

# Aufgabe 1: Weniger krumme Touren

Teilnahme-ID: 67739

Bearbeiter/-in dieser Aufgabe:  
Robert Vetter

16. April 2023

## Inhaltsverzeichnis

<b>1</b>	<b>Aufgabenstellung</b>	<b>2</b>
<b>2</b>	<b>Lösungsidee</b>	<b>2</b>
<b>3</b>	<b>Umsetzung</b>	<b>3</b>
<b>4</b>	<b>Beispiele</b>	<b>7</b>
<b>5</b>	<b>Quellcode</b>	<b>10</b>

## 1 Aufgabenstellung

Die Aufgabe 1 mit dem Namen "Weniger krumme Touren" fordert, dass eine Route in gegebenen Punkten gefunden wird, in welcher in keinem Fall der eingeschlossene Winkel zwischen zwei Kanten kleiner als  $90^\circ$  ist. Dabei waren von den Punkten lediglich die Koordinaten gegeben. Weiterhin war verlangt, eine möglichst kurze Route zu finden, jedoch nicht die allerbeste Strecke.

## 2 Lösungsidee

Die Aufgabe befasst sich mit dem Problem des minimalen aufspannenden Baumes, also ein Baum in einem Graphen, der jeden Knoten einmal besucht und dabei versucht, ein möglichst geringes Gesamtgewicht der Kanten einzuhalten. Bei dem Baum musste es sich zusätzlich um eine Route handeln, also ein Weg in einem Graphen. Ein verwandtes Problem ist auch das Travelling-Salesman-Problem. Da die Aufgabenstellung nicht die kürzeste mögliche Route forderte, schloss ich Algorithmen wie die Breitensuche oder die Tiefensuche direkt aus, da beide sehr kleinschrittig die Pfade im Graphen durchlaufen. Würde man in einem Netzwerk mit 100 Knoten von jedem Punkt zu allen anderen Knoten eine Kante ziehen, dann gäbe es

$$k = \frac{n \cdot (n - 1)}{2} = \frac{100 \cdot 99}{2} = 4950 \text{ Kanten.} \quad (1)$$

In Zuge dessen wäre der Graph zu komplex, um ihn mithilfe der DFS / BFS durchlaufen zu lassen. Für das Problem eignet sich daher eher ein Greedy-Algorithmus, welcher am Zeitpunkt der Entscheidung die beste Wahl trifft. Er agiert also auf lokaler Ebene. Auf die Aufgabe übertragen würde er also den nächsten Punkt mit der geringsten Entfernung vom aktuellen Knoten auswählen, bei welchem die Winkelbedingung eingehalten wird. Im folgenden Beispiel stellt der rote Kreis den Knoten dar, von welchem der nächste Punkt ausgewählt werden soll:

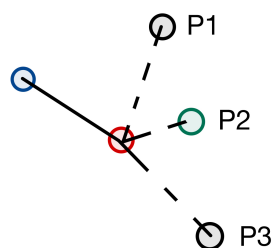


Abbildung 1: Veranschaulichung der Entscheidungsfindung

Hierbei fällt der Punkt P1 aus der Liste der möglichen Folgeknoten heraus, da der eingeschlossene Winkel der beiden Kanten kleiner als  $90^\circ$  wäre und es sich somit um eine "krumme Tour" handeln würde. Die Winkelbedingung ist bei P2 und P3 erfüllt. Da P3 jedoch eine höhere Distanz zu dem aktuellen Knoten aufweist, würde sich der Algorithmus für den Punkt P2 entscheiden. Natürlich hat der Greedy-Algorithmus eine sehr geringe Laufzeit. Im Gegenzug garantiert er jedoch auch nicht, dass die Lösung gefunden wird, wenn es eine gibt. Um die Wahrscheinlichkeit zu erhöhen, dass er tatsächlich eine Lösung findet, könnte man ihn von jedem möglichen Punkt aus starten lassen. Wenn er dann jedoch immer noch keine Lösung gefunden hat, wäre ein möglicher Lösungsansatz, die gefundene Route mit den meisten Punkten zu betrachten und zu versuchen, in den übrigen Knoten eine Route zu finden, um beide Routen anschließend zu verbinden. Wenn keine Verbindung beider Routen möglich ist, dann würde die nächstbeste Strecke in den verbleibenden Punkten betrachtet werden und wieder versucht werden, beide Wege zu kombinieren. Hierbei ist es logischerweise möglich, dass wiederum Punkte verbleiben, auch wenn zwei Routen schon verbunden wurden. In diesen müssten dann erneut Routen gesucht werden, um die Kombination zu ermöglichen. Stößt man auf eine Sackgasse (also es können keine Routen mehr verbunden werden), dann startet die gesamte Prozedur mit der anfänglich zweitbesten gefundenen Route erneut. Dieses könnte solange durchgeführt werden, bis letztlich eine Route gefunden wurde, die alle gegebenen Punkte beinhaltet. Hier ein Beispiel:

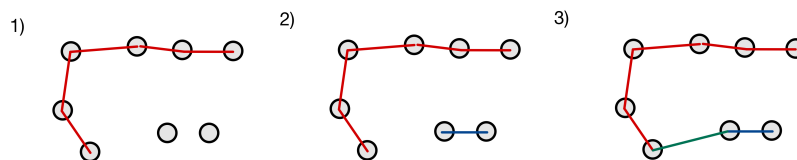


Abbildung 2: Veranschaulichung der Kombination zweier Routen

Angenommen im Abschnitt 1) der obigen Abbildung ist die Route eingezeichnet, die die meisten Punkte beinhaltet. Der Punkt links unten der roten Route stellt dabei den Startknoten dar. Der auf der vorherigen Seite beschriebene Algorithmus würde also nun in den zwei verbleibenden Punkten eine Route suchen. Dieses ist in Teilabschnitt 2) abgebildet. Zuletzt würden beide Routen kombiniert werden, sodass jeder Knoten eingeschlossen wird (3)). Wäre dies noch nicht der Fall, würde der Greedy-Ansatz wieder und wieder versuchen, neue Routen zu finden, um die schon bestehende Route mit der verbleibenden Route zu kombinieren. Natürlich steht die Frage im Raum, warum man denn nicht einfach, nachdem der Greedy-Algorithmus der roten Route keinen weiteren Punkt gefunden hat, den Algorithmus vom Startknoten aus auf die verbleibenden Punkte angewendet hätte. Dieses würde für das einfache abgebildete Beispiel noch funktionieren, bei komplexeren Routen jedoch nicht mehr. Die Kombination hat den Vorteil, dass der Greedy-Algorithmus von jedem verbleibenden Punkt aus gestartet wird und anschließend beide Routen kombiniert werden. So gibt es wesentlich mehr Routen, die zu einer zusammengefügt werden können, im Vergleich zum ausschließlichen Suchen einer Route vom Startknoten der ursprünglichen Route aus. Dies erhöht die Wahrscheinlichkeit signifikant, dass eine Route gefunden wird, die alle Punkte inkludiert. Es ist jedoch zu beachten, dass bei diesem Ansatz auch nicht garantiert die Lösung gefunden wird. Die Wahrscheinlichkeit hierfür wurde jedoch durch die beschriebene Logik deutlich erhöht.

### 3 Umsetzung

Die Lösungsidee wird in ein Programm der Sprache Python umgesetzt. Nachdem die TXT-Datei mit den gegebenen Koordinaten eingelesen wurde und sie als Liste von Tupeln gespeichert wurden, beginnt das eigentliche Programm. Hierbei widmete ich mich zuerst der Abstandsbestimmung zwischen zwei Punkten. Dies lässt sich über den Satz des Pythagoras realisieren. Da von den beiden Punkten die x- und die y-Koordinaten bekannt sind, lassen sich so die Längen der Katheten des rechtwinkligen Dreiecks bestimmen:

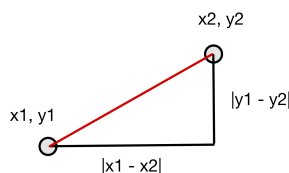


Abbildung 3: Abstandsbestimmung zweier Punkte

Letztlich ergibt sich somit für die Distanz zweier Punkte:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2)$$

Anschließend habe ich eine Methode "calculate\_angle" mit drei Punkten A, B und C als Parameter geschrieben, welche mithilfe der "calculate\_distance" Methode die Strecken  $\overline{AB}$ ,  $\overline{BC}$  sowie  $\overline{AC}$  berechnet.

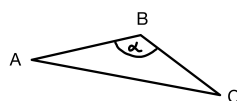


Abbildung 4: Winkelberechnung mithilfe des Kosinussatz

Sollte entweder  $\overline{AB}$  oder  $\overline{BC}$  gleich null sein, kann der Winkel als  $180^\circ$  angenommen werden. In jedem anderen Fall kann  $\alpha$  unter Verwendung des umgestellten Kosinussatz berechnet werden:

$$\alpha = \arccos \frac{(\overline{AB})^2 + (\overline{BC})^2 - (\overline{AC})^2}{2 \cdot \overline{AB} \cdot \overline{BC}} \quad (3)$$

Die wichtigste Komponente in meinem Programm ist jedoch der eigentliche Greedy-Algorithmus. Er geht alle möglichen Startpunkte durch und probiert eine möglichst lange Route zu finden. Den nächstbesten Punkt wählt er dabei basierend auf der Distanz aus, insofern die Winkelbedingung eingehalten ist. Nachfolgend habe ich ihn in einem Struktogramm abgebildet:

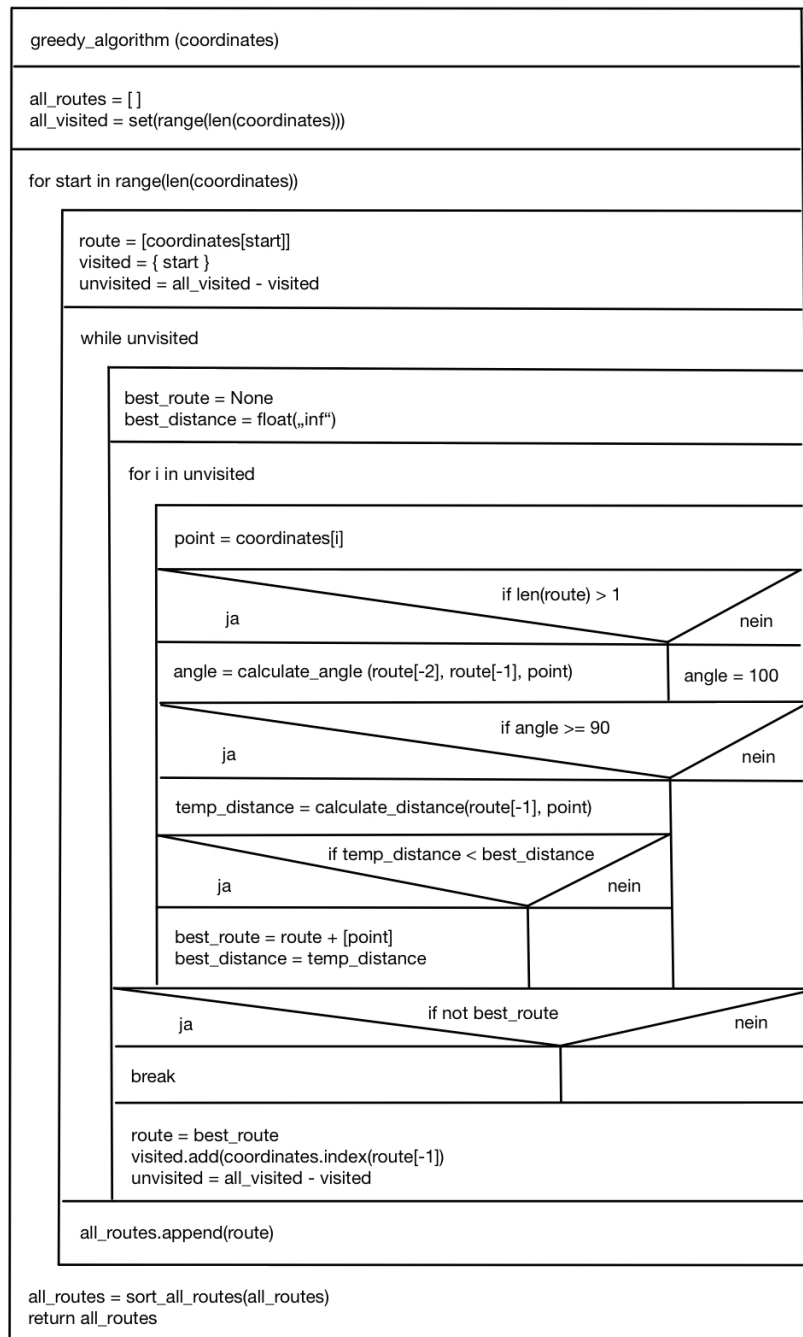


Abbildung 5: Struktogramm des Greedy-Algorithmus

Die Methode "sort\_all\_routes" erhält "all\_routes" als Parameter. Sie hat die Aufgabe, die Listen in "all\_routes" nach ihrer tatsächlichen Länge zu sortieren (Anzahl der Punkte). Wenn die tatsächliche Länge zweier Routen gleich ist, dann wird nach der Kilometerzahl der Routen sortiert, welche mithilfe der Methode "route\_length(route)" berechnet wird. In dieser werden einfach nur alle Distanzen der Abstände je zweier Punkte der Route addiert. Des weiteren entwickelte ich eine Methode "remaining\_points" mit den gesamten Koordinaten und einer Route als Parameter. Wie der Name schon vermuten lässt, gibt sie die in den Koordinaten verbleibenden Knoten zurück, welche noch nicht mit in der Route inkludiert wurden. Dies findet in der Methode "remaining\_cluster" mit den gesamten Koordinaten und der schon bestehenden Route als Parameter Verwendung. Sie ruft lediglich den Greedy-Algorithmus auf die verbleibenden Punkte auf. Wenn in diesen eine Route gefunden wurde, ist es natürlich essentiell, dass überprüft wird, ob die schon existierende Route mit einer neu gefundenen Strecke kombiniert werden kann. Daher implementierte ich eine weitere Methode namens "check\_for\_combining", welche zwei zu kombinierende Routen als Parameter erhält. Es gibt vier Möglichkeiten, wie zwei Routen kombiniert werden können:

1. Der Start der ersten Route verbindet sich mit dem Start der zweiten Route,
2. der Start der ersten Route verbindet sich mit dem Ende der zweiten Route,
3. das Ende der ersten Route verbindet sich mit dem Start der zweiten Route und
4. das Ende der ersten Route verbindet sich mit dem Ende der zweiten Route.

In der folgenden Abbildung sind Startpunkte durch einen grünen Kreis gekennzeichnet und Endpunkte einer Route werden mit einem roten Kreis dargestellt:

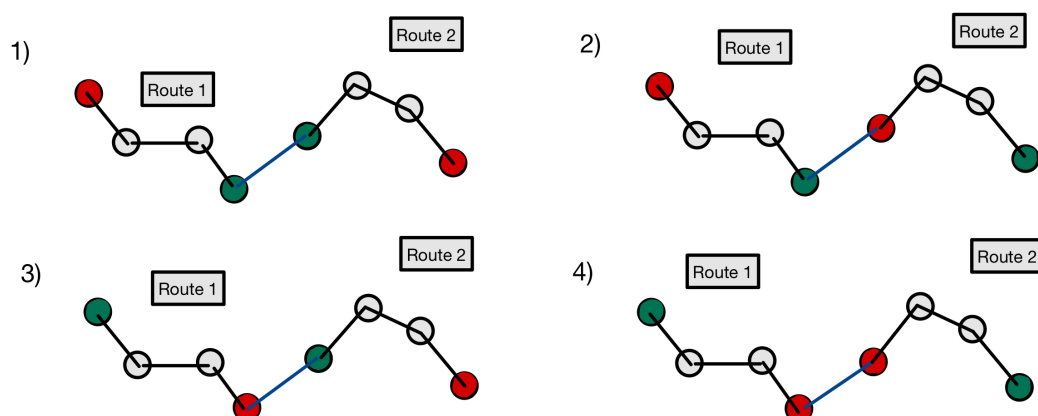


Abbildung 6: Möglichkeiten der Kombination zweier Routen

Nun habe ich mich einer Methode "find\_complete\_route" gewidmet, die über Schleifen versucht, mögliche Routen miteinander zu verbinden, um letztlich eine möglichst kurze Gesamtroute zu erhalten. Sie sollte sozusagen der Koordinator in meinem Programm sein. In Zuge dessen führt sie den Prozess aus, den ich schon im Kapitel "Lösungsidee" beschrieben habe. Sie geht alle gefundenen Originalrouten durch und ruft bei jeder Route die Methode "connect\_routes" auf mit den ursprünglichen Gesamtkoordinaten, der Hauptroute, der bisherigen minimalen Routenlänge sowie einem Schwellenwert als Parameter. Der Schwellenwert dient dabei zur Festlegung der maximal zu untersuchenden verbleibenden Routen, die in den Koordinaten gefunden wurden und verringert somit den Rechenaufwand des Programms. Ich habe ihn auf zehn festgelegt, da er bei diesem Wert eine gute Balance zwischen Qualität der Lösung und der Laufzeit des Programms besitzt. Dies hat jedoch zur Folge, dass nicht die Kombination aller möglicher Routen probiert wird. Die Methode "connect\_routes" ruft sich dabei rekursiv immer wieder auf. Dies geschieht solange, bis alle Kombinationen der Ausgangsroute mit den möglichen verbleibenden Routen ausprobiert wurden. Da die Methode durchaus einer der wichtigsten Bestandteile meines Programms ist, habe ich sie nachfolgend in einem Struktogramm abgebildet:

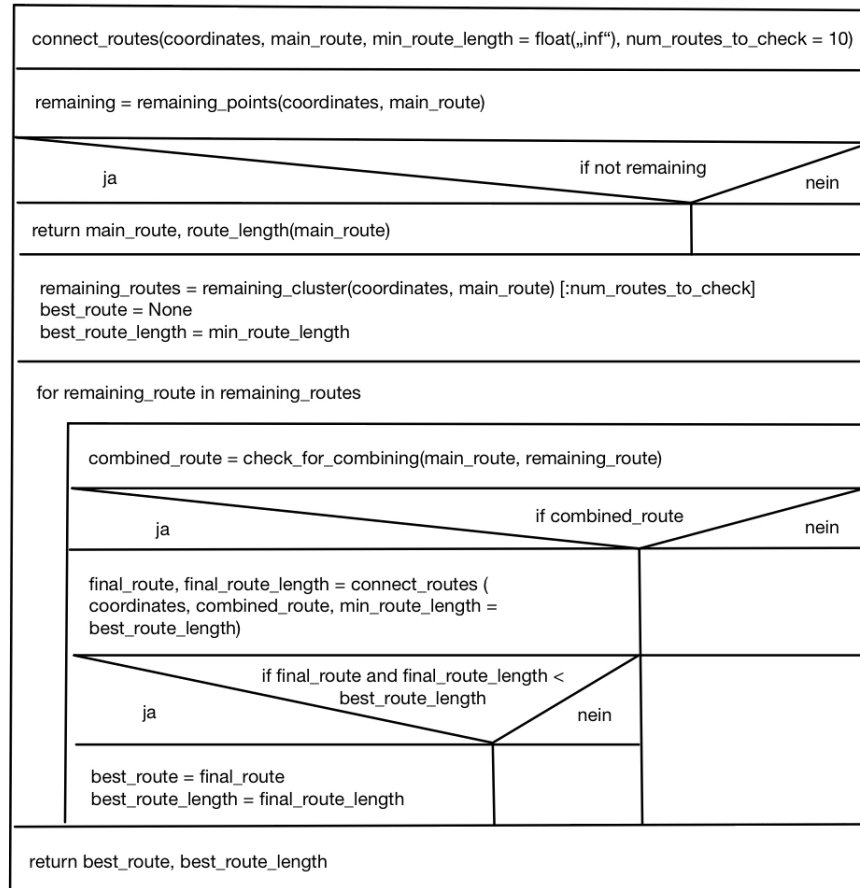



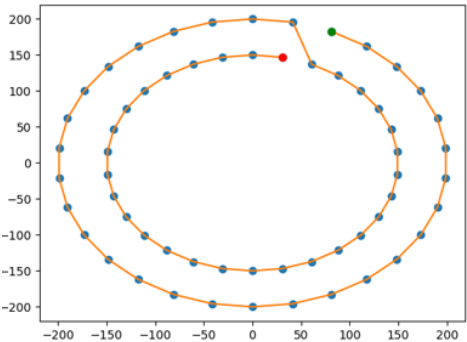
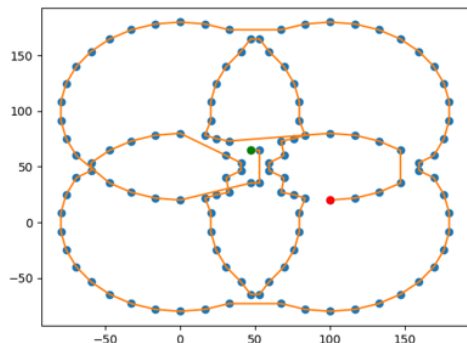
Abbildung 7: Struktogramm zur vollständigen Verknüpfung der Routen

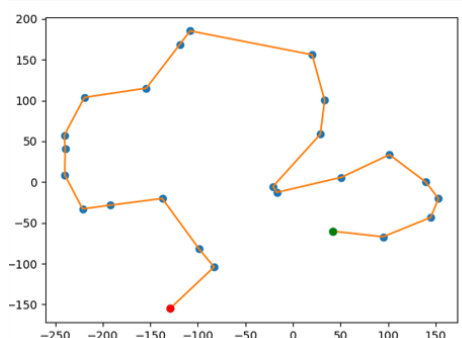
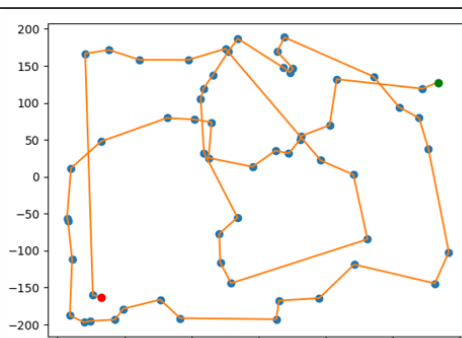
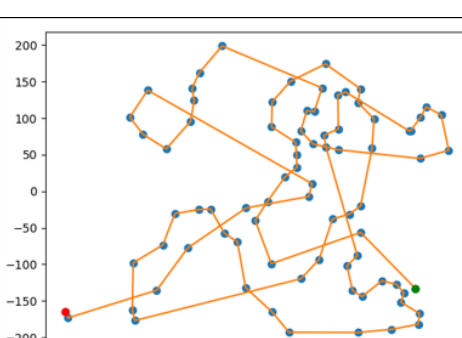
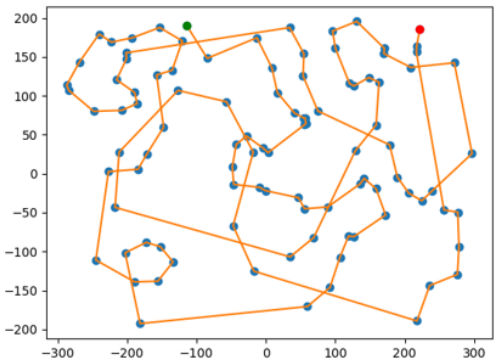
Abschließend habe ich die Route über die Methode "plot\_route" grafisch anzeigen lassen. Als Parameter erhält sie die Koordinaten aller Punkte und die Koordinaten der Route. Hierbei habe ich noch eine Methode "update" hinzugefügt, welche die Route stückweise plottet. Zusätzlich habe ich in einer anderen Funktion implementiert, dass automatisch eine Route mit einer gegebenen Anzahl an Punkten erstellt wird und diese in einer Datei mit dem Namen "points.txt" abgespeichert wird. Dabei wird bei jedem Schritt auf den Winkel geachtet. Hier ist jedoch zu berücksichtigen, dass die Punkte zufällig positioniert werden, weswegen es für den Greedy-Algorithmus teilweise schwierig sein kann, eine Lösung zu finden. Auch wenn es nicht von der Aufgabenstellung gefordert ist, entschied ich mich, noch die Reisedauer für solch eine Rundreise zu berechnen. Da das Vorgehen, um dieses umzusetzen, sehr simpel ist, erweiterte ich die Problemstellung: Die Aufenthaltsdauer des Reisenden beträgt in zentralen Städten zwei Tage, während er sich in den übrigen Städten lediglich einen Tag aufhält. Somit bestand die Schwierigkeit darin, erst einmal die zentralen Städte herauszufinden. Um dieses Problem zu lösen, verwendete ich einen Clustering-Algorithmus, mit welchen ich mich sowieso schon im Zuge meiner Seminarfacharbeit auseinandersetzte. Da viele der gegebenen Punktwolken eine besondere Form hatten, bei welchen unterschiedliche Clustering-Algorithmen stark unterschiedliche Resultate zu erzielen, entschied ich mich, den Algorithmus nur auf zufällig generierte Punkte anzuwenden. Dabei verwendete ich den Clustering-Algorithmus K-Means. Mithilfe diesem konnte ich die Zentren der gefundenen Cluster herausfinden. Die Anzahl der Cluster ermittelte ich dabei mithilfe der Ellenbogen-Methode. Ich probierte also verschiedene Anzahl an Clustern aus und ließ diese mithilfe des Silhouette-Scores bewerten. Die Verwendung dieser Anzahl ermöglichte es mir, durch K-Means die Cluster-Mittelpunkte zu bestimmen. Von diesen berechnete ich dann die Distanz zu den umliegenden Städten. Die Stadt, welche dem jeweiligen Mittelpunkt am nächsten liegt, ist also eine zentrale Stadt. Für die tatsächliche Reisedauer habe ich angenommen, dass das Flugzeug des Reisenden durchschnittlich mit  $800 \frac{\text{km}}{\text{h}}$  fliegt. Also bestand die Reisedauer lediglich aus folgender Rechnung:

$$t_{ges} = t_{Flugzeug} + t_{Stadt,zentral} + t_{Stadt,normal} = \frac{s_{ges}}{800 \frac{\text{km}}{\text{h}}} + t_{Stadt,zentral} + t_{Stadt,normal} \quad (4)$$

## 4 Beispiele

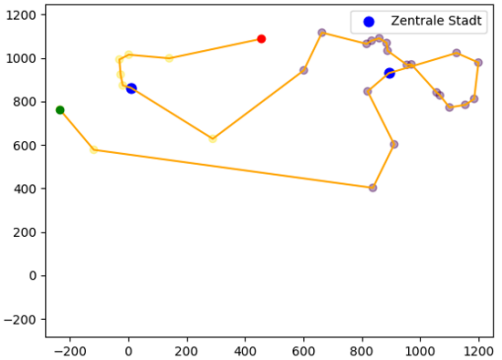
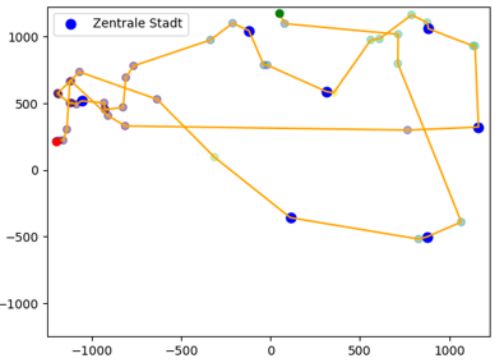
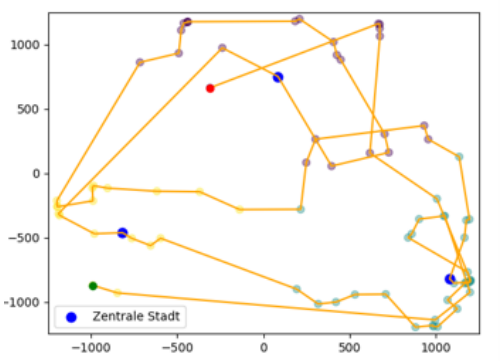
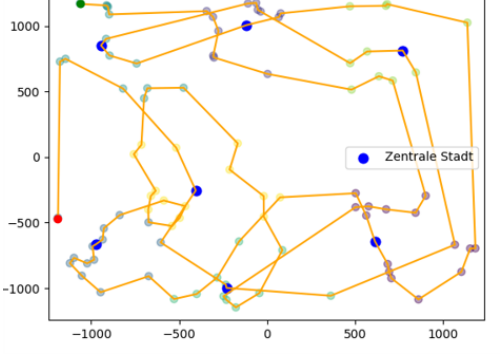
Im nachfolgenden werde ich das Ergebnis meines Programms mit den gegebenen BWInf-Beispielen darstellen. Der grüne Punkt symbolisiert dabei den Startpunkt der Route, wobei der rote Knoten für das Ende steht.

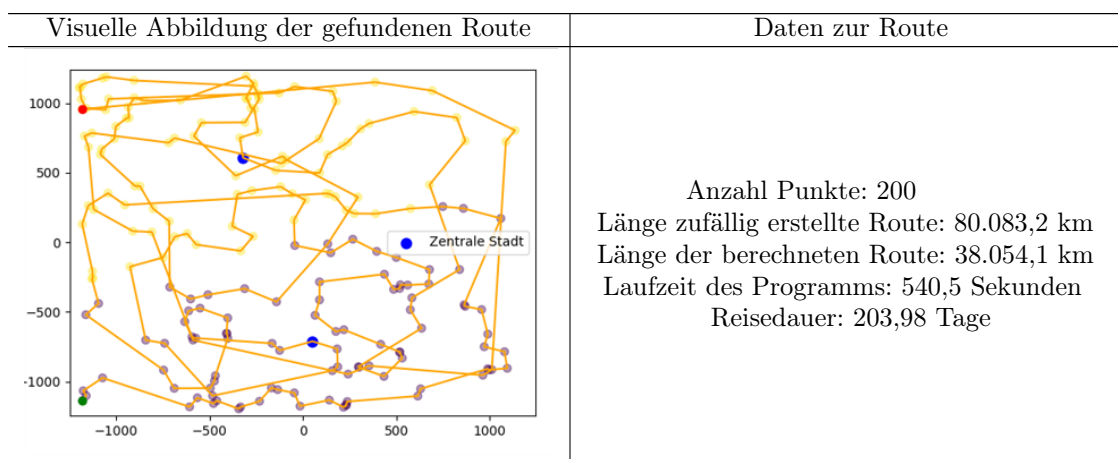
Visuelle Abbildung der gefundenen Route	Daten zur Route
	<p>Länge der Route: 847,4 Kilometer          Laufzeit des Programms: 0,8 Sekunden</p>
	<p>Länge der Route: 2183,7 Kilometer          Laufzeit des Programms: 1,38 Sekunden</p>
	<p>Länge der Route: 1952,2 Kilometer          Laufzeit des Programms: 3,78 Sekunden</p>

Visuelle Abbildung der gefundenen Route	Daten zur Route
	<p>Länge der Route: 1205,1 Kilometer          Laufzeit des Programms: 0,13 Sekunden</p>
	<p>Länge der Route: 3615,9 Kilometer          Laufzeit des Programms: 1,38 Sekunden</p>
	<p>Länge der Route: 4086,29 Kilometer          Laufzeit des Programms: 13,82 Sekunden</p>
	<p>Länge der Route: 4983,21 Kilometer          Laufzeit des Programms: 17,56 Sekunden</p>

Bei den vorgegebenen Beispiel ist mir aufgefallen, dass der Greedy-Algorithmus bis Beispiel drei selbstständig eine Lösung gefunden hat. Bei den folgenden Datensätzen konnte nur eine Lösung mithilfe der Kombination von mindestens zwei Routen gefunden werden. Nun möchte ich noch einige selbstgenerierte Beispiele vorstellen. In diesen stehen die hervorstechenden blauen Punkte für eine zentrale Stadt. Die jeweilige Farbe der restlichen Punkte steht für das Cluster, zu welchem sie zugeordnet wurden. Weiterhin ist mit der Information "Länge der zufällig erstellten Route" die Länge der Route gemeint, die von dem Programm unter zufälliger Positionierung der Punkte erstellt wurde. Die Länge der berechneten Route ist die Strecke, welche der Algorithmus in den erzeugten Knoten bestimmt hat. Die roten und grünen Knoten markieren wiederum Start- und Endpunkte:



Visuelle Abbildung der gefundenen Route	Daten zur Route
	<p>Anzahl Punkte: 30  Länge zufällig erstellte Route: 9539,7 km  Länge der berechneten Route: 4651,8 km  Laufzeit des Programms: 0,12 Sekunden  Reisedauer: 32,24 Tage</p>
	<p>Anzahl Punkte: 50  Länge zufällig erstellte Route: 17.278,0 km  Länge der berechneten Route: 12.017,4 km  Laufzeit des Programms: 3,34 Sekunden  Reisedauer: 57,63 Tage</p>
	<p>Anzahl Punkte: 80  Länge zufällig erstellte Route: 24.691,3 km  Länge der berechneten Route: 20.989,8 km  Laufzeit des Programms: 34,5 Sekunden  Reisedauer: 84,09 Tage</p>
	<p>Anzahl Punkte: 100  Länge zufällig erstellte Route: 42.929,1 km  Länge der berechneten Route: 25.384,2 km  Laufzeit des Programms: 46,6 Sekunden  Reisedauer: 108,32 Tage</p>



Es fällt auf, dass die Laufzeit des Programms stark variiert. Der Hauptgrund hierfür liegt in der Anzahl der Knoten. Aufgrund der zufälligen Positionierung der Punkte kann die Programmlaufzeit jedoch auch bei derselben Anzahl der Punkte stark variieren.

## 5 Quellcode

```

1 # notwendige Module/Bibliotheken importieren
import matplotlib.pyplot as plt
3 import numpy as np
import math
5 import matplotlib.animation as animation
import time
7 import random
from sklearn.cluster import KMeans
9 from sklearn.metrics import silhouette_score
import os

11
# Methode, um die Koordinaten aus der txt-Datei einzulesen, gibt sie als Liste von Tupeln zurueck
13 def read_coordinates(file):
    with open(file, "r") as f:
15         lines = f.readlines()
        coordinates = []
17         for line in lines:
            x, y = map(float, line.strip().split())
19             coordinates.append((x, y))
        return coordinates

21
# Funktion zur Berechnung der Distanz zwischen zwei Punkten mithilfe des S. d. Pythagoras
23 def calculate_distance(p1, p2):
    return math.sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

25
# Methode zur Berechnung des Winkels zwischen zwei Kanten, dabei ist p2 der Scheitelpunkt
27 def calculate_angle(p1, p2, p3):
    # Distanzen ausrechnen
29     a = calculate_distance(p1, p2)
    b = calculate_distance(p2, p3)
31     c = calculate_distance(p1, p3)

33     # wenn zwei Punkte uebereinander liegen, ist der Winkel valide und somit groesser als 90 Grad
    if a == 0 or b == 0:
35         return 180

37     # mithilfe des Kosinussatz cos(alpha) berechnen
    cos_angle = (a**2 + b**2 - c**2) / (2*a*b)

39
    # wenn numerische Ungenauigkeiten auftreten, dann runden
41     if cos_angle > 1:
        cos_angle = 1
43     elif cos_angle < -1:
        cos_angle = -1

45
    # Winkel in Grad zurueckgeben

```

```

47     angle = math.degrees(math.acos(cos_angle))
48     return round(angle, 2)
49
50 # berechnet die Laenge der Route ueber die Summe der Distanzen zwischen je zwei adjazenten Punkten
51 def route_length(coordinates):
52     length = 0
53     for i in range(len(coordinates) - 1):
54         length += np.linalg.norm(np.array(coordinates[i]) - np.array(coordinates[i + 1]))
55     return round(length, 3)
56
57 # Hauptalgorithmus des Programms, gibt alle gefundenen Routen zurueck
58 def greedy_algorithm(coordinates):
59     # all_routes und alle zu besuchenden Knoten initialisieren
60     all_routes = []
61     all_visited = set(range(len(coordinates)))
62
63     # alle Startknoten durchgehen
64     for start in range(len(coordinates)):
65         route = [coordinates[start]]
66         visited = {start}
67         unvisited = all_visited - visited
68         # solange es immer noch unbesuchte Knoten gibt
69         while unvisited:
70             best_route = None
71             best_distance = float("inf")
72             # jeden moeglichen naechsten Knoten durchgehen
73             for i in unvisited:
74                 point = coordinates[i]
75                 # Winkel berechnen
76                 if len(route) > 1:
77                     angle = calculate_angle(route[-2], route[-1], point)
78                 else:
79                     # vom ersten Punkt kann zu jedem moeglichen Punkt Kante gezogen werden
80                     angle = 100
81                 if angle >= 90:
82                     # waehlt Punkt aus, bei dem der Winkel passt und der am naechsten dran ist
83                     temp_distance = calculate_distance(route[-1], point)
84                     if temp_distance < best_distance:
85                         best_route = route + [point]
86                         best_distance = temp_distance
87
88             # wenn keine beste Route gefunden wurde, dann unterbreche
89             if not best_route:
90                 break
91
92             # lege die gefundene Route als die beste Route fest
93             route = best_route
94
95             # fuege zu den bisher besuchten Punkten den letzten neu hinzugekommenen
96             # Punkt der Route hinzu und aktualisiere 'unvisited'
97             visited.add(coordinates.index(route[-1]))
98             unvisited = all_visited - visited
99             all_routes.append(route)
100
101 # sortiert Liste nach len(list) und wenn Laenge gleich, dann nach route_length(list)
102 def sort_all_routes(lists):
103     lists.sort(key=len, reverse=True)
104     result = []
105     lengths = []
106     for lst in lists:
107         if len(lst) not in lengths:
108             lengths.append(len(lst))
109             result.append([lst])
110         else:
111             for i in range(len(result)):
112                 if len(result[i][0]) == len(lst):
113                     result[i].append(lst)
114                     break
115     for group in result:
116         group.sort(key=route_length)
117     return [item for sublist in result for item in sublist]
118
119 all_routes = sort_all_routes(all_routes)

```

```

    return all_routes
121
# gibt alle verbleibenden Punkte zurueck
123 def remaining_points(all_points, route):
    remaining = all_points[:]
125     for point in route:
        if point in remaining:
127             remaining.remove(point)
    return remaining
129
# gibt alle Routen in den verbleibenden Punkten zurueck
131 def remaining_cluster(coordinates, route):
    remaining_cl_route = greedy_algorithm(remaining_points(coordinates, route))
133     return remaining_cl_route

135 # checkt, ob man zwei Routen verbinden kann, in diesem Fall also eine Hauptroute (route) und
    eine Route im verbleibenden Cluster (remaining_cl_route)
137 def check_for_combining(route, remaining_cl_route):
    possible_routes = []
139
    # Start- und Endpunkte der Routen als Variablen abspeichern
141     remaining_cl_route_start, remaining_cl_route_end = remaining_cl_route[0], remaining_cl_route[-1]
    cl_route_start, cl_route_end = route[0], route[-1]
143
    # wenn Laenge der Route groesser 1 ist
145     if len(remaining_cl_route) > 1:
        # alle moeglichen Verbindungen ueberpruefen, Bedingungen der Verbindungen werden in
147         Liste abgespeichert
        # Moeglichkeit 1: Start des urspruenglichen Clusters verbindet
149         sich mit Start des remaining_clusters
        # Moeglichkeit 2: Start des urspruenglichen Clusters verbindet
151         sich mit Ende des remaining_clusters
        # Moeglichkeit 3: Ende des urspruenglichen Clusters verbindet
153         sich mit Start des remaining_clusters
        # Moeglichkeit 4: Ende des urspruenglichen Clusters verbindet
155         sich mit Ende des remaining_clusters
        route_conditions = [
157             (calculate_angle(remaining_cl_route[1], remaining_cl_route_start,
                cl_route_start) >= 90 and
159             calculate_angle(route[1], cl_route_start, remaining_cl_route_start) >= 90,
                route[::-1], remaining_cl_route),
161             (calculate_angle(remaining_cl_route[-2], remaining_cl_route_end,
                cl_route_start) >= 90 and
163             calculate_angle(route[1], cl_route_start, remaining_cl_route_end) >= 90,
                route[::-1], remaining_cl_route[::-1]),
165             (calculate_angle(remaining_cl_route[1], remaining_cl_route_start,
                cl_route_end) >= 90 and
167             calculate_angle(route[-2], cl_route_end, remaining_cl_route_start) >= 90,
                route, remaining_cl_route),
169             (calculate_angle(remaining_cl_route[-2], remaining_cl_route_end,
                cl_route_end) >= 90 and
171             calculate_angle(route[-2], cl_route_end, remaining_cl_route_end) >= 90,
                route, remaining_cl_route[::-1])
173         ]
    else:
175         # Route besteht aus einem Punkt -> dieser kann sich mit Start
        oder Ende der anderen Route verbinden
177         route_conditions = [
            (calculate_angle(remaining_cl_route[0], cl_route_start, route[1]) >= 90,
179             route[::-1], remaining_cl_route),
            (calculate_angle(remaining_cl_route[0], cl_route_end, route[-2]) >= 90,
181             route, remaining_cl_route)
        ]
183
    # durch Bedingungen durch iterieren und wenn erfuehrt, zusammengefuegte Route
185     zu possible_routes hinzufuegen
    for condition, part1, part2 in route_conditions:
187         if condition:
            possible_routes.append(part1 + part2)
189
    # Route mit minimaler Routenlaenge zurueckgeben
191     if possible_routes:
        return min(possible_routes, key=lambda r: route_length(r))

```

```

193     else:
194         return None
195
196 # Methode, die alle urspruenglich gefundenen Routen durchgeht und die Methode
197 # 'connect_routes' aufruft, welche alle moeglichen Verbindungen der Ursprungsrouten mit anderen probiert
198 def find_complete_route(coordinates, num_routes_to_check=10):
199     # initial_routes sind die im Graphen urspruenglich gefundenen Routen
200     initial_routes = greedy_algorithm(coordinates)
201     best_complete_route = None
202     best_complete_route_length = float("inf")
203
204     for initial_route in initial_routes:
205         complete_route, complete_route_length = connect_routes(coordinates, initial_route,
206             best_complete_route_length, num_routes_to_check)
207         # wenn neue Route gefunden wurde, die kuerzer als die bisher kuerzeste
208         # gefundene Route ist, dann aktualisiere die bisher kuerzeste gefundene Route
209         if complete_route and complete_route_length < best_complete_route_length:
210             best_complete_route = complete_route
211             best_complete_route_length = complete_route_length
212
213     return best_complete_route
214
215 # Methode, die ueber Rekursion alle moeglichen Routenkombinationen durchgeht
216 def connect_routes(coordinates, main_route, min_route_length=float("inf"), num_routes_to_check = 10):
217     # verbleibenden Punkte berechnen
218     remaining = remaining_points(coordinates, main_route)
219
220     # wenn es keine gibt, ist Route vollstaendig und kann zurueckgegeben werden
221     if not remaining:
222         return main_route, route_length(main_route)
223
224     # sonst suche Routen in den verbleibenden Punkten
225     remaining_routes = remaining_cluster(coordinates, main_route)[:num_routes_to_check]
226     best_route = None
227     best_route_length = min_route_length
228
229     # versuche, die Ursprungsrouten solange mit anderen verbleibenden Routen zu verbinden,
230     # bis alle Punkte Teil der Route sind
231     for remaining_route in remaining_routes:
232         combined_route = check_for_combining(main_route, remaining_route)
233         if combined_route:
234             # rekursiver Aufruf, um moeglichst viele Punkte in der Route zu inkludieren
235             final_route, final_route_length = connect_routes(coordinates, combined_route,
236                 min_route_length=best_route_length, num_routes_to_check=num_routes_to_check)
237             # wenn neue beste Route gefunden, dann speichere sie ab
238             if final_route and final_route_length < best_route_length:
239                 best_route = final_route
240                 best_route_length = final_route_length
241
242     return best_route, best_route_length
243
244 # Methode zur visuellen Anzeige der Route
245 def plot_route(coordinates, route, cluster = False):
246     # x- und y-Koordinaten festlegen
247     x_coo = coordinates[:, 0]
248     y_coo = coordinates[:, 1]
249     x = [p[0] for p in route]
250     y = [p[1] for p in route]
251
252     fig, ax = plt.subplots()
253
254     # wenn selbststaendig Punkte generiert werden sollen (also auch K-Means angewendet wird)
255     if cluster and len(route) > 10:
256         # fuehre K-Means auf Koordinaten aus und bestimme den Punkt, der dem jeweiligen
257         # Zentrum am naechsten liegt
258         optimal_clusters = determine_number_of_clusters(coordinates)
259         labels, cluster_centers = k_means_cluster(coordinates, optimal_clusters)
260         closest_points = find_closest_points_to_centers(coordinates, cluster_centers)
261         print("Die Reise dauert voraussichtlich ungefaehr ", time_of_travel(coordinates,
262             route, closest_points), " Tage.")
263
264     # Grenzen fuer Plot festlegen

```

```

min_val = min(min(x_coo), min(y_coo)) - 50
267 max_val = max(max(x_coo), max(y_coo)) + 50
ax.set_xlim(min_val, max_val)
269 ax.set_ylim(min_val, max_val)

271 # Cluster points
ax.scatter(x_coo, y_coo, c=labels, cmap='viridis', alpha=0.4)
273 # Closest points
ax.scatter(closest_points[:, 0], closest_points[:, 1], c='blue', marker='o',
275 s=60, label='Zentrale Stadt')
line, = ax.plot(x[:1], y[:1], '--', color='orange')
277 else:
ax.plot(x_coo, y_coo, 'o')
279 line, = ax.plot(x[:1], y[:1], '--')

281 # zeichnet die Route Stueck fuer Stueck in KOS, aktualisiert somit visuelle Darstellung
def update(num):
283     num += 1
    if num >= len(x) + 1:
285         ani.event_source.stop()
        return line,
287     line.set_data(x[:num], y[:num])
    return line,

289 ani = animation.FuncAnimation(fig, update, frames=len(x)+1, interval=150, blit=False)

291 # hier werden spezifische Parameter zur grafischen Darstellung festgelegt
293 if cluster and len(route) > 10:
    ax.plot(x[0], y[0], 'go', zorder=5)
295     ax.plot(x[-1], y[-1], 'ro', zorder=6)
    plt.legend()
297 else:
    ax.plot(x[0], y[0], 'go')
299     ax.plot(x[-1], y[-1], 'ro')

301 plt.show()

303 def time_of_travel(coordinates, route, closest_points):
305     # Flugzeug fliegt mit 800 km/h, Annahme: Reisender haelt sich 1 Tag in gewoehnlicher Stadt
    und 2 Tage in zentral gelegener Stadt auf
307     # -> tges = tFlug + tZentral + tGewoehnlich
    return round((route_length(route) / 800) / 24 + 1 * (len(coordinates) - len(closest_points))
309     + 2 * len(closest_points), 2)

311 # fuehrt KMeans-Clustering mithilfe von sklearn durch
def k_means_cluster(coordinates, n_clusters):
313     kmeans = KMeans(n_clusters=n_clusters, n_init=10).fit(coordinates)
    cluster_centers = kmeans.cluster_centers_
315     return kmeans.labels_, cluster_centers

317 # sucht Punkt mit geringster Entfernung zum Zentrum des Clusters
def find_closest_points_to_centers(coordinates, cluster_centers):
319     closest_points = []
    for center in cluster_centers:
321         distances = np.linalg.norm(coordinates - center, axis=1)
        closest_point_idx = np.argmin(distances)
323         closest_points.append(coordinates[closest_point_idx])
    return np.array(closest_points)

325 # bestimme optimales k fuer K-Means
327 def determine_number_of_clusters(coordinates):
    max_clusters = min(10, len(coordinates))
329     best_silhouette_score = -1
    best_n_clusters = 2

331     for n_clusters in range(2, max_clusters + 1):
333         labels, cluster_centers = k_means_cluster(coordinates, n_clusters)
        score = silhouette_score(coordinates, labels)

335         if score > best_silhouette_score:
337             best_silhouette_score = score
            best_n_clusters = n_clusters

```

```

339         return best_n_clusters
341
342     # erstelle eine Test-Route, in welcher die Winkelbedingung eingehalten wird
343 def create_points_with_right_angles(num_points, limit=1200):
344     points = []
345     x, y = random.uniform(-limit, limit), random.uniform(-limit, limit)
347
348     # die for-Schleife erzeugt num_points-Punkte im Bereich -limit bis limit
349     for _ in range(num_points):
350         angle = random.uniform(0, 2 * math.pi)
351         scale = random.uniform(0, limit)
353
354         dx, dy = math.cos(angle) * scale, math.sin(angle) * scale
355         new_x, new_y = x + dx, y + dy
357
358         # solange der Punkt noch ausserhalb des Bereichs liegt
359         while new_x < -limit or new_x > limit or new_y < -limit or new_y > limit:
360             scale = random.uniform(0, limit / 2)
361             dx, dy = math.cos(angle) * scale, math.sin(angle) * scale
362             new_x, new_y = x + dx, y + dy
364
365         x, y = new_x, new_y
366         points.append((x, y))
368
369     return points
370
371     # Test-Route wird in einer txt-Datei abgespeichert
372 def write_points_to_file(filename, points):
373     with open(filename, 'w') as f:
374         for point in points:
375             f.write(f"{point[0]} {point[1]}\n")
377
378 if __name__ == "__main__":
379     valid = False
381
382     # solange keine gueltige Eingabe eingelesen wurde, Frage erneut
383     while not valid:
384         decision = int(input("Moechten Sie neue zufaellige Koordinaten generieren (1) oder
385             schon vorbereitete Punkte verwenden? (2) "))
387
388         if decision == 1:
389             number_points = int(input("Wie viele Punkte moechten Sie generieren? "))
390             complete_route = None
391             start_time = time.time()
392             # es wird nicht immer eine Route gefunden -> probiere solange, bis eine gefunden wird
393             while not complete_route:
394                 created_coordinates = create_points_with_right_angles(number_points)
395                 # speichere "points.txt" in aktuellem Verzeichnis
396                 current_directory = os.path.dirname(os.path.abspath(__file__))
397                 file = os.path.join(current_directory, 'points.txt')
398                 write_points_to_file(file, created_coordinates)
399                 coordinates = read_coordinates(file)
400                 # wenn Anzahl Punkte hoeher als 70, suche mit geringerem Schwellenwert
401                 -> senkt Laufzeit
402                 if number_points > 70:
403                     complete_route = find_complete_route(coordinates, num_routes_to_check=2)
404                 else:
405                     complete_route = find_complete_route(coordinates)
406             print("Die Laenge der zufaellig generierten Route betraegt ",
407                 route_length(coordinates), " km.")
408             break
409
410         elif decision == 2:
411             file_found = False
412             # solange kein gueltiger Dateiname eingegeben wurde
413             while not file_found:
414                 file_name = str(input("Geben Sie die Bezeichnung der txt-Datei mit den Koordinaten
415                     an (Bsp.: wenigerkrumm1.txt). \nDie Datei muss ich dafuer in demselben
416                     Verzeichnis wie dieses Skript befinden.: "))
417                 # Datei im aktuellen Verzeichnis finden
418                 current_directory = os.path.dirname(os.path.abspath(__file__))
419                 file_path = os.path.join(current_directory, file_name)

```

```
413         # Ueberpruefen, ob die Datei existiert
414         if os.path.isfile(file_path):
415             file_found = True
416         else:
417             print("Datei nicht gefunden. Bitte geben Sie einen gueltigen Dateinamen ein.")
418
419         coordinates = read_coordinates(file_path)
420         start_time = time.time()
421         complete_route = find_complete_route(coordinates)
422         break
423     else:
424         print("Die Eingabe war ungueltig. Geben Sie entweder eine 1 oder eine 2 ein.")
425
426     print("Die Laenge der mit dem Algorithmus berechneten Route betraegt ",
427           route_length(complete_route), " km.")
428     end_time = time.time()
429     print("Laufzeit: ", round(end_time - start_time, 2), " Sekunden")
430
431     # Route plotten
432     if decision == 1:
433         plot_route(np.array(coordinates), complete_route, cluster=True)
434     if decision == 2:
435         plot_route(np.array(coordinates), complete_route, cluster=False)
```