Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 66993

Bearbeiter dieser Aufgabe:
Tim Krome

12. April 2023

Lösungsidee

Gegeben sind die Positionen von n Außenstellen, im Folgenden als „Stationen“ bezeichnet. Jede Position setzt sich aus zwei Koordinaten zusammen. Ich lege fest, dass es sich bei der ersten Koordinate um die x- und bei der zweiten Koordinate um die y-Koordinate handelt. Anton soll alle Stationen abfliegen, dabei darf sein Abbiegewinkel 90° nicht überschreiten (wie in der Aufgabe gezeigt).

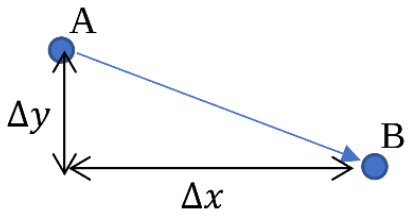
Die n Stationen lassen sich grundsätzlich in einem Graphen modellieren. Durch die Bedingung, dass der Abbiegewinkel <= 90° sein muss, handelt es sich aber nicht um einen klassischen Graphen mit gerichteten Kanten. Die Menge an Stationen, die Anton erreichen kann, ist nicht nur von seiner aktuellen Position abhängig, sondern auch von seiner vorherigen Position abhängig.

Um eine Lösung für die Aufgabe zu entwickeln, braucht man zunächst eine Methode zum Berechnen von Antons Abbiegewinkel.

Berechnen des Abbiegewinkels

Fliegt Anton von Punkt A zu Punkt B, dann bewegt er sich auf der Geraden AB, genauer gesagt bewegt er sich auf dem Vektor $\overrightarrow{AB}$.

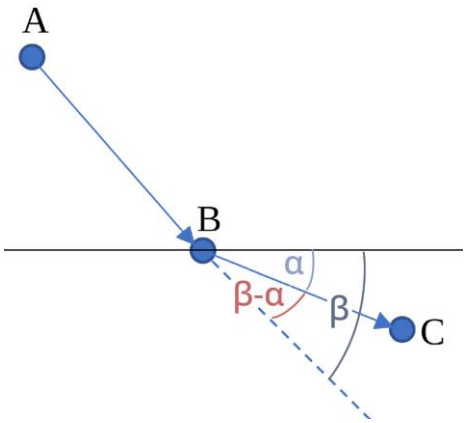
Bei jeder Gerade lässt sich mit der Formel $\alpha = \text{atan}\left(\frac{\Delta y}{\Delta x}\right)$ der Steigungswinkel berechnen (für $\Delta x \neq 0$). In unserem Fall ist $\Delta y = B_Y - A_Y$ und $\Delta x = B_X - A_X$. (A_X bezeichnet die x-Koordinate von A, A_Y die y-Koordinate von A). Schaubild zur Veranschaulichung:



Der Steigungswinkel der Geraden AB lässt sich also mit der Formel $\alpha = \text{atan}\left(\frac{B_Y - A_Y}{B_X - A_X}\right)$ berechnen (für $B_X \neq A_X$) . Im Sonderfall $B_X = A_X$ fliegt Anton orthogonal zur x-Achse und die Gerade AB hat einen Steigungswinkel von $\alpha = 90°$.

Zum Ermitteln des Abbiegewinkels ist es außerdem notwendig festzustellen, ob Anton nach „links“ (in negative x-Richtung) oder nach „rechts“ (in positive x-Richtung) fliegt, also ob $\overrightarrow{AB}$ nach rechts oder nach links zeigt. Dies kann geschehen durch Vergleichen der x-Koordinaten von A und B.
Wenn $B_X = A_X$, dann fliegt Anton weder nach rechts noch nach links. Damit dieser Sonderfall trotzdem in meine Modellierung des Szenarios passt, lege ich für ihn folgendes fest: Wenn Anton nach „oben“ fliegt (also in positive y-Richtung), dann wird das behandelt, wie wenn er nach rechts fliegt. Wenn Anton nach „unten“ fliegt (also in negative y-Richtung), dann das behandelt, wie wenn er nach links fliegt.

Fliegt Anton von Punkt A zu Punkt B und biegt dann zu Punkt C ab, dann bewegt er sich zuerst auf der Geraden AB mit dem Steigungswinkel α, danach auf der Geraden BC mit dem Steigungswinkel β. Sein Abbiegewinkel entspricht dem Schnittwinkel von AB und BC, also $abs(\beta - \alpha)$. Schaubild zur Veranschaulichung:



Wenn die Vektoren $\overrightarrow{AB}$ und $\overrightarrow{BC}$ beide nach rechts oder beide nach links zeigen, dann lässt sich also die Formel $abs(\beta - \alpha)$ zum Bestimmen von Antons Abbiegewinkel verwenden. Zeigt jedoch ein Vektor nach rechts und der andere nach links, dann muss dies miteinberechnet werden: In dem Fall ist der Schnittwinkel mit der Formel $180 - abs(\beta - \alpha)$ bestimmbar.

Finden des besten Startpunkts

Zum Ermitteln des geeignetsten Startpunkts für Antons Weg erstelle ich zuerst eine Tabelle. In dieser wird für jede Station gespeichert, welche Möglichkeiten Anton hat, sie zu erreichen (also von welchen Stationen er kommen muss). Es handelt sich bei der Tabelle also um eine Art umgedrehte Adjazenzliste.
Der optimale Startpunkt ist die Station, für die es am wenigsten Möglichkeiten gibt, sie zu erreichen: Wird diese Station als Startpunkt genommen, dann muss sie später nicht mehr abgeflogen werden. Hierdurch verringert sich die Wahrscheinlichkeit, in einer Sackgasse zu landen, da alle anderen Stationen besser erreichbar sind.

Finden eines Weges

Zum Finden eines möglichen Weges, der alle Stationen abdeckt, verwende ich eine klassische Tiefensuche, die alle möglichen Positionen beschreitet, bis ein Weg gefunden wurde, der alle Stationen abdeckt. Die Tiefensuche kann allerdings eine besser Evaluierungszeit bieten oder einen Weg mit kürzerer Strecke ergeben, wenn bei der Auswahl der als nächstes zu beschreitenden Position bestimmte Positionen priorisiert werden, also heuristisch vorgegangen wird.

Um eine kürzere Strecke zu erhalten, muss die Tiefensuche stets die Position priorisieren, die der aktuellen Position am nächsten ist. Dies sorgt allerdings dafür, dass die durchschnittliche Evaluierungszeit deutlich anwächst.
Um eine bessere Evaluierungszeit zu erhalten, muss die Tiefensuche die Position priorisieren, für die es am wenigsten Möglichkeiten gibt, sie zu erreichen – hierdurch sinkt die Wahrscheinlichkeit, dass die Tiefensuche in einer Sackgasse landet, da eine schlecht erreichbare Position, die besucht wurde, später nicht mehr besucht werden muss. Diese Heuristik sorgt allerdings dafür, dass Antons Weg deutlich länger werden kann.

Kann immer ein Weg gefunden werden?

Beweis durch Gegenbeispiel:

Angenommen, wir haben vier Stationen auf den vier Punkten A(0|0), B(1|0), C(0|1) und B(0|-1). Egal, welche Station als Startpunkt festgelegt wird: Um alle Stationen abzufliegen, muss Anton irgendwann um 135° abbiegen. Es ist somit nicht immer möglich, einen Weg zu finden, der die Bedingung (Abbiegewinkel <= 90°) erfüllt.

Umsetzung

Die Lösungsidee wird in Python 3.10 implementiert. Verwendete Python libraries:
- *math:* Verwendet für mathematische Operationen (atan, Quadratwurzel), Python standard library
- *matplotlib:* Verwendet zum Visualisieren der Flugroute in einem Plot

Einlesen der Stationen

In der Eingabe enthält jede Zeile die Koordinaten einer Station, getrennt durch ein Leerzeichen.

Die Eingabedatei wird vom Nutzer als erstes Kommandozeilenargument angegeben und vom Programm eingelesen, anschließend wird eine Liste erstellt, in der die Koordinaten der Stationen als Tupel gespeichert sind:

```
# Einlesen der Daten
# sys.argv[1] ist das erste Kommandozeilenargument
with open(sys.argv[1], "r") as f:
    data = f.read().split("\n")

# Entfernen leerer Zeilen aus data
data = list(filter(lambda x: x != "", data))

stations = [(float(i.split(" ")[0]), float(i.split(" ")[1])) for i in
data]
```

Berechnen des Abbiegewinkels

Ich definiere die Funktion `steigungswinkel(punkt0 : tuple, punkt1 : tuple)`, die den Steigungswinkel einer durch punkt0 und punkt1 gehenden Gerade zurückgibt und außerdem zurückgibt, ob sich Anton auf dem Koordinatensystem nach links oder nach rechts bewegt, wenn er von punkt0 zu punkt1 fliegt. Zum Berechnen des Steigungswinkels nutze ich die Funktion *math.atan*:

```
angle = math.atan((punkt1[1] - punkt0[1]) / (punkt1[0] - punkt0[0])) *
180/math.pi
```

Da *math.atan* den Tangens^{-1} im Bogenmaß zurückgibt, wird das Ergebnis von *math.atan* mit 180/pi multipliziert (Umrechnung in Gradmaß).

Anschließend definiere ich die Funktion `abbiegewinkel(punkt0 : tuple, punkt1 : tuple, punkt2 : tuple)`, die den Abbiegewinkel von Anton (wenn er von punkt0 kommt und bei punkt1 nach punkt2 abbiegt) berechnet und zurückgibt:

```
def abbiegewinkel(punkt0, punkt1):
    alpha, alpha_dir = steigungswinkel(punkt0, punkt1)
    beta, beta_dir = steigungswinkel(punkt1, punkt2)

    schnittwinkel = abs(beta – alpha)
    if alpha_dir != beta_dir:
        schnittwinkel = 180 - schnittwinkel


  return schnittwinkel
```

Durchführen der Tiefensuche

Zum Erstellen der umgekehrten Adjazenzliste (in der für jedes Feld gespeichert ist, über welche Felder es erreicht werden kann) verwende ich drei verschachtelte for-Schleifen, die alle durch die Stationen iterieren und jede Möglichkeit, die es zum Erreichen einer Station gibt, überprüfen:

```
adjazenzen = {}
for station0_id in range(len(stations)):
    for station1_id in range(len(stations)):
        if station1_id == station0_id:
            continue
        for station2_id in range(len(stations)):
            if station2_id not in adjazenzen:
                adjazenzen[station2_id] = []
            if station2_id == station1_id:
                continue
            angle = abbiegewinkel(
         stations[station0_id], stations[station1_id], stations[station2_id]
            )
            if angle <= 90:
                adjazenzen[station2_id].append((station0_id, station1_id))
```

Ich definiere anschließend eine Klasse `Node(station_id, visited, strecke, abstand)`, die zum Speichern einer Position auf Antons Weg dient. Die Klasse verfügt über eine `successors` Funktion, die alle von der Position aus erreichbaren Positionen als *Node*-Objekte zurückgibt. Die Funktion berechnet nicht jedes Mal aufs Neue, welche Positionen erreichbar sind, sondern macht von der vorab berechneten umgekehrten Adjazenzliste *adjazenzen* gebraucht.

Zum Finden eines Weges erstelle ich eine `tiefensuche` Funktion, die als Argument eine Liste *paths* mit den Anfangspositionen nimmt. Ich entscheide mich für eine iterative, keine rekursive Implementierung der Tiefensuche. In der Funktion wird in einer *while True*-Schleife bei jeder Iteration das erste Element aus *paths* genommen:

```
path = paths.pop(0)
```

Dann werden mit der *path.successors* alle von *path* aus erreichbaren Positionen ermittelt, der verwendeten Heuristik nach sortiert und vorne an paths angehängt.
Die *while True*-Schleife wird abgebrochen, sobald ein Weg gefunden wurde der alle Stationen abdeckt, oder alle möglichen Wege begangen wurden:

```
if len(path.visited) == len(stations):
    return path #Dieser Code bricht ab sobald ein Weg gefunden wurde
```

Die *tiefensuche*-Funktion wird insgesamt zweimal ausgeführt:

```
schlechteste_erreichbarkeit = min(
    adjazenzen.keys(), key=lambda station_id : len(adjazenzen[station_id])
)
path = tiefensuche(
    [Node(schlechteste_erreichbarkeit)], prioritise_strecke=True
)
```

```
if path is None:
    path = tiefensuche(
        [Node(schlechteste_erreichbarkeit)], prioritise_strecke=False
    )
```

Beim Durchführen der ersten Tiefensuche wird eine Heuristik verwendet, die für einen kürzeren Weg sorgt. Findet diese innerhalb von 1000 Iterationen kein Ergebnis, wird die erste Tiefensuche abgebrochen. Es wird eine zweite Tiefensuche gestartet, diese verwendet eine andere Heuristik, die die durchschnittliche Evaluierungszeit verringert und die Länge des Wegs nicht berücksichtigt. Auf die Art wird in angemessener Zeit ein Weg gefunden (wenn es einen geben sollte), und es wird auch versucht, die Strecke möglichst kurz zu halten, so wie in der Aufgabenstellung gefordert.

Ausgeben und Visualisieren des Ergebnisses

Wenn ein Weg gefunden wird, dann werden der Weg und dessen Länge in der Konsole ausgegeben. Außerdem wird der Weg mit *matplotlib* aus Plot visualisiert und ausgegeben, sofern das Library installiert ist:

```
if path is None:
    print("Es gibt keinen Weg, der die Bedingung erfüllt.")
else:
    print("Streckenlänge:", path.strecke, "km")
    str_visited, visited = output_path(path)
    print("Weg:", str_visited)

    # Visualisierung des gefundenen Weges:
    try:
        import matplotlib.pyplot as plt

        x, y = list(zip(*visited))
        plt.plot(x, y, '-o', markersize=3)
        ...
        # Plot anzeigen
        plt.show()
    except ModuleNotFoundError:
        print("matplotlib nicht installiert“)
```

Laufzeitanalyse

Im Programm kommen an mehreren Stellen Schleifen vor. Der Programmteil, der die umgekehrte Adjazenzliste bestimmt, besteht aus drei verschachtelten for-Schleifen und hat somit eine Laufzeit von n^3, n = Anzahl Außenstellen.

Die Tiefensuche wird so lange ausgeführt, bis ein Weg gefunden wurde – oder bis alle möglichen Wege überprüft wurden und keiner gefunden wurde, der alle Außenstellen abdeckt. Bei der Bestimmung der Laufzeit muss berücksichtigt werden, dass der Startpunkt der Tiefensuche festgelegt ist und somit nur noch n-1 Stationen besucht werden müssen.
Im Worst-Case-Szenario enden alle Wege kurz vor Vollendung in einer Sackgasse. Die Laufzeit der Tiefensuche ist im Worst-Case-Szenario (n-1)!:

Nach Besuchen der ersten Außenstelle gibt es noch n-2 Außenstellen, die besucht werden können, nach Besuchen der zweiten Außenstelle gibt es noch n-3 Außenstellen, die besucht werden können usw.

Die for-Schleife in der output_path-Funktion hat eine lineare Laufzeit von n. Die Gesamtlaufzeit des Programms beträgt in O-Notation also O((n-1)!), da eine Fakultätsfunktion asymptotisch schneller wächst als jede Polynomfunktion.

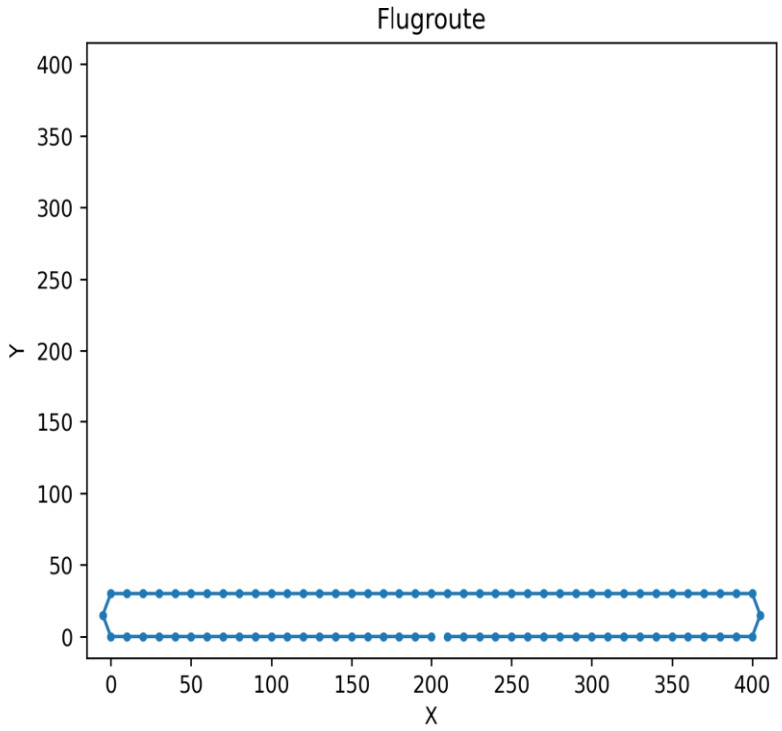
Beispiele

Die in den Beispielen 1 bis 7 verwendeten Eingabedateien stammen von der BWinf-Webseite. Die Eingabedateien und die vollständigen Ausgaben meines Programms sind in Ordner *Aufgabe 1* zu finden.
Für das Messen der Evaluierungszeit wurde das Programm auf einem Gerät mit 16 GB RAM ausgeführt.

Beispiel 1

Eingabedatei:
wenigerkrumm1.txt

Ausgabe:
```
Streckenlänge: 853.2455532033676 km
Weg: (200.0, 0.0) -> (190.0, 0.0) -> (180.0, 0.0) -> (170.0, 0.0) -> (160.0,
0.0) -> (150.0, 0.0) -> (140.0, 0.0) -> (130.0, 0.0) -> (120.0, 0.0) -> (110.0,
0.0) -> (100.0, 0.0) -> (90.0, 0.0) -> (80.0, 0.0) -> (70.0, 0.0) -> (60.0,
0.0) -> (50.0, 0.0) -> (40.0, 0.0) -> (30.0, 0.0) -> (20.0, 0.0) -> (10.0, 0.0)
-> (0.0, 0.0) -> (-5.0, 15.0) -> (0.0, 30.0) -> (10.0, 30.0) -> (20.0, 30.0) ->
(30.0, 30.0) -> (40.0, 30.0) -> (50.0, 30.0) -> (60.0, 30.0) -> (70.0, 30.0) ->
(80.0, 30.0) -> (90.0, 30.0) -> (100.0, 30.0) -> (110.0, 30.0) -> (120.0, 30.0)
-> (130.0, 30.0) -> (140.0, 30.0) -> (150.0, 30.0) -> (160.0, 30.0) -> (170.0,
30.0) -> (180.0, 30.0) -> (190.0, 30.0) -> (200.0, 30.0) -> (210.0, 30.0) ->
(220.0, 30.0) -> (230.0, 30.0) -> (240.0, 30.0) -> (250.0, 30.0) -> (260.0,
30.0) -> (270.0, 30.0) -> (280.0, 30.0) -> (290.0, 30.0) -> (300.0, 30.0) ->
(310.0, 30.0) -> (320.0, 30.0) -> (330.0, 30.0) -> (340.0, 30.0) -> (350.0,
30.0) -> (360.0, 30.0) -> (370.0, 30.0) -> (380.0, 30.0) -> (390.0, 30.0) ->
(400.0, 30.0) -> (405.0, 15.0) -> (400.0, 0.0) -> (390.0, 0.0) -> (380.0, 0.0)
-> (370.0, 0.0) -> (360.0, 0.0) -> (350.0, 0.0) -> (340.0, 0.0) -> (330.0, 0.0)
-> (320.0, 0.0) -> (310.0, 0.0) -> (300.0, 0.0) -> (290.0, 0.0) -> (280.0, 0.0)
-> (270.0, 0.0) -> (260.0, 0.0) -> (250.0, 0.0) -> (240.0, 0.0) -> (230.0, 0.0)
-> (220.0, 0.0) -> (210.0, 0.0)
```



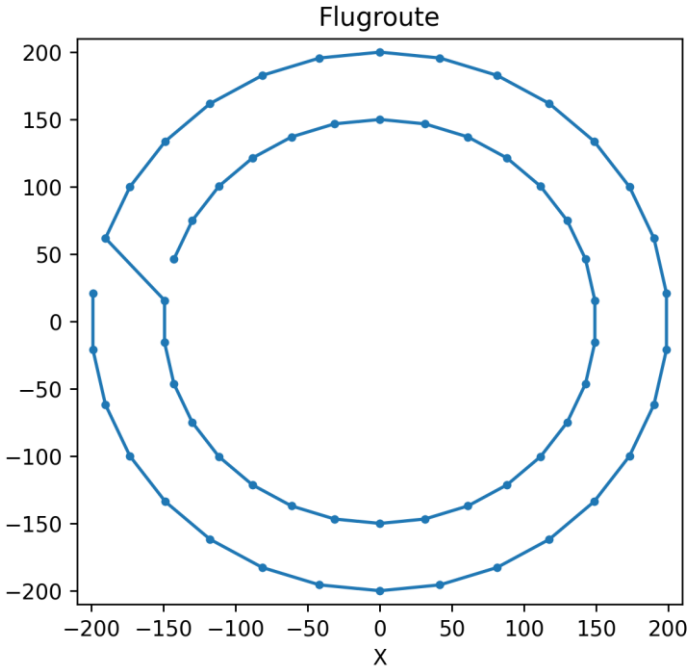
Evaluierungsdauer:
3,5 Sekunden

Erläuterung:
In Beispiel 1 bis 4 kann das Programm eine kurze Route finden, deren Strecke nahe an der kürzesten Route dran ist.

Beispiel 2

Eingabedatei:
wenigerkrumm2.txt

Ausgabe (gekürzt):
```
Streckenlänge: 2183.6622677032137 km
Weg: (-142.658477, 46.352549) -> (-129.903811, 75.0) -> (-111.471724,
100.369591) -> (-88.167788, 121.352549) -> (...) -> (-117.55705, -161.803399)
-> (-148.628965, -133.826121) -> (-173.205081, -100.0) -> (-190.211303, -
61.803399) -> (-198.904379, -20.905693) -> (-198.904379, 20.905693)
```

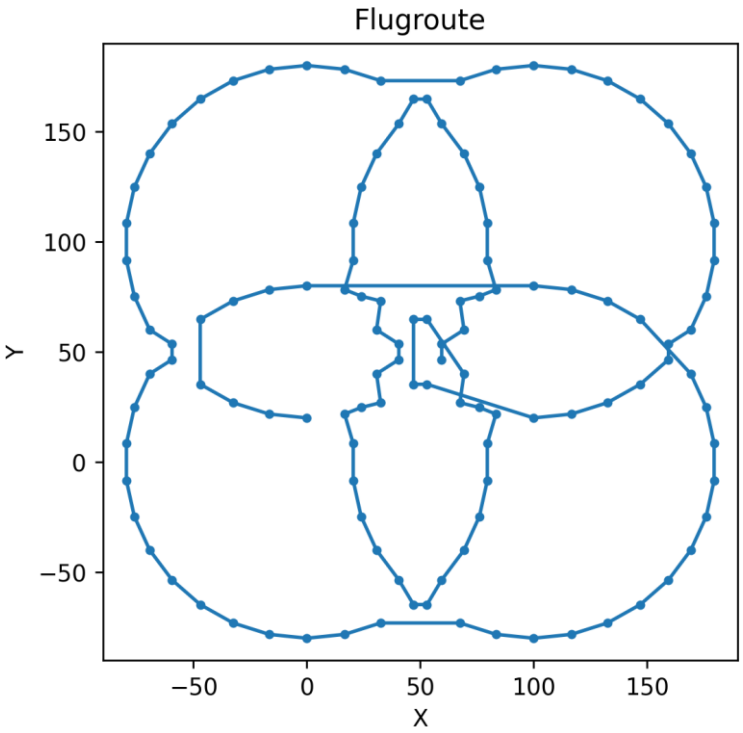


Evaluierungsdauer:
2,3 Sekunden

Beispiel 3

Eingabedatei:
wenigerkrumm3.txt

Ausgabe (gekürzt):
```
Streckenlänge: 1978.7913030438574 km
Weg: (59.451586, 46.469551) -> (59.451586, 53.530449) -> (69.282032, 60.0) ->
(...) -> (-16.632935, 78.251808) -> (-32.538931, 73.083637) -> (-47.02282,
64.72136) -> (-47.02282, 35.27864) -> (-32.538931, 26.916363) -> (-16.632935,
21.748192) -> (0.0, 20.0)
```



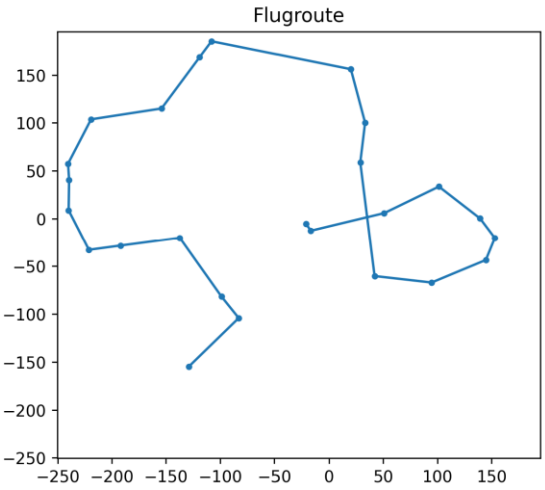
Evaluierungsdauer:
6,3 Sekunden

Beispiel 4

Eingabedatei:
wenigerkrumm4.txt

Ausgabe:
```
Streckenlänge: 1243.4096728129146 km
Weg: (-20.971208, -5.637107) -> (-16.72313, -12.689542) -> (51.00814, 5.769601)
-> (101.498782, 33.484198) -> (139.446709, 0.233238) -> (153.130159, -20.36091)
-> (144.832862, -43.476284) -> (94.789917, -67.087689) -> (42.137753, -
60.319863) -> (28.913721, 58.69988) -> (33.379688, 100.161238) -> (20.212169,
156.013261) -> (-107.988514, 185.173669) -> (-119.026308, 168.453598) -> (-
154.088455, 115.022553) -> (-219.148505, 103.685337) -> (-240.369194,
57.426131) -> (-239.414022, 40.427118) -> (-239.848226, 8.671399) -> (-
221.149792, -32.862538) -> (-191.716829, -28.360492) -> (-137.317503, -
20.146939) -> (-98.760442, -81.770618) -> (-82.864121, -104.1736) -> (-
129.104485, -155.04164)
```

Evaluierungsdauer:
6,3 Sekunden

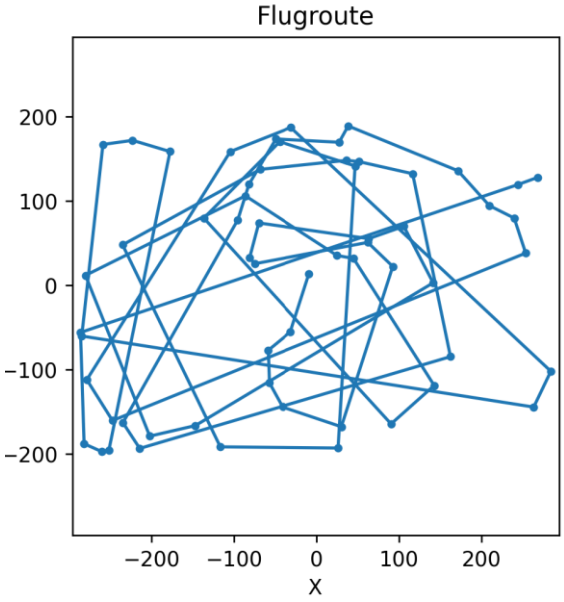


Evaluierungsdauer:
1,6 Sekunden

Beispiel 5

Eingabedatei:
wenigerkrumm5.txt

Ausgabe (gekürzt):
```
Streckenlänge: 8691.64393050238 km
Weg: (-8.936916, 13.543851) -> (-31.548604, -55.223912) -> (-58.684205, -
76.988884) -> (...) -> (-286.024059, -55.955204) -> (244.228552,
119.192512) -> (267.845908, 127.627482)
```

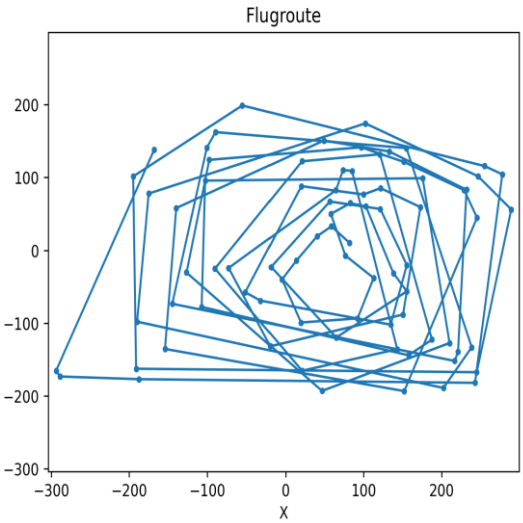


Evaluierungsdauer:
2,72 Sekunden

Beispiel 6

Eingabedatei:
enigerkrumm6.txt

Ausgabe (gekürzt):
```
Streckenlänge: 11739.204532348716 km
Weg: (81.740403, 10.276251) -> (58.71662, 32.83593) -> (40.327635,
19.216022) -> (14.005617, -14.015334) -> (-4.590656, -40.067226) ->
(19.765322, -99.2364) -> (...) -> (-189.988471, -98.043874) -> (-
175.118239, 77.842636) -> (102.223372, 174.201904) -> (246.621634,
101.705861) -> (289.298882, 56.051342) -> (245.020791, -167.448848) -> (-
191.216327, -162.689024) -> (-194.986965, 101.363745) -> (-55.091518,
198.826966) -> (255.134924, 115.594915) -> (277.821597, 104.262606) ->
(242.810288, -182.054289) -> (-187.485329, -177.031237) -> (-288.744132,
-173.349893) -> (-293.833463, -165.440105) -> (-167.994506, 138.195365)
```

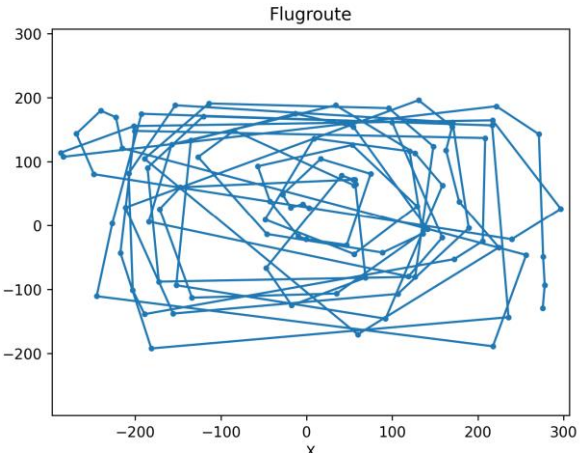


Evaluierungsdauer:
3,7 Sekunden

Beispiel 7

Eingabedatei:
wenigerkrumm7.txt

Ausgabe (gekürzt):
```
Streckenlänge: 15455.590442508 km
Weg: (3.152113, 27.10389) -> (-4.434919, 33.164884) -> (-18.316063,
27.75586) -> (-27.911955, 48.326745) -> (16.573231, 104.020979) ->
(74.8875, 80.586458) -> (...) -> (278.105364, -93.771765) -> (275.793495,
-129.415477)
```



Evaluierungsdauer:
10,5 Sekunden

Erläuterung:
Das Programm kann trotz der langen Eingabe (100 Stationen) in wenigen Sekunden einen Weg finden. Es handelt sich bei diesem allerdings um keinen besonders kurzen Weg.

Beispiel 8

Eingabedatei:
wenigerkrumm8.txt

Ausgabe:
```
Es gibt keinen Weg, der die Bedingung erfüllt.
```

Evaluierungsdauer:
1,8 Sekunden

Erläuterung:
Die Eingabe für Beispiel 8 ist das von mir in Lösungsidee beschriebene Szenario, für das es keinen Weg gibt, der die Bedingungen erfüllt. (Siehe Seite 3 „Kann immer ein Weg gefunden werden?“)

Quellcode

Im Folgenden einige Auszüge aus dem Quellcode meiner Implementierung.

Hilfsfunktion, berechnet den Steigungswinkel eines Vektors, der von punkt0 nach punkt1 geht:

```
def steigungswinkel(punkt0, punkt1):
    if punkt0[0] == punkt1[0]:
        # -> Beide Punkte haben dieselbe x-Koordinate
        # Der Steigungswinkel ist 90 Grad, der Vektor geht senkrecht zur x-
Achse
        angle = 90
        # Um diesen Sonderfall in das Modell aufzunehmen, definiere ich:
        # Wenn die Y-Koordinate von Punkt 1 größer ist als die von Punkt 0,
dann zeigt Vektor nach rechts, sonst nach links.
        if punkt1[1] > punkt0[1]:
```

```
            direction = "r"
        else:
            direction = "l"
    else:
        # -> Die Punkte haben verschiedene x-Koordinaten
        # Ermitteln des Steigungswinkels mit atan(Differenzenquotient):
        angle = math.atan((punkt1[1] - punkt0[1]) / (punkt1[0] - punkt0[0])) *
180/math.pi
        # Ermitteln, ob Vektor nach rechts oder links zeigt
        if punkt0[0] < punkt1[0]:
            direction = "r"
        else:
            direction = "l"
    return angle, direction
```

Hilfsfunktion, berechnet die Distanz von punkt0 zu punkt1:

```
def distanz(punkt0, punkt1):
    return math.sqrt((punkt1[0]-punkt0[0])**2 + (punkt1[1]-punkt0[1])**2)
```

Hilfsfunktion, ermittelt Antons Abbiegewinkel, wenn er von punkt0 über punkt1 zu punk2 fliegt:

```
def abbiegewinkel(punkt0, punkt1, punkt2):

    alpha, alpha_dir = steigungswinkel(punkt0, punkt1)
     #Steigungswinkel eines Vektors, der von punkt0 zu punkt1 geht
    beta, beta_dir = steigungswinkel(punkt1, punkt2)
     #Steigungswinkel eines Vektors, der von punkt1 zu punkt2 geht

    schnittwinkel = abs(beta - alpha)
     #Schnittwinkel ist die Differenz der beiden Schneidungswinkel
    if alpha_dir != beta_dir:
        schnittwinkel = 180 - schnittwinkel

    return schnittwinkel
```

Node-Klasse für das Speichern einer Position auf Antons Weg:

```
class Node:
    def __init__(self, station_id, visited=[], strecke=0, abstand=0):
        self.station_id = station_id  # Aktuelle Position
        self.strecke = strecke # Antons bisher zurückgelegte Strecke
        self.strecke += abstand
        self.abstand = abstand # Abstand zur vorherigen Position
        self.visited = list(visited) # Liste, die die bereits besuchten
           Positionen enthält
        self.visited.append(station_id)
        self.distance_to_avg = distanz(avg, stations[station_id])
        self.ways_to_reach_station = len(adjazenzen[station_id])
```

```
    def successors(self):
        # Gibt alle erreichbaren Positionen als Node-Objekte zurück
        successors = []
        for station_id in range(len(stations)):
            if station_id not in self.visited:
                abstand = distanz(stations[self.station_id],
stations[station_id])
                if self.visited == [self.station_id]:
                    successors.append(Node(station_id, self.visited,
self.strecke, abstand))
                else:

                    if (self.visited[-2], self.visited[-1]) in
adjazenzen[station_id]:
                        successors.append(Node(station_id, self.visited,
self.strecke, abstand))
        return successors
```

Programmcode, der eine umgekehrte Adjazenzliste erstellt:

```
adjazenzen = {}
for station0_id in range(len(stations)):
    for station1_id in range(len(stations)):
        if station1_id == station0_id:
            continue
        for station2_id in range(len(stations)):
            if station2_id not in adjazenzen:
                adjazenzen[station2_id] = []
            if station2_id == station1_id:
                continue
            angle = abbiegewinkel(stations[station0_id],
stations[station1_id], stations[station2_id])
            if angle <= 90:
                adjazenzen[station2_id].append((station0_id,
station1_id))
```

Funktion, die eine Tiefensuche zur Bestimmung eines möglichen Wegs vornimmt:

```
def tiefensuche(paths, *, prioritise_strecke):
    i = 0
    while True:
        i += 1
        if paths == []:
            return None
        path = paths.pop(0)
        if len(path.visited) == len(stations):
            return path
```

```
        successors = path.successors() #Vom Punkt path aus erreichbare
Stationen
        successors = sorted(successors, key=lambda path : path.strecke)
#Sortieren der Stationen -> Priorisierung kurzer Strecken

        if prioritise_strecke:
            if i > 1000:
                return None
        else:
            successors = sorted(successors, key=lambda path :
path.ways_to_reach_station) #Sortieren der Stationen -> Priorisierung von
Stationen, die nur von wenigen Stationen aus erreichbar sind
        paths = successors + paths
```