Aufgabe 1: Lisa Rennt

Teilnahme-Id: 01189

Bearbeiter/-in dieser Aufgabe:
Georgijs Vilums

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Wie bereits in der Aufgabenstellung erwähnt, ist es sinnvoll, sich zu beginn zu überlegen, welchen Weg Lisa einschlagen sollte, wenn ihr keine Hindernisse im Weg stehen. Die verfügbaren Informationen sind Geschwindigkeit v_B des Busses, Geschwindigkeit v_L von Lisa, sowie die x und y-Koordinate von Lisas Haus, im Folgenden mit x_H und y_H bezeichnet. Wenn Lisa den Bus im letztmöglichen Moment erreichen soll, müssen sie und der Bus den Treffpunkt gleichzeitig erreichen. Da der Bus sich auf der Geraden $x = 0$ bewegt, ist nur offen, bei welcher y-Koordinate sie sich treffen. Diese wird im Folgenden schlichtweg mit y bezeichnet.
Um den Treffpunkt zu erreichen, müssen der Bus und Lisa jeweils die Distanzen d_B und d_L zurücklegen. Diese sollen in einer Zeit t zurückgelegt werden. Außerdem hat Lisa die Möglichkeit, früher als der Bus loszulaufen. Diese Zeitspanne wird mit h gekennzeichnet. So ergibt sich ein lineares Gleichungssystem:

$$d_B = v_B \cdot t \tag{1}$$
$$d_L = v_L \cdot (t + h) \tag{2}$$

Wenn dieses Gleichungssystem nach h aufgelöst wird, ergibt sich der folgende Zusammenhang

$$h = \frac{d_L}{v_L} - \frac{d_B}{v_B} \tag{3}$$

für die Zeit, die Lisa vor dem Bus loslaufen muss. Dabei gelten $d_B = y$ und $d_L = \sqrt{x_H^2 + (y - y_H)^2}$ für die jeweiligen zurückzulegenden Distanzen. Für die Zusatzzeit h in Abhängigkeit von Höhe y des Treffpunktes gilt also:

$$h(y) = \frac{\sqrt{x_H^2 + (y - y_H)^2}}{v_L} - \frac{y}{v_B} \tag{4}$$

Von dieser Funktion ist das globale Minimum gesucht, denn an diesem Wert für y braucht Lisa am wenigsten Vorsprung, kann also möglichst spät losgehen. Dieses globale Minimum findet sich an der Stelle

$$y = y_H + \frac{v_L \cdot x_H}{\sqrt{v_B^2 - v_L^2}} \tag{5}$$

Für einen gegebenen Startpunkt kann mit dieser Formel der ideale Endpunkt berechnet werden, der in einem geraden Weg erreicht werden sollte.

Um festzustellen, wie Hindernisse am besten zu umgehen sind, gibt es eine Eigenschaft, die man bei den idealen Wegen um Polygone herum ausnutzen kann: Alle idealen Wege sind entweder eine gerade Strecke zum Ziel, oder bestehen aus mehreren Geradensegmenten. Außerdem liegen die Ecken eines solchen Weges immer an den Ecken der zu umgehenden Polygone, da, wenn dies nicht der Fall wäre, der Weg kürzer gemacht werden könnte und damit nicht ideal wäre. Hierbei ist wichtig zu beachten, dass dies nur in dem Fall gilt, dass alle Hindernisse Polygone mit geraden Kanten sind.

Diese Eigenschaft kann man sich zunutze machen, indem man alle möglichen Wege vom Startpunkt bis zum Ziel als einen Graphen auffasst, dessen Knoten die Ecken der Polygone und dessen Kanten die Wege zwischen den Ecken repräsentieren. Auf dem Graphen sollen dabei nur Wege möglich sein, die im geometrischen Sinne nicht durch Polygone blockiert werden, also Wege zwischen Ecken, zwischen denen sich Lisa in einer geraden Strecke bewegen kann. Der Startpunkt wird dabei auch als eine Einzelne Ecke gesehen und in den Graphen als Knoten eingefügt. Interessant wird es bei der Betrachtung möglicher Endpunkte, da diese kontinuierlich auf der Geraden $x = 0$ verteilt sind. Aber auch hier lassen sich diskrete Punkte in den Graphen aufnehmen: Von allen nicht-Endpunkten wird der ideale Endpunkt mit der o. g. Formel berechnet und in den Graphen hinzugefügt, falls der Weg nicht durch ein Polygon blockiert wird. Man hat nun also einen Graphen, der unter anderem die Punkte enthält, die beim idealen Weg durchkreuzt werden müssen, sowie eine Menge an potentiellen Endpunkten. Nun muss nur noch für jeden potentiellen Endpunkt der kürzeste Weg zwischen letzterem und dem Startpunkt berechnet und mithilfe von Formel (3) herausgefunden werden, welcher Weg und dessen zugehöriger Endpunkt den Wert von h minimieren. Die Route, für die der Wert von h minimal ist, ist die beste Route. Das finden des kürzesten Weges lässt sich mit Hilfe des A*-Algorithmus durchführen, der in Abschnitt 2 erläutert wird.

2 Umsetzung

Das Programm wurde in Python implementiert. Zur Umsetzung der o. g. Ideen wurden drei Python-Libraries verwendet: shapely zur Modellierung der geometrischen Objekte und zur Berechnung von Schnittmengen, networkx zur Modellierung des Graphen und zur Berechnung der kürzesten Wege innerhalb des Graphen und matplotlib zur Visualisierung der berechneten Route.

Zu Beginn wird die gegebene Datei eingelesen und eine Liste von Polygonen erstellt, welche einerseits die Eckpunkte, welche zum Graphen hinzugefügt wurden, andererseits das Polygon selbst als shapely-Objekt enthalten. Diese shapely-Objekte sind eine geometrische Repräsentation der Polygone, und erlauben es, geometrische Operationen wie das Herausfinden von Schnittmengen durchzuführen. Im folgenden Schritt werden Polygone, die eine gemeinsame Kante haben, zu einem Polygon kombiniert, unter Zuhilfenahme der Funktion `union` von shapely, welche zwei Polygone zu einem verbindet

Im darauf folgenden Schritt wird der Graph erstellt. Der erste Schritt in diesem Prozess ist die Erstellung aller Knoten bis auf die Endknoten. Hierfür wird für jedes Polygon die Liste seiner Eckpunkte durchlaufen, und jeder Eckpunkt als Knoten in den Graphen eingefügt. Auch der Startpunkt gilt als Knoten und wird in den Graphen eingefügt. Die nun vorhandenen Knoten müssen vernetzt werden, wobei nur Verbindungen erlaubt werden, die kein Polygon schneiden. Dafür werden alle Knoten, die ja jeweils eine Ecke eines Polygons repräsentieren, durchlaufen und es wird für den Weg zu jedem anderen Knoten geprüft, ob dieser Weg erlaubt ist oder nicht.
Zur Überprüfung von Wegen wurde die Funktion `check_edge_valid(p1, p2, ...)` implementiert. Diese nutzt vor allem die Funktion `intersection` von shapely, die die Schnittmenge zweier geometrischer Objekte liefert. Die erstere verfährt wie folgt:

1. Es wird eine Kante erstellt, welche die Verbindung zwischen zwei Punkten repräsentiert

2. Die Schnittmenge jedes Polygons mit der Kante wird berechnet und einer Liste angefügt

3. Für jede Schnittmenge aus der Liste wird geprüft, ob diese entweder punktförmig oder innerhalb des Umfanges des Polygons enthalten ist, auf dem der Ausgangspunkt liegt. Trifft dies zu, gibt die Methode `True` zurück, sonst `False`. So wird sichergestellt, dass nur Verbindungen, die zwischen Polygonen oder auf der Kante eines Polygons liegen, akzeptiert werden. Eine unerlaubte Schnittmenge wäre bspw. ein Geradensegment, welches nicht auf der Kante eines Polygons liegt, da ein solches Segment zwingend ein Polygon schneidet.

Nachdem eine Verbindung geprüft wurde, wird sie, sofern sie zulässig ist, in den Graphen eingefügt. Die Gewichtung der Kante entspricht dabei der Distanz der beiden Punkte, deren Verbindung sie repräsentiert. So kann später mithilfe eines Suchalgorithmus der kürzeste Weg zwischen zwei beliebigen Punkten gefunden werden.
Strukturell ist der Graph fast fertig. Es müssen nur noch mögliche Endpunkte, die an der y-Achse liegen, eingefügt werden, sowie ihre Verbindungen zu den restlichen Knoten. In Abschnitt 1 wurde bereits die Formel hergeleitet, mit der für einen beliebigen Punkt der ideale Endpunkt berechnet werden kann. Diese wird nun auf jeden Punkt innerhalb des Graphen angewendet. Dann wird wieder mit der Funktion `check_edge_valid(p1, p2, ...)` geprüft, ob die Verbindung erlaubt ist. Wenn dies der Fall ist, so wird sowohl der berechnete Endpunkt als Knoten, als auch die Verbindung als Kante in den Graphen eingefügt. Da später außerdem Wege zu allen möglichen Endpunkten berechnet werden müssen, werden die Endpunkte gesondert in einer Liste gespeichert.

Innerhalb des Graphen werden viele verschiedene Wege abgebildet, die zur y-Achse führen, worunter sich auch die ideale Lösung befindet. Es wird fortgefahren, indem für jeden möglichen Endpunkt, der in der Liste der Endpunkte enthalten ist, die Länge des Kürzesten Weges innerhalb des Graphen zwischen dem jeweiligen Endpunkt und dem Startpunkt berechnet wird. Die Länge des kürzesten Weges zwischen Zwei Knoten wird mithilfe der Funktion `astar_path_length` von networkx berechnet. Diese nutzt den A*-Algorithmus, um den kürzesten Pfad innerhalb eines Graphen zu berechnen. Dieser soll im folgenden erläutert werden.

Der A*-Algorithmus ist ein Suchalgorithmus, der einen Graphen traversiert, um den besten Weg zwischen zwei Knoten zu Berechnen. A* verwendet eine Heuristik, um abzuschätzen, wie hoch die Kosten von einem Folgeknoten aus sind, um an das Ziel zu gelangen, wobei Knoten mit niedrigeren Kosten und damit einer besseren erwarteten Leistung bevorzugt betrachtet werden. Im Falle der Wegfindung wird als Heuristik oft die direkte Distanz zum Ziel gewählt. Unter Zuhilfenahme einer Heuristik ist A* oft schneller als andere Suchalgorithmen.
Während der Ausführung Teilt A* alle Knoten in drei Kategorien: Unbekannt, Bekannt und Abgeschlossen. Zunächst sind alle Knoten bis auf den Startknoten unbekannt. Zu ihnen ist noch kein Weg berechnet worden. Bekannt werden alle die Knoten genannt, zu denen bereits ein Weg bekannt ist, bei dem aber eine Unsicherheit besteht, ob es der beste Weg ist. Abgeschlossen sind die Knoten, zu denen der beste Weg bekannt ist. Diese werden nicht weiter untersucht.
Während der Suche wird jedem Knoten sein bisher bester bekannter Vorgängerknoten zugewiesen. Der Algorithmus arbeitet dann die folgenden Schritte in einer Schleife ab, entweder, bis der gesuchte Knoten in die Liste der abgeschlossenen Knoten aufgenommen wird, oder, wenn die Liste der bekannten Knoten leer ist.

1. Der Knoten mit den geringsten vermuteten Kosten wird aus der Liste der bekannten Knoten entnommen und in die Liste der abgeschlossenen Knoten aufgenommen

2. Jeder noch unbekannte Folgeknoten dieses Knoten wird in die Liste der bekannten Knoten aufgenommen, zusammen mit seinen vermuteten Kosten und dem aktuellen Knoten als Vorgänger. War der Knoten bereits bekannt, aber sind die aktuellen Kosten kleiner als die bisher berechneten, werden diese aktualisiert, und der Vorgängerknoten aktualisiert.

3. Wenn der Gesuchte Knoten in die Liste der abgeschlossenen Knoten übernommen wird, terminiert der Algorithmus. Der kürzeste Weg lässt sich nun über die jeweiligen Vorgängerknoten finden, bis der Anfangsknoten erreicht wird

Nun kann für jeden Endpunkt der kürzeste Weg bis zum Startpunkt berechnet werden, und umgekehrt. Es muss nur noch mithilfe der Formel (3) die benötigte Zeit für jeden Knoten berechnet werden, und der Endknoten ausgewählt werden, bei dem dieser Wert minimal ist. Der endgültige Pfad wird dann mithilfe der Funktion `astar_path` von networkx berechnet und, zusammen mit weiteren Informationen, an die Konsole bzw. graphisch ausgegeben.

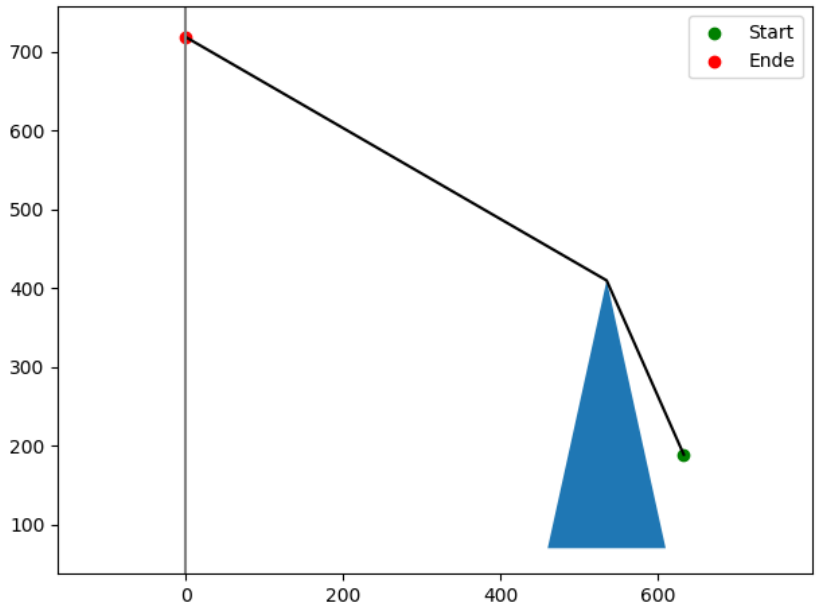
3 Beispiele

3.1 lisarennt1.txt

Eingabe: `python 2_1.py lisarennt1.txt`
Ausgabe:

```
Laufdistanz: 859.5187952385713 m
Laufzeit: 206.28451085725712 s
Startzeit: 07:27:59.981376
Ankunftszeit: 07:31:26.265887
Treffpunkt mit Bus: (0, 718.8823940164498)

Route:
1. (633, 189, 'L')
2. (535, 410, 'P1')
3. (0, 718.8823940164498, 'Bus')
```



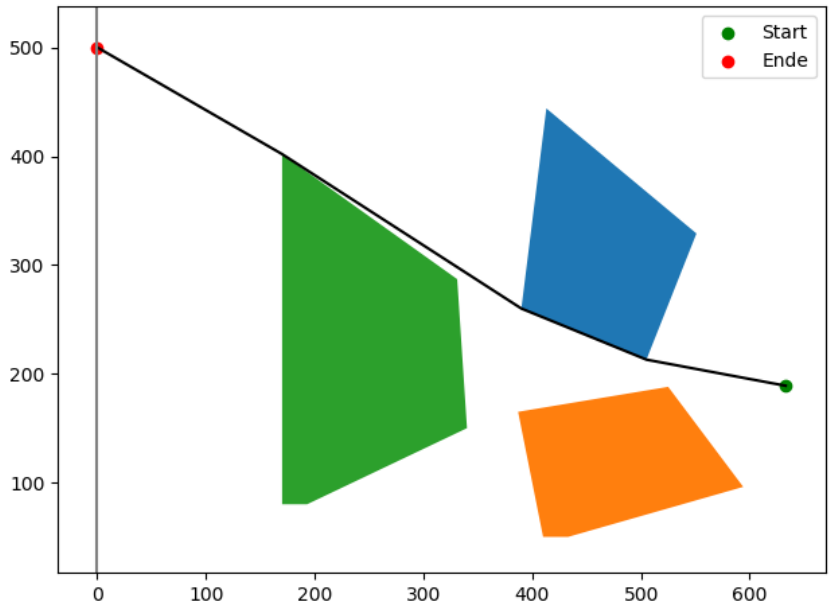
3.2 lisarennt2.txt

Eingabe: `python 2_1.py lisarennt2.txt`
Ausgabe:

```
Laufdistanz: 712.6105908652551 m
Laufzeit: 171.0265418076612 s
Startzeit: 07:28:08.991404
Ankunftszeit: 07:31:00.017946
Treffpunkt mit Bus: (0, 500.14954576223636)

Route:
1. (633, 189, 'L')
2. (505, 213, 'P1')
```

```
3. (390, 260, 'P1')
4. (170, 402, 'P3')
5. (0, 500.14954576223636, 'Bus')
```

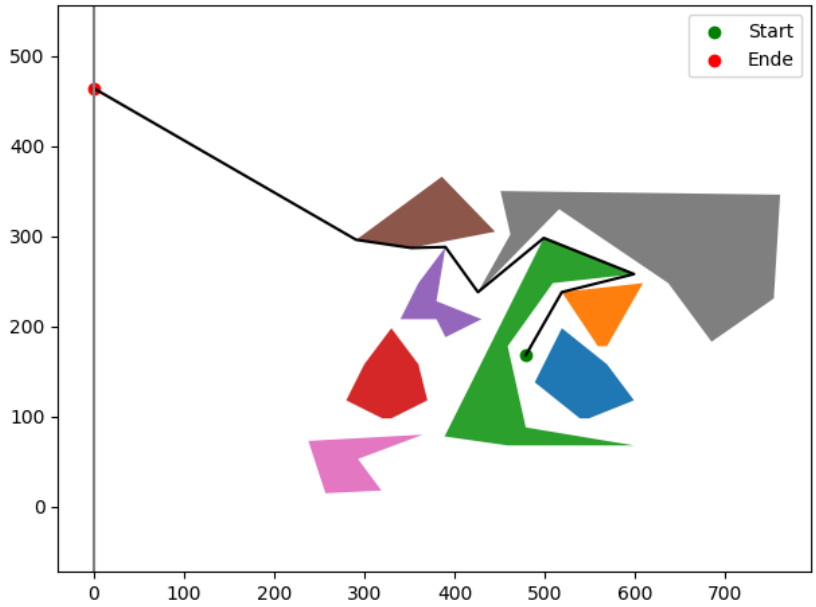


3.3 lisarennt3.txt

Eingabe: `python 2_1.py lisarennt3.txt`
Ausgabe:

```
Laufdistanz: 862.5844316766888 m
Laufzeit: 207.0202636024053 s
Startzeit: 07:27:28.660808
Ankunftszeit: 07:30:55.681072
Treffpunkt mit Bus: (0, 464.00892833418106)

Route:
1. (479, 168, 'L')
2. (519, 238, 'P2')
3. (599, 258, 'P3')
4. (499, 298, 'P3')
5. (426, 238, 'P8')
6. (390, 288, 'P5')
7. (352, 287, 'P6')
8. (291, 296, 'P6')
9. (0, 464.00892833418106, 'Bus')
```

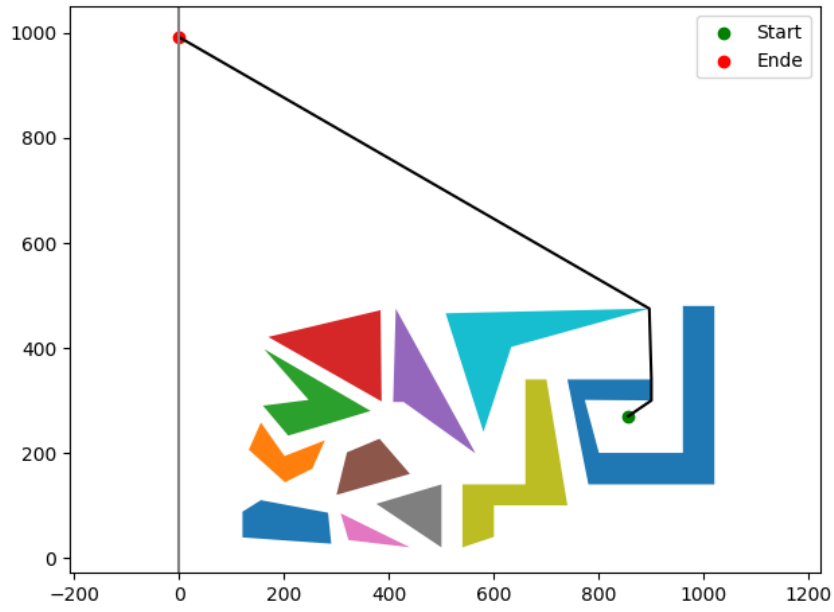


3.4 lisarennt4.txt

Eingabe: `python 2_1.py lisarennt4.txt`
Ausgabe:

```
Laufdistanz: 1262.9250364694046 m
Laufzeit: 303.10200875265707 s
Startzeit: 07:26:55.974692
Ankunftszeit: 07:31:59.076701
Treffpunkt mit Bus: (0, 992.3058411939047)

Route:
1. (856, 270, 'L')
2. (900, 300, 'P11')
3. (900, 340, 'P11')
4. (896, 475, 'P10')
5. (0, 992.3058411939047, 'Bus')
```

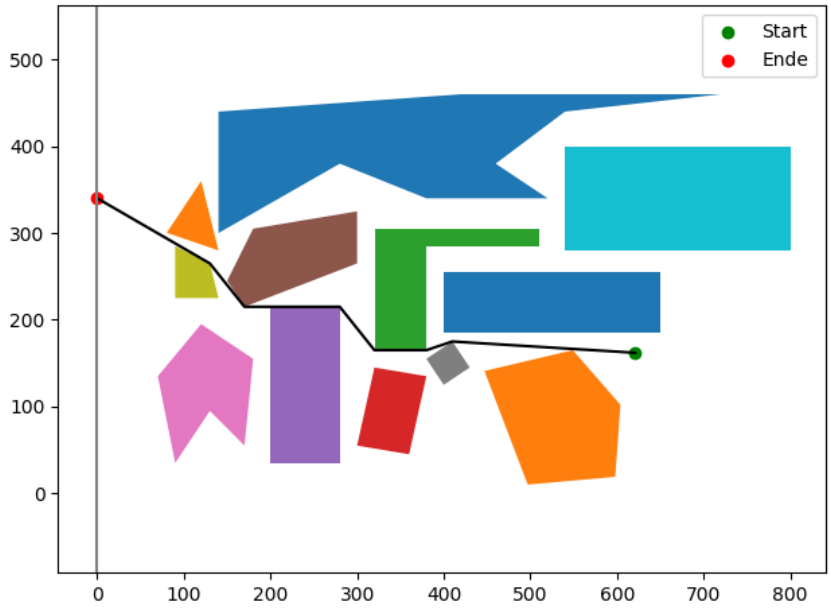


3.5 lisarennt5.txt

Eingabe: `python 2_1.py lisarennt5.txt`
Ausgabe:

```
Laufdistanz: 691.1964259470017 m
Laufzeit: 165.88714222728038 s
Startzeit: 07:27:54.919522
Ankunftszeit: 07:30:40.806664
Treffpunkt mit Bus: (0, 340.05553499465134)

Route:
1. (621, 162, 'L')
2. (410, 175, 'P8')
3. (380, 165, 'P3')
4. (320, 165, 'P3')
5. (280, 215, 'P5')
6. (170, 215, 'P6')
7. (130, 265, 'P9')
8. (0, 340.05553499465134, 'Bus')
```



3.6 Diskussion

Die Beispiele zeigen, dass das Programm es in den verschiedensten Fällen schafft, eine sehr gute, vermutlich auch optimale Lösung zu finden. Beispiele 1 und 2 beleuchten sehr simple Fälle, in denen das Programm ohne Probleme einen Weg um die Hindernisse findet. Beispiel 3 zeigt einen für einige Lösungsansätze vielleicht sehr schwierig zu bewältigenden Fall, wo ein konkaves Polygon umgangen werden muss. Aber auch hier wird problemlos der ideale Weg gefunden. Beispiel 4 zeigt, dass das Programm verschiedene mögliche Endpunkte berücksichtigt und feststellt, dass es gar nicht nötig ist, einen Weg durch die vielen Polygone zu finden, da der eigentlich schnellste Zielpunkt durch einen Weg fast komplett ohne Hindernisse erreicht werden kann. Zuletzt weist Beispiel 5 auf einen möglichen Schwachpunkt des Programms hin. Es wird zwar wieder ein sehr guter Weg gefunden, allerdings dauert die Berechnung nun mehrere Sekunden. Die Laufzeit des Programms zeigt wegen der Überprüfung von allen möglichen Verbindungen quadratisch an. Die Qualität der Lösungen wird aber mit zunehmender Polygonmenge nicht schlechter.

4 Quellcode

```python
#!/usr/bin/env python3
# Python 3.7

import sys
import os
import math
import matplotlib.pyplot as plt
import shapely.geometry as geometry
import networkx as nx
import datetime

BUS_SPEED = 30 / 3.6
RUN_SPEED = 15 / 3.6


def main():
    # Festlegen der Standarddatei
    file_name = os.path.join(os.path.dirname(__file__), "lisarennt5.txt")

    if len(sys.argv) == 2:
        file_name = os.path.join(os.path.dirname(__file__), sys.argv[1])

    # Einlesen der Datei
    input_file = open(file_name, 'r')
    str_data = input_file.read().splitlines()
    input_file.close()

    str_data_split = [list(map(int, string.split(' '))) for string in str_data]

    poly_data = str_data_split[1:len(str_data_split) - 1]
    raw_polygons = []
    all_points = []

    # Erstellen der Polygone
    for data_line in poly_data:
        vertices = []
        for _ in range(1, len(data_line) - 1, 2):
            vertices.append((data_line[_], data_line[_ + 1]))
        raw_polygons.append(Polygon(vertices))
        all_points += vertices

    start_point = (str_data_split[-1][0], str_data_split[-1][1])
    all_points.append(start_point)

    # Vereinen von Polygonen, die einander beruehren
    polygons = []
    for poly1 in raw_polygons:
        for poly2 in raw_polygons:
            if poly1 is not poly2 and poly1.poly_obj.intersects(poly2.poly_obj):
                    poly1 = poly1.poly_obj.union(poly2.poly_obj)
                    # Aktualisieren der Liste der Ecken des neuen kombinierten Polygons
                    poly1.vertices = [(round(p[0]), round(p[1])) for p in
                                      geometry.mapping(poly1.poly_obj)['coordinates'][0]]
                    raw_polygons.remove(poly2)
        polygons.append(poly1)

    # Erstellen eines Graphen mit allen bisherigen Punkten
    graph = nx.Graph()
```

```
    graph.add_nodes_from(all_points)

    # Erstellen von erlaubten Verbindungen innerhalb des Graphen
    checked_connections = []
    for polygon in polygons:
        for p1 in polygon.vertices:
            for p2 in all_points:
                if p1 != p2 and (p2, p1) not in checked_connections:
                    valid_connection, connecting_line = check_edge_valid(p1, p2, polygons, polygon)
                    if valid_connection:
                        graph.add_edge(p1, p2, weight=connecting_line.length)
                    checked_connections.append((p1, p2))

    # Hinzufuegen moeglicher Endpunkte an der Gerade x = 0
    destinations = []
    for p1 in all_points:
        p2 = calculate_ideal_finish(p1)
        valid_connection, connecting_line = check_edge_valid(p1, p2, polygons)
        if valid_connection:
            graph.add_node(p2)
            graph.add_edge(p1, p2, weight=connecting_line.length)
            destinations.append((p2, nx.astar_path_length(graph, start_point, p2,
                                                          lambda p, q: geometry.LineString(
                                                          [p, q]).length)))

    # Ausrechnen der Guete jedes Endpunktes
    early_start = []
    for point, distance in destinations:
        early_start.append(distance / RUN_SPEED - point[1] / BUS_SPEED)

    best_destination = destinations[early_start.index(min(early_start))]
    best_path = nx.astar_path(graph, start_point, best_destination[0],
                              lambda p, q: geometry.LineString([p, q]).length)

    # Ausgabe von Routeninformationen
    start_time = (datetime.datetime(year=1, month=1, day=1, hour=7, minute=30)
                  - datetime.timedelta(seconds=min(early_start)))
    print("Laufdistanz: " + str(best_destination[1]) + " m")
    print("Laufzeit: " + str(best_destination[1] / RUN_SPEED) + " s")
    print("Startzeit: " + start_time.time().strftime('%H:%M:%S.%f'))
    print("Ankunftszeit: " + (start_time + datetime.timedelta(seconds=best_destination[1] / RUN_SPEED))
          .time().strftime('%H:%M:%S.%f'))
    print("Treffpunkt mit Bus: " + str(best_destination[0]))
    print()
    print("Route:")
    # Ausgabe der Route, Feststellen der jeweiligen Polygon-ID
    for i in range(len(best_path)):
        point = best_path[i]
        point_tag = ""
        if i == 0:
            point_tag = "L"
        elif i == (len(best_path) - 1):
            point_tag = "Bus"
        else:
            for j in range(len(raw_polygons)):
                if raw_polygons[j].poly_obj.intersects(geometry.Point(point)):
                    point_tag = "P" + str(j + 1)
                    break
        print(str(i + 1) + ". " + str((point[0], point[1], point_tag)))

    # Zeichnen des Weges
```

```
    plot_polygons(polygons)
    plot_line(best_path, 'black')
    plt.axvline(x=0, color='grey')
    plt.scatter(best_path[0][0], best_path[0][1], color='green', label='Start')
    plt.scatter(best_path[-1][0], best_path[-1][1], color='red', label='Ende')
    plt.legend()
    plt.axis('equal')
    plt.tight_layout()
    plt.show()


# Klasse, um Informationen ueber Polygone zu speichern
class Polygon:
    def __init__(self, vertices):
        self.vertices = vertices
        self.poly_obj = geometry.Polygon(vertices)


# Prueft, ob eine verbindende Kante zwischen zwei Punkten keine anderen Polygone schneidet
def check_edge_valid(p1, p2, polygons, polygon=None):
    connecting_line = geometry.LineString([p1, p2])
    intersecting_geometry = []
    for polygon2 in polygons:
        current_intersection = polygon2.poly_obj.intersection(connecting_line)
        if isinstance(current_intersection, geometry.Point) or \
                isinstance(current_intersection, geometry.LineString):
            intersecting_geometry.append(current_intersection)
        else:
            intersecting_geometry += list(current_intersection)
    valid_connection = True
    for geom in intersecting_geometry:
        validated = False
        if polygon is not None:
            if polygon.poly_obj.boundary.contains(geom):
                validated = True
        else:
            if geometry.Point(p1).contains(geom):
                validated = True
        if geometry.Point(p2).contains(geom):
            validated = True
        if not validated:
            valid_connection = False
    return valid_connection, connecting_line


# Zeichnet eine Linie im pyplot-Diagramm
def plot_line(vertices, color='#333333'):
    x = []
    y = []
    for vertex in vertices:
        x.append(vertex[0])
        y.append(vertex[1])
    plt.plot(x, y, color)


# Zeichnet ein Polygon
def plot_polygons(polygons):
    for polygon in polygons:
        x = []
        y = []
        for vertex in polygon.vertices:
```

```
            x.append(vertex[0])
            y.append(vertex[1])
        plt.fill(x, y)


# Berechnet die ideale ziel-y-Koordinate abhaengig vom aktuellen Standort
def calculate_ideal_finish(coords):
    return 0, coords[1] + ((RUN_SPEED * coords[0]) / math.sqrt(
                           math.pow(BUS_SPEED, 2) - math.pow(RUN_SPEED, 2)))


if __name__ == '__main__':
    main()
```