalable Vector Grafics) Format, dass es ermöglicht Grafiken auf der Basis von Koordinaten zu erstellen. Zudem ist es unendlich Skalierbar, was das Problem des Herunter- / Hochskalieren von Koordinaten auf einzelne Pixel wegfallen lässt. Dafür wird die Python-Bibliothek „svgwrite“ verwendet, da sie über ein relativ simples Interface verfügt. Da ich Faul bin und die entstandene SVG-Datei nicht noch zusätzlich Öffnen möchte, wird eine Funktion „openWithSystemViewer(path)“ definiert, die die Datei am ende einfach öffnet. (Achtung bei Ausführung: nur mit Linux *xdg−open* getestet. Hatte gerade kein Windows oder MacOS da) Zusätzlich werden die Punkte auch noch den Eingaben nachempfunden in der Konsole sortiert ausgegeben. *Nachsatz: Offenbar invertiert mein Darstellungsscript die Punkte entlang der X-Achse. Dies wurde nicht verändert, da es prinzipiell keinen Einfluss auf die Ergebnisse usw. hat.*

Beispiele

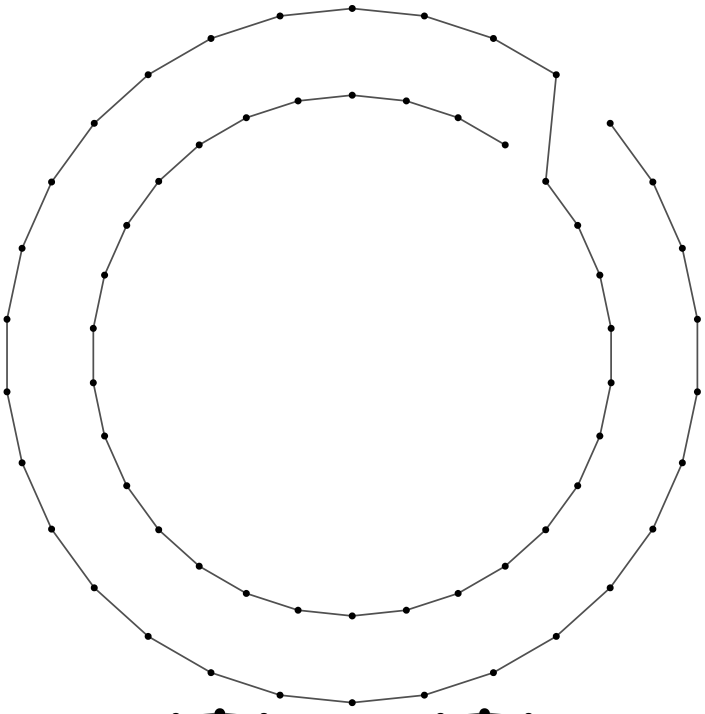
wenigerkrumm1.txt

Punkte: 84 ; Länge: 847.4341649025257LE; Zeit: 0.336s



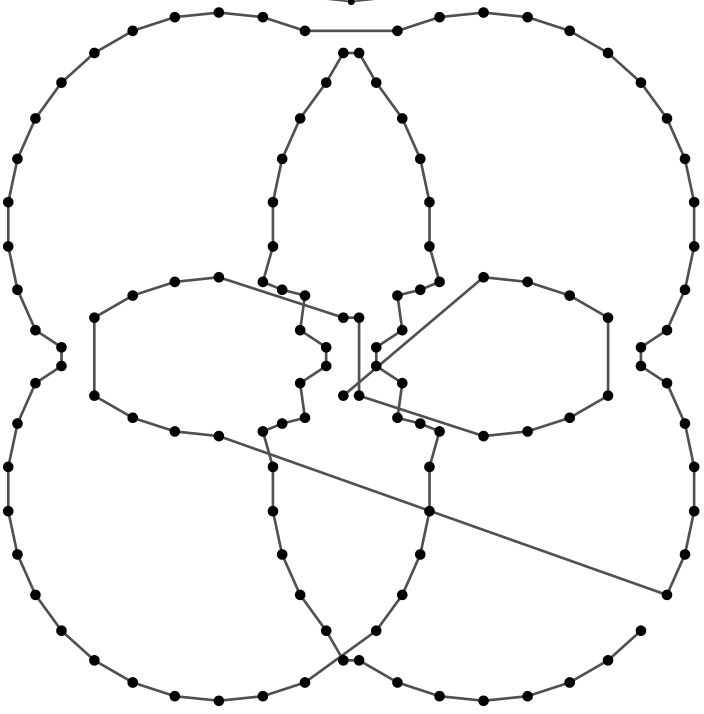
wenigerkrumm2.txt

Punkte: 60
Länge: 2183.662266989629LE
Zeit: 0.33s



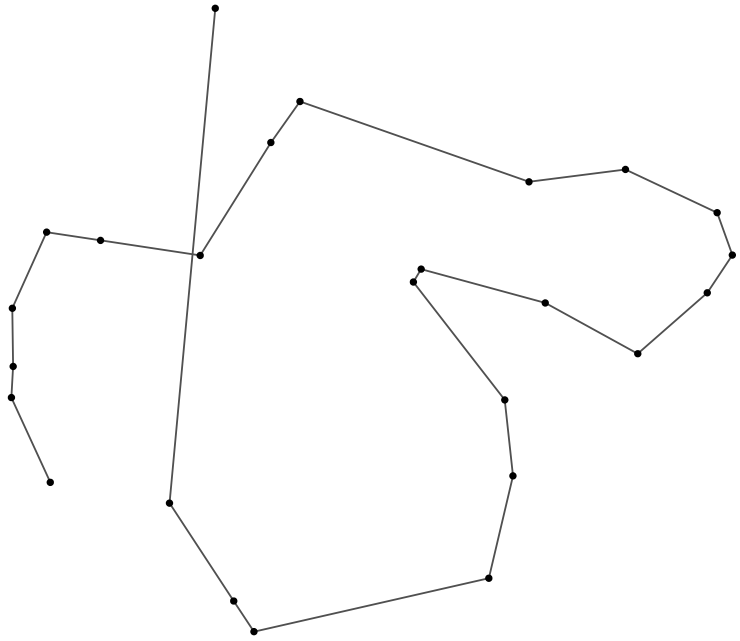
wenigerkrumm3.txt

Punkte: 120
Länge: 2129.9995765188946LE
Zeit: 0.347s



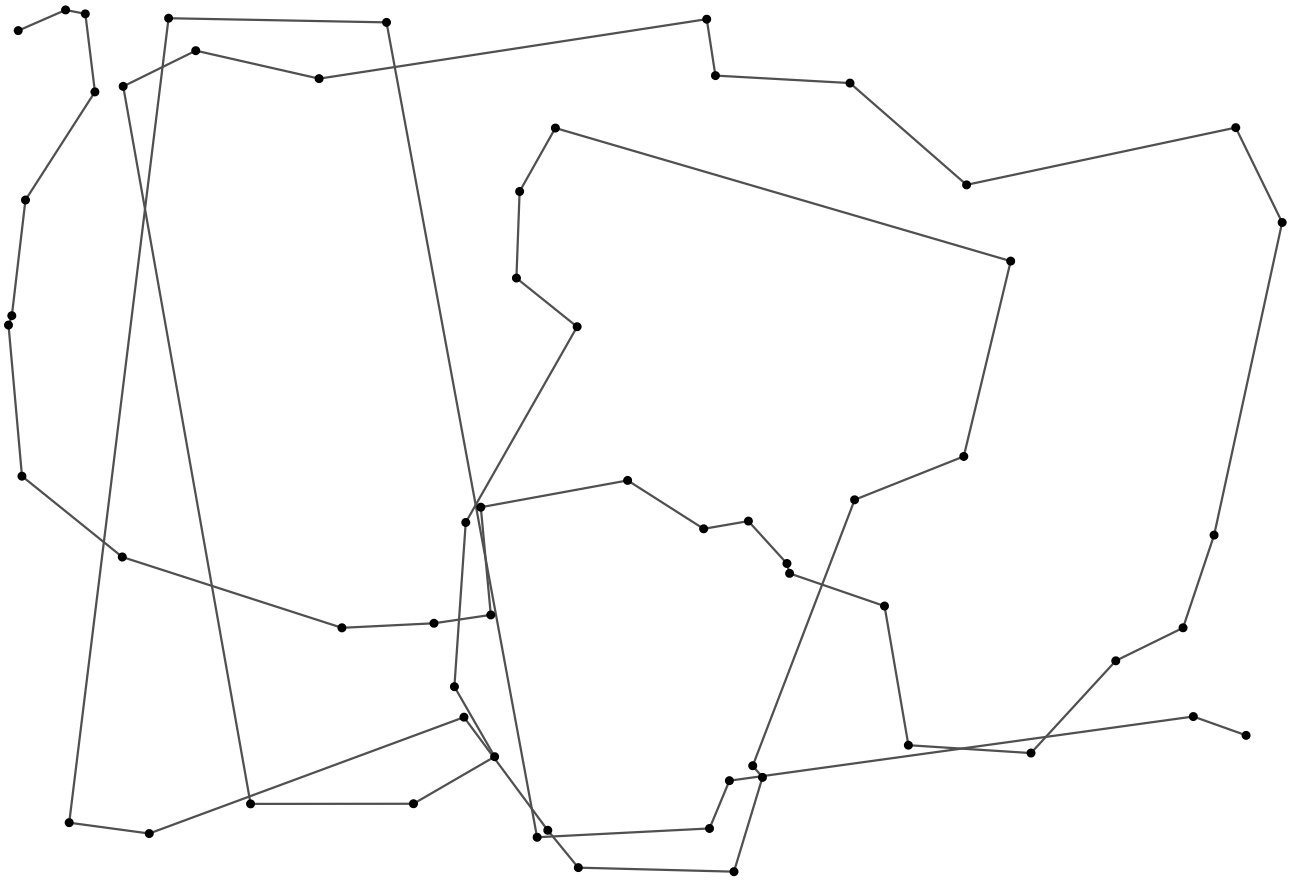
wenigerkrumm4.txt

Punkte: 25
Länge: 1473.9727369660593LE
Zeit: 0.32s



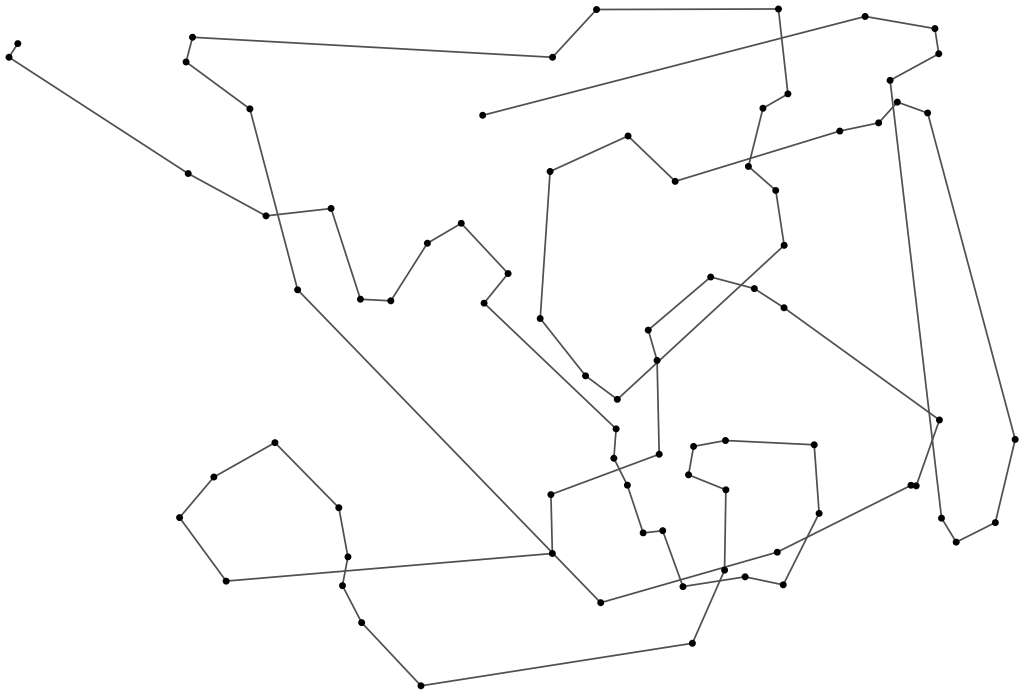
wenigerkrumm5.txt

Punkte: 60
Länge: 4514.0163972060955LE
Zeit: 1.02s



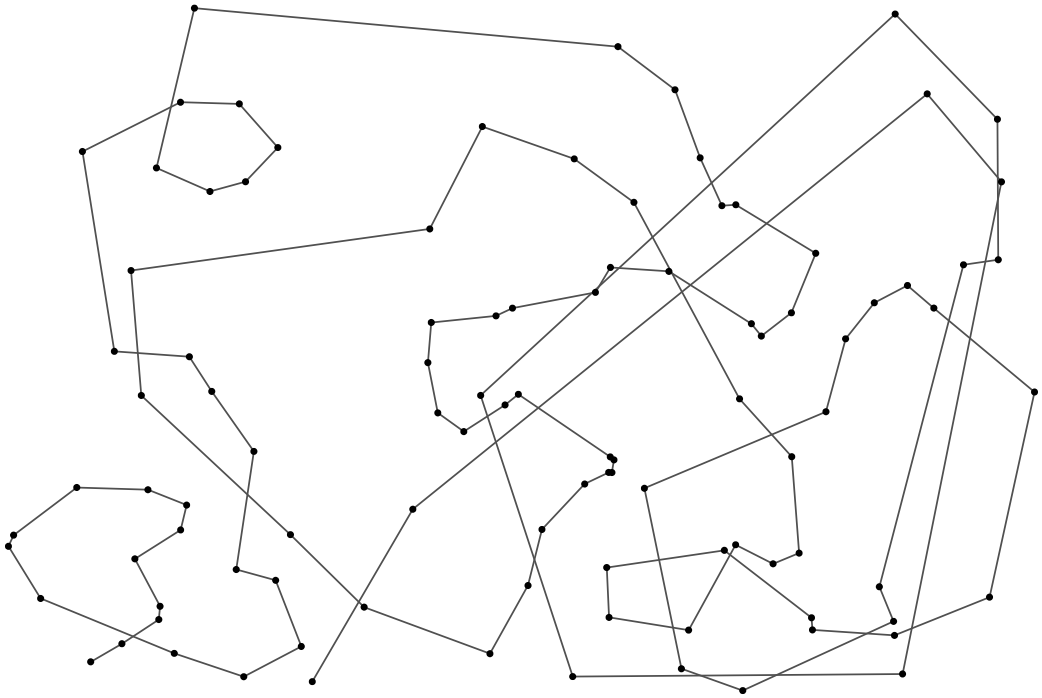
wenigerkrumm6.txt

Punkte: 80
Länge: 4552.89661277764LE
Zeit: 0.33s



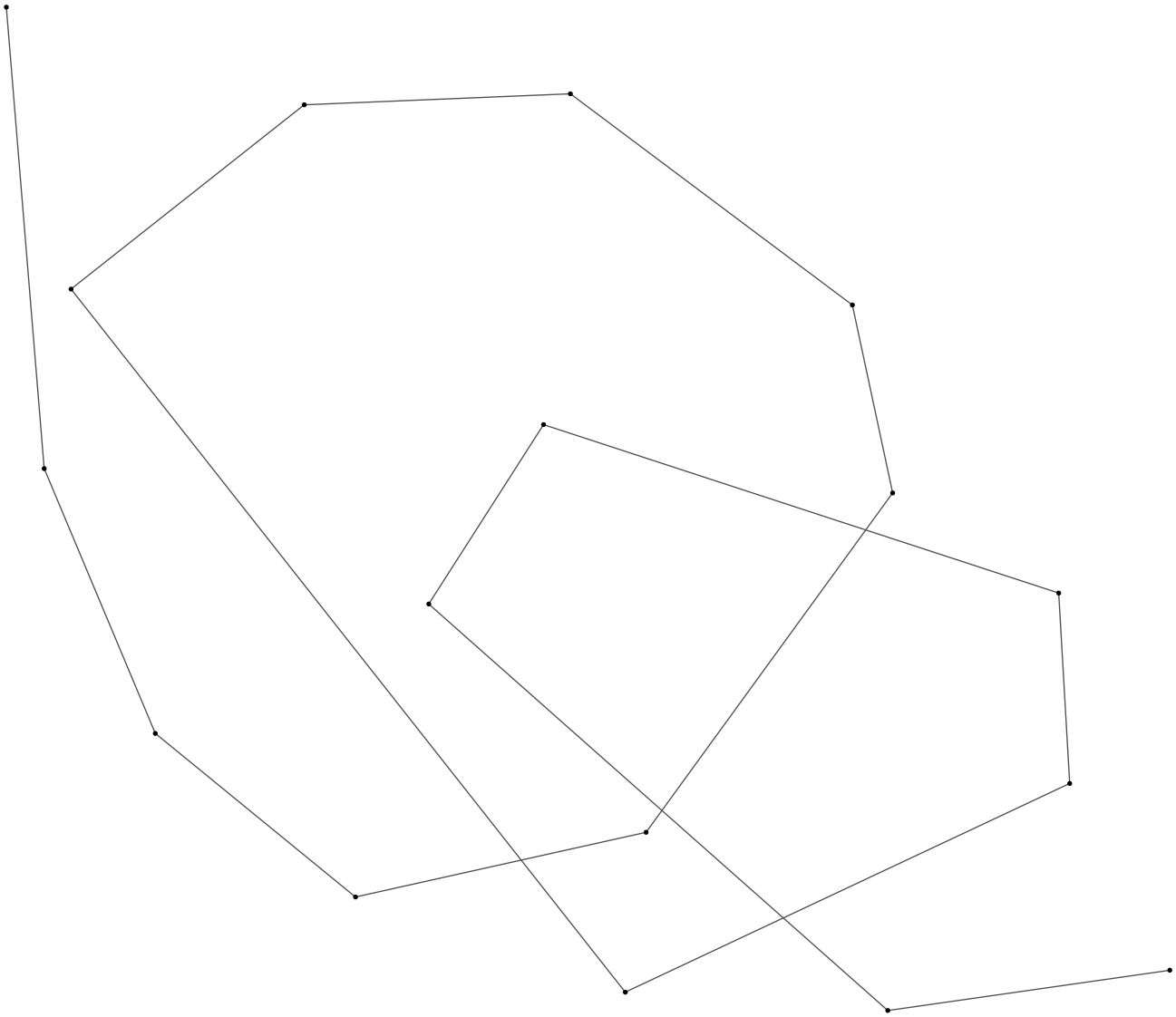
wenigerkrumm7.txt

Punkte: 100
Länge: 5953.856638117417LE
Zeit: 3.69s



Zwei „unmögliche Winkel“

Punkte: 17
Länge: 4967.507282545895LE
Zeit: 0.32



Theoretische Analyse

Ergebnisse

Offensichtlich ist, dass alle Ergebnisse korrekt sind und nicht unbedingt eine unnötig lange Route bilden. Eine offensichtliche schwäche dieses Algorithmus ist jedoch das Verhalten der Route an den Enden dieser. Es ist zu beobachten, dass die Routensegimente zunehmend Länger werden. Dies liegt daran, dass der Algorithmus in seiner Routenkonstruktion Punkte, die in der Nähe des Weges sind, aber nicht verbunden werden „zurücklässt“. Diese müssen am Ende alle „nachgeholt“ werden, um eine Valide Route zu finden. Eine Lösung für dieses Problem bietet die in der Implementierung beschriebene Optimierung der „Sichtbarkeit von Punkten“. Wenn die Punkte nur Relativ kurze Verbindungen eingehen würden sollte die Endroute eine kürzere Gesamtlänge aufweisen. Es wurden mit dem „nearest-neighbour“-Algorithmus auch bessere Ergebnisse erzielt. Diese sind in der folgenden Erweiterung zu sehen.

Laufzeit

Die Laufzeit des Programms hängt von vielen verschiedenen Faktoren ab.

Zunächst wird für jeden Punkt die Distanz zu jedem anderen Punkt berechnet. Da jeder Punkt alle Punkte aufruft kann die Laufzeit der Berechnungen der Abstände mit $O(n^2)$ dargestellt werden.

Da für die Startpunktberechnung der Graham-Scan Algorythmus mit der Laufzeit $O(n\log(n))$ verwendet wird, muss dieser ebenfalls berücksichtigt werden. Das überprüfen auf „unmögliche Winkel“ findet für jeden Punkt der Konvexen Hülle nur einmal statt, deshalb kann dies Nährungsweise mit $O(4\sqrt{n})$ dargestellt werden. Zusammengefasst ist die Laufzeit der Startpunktberechnung also $O(4\sqrt{n}+n\log(n))$

Den größten Teil der Laufzeit nimmt die rekursive Berechnung der verschiedenen Wege in Anspruch. Diesen Teil zu Modellieren ist deutlich schwerer, da er nicht nur von der Variable *n*, sondern auch von der Variable *depth* abhängt. Ohne das Limit der *depth*-Variable kann die Laufzeit der Funktion wie folgt Dargestellt werden:

$$O(n*(\frac{(n-1)}{2}*\frac{(n-2)}{2}*\frac{(n-3)}{2}*...*\frac{1}{2})) \quad => \quad O(n\frac{(n-1)!}{2(n-1)}) \quad => \quad O(\frac{n!}{2(n-1)})$$

Hier wird davon ausgegangen, dass in jeder Verzweigung die Hälfte der Äste valide sind und somit verfolgt werden können.

Nun setzt die *depth* Variable ein Limit für die Variable *n* über dem Bruchstrich, da nur *depth*-viele Wege verfolgt werden.

Sei die Funktion *l* wie Folgt definiert: $l:\to n\{\begin{matrix} depth, n>depth \\ n\leq depth \end{matrix}$ wobei $\{depth\in\mathbb{N}\}$ und $\{n\in\mathbb{N}\}$, dann kann die Oben stehende Bedingung wie folgt umgeschrieben werden und die maximale Laufzeit der Rekursion wie folgt modelliert werden:

$O(l(n)*(\frac{(l(n)-1)}{2}*\frac{(l(n)-2)}{2}*\frac{(l(n)-3)}{2}*...*\frac{1}{2}))$ => $O(\frac{l(n)!}{2^{(n-1)}})$

Die endgültige Laufzeit des Algorithmus ist nun die Summe dieser drei Laufzeiten. Die Gesamtlaufzeit, kann also nährungsweise mit $O(n^2+4\sqrt{n}+n*\log(n)+(\frac{l(n)!}{2^{(n-1)}}))$ modelliert werden, wobei für *l* die oben genannte Definition gilt. Dabei ist zu beachten, dass es sich hierbei um die Worst-Case Laufzeit handelt, da realistischerweise niemals im letzten Zweig die erste Lösung gefunden werden wird. *(Anmerkung zur O-Notation: Diese wurde noch nie verwendet)*

Speicherbedarf

Der Speicherbedarf des Algorithmus ist relativ gering, da nie mehr, als eine Route gleichzeitig betrachtet wird. Auch sind nahezu alle Variablen innerhalb der rekursiven Funktion temporär und werden deshalb automatisch gelöscht, sobald der aktuelle Weg als Invalide gewertet wird.

So werden für die Eingabe *wenigerkrumm7.txt* maximal 14Mib Speicherplatz (in meinen Tests) in Anspruch genommen. Andere Algorithmen, die auf dem Verglichen verschiedener Wege basieren haben einen deutlich höheren Speicherbedarf, da sie diese Wege Speichern müssen.

Die Größen, die hier mit steigendem *n* größer werden sind lediglich die zu speichernden Distanzen und die Routenlänge in den Rekursiven aufrufen. Da alle Vorheringen Teilrouten mit der aktuellen Route gespeichert werden kann der Speicheraufwand dieser mit der Summenformel $\frac{n^2+n}{2}$ dargestellt werden. Der Speicheraufwand der Segmentlänge ist n^2, da hier für jeden Punkt alle weiteren Punkte berechnet und abgespeichert wurden. Alle weiteren hier nicht aufgeführten Größen, wie beispielsweise die Punkte selbst, verfügen über einen Linearen Zusammenhang. Diese werden eher Symbolisch mit *n* Dargestellt. Dadurch ergibt sich ein Speicherbedarf in Abhängigkeit von *n*:

$n+n^2+\frac{n^2+n}{2}=n+n^2+\frac{1}{2}n^2+\frac{1}{2}n=\frac{3}{2}n+\frac{3}{2}n^2$

Wichtig ist auch noch zu beachten, dass die Einträge in den Listen lediglich auf ein Objekt verweisen und es nicht Kopieren. Deshalb spielt es für die Speicherung einer Liste keine Rolle, ob Ein einzelner Punkt 1Kib oder 1Mib an Speicherbedarf hat.

Quellcode

```python
###   point.py
class P: #point
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.distancemap = {}

    def calcDistanceMap(self, points):
        for point in points:
            self.distancemap[point] = self._lenghtTo(point)

    def _lenghtTo(self, point2):
        x1 = self.x
        x2 = point2.x
        y1 = self.y
        y2 = point2.y
        return sqrt(((x1-x2)**2) + ((y1-y2)**2))

    def lenghtTo(self, point2):
        return self.distancemap[point2]

### main.py
def main(filename, depth):
    points = []

    with open(filename, "r") as file:
        lines = file.read().splitlines()
        for line in lines:
            xy = line.split(" ")
            points.append(
                P(float(xy[0]), float(xy[1]))
            )

    for point in points:
        point.calcDistanceMap(points)

    endLen = len(points)

    path = func.getStartPoints(points)

    for p in path:
        points.remove(p)

    return func.step(path, points, endLen, depth)
```

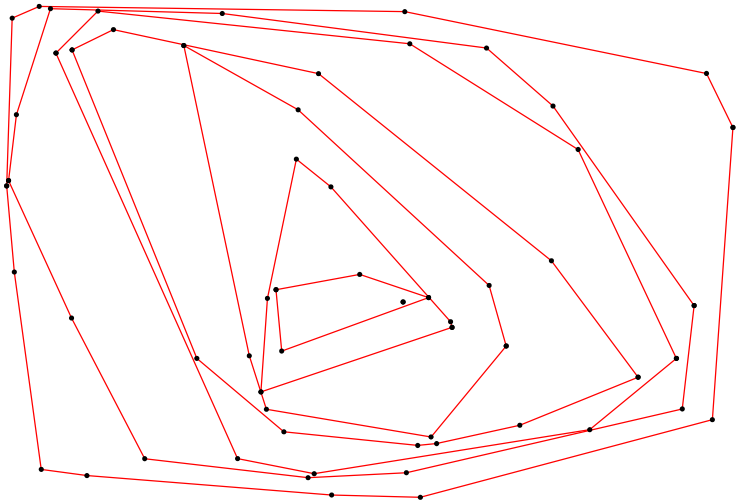
```python
### Winkelvalidierung
def isValidAngleList(path, distancemap):
    if not (len(path) <2):
        A:P = path[-2]
        B:P = path[-1]

        if(A.x==B.x):
            if(A.y<B.y):
                return list(filter(lambda C: C[1].y>=B.y, distancemap))
            elif(A.y>B.y):
                return list(filter(lambda C: C[1].y<= B.y, distancemap))
            else: return True #Both points are the same
        elif(A.y==B.y):
            if(A.x<B.x):
                return list(filter(lambda C: C[1].x>= B.x, distancemap))
            elif(A.x>B.x):
                return list(filter(lambda C: C[1].x<= B.x, distancemap))
        elif(A.x>B.x):
            m = -((A.x-B.x)/(A.y-B.y))
            b = B.y - (m*B.x)

            if (A.y > B.y):
                return list(filter(lambda C: (C[1].y <= m*C[1].x+b), distancemap))
            elif (A.y < B.y):
                return list(filter(lambda C: (C[1].y >= m*C[1].x+b), distancemap))

        elif(A.x < B.x):
            m = -((A.x-B.x)/(A.y-B.y))
            b = B.y - (m*B.x)

            if (A.y > B.y):
                return list(filter(lambda C: (C[1].y <= m*C[1].x+b), distancemap))
            elif (A.y < B.y):
                return list(filter(lambda C: (C[1].y >= m*C[1].x+b), distancemap))

            return []
        else: return []
    else: return distancemap
```

```python
### rekursive Wegfindungsfunktion
def step(path:list[P], points:list[P] ,endLen:int, depth:int=2):
    if (len(path) == endLen):
        vis.generateAndShowWithSystemViewerLineSVG(path)
        print("points: "+str(endLen))
        print("total lenght: "+str(getTotalLenght(path)))
        return 0

    lastpoint = path[-1]
     distancemap = list(map(lambda x: [lastpoint.lenghtTo(x), x], points.copy()))
     distancemap = isValidAngleList(path, distancemap)


     if(len(distancemap) == 0):
        return -1

     distancemap = sorted(distancemap, key=lambda x: x[0])

     for point in first(depth, distancemap):
        newPath = path.copy()
        newpoints = points.copy()
        newPath += [point[1]]
        newpoints.remove(point[1])
        c = step(newPath, newpoints, endLen, depth)
        if (c == -1):
            continue
        elif (c==0):
            return 0
     return -1
```

Ein anderer Algorithmus: Konvexe Hüllen

In der „nearest-neighbour“-Lösung werden Konvexe hüllen verwendet, um optimale Startpunkte zu finden. Das interessante an konvexen Hüllen ist, dass nur eher selten Spitze Winkel auftreten, was eine Lösungsidee aufwirft:

Die Punkte werden in dem Muster einer Spirale verbunden. Um sie in einer Spirale zu verbinden wird die konvexe Hülle der Punkte berechnet. Anschließend wird die konvexe Hülle der Restmenge gebildet und danach wieder aus der Restmenge die Konvexe Hülle berechnet und … . Visuell Dargestellt ergibt sich eine reihe an konvexen Hüllen, die alle ineinander geschachtelt sind. Es müssen lediglich diese Verbunden werden. Bei genauerem Betrachten fällt auf, dass Punkte, die von der Strecke aus einfach erreicht werden können dieser nicht angehören, da sie nicht dieser konvexen Hülle angehört. Mit einer Berechnung wird überprüft, ob es sinnvoll ist diesen Punkt der Strecke hinzuzufügen. Nun kann die konvexe Hülle auch einen Winkel <90° vorweisen. An diesen Stellen müssen die verschiedenen Layer miteinander verbunden werden um den >90° Winkel zu gewährleisten. Auch werden Verbindungen mit Punkten aus den vorherigen Layern in Betracht gezogen. Dabei wird der Punkt aus der vorherigen konvexen Hülle gelöscht, sofern dadurch kein weiterer Spitzer Winkel entsteht.



Dieser Ansatz ist besonders gut für Datensätze, deren Punkte alle Ähnlich weit voneinander entfernt sind, geeignet, da so keine sehr kleinen Winkel durch „Ausreißer“ in den konvexen Hüllen. (Wie beispielsweise einem gleichmäßigen Raster an Punkten). Jedoch scheitert er schnell an Eingaben, die auf der Basis eines Musters arbeiten, in dem größere Lücken zu finden sind. (z.B. wenigerkrumm3.txt) Vor allem aber spricht gegen diese Lösungsidee die unnötig langen Wege, die generiert werden. Dies ist ebenfalls der Hauptgrund, aus dem dieser Algorithmus verworfen wurde.

Laufzeit und Speicher

Einen klaren Vorteil hat dieser Algorithmus jedoch verglichen mit der „nearest-neighbour“-Lösung. Die Laufzeit ist bedeutend geringer, da nicht so viele Iterationen durchgeführt werden müssen, wie in der „nearest-neighbour“-Lösung rekursive aufrufe getätigt werden wenn angenommen wird, dass jede konvexe Hülle etwa $4\sqrt{n}$ Punkte der Punktmenge beinhaltet. Der Speicheraufwand sollte höher ausfallen, da das Speichern der verschiedenen konvexen Hüllen mehr Speicher beanspruchen sollte. (eher eine Theorie und von der Implementierung stark abhängig)

Erweiterung: Infinity-Search

Wird nochmal die Anfangsveranschaulichung mit dem Baumdiagramm betrachtet stellt sich die Frage, weshalb nur die erste Lösung zurückgegeben wird. Wenn der Baum weiter Abgesucht wird, dann kann der bisherige Weg bis hin zu einem optimalen Weg abgesucht werden. Diese Berechnung kommt einer Bruteforce-Suche recht nahe, da sie für den gegebenen Startpunkt alle möglichen Wege berechnen kann. Auch benötigt sie tatsächlich $O(n^2+(n+n\log(n))+(\frac{l(n)!}{2(n-1)}))$ Zeit, um den gesamten Baum abgefahren zu haben. Die Suchtiefe wurde für diese Implementierung auf 5 gestellt, um mehr Wege zu berücksichtigen.

Nochmal zurück zum Sachverhalt

„Ein Anteil von Antons Arbeit ist das Abfliegen aller Außenstellen in Australien. Auf seiner Tour muss er vorgegebene Orte besuchen, kann sich aber aussuchen, in welcher Reihenfolge dies geschieht.“ heißt es in der Ausgabenstellung. Dies bedeutet, dass Anton eine Routen Optimierung während dem Flug durchführen könnte. Wird der Baum mit dem „Infinity-Seach“ überarbeitet und für jeden Validen Weg überprüft, ob dieser besser als der bisherig beste Weg war, kann eine Ähnlichkeit der beiden Wege festgestellt werden. Da das rekursive Suchverhalten des Algorithmus vor allem das ende des Baumes optimiert und nicht die Anfangsstrecke, kann Anton den ersten Teil der Strecke abfliegen, während der Algorithmus die Restliche strecke optimiert.

Implementierung

Da diese gesamte Erweiterung eher aus einer fixen Idee heraus entstanden ist, wurde bei der Implementierung (vor allem bei dem networking code!) nicht sehr auf die Effizienz und die Stabilität des Algorithmus geachtet. Es steht hier im Vordergrund, was möglich ist, und nicht wie effizient dies ist.

Um die Routen dynamisch anzuzeigen wird eine Verbindung zu einem Client-Webbrowser aufgebaut und die neuen Routen SVG’s zu diesem Browser geschickt. Dazu werden die Module *flask* und $flask-socketio$ verendet, da diese einen relativ Stabilen HTTP bzw. Websocket Server aufbauen können.

Da Python kein direktes asynchrones Handling der Flask/Socketio Instanz und des Algorithmus ermöglicht wird mit dem *threading* Modul ein zweiter Thread für den Algorithmus erstellt. Dieser kann über anonyme Funktionen (oder callbacks) auf die Funktionen des Flask/Socketio Servers zugreifen und so mit dem Client kommunizieren.

Damit eine unendliche suche möglich wird, muss das Rekursionsverfahren des Algorithmus leicht verändert werden. Im falle eines Validen Weges wird nach dem selben Prinzip, wie im falle eines Invaliden Weges einfach weiter gesucht.

Sobald eine valide Route gefunden wurde wird der String der SVG-Datei gebildet und an den Client gesendet. Dieser rendert anschließend den String. Effizienter wäre es nur die Punktkoordinaten an

den Browser zu schicken und diesen das SVG selber zusammenfügen lassen. Alternativ könnte das SVG-Zusammensetzten auch von einem weiteren Thread übernommen werden. *(Aufgrund von Faulheit keine Implementierung)*

Ausführung

Zur Ausführung der Erweiterung müssen die Python Module flask, flask_socketio und svgwrite zwingend installiert sein. Diese können über den pip befehl problemlos installiert werden.

Das Programm wird über den Befehl

```
python3 main.py --file /path/zur/eingabe.txt --port <PORT>
```

gestartet und kann mit ^C abgebrochen werden, sobald eine gewünschte Route gefunden wurde.

Es ist dennoch möglich, dass bei der Ausführung des Programms Fehler auftreten. (vor allem mit dem flask_socketio Modul) Deshalb wurde ein Video mit einer Beispielausführung des Programms in der Zip mit hochgeladen.

Theoretische Analyse

Die Ergebnisse zeigen, dass die Route nahezu immer optimiert werden kann. (Ausnahme: *wenigerkrumm*1.*txt ; wenigerkrumm*2.*txt*) auch zeigen sie uns, dass die erste Gefundene Route (also die der „Hauptlösung“) nicht nur eine von vielen Routen ist, sondern eine Route relativ weit oben in einer Hirarchie nach Routenlänge.

Die Laufzeit beträgt wie schon gesagt die selbe wie der vorherige Algorithmus mit der Ausnahme, dass diesmal die gesamte Laufzeit tatsächlich gilt, da alle Zweige abgefahren werden.

Der Speicherbedarf des Algorithmus wird nur durch die Serverinstanzen ergänzt, die jedoch nicht von der eingegebenen Menge *n* abhängt. Je nach Browser nimmt der Client jedoch eine Menge an Speicherbedarf ein, die indirekt auf die Erweiterung zurückzuführen sind.

Quellcode

Da Prinzipiell der selbe Algorithmus für die Erweiterung gewählt wurde gibt es keinen „besonderen Code“, der eine Explizite Darstellung wert ist. Der einzige als „besonders“ zu erachtende Code wäre vermutlich der Networking / Threading Code. Da dieser jedoch für die Lösung unerheblich ist, wurde dieser hier weggelassen.

Der Code für die Erweiterung ist in dem Ordner erweiterung (kreativ!) zu finden.