Aufgabe 1: Lisa rennt

Team-ID: 00772

Team-Name: T.S

28. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Die Aufgabenstellung kann im Prinzip in zwei Schritte geteilt werden. Zum einen muss man einen kürzesten Weg durch das Feld mit Beachtung der Hindernisse finden können, zum anderen muss man aus all den möglichen Wegen den Weg finden, bei dem Lisa am spätesten aufstehen kann. Natürlich werden für die Lösung einige Annahmen gemacht, die im echten Leben nicht möglich wären: Lisas Geschwindigkeit bleibt konstant während sie rennt, sie wird als Punkt dargestellt und kann somit direkt am Rand eines Hindernisses entlang laufen, und es reicht, dass sie den Bus auf die Sekunde genau erreicht (d.h mögliche Bremszeiten werden nicht in Betrachtung gezogen).

1.1 Kürzester Weg mit Hindernissen

Die Frage nach dem kürzesten Weg durch eine Ebene mit Polygonhindernissen ist allgemein bekannt als das Problem der Suche nach **euklidischen kürzesten Wegen**. Dieses ist eng verwandt mit dem Problem der Konstruktion von sogenannten **Visibility Graphen**.

Definition 1.1. Gegeben ist eine Menge von Polygonen P, ein Startpunkt S und ein Endpunkt E. Der Visibility Graph besteht aus den n Eckpunkten der Polygone $v_1, v_2, ..., v_n$ und S und E als Knoten. Eine Verbindung von einem Knoten v zu einem anderen Knoten w ist eine Kante falls w von v (und somit auch v von w) sichtbar ist, d.h die Kante nicht die Seite eines Polygons schneidet. Folglich ist die Kantenmenge eine Untermenge aller möglichen Kanten zwischen allen Knoten.

Im euklidischen Raum besteht ein kürzester Weg immer aus geraden Linien, somit liegt der kürzeste Weg von S zu E auf dem Visibility Graphen. Hat man einen Visibility Graph konstruiert, kann darauf ein Algorithmus zum Finden von kürzesten Wegen in Graphen angewandt werden, wie zum Beispiel der Dijkstra Algorithmus.

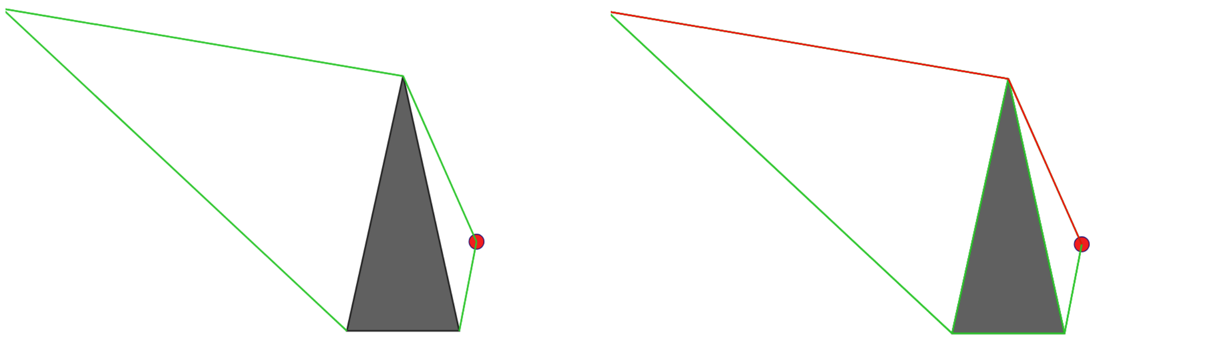


Abbildung 1: Visibility Graph (grün, noch ohne inklusive Polygone) und kürzester Weg (rot) auf Basis von Beispiel 1

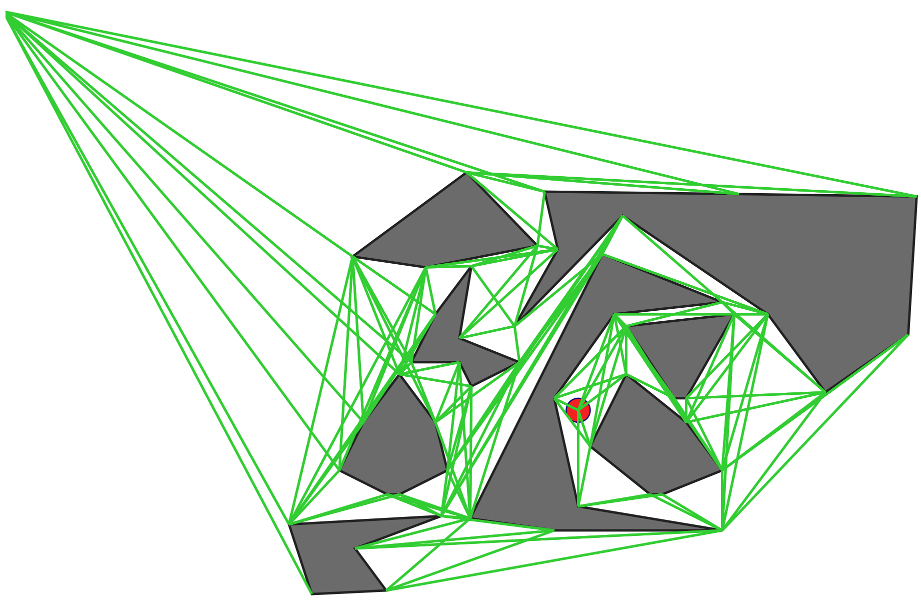


Abbildung 2: Visibility Graph (grün, noch ohne inklusive Polygone) auf Basis von Beispiel 3

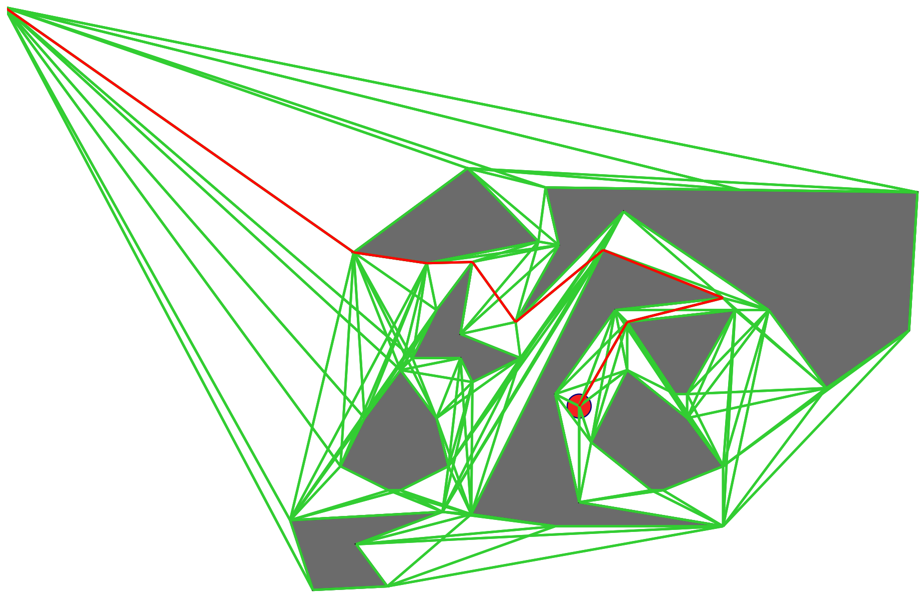


Abbildung 3: Kürzester Weg (rot) auf Basis von Beispiel 3

1.1.1 Überschneiden von Linien

Ein wichtiger Bestandteil des Algorithmus zur Konstruktion des Graphen ist das Überprüfen zweier Linien auf Überschneidung. Dazu ist das Konzept der **Orientierung** wichtig. Wenn 3 Punkte in einer bestimmten Reihenfolge gegeben sind, lassen sich 3 mögliche Orientierungen feststellen: im Uhrzeigersinn, gegen den Uhrzeigersinn, und parallel.

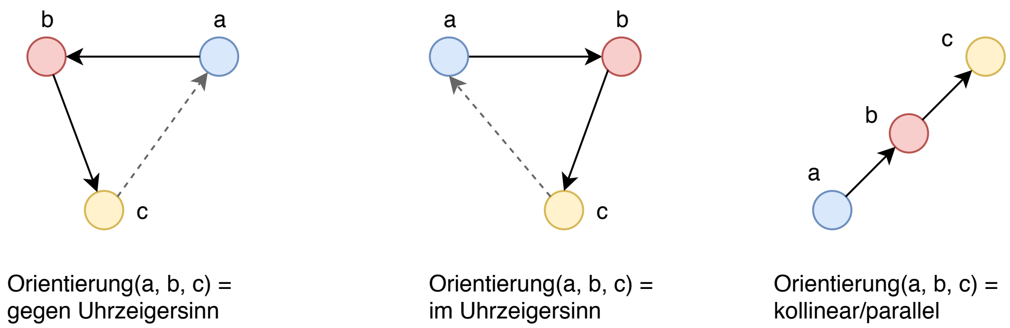


Abbildung 4: Mögliche Orientierungen

Man kann die Orientierung mit Hilfe des **Kreuzproduktes** zwischen zwei Vektoren bestimmen:

Gegeben sind 3 Punkte $(a_1|a_2), (b_1|b_2), (c_1|c_2)$. Dann

$$\vec{v} = \begin{pmatrix} b_1 - a_1 \\ b_2 - a_2 \\ 0 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ 0 \end{pmatrix} \qquad \vec{w} = \begin{pmatrix} c_1 - b_1 \\ c_2 - b_2 \\ 0 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ 0 \end{pmatrix}$$

Falls $\vec{v}$ und $\vec{w}$ kollinear, dann ist Orientierung parallel. Sonst:

$$\vec{v} \times \vec{w} = \begin{pmatrix} 0 \\ 0 \\ v_1 w_2 - v_2 w_1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ d \end{pmatrix}$$

Falls $d < 0$ dann Orientierung im Uhrzeigersinn, sonst wenn $d > 0$ gegen Uhrzeigersinn. Das liegt daran, dass der resultierende Vektor des Kreuzprodukts zweier Vektoren im $\mathbb{R}^2$ ein zur Ebene senkrechter Vektor ist, der in negative x_3-Richtung verläuft falls $\vec{w}$ rechts von $\vec{v}$ liegt, und in positive Richtung falls er links liegt ("Rechte-Hand-Regel").

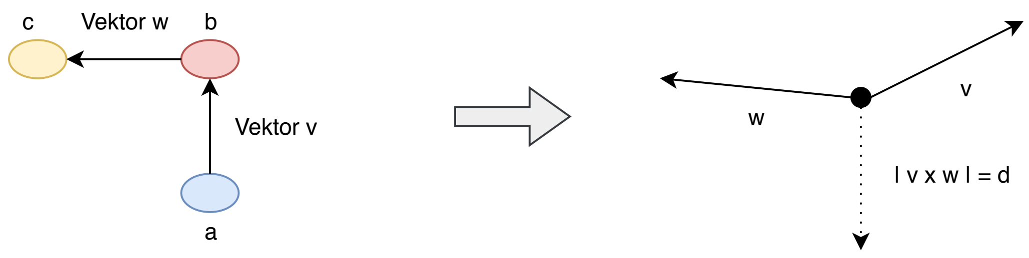


Abbildung 5: Anwendung des Kreuzprodukts

Diese Eigenschaft kann folgendermaßen benutzt werden:

1. Hat man zwei Linien, eine mit den Endpunkten p_1 und q_1, die andere mit den Endpunkten p_2 und q_2, prüft man zunächst ob sie die selbe Steigung haben.
2. Wenn ja, dann gibt es keine Überschneidung da selbst übereinander liegende Strecken in diesem Fall belaufbar sein sollen.
3. Wenn nein, dann überprüft man ob gilt: Orientierung($p_1 \to q_1 \to p_2$) $\neq$ Orientierung($p_1 \to q_1 \to q_2$) UND Orientierung($p_2 \to q_2 \to p_1$) $\neq$ Orientierung($p_2 \to q_2 \to q_1$).
   - Wenn ja, dann schneiden sich die zwei Linien.
   - Wenn nein, dann schneiden sich die Linien nicht.

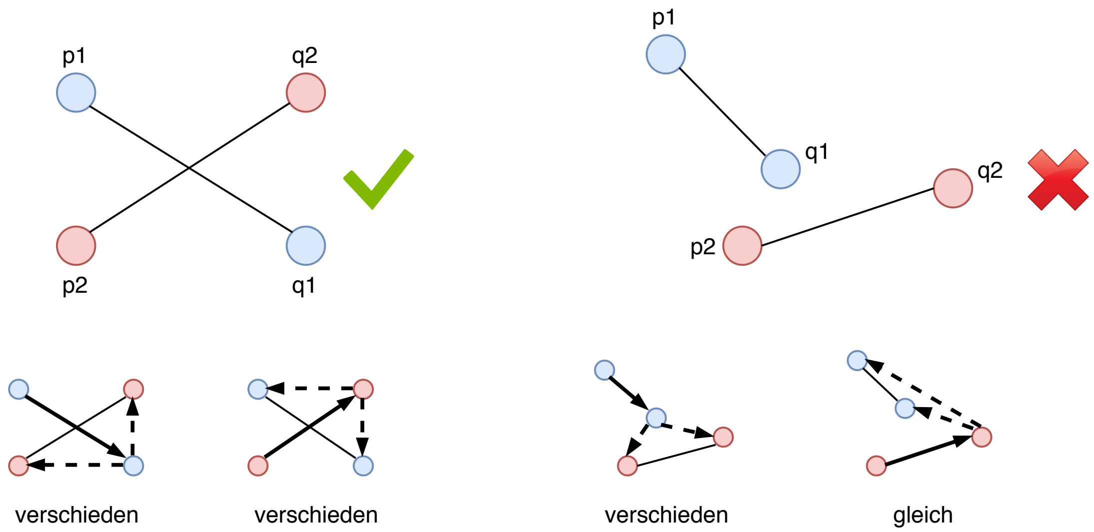


Abbildung 6: Beispiel zur Anwendung von Orientierung zur Überprüfung von Überschneidungen.

1.1.2 Konstruktion des Visibility Graphen

Eine Methode um den Visibility Graphen zu konstruieren, ist für jeden Knoten des Graphen jede Verbindung zu allen anderen Knoten auf Überschneidungen mit Kanten und Knoten des Polygongraphen (Graph mit unzusammenhängenden Komponenten die jeweils ein Hinderniss darstellen) zu überprüfen und nur Kanten in den Visibility Graph einzufügen, die keine Überschneidungen aufweisen. Für einen Knoten würde der Algorithmus folgendermaßen aussehen:

Algorithmus 1 : Visibility

Data : Knoten v, Polygongraph P
Result : Liste von Kanten die von v im Visibility Graph ausgehen
Initialisiere die Listen visible-edges, checked-nodes
for $n \in P$ **do**
  if $v \neq n$ *und* $n \notin$ *selbes Polygon wie* v **then**
    possible-edge = Kante von v zu n
    if *possible-edge* $\notin$ *checked-nodes* **then**
      Füge possible-edge zu checked hinzu
      obstacles = alle Kanten $\in P$ die sich mit possible-edge schneiden
      if *obstacles* $= \emptyset$ *und possible-edge schneidet keine Knoten* $\in P$ **then**
        Füge possible-edge zu visible-edges hinzu
      end
    end
  end
end
visible-edges zurück geben

Dieser Algorithmus wird für alle Knoten des Polygongraphen und den Start- und Endknoten (d.h Lisas Haus und Bus) ausgeführt.

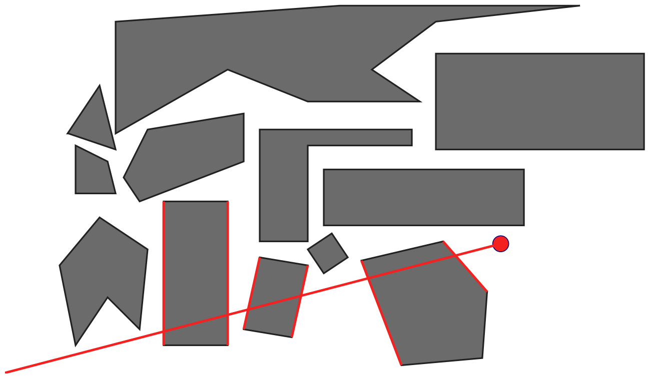


Abbildung 7: In diesem Fall wurden Überschneidungen gefunden.

Für n Knoten heißt das, dass für jeden Knoten $n-1$ Verbindungen gibt die mit $n-2$ Kanten (End- und Startknoten haben im Polygongraph noch keine Kanten) auf Überschneidung kontrolliert werden müssen. Dies bedeutet eine **Zeitkomplexität** von $O(n^3)$ für die Konstruktion.

1.1.3 Dijkstras Algorithmus

Dijkstras Algorithmus ist ein greedy Algorithmus zum Finden von kürzesten Wegen in Graphen (mit einer Zeitkomplexität von $O(n^2)$ im schlechtesten Fall, wird allerdings zum Beispiel ein Heap verwendet wie hier verbessert sich diese deutlich):

Algorithmus 2 : Dijkstra

Data : Startknoten start, Visibility Graph G
Result : In den Knoten von G ist jeweils die kürzeste Distanz zu start und der Vorgänger auf dem Weg zu start gespeichert

```
Initialisiere bei allen Knoten ∈ G Vorgänger prev = null, Distanz d = ∞ und visited = False
Initialisiere start.d = 0 und start.visited = True
Initialisiere einen Heap toexplore und füge start hinzu
while toexplore ≠ ∅ do
    v = Knoten mit geringsten d in toexplore
    for w ∈ Nachbarn von v do
        dist = v.d + Kantengewicht(v → w)
        if dist < w.d then
            w.d = dist
            w.prev = v
            if w.visited = False then
                w.visited = True
                Füge w zu toexplore hinzu
            end
        end
    end
end
```

1.2 Letztmöglicher Zeitpunkt Haus zu verlassen

1.2.1 Ohne Hindernisse

Man kann die Zeit zu der Lisa das Haus verlassen muss gewissermaßen als **Funktion** $t(m)$ sehen, die für jede y-Koordinate des Buses in Meter (gegeben als m) die korrespondierende Zeit t ausgibt. Dafür kann man zum Beispiel t als vergangene Sekunden seit Mitternacht sehen.
Sei $b(m)$ die Zeit die der Bus zur Position m braucht, $l(d)$ die Zeit die Lisa braucht, $d(m)$ die Distanz von Lisas Haus zur Position m, z die Zeit zu der der Bus los fährt, und x und y die Koordinaten von Lisas Haus. Damit entsteht mit Hilfe des Satz des Pythagoras die Funktion

$$t(m) = z + b(m) - l(d(m)) = 7,5 \cdot 60 \cdot 60 + \frac{m}{30/3,6} - \frac{\sqrt{x^2 + (m-y)^2}}{15/3,6}.$$

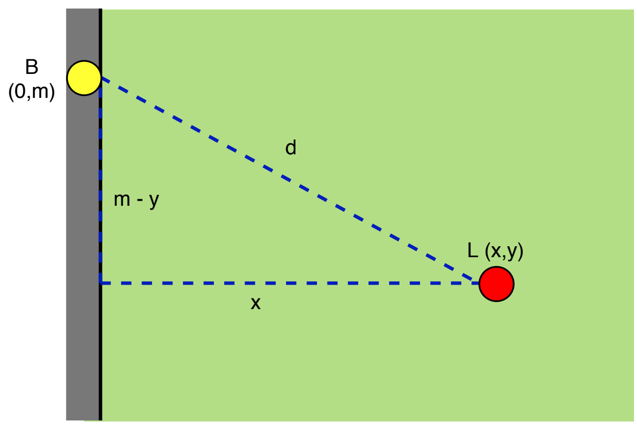


Abbildung 8: Situation bei gerader Strecke.

Der späteste Zeitpunkt kann durch das Maximum der Funktion gefunden werden. Im ersten Beispiel mit $x = 633$ und $y = 189$ würde das Maximum zum Beispiel bei $m = 189 + 211\sqrt{3} \approx 554.46$ liegen, was einer Uhrzeit von ungefähr 7:28:11 entspricht.

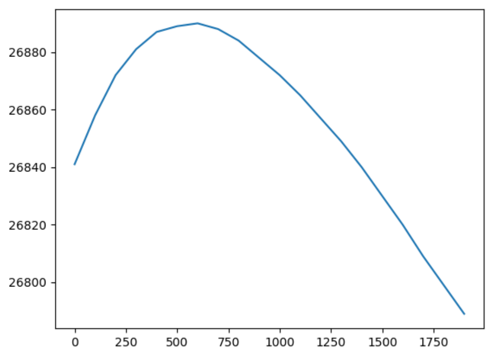


Abbildung 9: Das Maximum ist auch auf dem Graphen der Funktion $t(m)$ erkennbar.

1.2.2 Mit Hindernissen

Es ändern sich vor allem ein Aspekte sobald man das Problem inklusive der Hindernisse betrachtet. Man nicht mehr von einer konkreten Funktion für $d(m)$ ausgehen, da sich für jede Busposition der Visibility Graph ändern kann und somit auch die Distanz jedes Mal neu durch Dijkstra berechnet werden muss. Das bedeutet auch, das man für eine genaue Lösung nicht darum herum kommt, tatsächlich zumindest für einem Bereich für jedes mögliche m die benötigten Methoden auszuführen.

Um dies effizienter zu gestalten, wird zum einen eine weitere Methode eingeführt, die den Visibility Graphen aktualisiert wenn sich der Endpunkt des Buses ändert: da sich alle Knoten abgesehen vom Endknoten nicht veränderen, verändert sich ihre Sichtbarkeit auch nicht, deswegen muss nur für den Endknoten der Sichbarkeitsalgorithmus wieder ausgeführt werden.

Zum anderen findet man zunächst nur für jedes hundertste m die Startzeit um grob Abschätzen zu können wo das Maximum liegt. In der näheren Umgebung des gefundenen m mit der spätesten Startzeit wird das Program dann für jeden Meter ausgeführt, um eine genaue Startzeit zu finden.

1.3 Mögliche Verbesserungen und Erweiterungen

Es ist offensichtlich, dass eine Laufzeit von $O(n^3)$ für die Konstruktion des Visibility Graphen weit von optimal ist, weswegen es hilfreich wäre einen anderen Algorithmus zu verwenden. Es wäre zum Beispiel der **Rotational Sweep** Algorithmus mit einer Laufzeit von $O(n^2 log(n))$ möglich, der auf der Idee basiert, dass die Überprüfung von Überschneidungen in einer bestimmten Reihenfolge (in diesem Fall kreisförmig) dazu führt, dass die Informationen schon überprüfter Verbindungen helfen können weniger Verbindungen zu prüfen.

Des Weiterem wird im Moment beim Finden des spätesten Startzeitpunkts der erstmögliche Weg genommen, ohne zu kontrollieren ob es bei mehreren Möglichkeiten einen kürzeren Weg gibt (was für Lisa wahrscheinlich auch vom Interesse wäre).

Zudem wird momentan angenommen, dass die vorgegeben Hindernisse sich nicht überschneiden. Da diese aber existieren könnten, wäre es auch interessant zuvor zu prüfen ob manche Polygone kollidieren und falls ja, diese zusammenzuführen.

2 Umsetzung

Die Lösungsidee wurde in **Python** implementiert.

Dabei hat das Program folgende Elemente:

- **LisaRun:** Hauptdatei aus der das Program startet, mit Einlesen der Vorgabe und Ausgabe des Ergebnisses.
- **Graph:** Datei mit Knoten und Kanten Klassen (zur Darstellung der verschidenen Graphen in Form von einer **Adjazenzliste**) und allen Methoden relevant zur Graphenverarbeitung (unter anderem Dijkstra und Konstruktion des Visibility Graphen).

- **Vectors:** Datei mit einer Koordinaten Klasse und Vektormethoden wie Kreuzprodukt und Betrag.
- **Time:** Datei mit Klasse zur Darstellung von Zeit im Format Stunde:Minute:Sekunde und Methoden zum Verrechnen von Zeitangaben in Sekunden mit diesem Format.
- **Graphics** Datei mit Methoden zum Einlesen, Bearbeiten und Ausgeben von svg-Dateien mit Hilfe der xml Bibliothek lxml.etree.

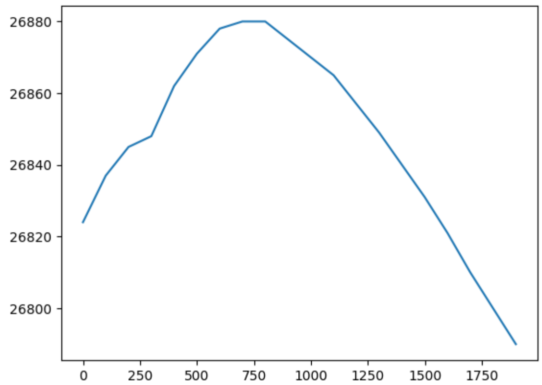
Da zur Lösung des Problems relativ viele Schleifen benötigt und Python ohnehin eine eher langsame Programmiersprache ist, wurde außerdem **Numba** als JIT-Compiler für einige Methoden benutzt um die Laufzeit zzu optimieren.

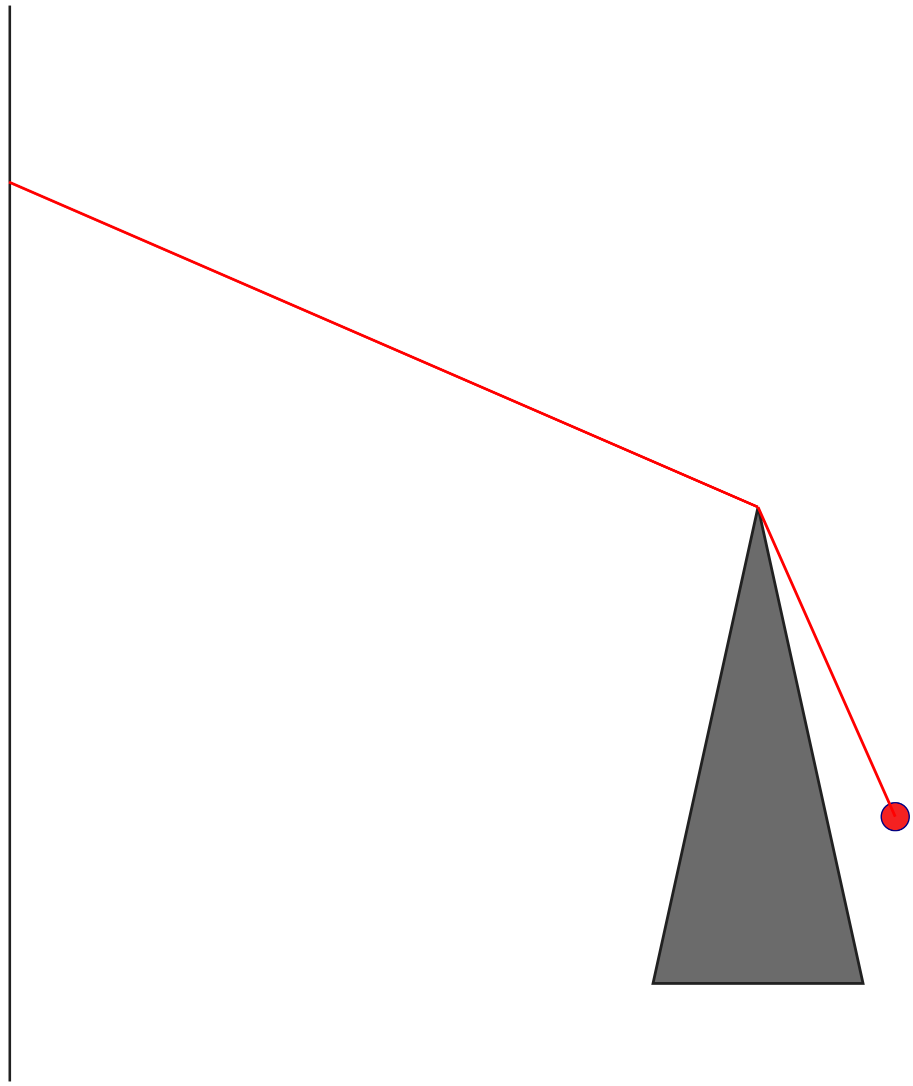
3 Beispiele

Für jedes Beispiel werden Lösungsdaten (Startzeit, Endzeit, y-Koordinate des Buses zum Treffpunkt, Lisas Laufdistanz und Laufdauer, und die durchlaufenen Eckpunkte in umgekehrter Reihenfolge), ein Graph der für jede Busposition in einem bestimmten Bereich die Zeit angibt, zu der Lisa das Haus verlassen muss (in Sekunden nach Mitternacht), und schließlich eine Visualisierung des Laufwegs.

Beispiel 1 (lisarennt1.txt):

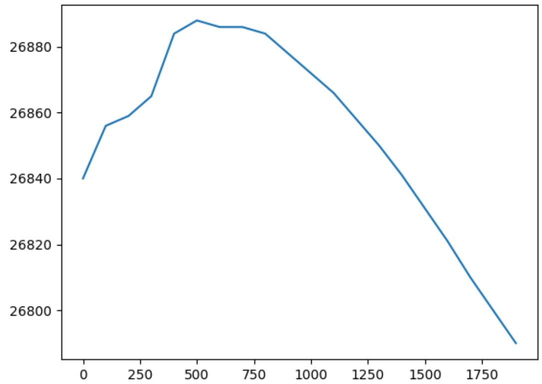
```
Startzeit: 7:28:00
Endzeit: 7:31:17
y-Koordinate Bus: 642 m
Laufdistanz: 824.89 m
Laufdauer: 3.28 min (= 3 min 17 sec)
B at (0|642)
<-- P1.0 at (535|410)
<-- L at (633|189)
```

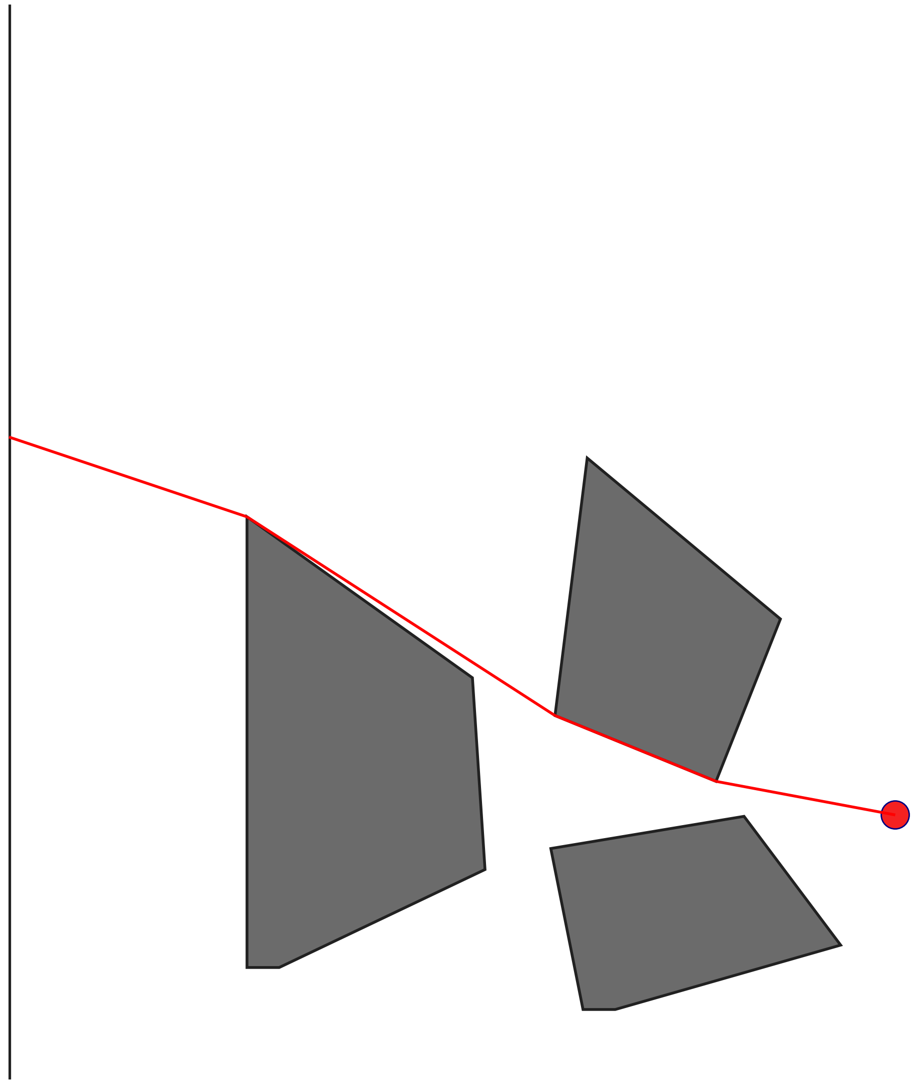




Beispiel 2 (lisarennt2.txt):

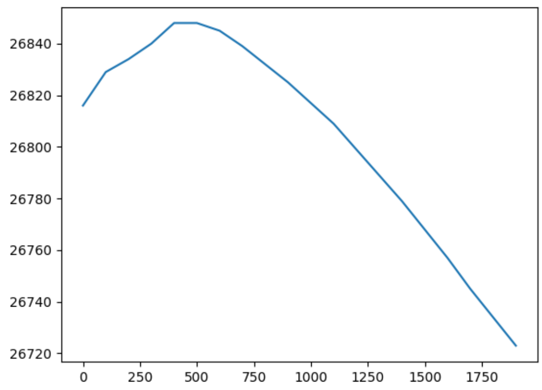
```
Startzeit: 7:28:09
Endzeit: 7:30:55
y-Koordinate Bus: 459 m
Laufdistanz: 695.61 m
Laufdauer: 2.77 min (= 2 min 46 sec)
B at (0|459)
<-- P3.4 at (170|402)
<-- P1.0 at (390|260)
<-- P1.1 at (505|213)
<-- L at (633|189)
```

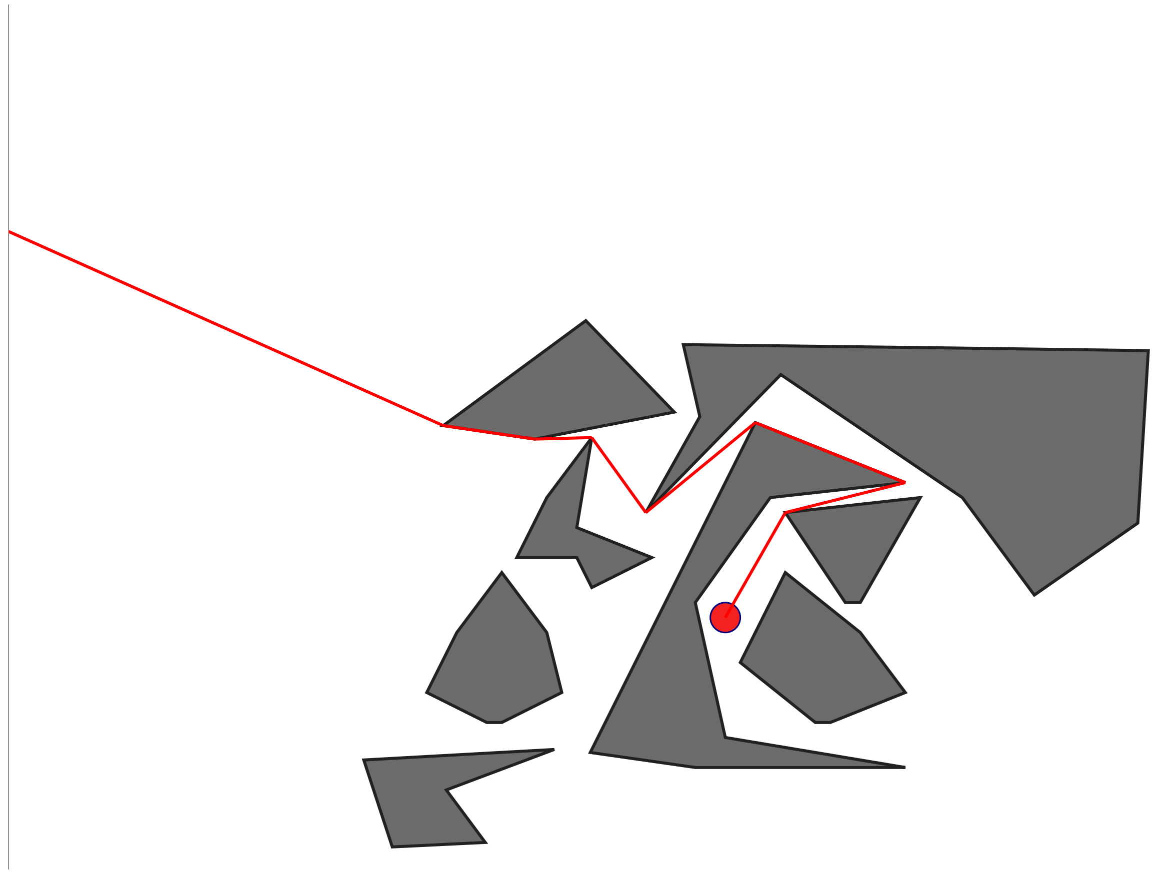




Beispiel 3 (lisarennt3.txt):

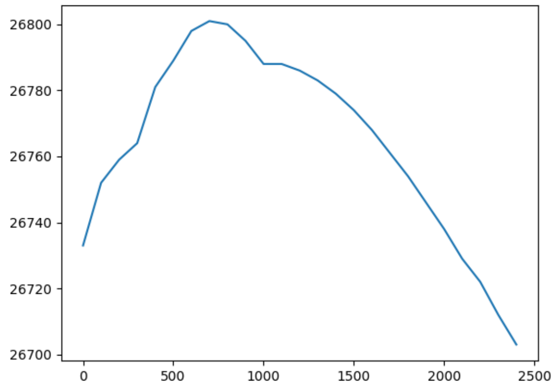
```
Startzeit: 7:27:29
Endzeit: 7:30:51
y-Koordinate Bus: 426 m
Laufdistanz: 845.28 m
Laufdauer: 3.37 min (= 3 min 22 sec)
B at (0|426)
<-- P6.3 at (291|296)
<-- P6.0 at (352|287)
<-- P5.4 at (390|288)
<-- P8.2 at (426|238)
<-- P3.7 at (499|298)
<-- P3.6 at (599|258)
<-- P2.3 at (519|238)
<-- L at (479|168)
```

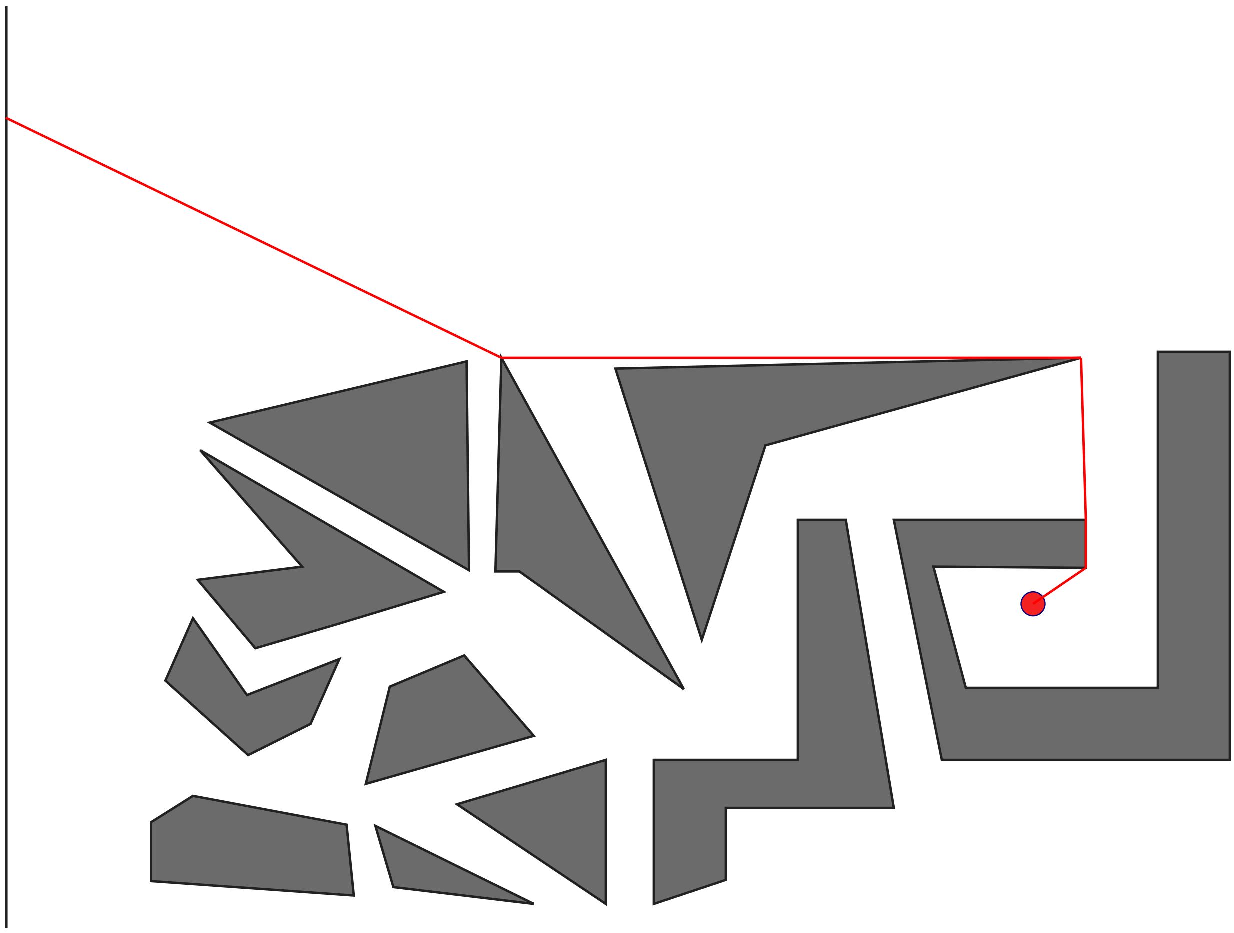




Beispiel 4 (lisarennt4.txt):

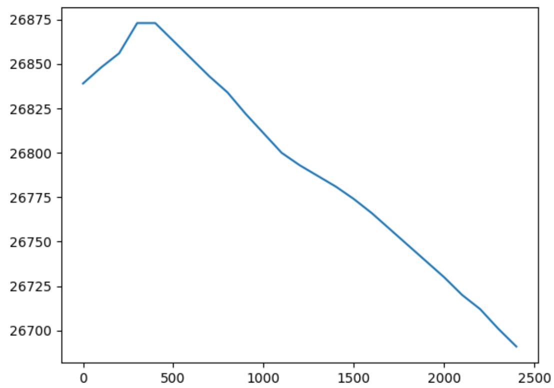
```
Startzeit: 7:26:41
Endzeit: 7:31:21
y-Koordinate Bus: 675 m
Laufdistanz: 1170.19 m
Laufdauer: 4.67 min (= 4 min 40 sec)
B at (0|675)
<-- P5.3 at (413|475)
<-- P10.2 at (896|475)
<-- P11.8 at (900|340)
<-- P11.7 at (900|300)
<-- L at (856|270)
```

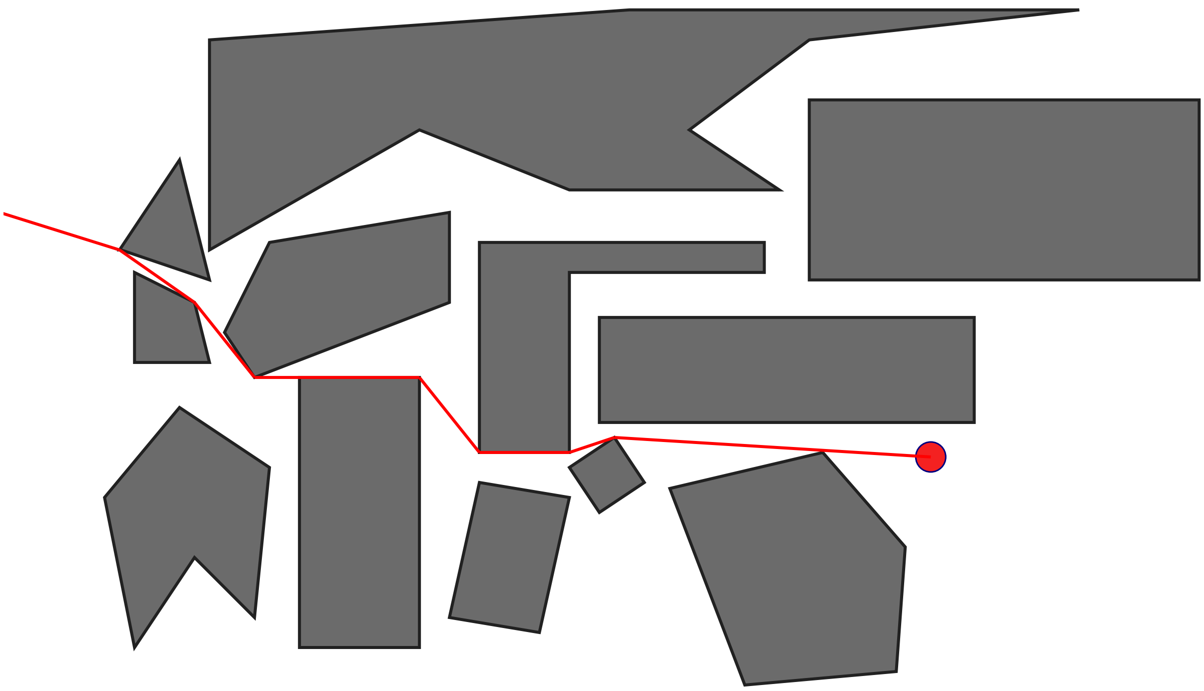




Beispiel 5 (lisarennt5.txt):

```
Startzeit: 7:27:55
Endzeit: 7:30:39
y-Koordinate Bus: 325 m
Laufdistanz: 685.93 m
Laufdauer: 2.73 min (= 2 min 44 sec)
B at (0|325)
<-- P12.0 at (80|300)
<-- P9.2 at (130|265)
<-- P6.0 at (170|215)
<-- P5.2 at (280|215)
<-- P3.0 at (320|165)
<-- P3.1 at (380|165)
<-- P8.2 at (410|175)
<-- L at (621|162)
```





4 Quellen

Theoretische Grundlagen:

- Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: *Computational Geometry - Algorithms and Applications*
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*
- Vorlesungsfolien aus: Simonas Šaltenis: *Advanced Algorithm Design and Analysis* (http://people.cs.aau.dk/ simas/aalg04/slides/aalg9.pdf)
- Vorlesungsfolien aus: Yaron Ostrovsky-Berman: *Computational Geometry* (http://www.cs.huji.ac.il/course/2004/compgeom/slides/CG-tirgul03-6spp.pdf)
- https://fribbels.github.io/shortestpath/writeup.html

Bibliotheken für Umsetzung:

- lxml.etree https://lxml.de/index.html
- numba http://numba.pydata.org/
- matplotlib https://matplotlib.org/index.html

Die Abbildungen in dieser Ausarbeitung sind alle selbständing erstellt, entweder mittels svg-Dateien oder https://draw.io/.

5 Quellcode

LisaRun.py:

```python
import ...

# Szenario
case = "lisarennt1"

# Geschwindigkeit in m/s
speed_lisa = 15.0 / 3.6
speed_bus = 30.0 / 3.6

# Daten einlesen
f = open(case+".txt", "r")
p = int(f.readline())
# Koordinaten Polygone
polygons = []
for i in range(p):
  line = f.readline()
  coord = []
  count = 0
  for number in line.split():
    if count % 2 != 0:
      coord.append(vec.Coord(pos_x=int(number)))
    if count % 2 == 0 and count != 0:
      coord[int((count / 2) - 1)].set_y(int(number))
    count += 1
  polygons.append(coord)
# Koordinaten Haus
line = f.readline()
coord_h = []
for number in line.split():
  coord_h.append(int(number))
start_x = coord_h[0]
start_y = coord_h[1]
f.close()
# Startkoordinaten
start = vec.Coord(start_x, start_y)

def shortest_runtime(d, obj):
  if obj == "Lisa":
    return int(d / speed_lisa)
  if obj == "Bus":
    return int(d / speed_bus)

# Durchgehen moeglicher Buspunkte
@jit
def find_leaving_time(lower, higher, step, p_graph, v_graph, start_node,
    end_node):
  departure = Time.Time(7, 30, 0)
  meter = 0
  gr.dijkstra(v_graph, start_node)
  distance = end_node.distance
  # Zeit die Lisa braucht
  lt = shortest_runtime(distance, "Lisa")
```

```
  # Zeit die Bus braucht
  bt = shortest_runtime(meter, "Bus")
  # Zeit bei Treffpunkt, d.h Endzeit
  meeting_time = Time.add_seconds(departure, bt)
  # Startzeit
  leaving_time = Time.subtract_seconds(meeting_time, lt)
  # Initialisierung der Bestzeit (mit dazugeh[U+FFFD]riger y-Koordinate)
  current_best_time = leaving_time
  current_best_meter = meter
  times = []

  # Finden des spaetesten Hausverlasszeitpunkts
  for meter in range(lower, higher):
    if meter % step == 0:
      end_node = gr.Node(vec.Coord(0, meter), "B")
      gr.update_visibility(v_graph, p_graph, end_node)
      gr.dijkstra(v_graph, start_node)
      distance = end_node.distance
      lt = shortest_runtime(distance, "Lisa")
      bt = shortest_runtime(meter, "Bus")
      meeting_time = Time.add_seconds(departure, bt)
      leaving_time = Time.subtract_seconds(meeting_time, lt)
      if leaving_time > current_best_time:
        current_best_time = leaving_time
        current_best_meter = meter
      times.append(leaving_time.seconds_from_midnight())

  # Plot of possible leaving times
  x = np.arange(lower, higher, step)
  plt.plot(x, times)
  plt.show()

  return current_best_meter

# Ausgabe der Ergebnisse
def get_information(meter, v_graph, p_graph, start_node):
  departure = Time.Time(7, 30, 0)
  end_node = gr.Node(vec.Coord(0, meter), "B")
  gr.update_visibility(v_graph, p_graph, end_node)
  gr.dijkstra(v_graph, start_node)
  distance = end_node.distance
  bt = shortest_runtime(meter, "Bus")
  lt = shortest_runtime(distance, "Lisa")
  meeting_time = Time.add_seconds(departure, bt)
  leaving_time = Time.subtract_seconds(meeting_time, lt)

  print("Startzeit:␣{}".format(leaving_time))
  print("Endzeit:␣{}".format(meeting_time))
  print("y-Koordinate␣Bus:␣{}␣m".format(meter))
  print("Laufdistanz:␣{}␣m".format(distance))
  print("Laufdauer:␣{}␣min".format(Time.sec_in_min(lt)))
  print(gr.traversed_nodes(v_graph, end_node))

  svg = Graphics.read_svg(case + ".svg")
  Graphics.visualise_path(svg, v_graph, end_node)
```

```
  Graphics.output_svg(svg, "shortest_path_final")

# Initialisierung
# Polygonhindernisse
polygon_graph = gr.build_polygon_graph(polygons)
# Lisas Haus
start_n = gr.Node(start, "L")
# Position Bus am Anfang
end_n = gr.Node(vec.Coord(0, 0), "B")
# Visibility Graph am Anfang
visibility_edges = gr.construct_visibility_graph_brute_force(start_n, end_n,
    polygon_graph)
visibility_graph = gr.combine_graphs(visibility_edges, polygon_graph, start_n,
    end_n)

# Aufrufen der Hauptmethoden (mit Optimisierung des betrachteten Wertebereichs)
best1 = find_leaving_time(0, 2000, 100, polygon_graph, visibility_graph, start_n
    , end_n)
best2 = find_leaving_time(best1-100, best1+100, 1, polygon_graph,
    visibility_graph, start_n, end_n)
get_information(best2, polygon_graph, visibility_graph, start_n)
```

Graph.py:

```
import ...

# Klassen und Methoden zur Konstruktion und Verarbeitung von Graphen

# Knoten eines als Adjazenzliste dargestellten Graphen
class Node:
  def __init__(self, coordinates=vec.Coord(-1, -1), id="undefined"):
    self.coord = coordinates
    self.neighbours = []
    self.id = id
    # Hilfsattribute fuer Dijkstra
    self.distance = 0
    self.visited = False
    self.prev = ""

  def add_neighbour(self, edge):
    contains = False
    for n in self.neighbours:
      if edge.neighbour2.id == n.neighbour2.id:
        contains = True
    if not contains:
      self.neighbours.append(edge)

  def is_proper(self):
    return self.coord.x > 0 and self.coord.y > 0

  def __lt__(self, other):
    return self.distance < other.distance

```

```
# Um gewichteten Graphen darzustellen, werden Nachbarn als Kanten mit Gewicht
    und 2 Knoten gespeichert
class Edge:
  def __init__(self, weight, neighbour1, neighbour2, id="undefined"):
    self.weight = weight
    self.neighbour1 = neighbour1
    self.neighbour2 = neighbour2
    self.id = id


# baut unzusammenhaengenden Graphen mit Polygonen als Komponenten
# (Annahme: Koordinaten eines Polygons sind in Reihenfolge der Verbindungen
    angegeben)
# input: Liste mit Koordinatenlisten fuer jedes Polygon
# output: Liste mit Knoten des Graphen
@jit
def build_polygon_graph(polygons):
  polygon_graph = {}
  count_p = 0
  for p in polygons:
    previous = Node()
    first = Node()
    count_n = 0
    for n in p:
      new_id = "P{}.{}".format(count_p+1, count_n)
      polygon_graph[new_id] = Node(vec.Coord(n.x, n.y), new_id)
      current = polygon_graph[new_id]
      if previous.is_proper():
        edge_id = "E{}.{}".format(count_p+1, count_n)
        w = compute_weight(previous, current)
        previous.add_neighbour(Edge(w, previous, current, edge_id))
        current.add_neighbour(Edge(w, current, previous, edge_id))
      if count_n == 0:
        first = current
      previous = current
      count_n += 1
      if count_n == len(p):
        edge_id = "E{}.{}".format(count_p + 1, count_n)
        w = compute_weight(current, first)
        current.add_neighbour(Edge(w, current, first, edge_id))
        first.add_neighbour(Edge(w, first, current, edge_id))
    count_p += 1
  return polygon_graph


# Hilfsmethode um Distanz (= Kantengewicht) zwischen zwei Knoten zu berechnen
def compute_weight(n1, n2):
  x = n2.coord.x - n1.coord.x
  y = n2.coord.y - n1.coord.y
  return math.sqrt((x ** 2) + (y ** 2))


# Hilfsmethode zur schriftlichen Visualisierung eines Graphen
def print_graph(graph):
  string = ""
```

```
  for n in graph.values():
    string += "Node {} has {} neighbours ( ".format(n.id, len(n.neighbours))
    for edge in n.neighbours:
      string += "{} ".format(edge.neighbour2.id)
    string += ")"
    string += " -- and previous is {}\n".format(n.prev)
  print(string)


# Anzahl von Polygonkomponenten in einem Polygongraph
def number_of_polygons(graph):
  polygons = []
  for n in graph.values():
    pol_id = n.id.split(".")[0]
    if not polygons.__contains__(pol_id):
      polygons.append(pol_id)
  return len(polygons)


# Ueberprueft ob zwei Knoten zum selben Polygon gehoeren
def element_of_same_polygon(n1, n2):
  if len(n1.id) > 1 and len(n2.id) > 1:
    id1 = n1.id
    id2 = n2.id
    s1 = id1.replace("P", "").split(".")
    s2 = id2.replace("P", "").split(".")
    return s1[0] == s2[0]
  else:
    return False


# Kombiniert eine Listen von Kanten (Visibility Graph) mit einem Graph (Polygon Graph)
@jit
def combine_graphs(edges, pol_graph, start, end):
  graph = pol_graph
  graph[start.id] = start
  graph[end.id] = end
  for e in edges:
    graph[e.neighbour1.id].add_neighbour(e)
    graph[e.neighbour2.id].add_neighbour(Edge(e.weight, e.neighbour2, e.neighbour1, e.id))
  return graph


# Dijkstra Algorithmus zum finden von kuerzesten Wegen
@jit
def dijkstra(graph, start):
  for node in graph.values():
    node.distance = float("inf")
    node.previous = ""
    node.visited = False
  start.distance = 0
  start.visited = True
  toexplore = []
```

```
  heapq.heappush(toexplore, start)
  while toexplore:
    v = heapq.heappop(toexplore)
    for edge in v.neighbours:
      w = edge.neighbour2
      dist_w = v.distance + edge.weight
      if dist_w < w.distance:
        w.distance = dist_w
        w.prev = v.id
        if w.visited is False:
          w.visited = True
          heapq.heappush(toexplore, w)


def traversed_nodes(graph, end):
  if end.distance == 0:
    return "{} at {}".format(end.id, end.coord)
  return "{} at {} <-- ".format(end.id, end.coord) + traversed_nodes(graph, graph[end.prev])

# Methoden zur Konstruktion eines Visibility Graphs


# Hauptmethode
@jit
def construct_visibility_graph_brute_force(start, end, graph):
  visibility_graph = []
  full_graph = graph
  full_graph[start.id] = start
  full_graph[end.id] = end
  for v in graph.values():
    visibility_graph = visibility_graph + visible(v, full_graph)
  return visibility_graph


# Findet alle Verbindungen die vom Knoten v sichtbar sind
@jit
def visible(v, graph):
  v_graph = []
  checked = []
  for n in graph.values():
    if v != n and not element_of_same_polygon(v, n):
      poss_edge = Edge(compute_weight(v, n), v, n, id="{}-{}".format(v.id, n.id))
      if not checked.__contains__("{}-{}".format(n.id, v.id)):
        checked.append(poss_edge.id)
        obstacles = intersected_lines(poss_edge, graph)
        if len(obstacles) == 0 and intersected_nodes(poss_edge, graph):
          v_graph.append(poss_edge)
  return v_graph


# Input: zwei Knoten und ein Polygongraph
# Output: Liste mit allen Kanten die die Verbinding zwischen den Knoten schneidet
```

```
@jit
def intersected_lines(poss_edge, pol_graph):
  checked = []
  intersected = []
  for node in pol_graph.values():
    for edge in node.neighbours:
      if not checked.__contains__(edge.id):
        checked.append(edge.id)
        if intersect(edge, poss_edge):
          intersected.append(edge)
  return intersected


# Ueberprueft ob zwei Strecken (gegeben als Kanten) sich ueberschneiden
@jit
def intersect(e1, e2):
  p1 = e1.neighbour1.coord
  q1 = e1.neighbour2.coord
  p2 = e2.neighbour1.coord
  q2 = e2.neighbour2.coord
  if slope(p1, q1) == slope(p2, q2) and slope(p1, q1) != 0 and slope(p2, q2) !=
   0:
    return False
  else:
    if compare_orientation(orientation(p1, q1, p2), orientation(p1, q1, q2)) \
        and compare_orientation(orientation(p2, q2, p1), orientation(p2, q2, q1)
   ):
      return True
    else:
      return False


# Da Beruehrung nicht als Ueberschneiden gilt
def compare_orientation(o1, o2):
  if o1 == "collinear" or o2 == "collinear":
    return False
  else:
    return o1 != o2


# Orientierung der Verbindung dreier Punkte c1 -> c2 -> c3
def orientation(coord1, coord2, coord3):
  c1 = vec.get_vector(coord1, coord2)
  c2 = vec.get_vector(coord2, coord3)
  d = vec.cross_product_direction(c1, c2)
  if d > 0:
    return "counter"
  else:
    if d < 0:
      return "clock"
    else:
      return "collinear"


# Steigung zwischen 2 Punkten
```

```
def slope(p1, p2):
  if p1.x == p2.x:
    return 0  # Senkrechte
  else:
    return (p2.y - p1.y)/(p2.x - p1.x)


# Ueberprueft ob eine moegliche Kante einen Knoten schneidet
@jit
def intersected_nodes(poss_edge, pol_graph):
  for n in pol_graph.values():
    if node_intersect(n, poss_edge):
      return False
  return True


# Ueberprueft eine Kante einen bestimmten Knoten schneidet
@jit
def node_intersect(n, e):
  m = slope(e.neighbour1.coord, e.neighbour2.coord)
  t = e.neighbour1.coord.y - m * e.neighbour1.coord.x
  con1 = min(e.neighbour1.coord.x, e.neighbour2.coord.x) < n.coord.x < max(e.
   neighbour1.coord.x, e.neighbour2.coord.x)
  con2 = min(e.neighbour1.coord.y, e.neighbour2.coord.y) < n.coord.y < max(e.
   neighbour1.coord.y, e.neighbour2.coord.y)
  if m*n.coord.x + t == n.coord.y and con1 and con2:
    return True
  else:
    return False


# bei Positionsveraenderung des Buses
@jit
def update_visibility(v_graph, p_graph, end_node):
  for edge1 in v_graph[end_node.id].neighbours:
    for edge2 in edge1.neighbour2.neighbours:
      if edge2.neighbour2.id == end_node.id:
        edge1.neighbour2.neighbours.remove(edge2)
  v_graph[end_node.id] = end_node
  end_node.neighbours = visible(end_node, p_graph)
  for edge3 in end_node.neighbours:
    edge3.neighbour2.add_neighbour(Edge(compute_weight(edge3.neighbour2,
   end_node), edge3.neighbour2, end_node))
```

Vectors.py:

```
import ...

# Vektormethoden zur Anwending auf Koordinaten (R^2)

# Vektor zwischen 2 Koordinatenpunkten
def get_vector(c1, c2):
    vector = Coord()
    vector.set_x(c2.x - c1.x)
    vector.set_y(c2.y - c1.y)
```

```
    return vector


# Skalarprodukt
def dot_product(c1, c2):
    return c1.x*c2.x + c1.y*c2.y


# Betrag eines Vektors
def norm(v):
    return math.sqrt((v.x ** 2) + (v.y ** 2))


# Winkel zwischen 2 Vektoren
def compute_angle(v1, v2):
    alpha = math.acos(dot_product(v1, v2)/(norm(v1)*norm(v2)))
    return math.degrees(alpha)


# Laenge des aus zwei 2-dimensionalen Vektoren durch Kreuzprodukt entstehenden
    senkrechten Vektors
def cross_product_direction(v1, v2):
    return v1.x*v2.y - v1.y*v2.x


# Koordinatenklasse zur besseren Handhabung von x und y Werten, inklusive
    Vektormethoden
class Coord:
    def __init__(self, pos_x=0, pos_y=0):
        self.x = pos_x
        self.y = pos_y

    def set_x(self, x):
        self.x = x

    def set_y(self, y):
        self.y = y

    def __str__(self):
        return "({}|{})".format(self.x, self.y)

    def __repr__(self):
        return "({}|{})".format(self.x, self.y)
```

Time.py:

```
class Time:
  def __init__(self, hour, minute, second):
    self.hour = hour
    self.minute = minute
    self.second = second

  # Vergleichsmethode
  def __lt__(self, other):
    if self.hour < other.hour:
```

```
      return True
    else:
      if self.hour == other.hour:
        if self.minute < other.minute:
          return True
        else:
          if self.minute == other.minute:
            if self.second < other.second:
              return True
    return False

  def __str__(self):
    return "{}:{}:{}".format(self.hour, self.minute, self.second)

  def __repr__(self):
    return "{}:{}:{}".format(self.hour, self.minute, self.second)

  def seconds_from_midnight(self):
    return min_in_sec(self.hour * 60) + min_in_sec(self.minute) + self.second


# Addition von Sekunden auf eine Uhrzeit (Uebergang auf anderen Tag nicht moeglich)
def add_seconds(t, s):
  new_time = Time(t.hour, t.minute, t.second)
  if s < 60:
    sec = t.second + s
  else:
    if s < 3600:
      mod = s % 60
      sec = t.second + mod
      minu = t.minute + (s - mod)/60
    else:
      mod1 = s % 3600
      mod2 = mod1 % 60
      sec = t.second + mod2
      minu = t.minute + (mod1 - mod2)/60
      new_time.hour = t.hour + (s - mod1)/3600
    if minu < 60:
      new_time.minute = minu
    else:
      new_time.hour += 1
      new_time.minute = minu - 60
  if sec < 60:
    new_time.second = sec
  else:
    new_time.minute += 1
    new_time.second = sec - 60
  new_time.hour = int(new_time.hour)
  new_time.minute = int(new_time.minute)
  new_time.second = int(new_time.second)
  return new_time


# Subtraktion von Sekunden von einer Uhrzeit (Uebergang auf anderen Tag nicht
```

```
    moeglich)
def subtract_seconds(t, s):
  ... (selbes Prinzip wie Addition)


# Hilfsmethode
def min_in_sec(mins):
  return mins * 60


# Hilfsmethode
def sec_in_min(secs):
  return secs / 60
```

Graphics.py:

```
from lxml import etree as et

# Methoden zur graphischen Ausgabe mittels svg


def read_svg(xml_file):
    with open(xml_file) as xf:
        xml = xf.read()
    root = et.fromstring(xml)
    return root


def output_svg(root, filename):
    tree = et.ElementTree(root)
    tree.write("{}.svg".format(filename), pretty_print=True)


def add_line(root, x1, y1, x2, y2, colour):
    attr = {
        "fill": "none",
        "stroke": colour,
        "stroke-width": "2"
    }
    root[0][0].append(et.Element("line", attr, x1=x1, y1=y1, x2=x2, y2=y2))
    return root


def visualise_graph(root, graph):
    for n in graph.values():
        for e in n.neighbours:
            root = add_line(root, str(n.coord.x), str(n.coord.y),
                                str(e.neighbour2.coord.x), str(e.neighbour2.coord.y)
    , "#32CD32")


def visualise_lines(root, lines):
    ... (gleiches Prinzip wie visualise_graph)

```

```
def visualise_path(root, graph, end):
    ... (gleiches Prinzip wie visualise_graph))

def add_text(root, text, x, y):
    ... (gleiches Prinzip wie visualise_graph)
```