Aufgabe 3: Abbiegen

Richard Wohlbold
Team-ID: 00487

16. April 2020

Inhaltsverzeichnis

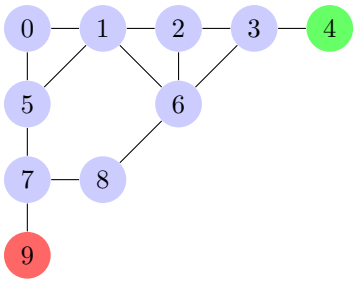


Abbildung 1: Beispiel aus der Aufgabe, als *node-based-graph*, zur Übersichtlichkeit ohne entsprechende Gewichte, repräsentiert. Knoten 9 ist dabei der Ausgangsknoten und Knoten 4 der Zielknoten

1 Lösungsidee

Für die Beantwortung von Bilals Frage muss das Straßennetz zunächst als Datenstruktur repräsentiert werden. Daraufhin arbeite ich mit einer modifizierten Version des Dijkstra-Shortest-Path-Algorithmus, um einen Weg zu finden, dessen Länge innerhalb der gegebenen Grenzen liegt und auf dem so wenig wie möglich abgebogen werden muss.

1.1 Darstellung des Straßennetzes

Straßennetze werden allgemein als gewichtete Graphen, d.h. Mengen von Knoten und Kanten repräsentiert. Grafisch wird ein Knoten dabei das Punkt oder Kreis repräsentiert, eine Kante als Linie zwischen zwei Knoten. Normalerweise entsprechen dabei Knoten den Kreuzungen und Kanten den Straßen zwischen den Kreuzungen, ein sog. *node-based-graph* [1]. Dabei hat jede Kante ein Gewicht, d.h. einen ihr zugewiesenen numerischen Wert, der meistens dem Zeit entspricht, der für das Befahren aufgewendet wird. In einfachen Szenarien, in denen die Geschwindigkeit auf allen Straßen als konstant angenommen wird, wird oft die Länge der entsprechenden Straße als Gewicht verwendet, wie es in dieser Aufgabe der Fall ist. Das Beispiel aus der Aufgabenstellung ist in Abbildung 1.1 als *node-based-graph* repräsentiert (ohne Gewichte).

Wenn jedoch auch das Abbiegen miteinbezogen werden soll, wird die Darstellung des Straßennetzes schwieriger. Oft soll beispielsweise auf einer Route weniger abgebogen werden, weshalb Abbiegevorgängen Gewichte zugeordnet werden, sogenannte *turn costs* [1]. Bei diesen Gewichten stößt die traditionelle Repräsentation der Straßennetze an ihre Grenzen, sodass bei *turn costs* der Ansatz eines *edge-based-graph*s gewählt wird. Dabei wird jede Straße als Knoten repräsentiert. Für jede Möglichkeit, von einer Straße auf eine andere zu gelangen, wird eine Kante zwischen den zwei entsprechenden Knoten hinzugefügt. Falls zwischen den Straßen abgebogen werden muss, kann dies entweder als Gewicht der Kante dargestellt werden oder ein einfaches Wahr oder Falsch für jede Kante gespeichert werden. Die Länge jeder Straße muss für einen *edge-based-graph* für jeden Knoten statt für jede Kante gespeichert werden. Das Beispiel aus der Aufgabe ist in Abbildung 2 als *edge-based-graph* repräsentiert.

1.2 Finden des optimalen Weges

Das Finden des optimalen Weges löse ich mithilfe einer modifizierten Version des Dijkstra-Shortest-Path-Algorithmus. Dieser benutzt eine PriorityQueue, um Kreuzungen geordnet nach ihrer Distanz zum Ausgangspunkt zu „besuchen“. Wird eine Straße ”besucht”, werden alle Straßen, auf die man ausgehend von dieser Straße gelangen kann, der PriorityQueue mit ihrer Distanz vom Ursprung hinzugefügt. Dabei wird die Distanz vom Ursprung einer Straße aus der Distanz vom Ursprung der gerade besuchten Straße plus die Länge der nächsten Straße berechnet. Wird eine Straße besucht, die an die Zielkreuzung angrenzt, terminiert der Algorithmus und der kürzeste Weg kann rekonstruiert werden. Durch die Verwendung des *edge-based-graph* kann neben der Distanz vom Urspung jeweils auch die Anzahl an Abbiegevorgängen der PriorityQueue hinzugefügt werden. So kann für jede besuchte Straße auch die Anzahl an Abbiegevorgängen bestimmt werden, die man braucht, um auf sie zu gelangen. Die bereits besuchten Straßen werden einer Menge hinzugefügt. Eine Straße wird nicht besucht, falls sie bereits besucht wurde.

Der erste Schritt meines Programms ist es, den Dijkstra-Shortest-Path-Algorithmus wie oben beschrieben auf den *edge-based-graph* anzuwenden. Dadurch wird die kürzeste mögliche Distanz und die entsprechende Anzahl an Abbiegevorgängen vom Start zum Ziel n ermittelt.

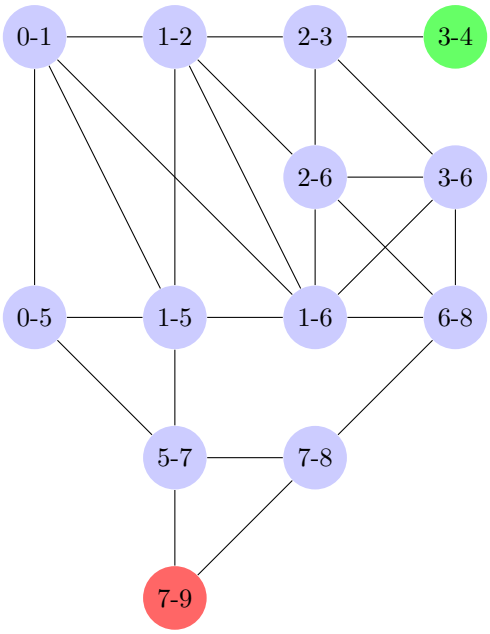


Abbildung 2: Beispiel aus der Aufgabe, als *edge-based-graph* repräsentiert, hier ohne Gewicht

Im Folgenden soll nun versucht werden, den kürzesten Weg mit einer gewissen Anzahl an Abbiegevorgängen m zu finden. Anfangs ist $m = n - 1$, falls ein solcher Weg existiert und Bilals Toleranz nicht überschreitet, wird m um jeweils 1 verringert, bis eine der Bedingungen verletzt ist. Dadurch wird der Weg mit der niedrigsten Anzahl an Abbiegevorgängen, der Bilals Anforderungen genügt, gefunden.

Wie auch anfangs, verfolgt die modifizierte Version des Dijkstra-Algorithmus die Anzahl der Abbiegevorgänge, die benötigt werden, um auf eine Straße (zu einem Knoten) zu gelangen. Anders als zuvor fügt jedoch der Algorithmus der PriorityQueue keine Straßen hinzu, für die $k > m$ Mal abgebogen werden muss. Zusätzlich wird es erlaubt, Knoten öfter zu besuchen, sofern bei den neuen Besuchen die Zahl der Abbiegevorgänge auf der Route geringer ist, als vorher. Dazu wird die oben beschriebene Menge so abgeändert, dass sie zusätzlich zur Straße die Anzahl an Abbiegevorgängen speichert.

Somit werden nur Routen gefunden, die maximal m Abbiegevorgänge haben. Auch werden keine Routen ignoriert, die zwar etwas länger sind, aber trotzdem die entsprechende Anzahl an Abbiegevorgängen aufweisen könnten. Von allen Routen, die k Abbiegevorgänge haben, ist die Resultierende aufgrund des Dijkstra-Shortest-Path-Algorithmus auch die Kürzeste.

2 Umsetzung

Ich habe die Lösungsidee in Python 3 umgesetzt.

2.1 Aus der Eingabedatei zum node-based-graph

Dazu wird zuerst die Eingabedatei in einen *node-based-graph* eingelesen. Die eingelesenen Kreuzungen werden als `dict` repräsentiert: Dabei erhält jede Kreuzung einen Schlüssel, die sie eindeutig identifiziert. Die Schlüssel der Kreuzung sind jeweils die Schlüssel des `dict`s. Die Werte des `dict`s sind die Koordinaten der Kreuzung.

Auch die Straßen werden als `dict` repräsentiert: Dabei werden als Schüssel der Schlüssel der Startkreuzung der Straße verwendet und als Wert eine Menge mit den Schlüsseln aller Kreuzungen, mit denen die Startkreuzung verbunden ist.

Das Beispiel aus der Aufgabe wird folgendermaßen repräsentiert:

```
junctions = {
  0: (0, 0),
  1: (0, 1),
  2: (0, 2),
  3: (0, 3),
  4: (1, 1),
```

```
  5: (1, 3),
  6: (2, 2),
  7: (2, 3),
  8: (3, 3),
  9: (4, 3)
}

roads = {
  0: {1},
  1: {0, 2, 4},
  2: {1, 3, 4, 5},
  3: {2, 5},
  4: {1, 2, 6},
  5: {2, 3, 6, 7},
  6: {8, 4, 5, 7},
  7: {8, 5, 6},
  8: {9, 6, 7},
  9: {8}
}
```

2.2 Vom node-based-graph zum edge-based-graph

Als nächster Schritt wandelt das Programm den *node-based-graph* in einen *edge-based-graph* um. Alle Straßen erhalten zuerst einheitliche Namen: Dazu werden die Schlüssel ihrer Start- und Endkreuzungen sortiert und mit einem Unterstrich verbunden. Daraufhin werden sie mit ihrer Länge als Knoten in einem `dict` (`nodes`) gespeichert. Die Kanten des *edge-based-graphs* bilden die Kreuzungen. Dabei wird jede Möglichkeit, auf eine andere Straße zu wechseln, einzeln zusammen mit der Information, ob für diesen Wechsel abgebogen werden muss, gespeichert. Dies geschieht erneut in einem `dict`, dessen Schlüssel der Name einer Straße bildet. Der Wert ist jeweils eine Liste von `tuplen`, in denen der Name der nächsten Straße und ob man geradeaus weiterfahren muss, gespeichert ist. Durch die Umwandlung in einen *edge-based-graph* gibt es nicht mehr nurnoch ein Ziel, sondern eine Menge an Straßen, von denen man das Ziel sofort erreicht. Im Programm wird dies als *set* repräsentiert, das die Namen aller Straßen, die mit der Zielkreuzung verbunden sind, enthält. Analog wird auch ein `set` erstellt, das die Namen aller Ausgangsstraßen enthält.

Um herauszufinden, ob man bei einem Straßenwechsel abbiegen muss, wird der Winkel zwischen den beiden Vektoren der Straßen berechnet. Es gilt:

$$\vec{a} * \vec{b} = |\vec{a}||\vec{b}| \cos\alpha$$
$$\Leftrightarrow \cos\alpha = \frac{\vec{a} * \vec{b}}{|\vec{a}||\vec{b}|}$$

Da $\alpha = 180°$

$$\frac{\vec{a} * \vec{b}}{|\vec{a}||\vec{b}|} = -1$$

Für das Beispiel aus der Aufgabe sehen die Datenstrukturen folgendermaßen aus (mit Auslassungen):

```
nodes = {
  '0_1': 1.0,
  '1_2': 1.0,
  '1_4': 1.0,
  '2_3': 1.0,
  '2_4': 1.4142135623730951,
  '2_5': 1.4142135623730951,
  '3_5': 1.0,
  '4_6': 1.4142135623730951,
  '5_6': 1.4142135623730951,
  '5_7': 1.0,
  '6_8': 1.4142135623730951,
  '6_7': 1.0,
  '7_8': 1.0,
```

```
  '8_9': 1.0
 }

edges = {
 '0_1': {
  ('1_2', True),
  ('1_4', False)
},
'1_2': {
  ('2_5', False),
  ('2_3', True),
  ('2_4', False),
  ('0_1', True),
  ('1_4', False)
  },
'1_4': {
  ('2_4', False),
  ('4_6', False),
  ('0_1', False),
  ('1_2', False)},
'2_3': {
  ('2_4', False),
  ('1_2', True),
  ('3_5', False),
  ('2_5', False)
},
...
}

sources = {'0_1'}

targets = {'8_9'}
```

Im Folgenden wird der Algorithmus aus Abschnitt 1 ausgeführt. Dazu wird eine Funktion `dijkstra` definiert, der den kürzesten Weg zum Ziel mit höchstens `number_turns` Abbiegevorgängen findet und zurückgibt. Diese arbeitet mithilfer der `PriorityQueue` aus dem Paket `queue`. Ein Element der PriorityQueue ist ein `tuple` und enthält die bisherige Weglänge (Priorität, mit der die `PriorityQueue` die Elemente ordnet, nach dem Dijkstra-Shortest-Path-Algorithmus), den vorherigen Knoten des *edge-based-graph*s sowie der gesamte Weg bis zum Knoten und die Anzahl an Abbiegevorgängen. Wird ein neuer Knoten von dem vorherigen aus erreicht, werden die Gesamtweglänge, der vorherige Knoten, der Gesamtweg und die Anzahl an Abbiegevorgängen entsprechend angepasst und verändert der PriorityQueue hinzugefügt.

Falls auf einem Weg bereits `number_turns` abgebogen wurde und für den nächsten Knoten ein Abbiegevorgang benötigt wird, wird dieser nicht der `PriorityQueue` hinzugefügt, sodass nur Wege mit einer Maximalanzahl an Abbiegevorgängen von `number_turns` zurückgegeben werden können.

Ist die `PriorityQueue` leer, so existiert kein Weg mit `number_turns` oder weniger Abbiegevorgängen. Die Funktion `dijkstra` gibt entsprechend `None` zurück.

Wird kein Weg mit m Abbiegevorgängen gefunden oder überschreitet dessen Länge Bilals Toleranz, wird der letztgefundene Weg verwendet. Zur Ausgabe müssen die Knoten des *edge-based-graph*s wieder in Knoten des *node-based-graph*s umgewandelt werden. Dazu werden die Schlüssel der Knoten des *edge-based-graph*s aufgeteilt, sodass je zwei Schlüssel des *node-based-graph*s erhalten werden (siehe oben). Nun wird überprüft, von welcher Kreuzung Bilal auf die Straße fährt und die neue Kreuzung wird an die Liste an Kreuzungsschlüsseln `path` angehängt. Mithilfe des `dicts` `junctions` werden nun den Kreuzungsschlüsseln Koordinaten zugeordnet, die mit der Weglänge und der Anzahl an Abbiegevorgängen ausgegeben werden.

2.3 Visualisierung

Mithilfe des LaTeX-Pakets `tikz` lassen sich Graphen einfach visualisieren. Im Python-Code ist deshalb auch eine Funktion `visualize` enthalten, die eine Datei `_visualize.tex` erstellt, die, durch LaTeXkompiliert, eine grafische Repräsentation der Eingabedatei zeigt. Bei der Visualisierung wird die Startkreuzung grün markiert und die Endkreuzung rot markiert. x und y-Koordinaten können wie im ersten Quadranten in einem Koordinatensystem verwendet werden.

2.4 Laufzeit

Die Laufzeit des Programms ist in allen getesteten Beispielen kein Problem. Selbst für das längste Beispiel benötigt das Programm weniger als 0.2s.

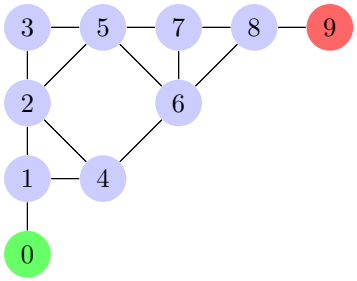
3 Beispiele

Zu jedem Beispiel drucke ich die grafische Repräsentation des Beispiels sowie die Programmausgabe und die Zeit, die zur Ausführung benötigt wird

3.1 Beispiel aus der Aufgabenstellung

Die Ergebnisse des Programms beim Beispiel aus der Aufgabenstellung entsprechen in allen Aspekten der Aufgabenstellung.

3.1.1 Visualisierung



3.1.2 0% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen0.txt 0
Distance 5.83
3 turns
(0, 0) -> (0, 1) -> (1, 1) -> (2, 2) -> (3, 3) -> (4, 3)
```

3.1.3 15% Toleranz

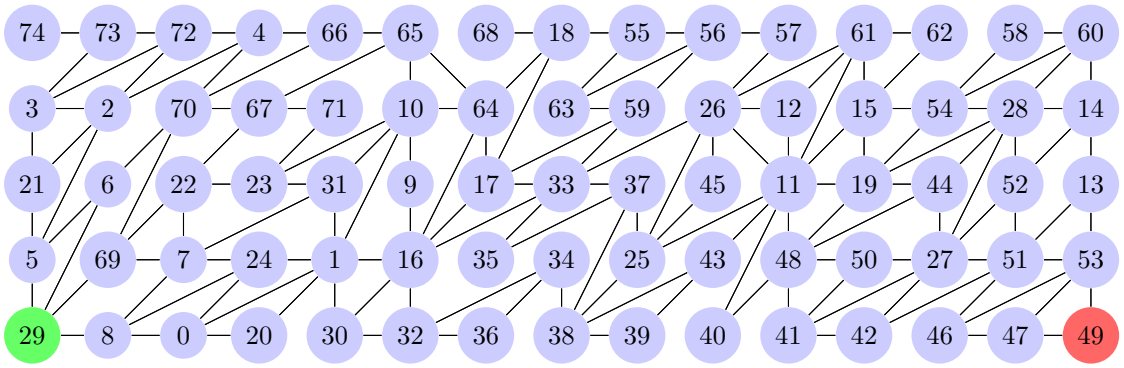
```
$ python main.py ../../material/a3-abbiegen/abbiegen0.txt 15
Distance 6.41
2 turns
(0, 0) -> (0, 1) -> (0, 2) -> (1, 3) -> (2, 3) -> (3, 3) -> (4, 3)
```

3.1.4 30% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen0.txt 30
Distance 7.0
1 turn
(0, 0) -> (0, 1) -> (0, 2) -> (0, 3) -> (1, 3) -> (2, 3) -> (3, 3) -> (4, 3)
```

3.2 Beispiel 1 von der Website

3.2.1 Visualisierung



3.2.2 0% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen1.txt 0
Distance 17.3
6 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (10, 2)
    ↪ -> (10, 1) -> (11, 1) -> (12, 1) -> (13, 1) -> (14, 1) -> (14, 0)
```

3.2.3 15% Toleranz

Mit 15% Toleranz lässt sich die Anzahl an Abbiegevorgängen von 6 auf 5 reduzieren:

```
$ python main.py ../../material/a3-abbiegen/abbiegen1.txt 15
Distance 19.12
5 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (11, 4)
    ↪ -> (11, 3) -> (12, 3) -> (13, 3) -> (14, 3) -> (14, 2) -> (14, 1) -> (14, 0)
```

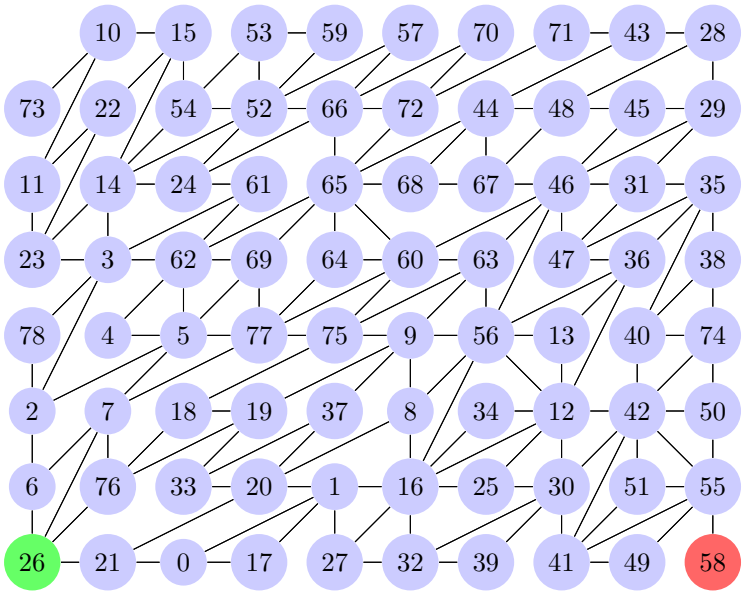
3.2.4 30% Toleranz

Auch mit 30% Toleranz findet sich kein Weg mit weniger Abbiegevorgängen:

```
$ python main.py ../../material/a3-abbiegen/abbiegen1.txt 30
Distance 19.12
5 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (11, 4)
    ↪ -> (11, 3) -> (12, 3) -> (13, 3) -> (14, 3) -> (14, 2) -> (14, 1) -> (14, 0)
```

3.3 Beispiel 2 von der Website

3.3.1 Visualisierung



3.3.2 0% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen2.txt 0
Distance 10.89
6 turns
(0, 0) -> (1, 0) -> (2, 0) -> (4, 1) -> (5, 1) -> (7, 2) -> (8, 2) -> (9, 1) -> (9, 0)
```

3.3.3 15% Toleranz

Mit 15% Toleranz kann die Anzahl an Abbiegevorgängen von 6 auf 5 verringert werden:

```
$ python main.py ../../material/a3-abbiegen/abbiegen2.txt 15
Distance 11.06
5 turns
(0, 0) -> (1, 0) -> (2, 0) -> (4, 1) -> (5, 1) -> (6, 1) -> (7, 1) -> (8, 2) -> (9, 1) ->
    ↪ (9, 0)
```

3.3.4 30% Toleranz

Mit 30% Toleranz kann die Anzahl an Abbiegevorgängen von 5 auf 4 verringert werden:

```
$ python main.py ../../material/a3-abbiegen/abbiegen2.txt 30
Distance 13.06
4 turns
(0, 0) -> (1, 0) -> (2, 0) -> (4, 1) -> (5, 1) -> (6, 1) -> (7, 1) -> (8, 2) -> (9, 3) ->
    ↪ (9, 2) -> (9, 1) -> (9, 0)
```

3.3.5 50% Toleranz

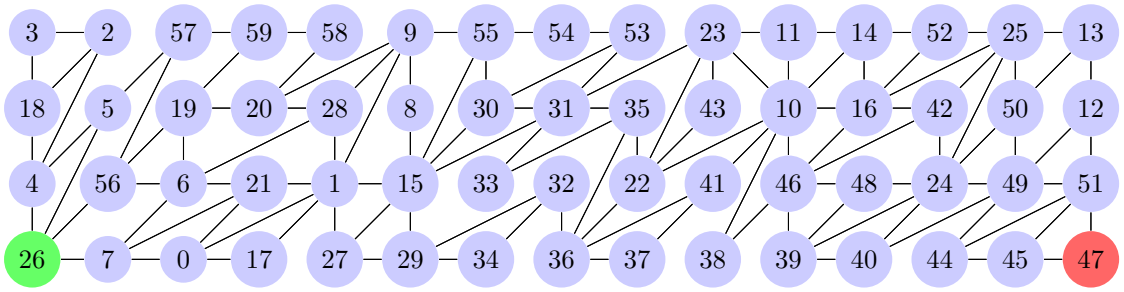
Mit 50% Toleranz kann die Anzahl an Abbiegevorgängen von 4 auf 3 verringert werden:

```
$ python main.py ../../material/a3-abbiegen/abbiegen2.txt 50
Distance 15.94
3 turns
(0, 0) -> (1, 2) -> (3, 3) -> (5, 4) -> (7, 5) -> (8, 5) -> (9, 5) -> (9, 4) -> (9, 3) ->
    ↪ (9, 2) -> (9, 1) -> (9, 0)
```

3.4 Beispiel 3 von der Website

Bei diesem Beispiel ist der kürzeste Weg bereits der mit der geringsten Anzahl an Abbiegevorgängen.

3.4.1 Visualisierung



3.4.2 0% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen3.txt 0
Distance 17.3
6 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (10, 2)
    ↪ -> (10, 1) -> (11, 1) -> (12, 1) -> (13, 1) -> (14, 1) -> (14, 0)
```

3.4.3 15% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen3.txt 15
Distance 17.3
6 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (10, 2)
    ↪ -> (10, 1) -> (11, 1) -> (12, 1) -> (13, 1) -> (14, 1) -> (14, 0)
```

3.4.4 30% Toleranz

```
$ python main.py ../../material/a3-abbiegen/abbiegen3.txt 30
Distance 17.3
6 turns
(0, 0) -> (1, 1) -> (2, 1) -> (3, 1) -> (4, 1) -> (5, 1) -> (7, 2) -> (9, 3) -> (10, 2)
    ↪ -> (10, 1) -> (11, 1) -> (12, 1) -> (13, 1) -> (14, 1) -> (14, 0)
```

3.5 Eingabedatei existiert nicht

Falls die Eingabedatei nicht existiert, wird ein Fehler zurückgegeben:

```
$ python main.py ../../material/a3-abbiegen/abbiegen.txt 0
Traceback (most recent call last):
  File "main.py", line 169, in <module>
    junctions, roads, source, target = parse_input(sys.argv[1])
  File "main.py", line 103, in parse_input
    with open(file) as f:
FileNotFoundError: [Errno 2] No such file or directory: '../../material/a3-abbiegen/
    ↪ abbiegen.txt'
```

3.6 Toleranz keine Zahl oder negativ

Ist die Toleranz keine Zahl oder negativ, wird ein Fehler zurückgegeben:

```
$ python main.py ../../material/a3-abbiegen/abbiegen.txt -1
Usage main.py <filename> <tolerance (percent)>
```

3.7 Zu wenig Parameter

Sind zu wenig Parameter angegeben, wird ein Fehler zurückgegeben:

```
$ python main.py
Usage main.py <filename> <tolerance (percent)>
```

3.8 Kein Weg zum Ziel

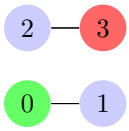
Führt kein Weg zum Ziel, wird ein Fehler zurückgegeben.

3.8.1 Eingabedatei

Die Eingabedatei `error.txt`:

```
2
(0,0)
(1,1)
(0,0) (1,0)
(0,1) (1,1)
```

3.8.2 Visualisierung



3.8.3 Ausgabe

Die Programmausgabe:

```
$ python main.py error.txt 0
Error: Cannot find any path from source to target in road network!
```

4 Quellcode (ausschnittsweise)

Die Auslassungen belaufen sich auf die Parsen der Parameter und die Visualisierungsfunktion und sind mit `...` gekennzeichnet.

```
import sys
import math
import queue
import numpy as np
import collections
import itertools
from collections import defaultdict

# Auf wie viele Ziffern wird der Quotient gerundet, wenn ermittelt wird, ob an einer
    ↪ Kreuzung abgebogen werden muss?
FLOAT_ERROR_DIGITS = 8

def straight(p1, junc, p2):
    """
    Kann man von Kreuzung Punkt p1 über Kreuzung junc auf Punkt p2 gehen, ohne abbiegen
    ↪ zu müssen?
    """
    delta1_x = junc[0] - p1[0]
    delta1_y = junc[1] - p1[1]
    delta_1 = distance(p1, junc)

    delta2_x = p2[0] - junc[0]
```

```
    delta2_y = p2[1] - junc[1]
    delta_2 = distance(p2, junc)

    return abs(round((delta1_x*delta2_x+delta1_y*delta2_y)/(delta_1*delta_2), FLOAT_ERROR
    ↪ _DIGITS))==1

def distance(p1, p2):
    """
    Berechnet die Distanz von Punkt p1 zu Punkt p2
    """
    return math.sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)

def format_roadnode(j1, j2):
    """
    Knoten eines edge-based-graphs (Straße), der im node-based-graph eine Kante zwischen
    ↪ zwei Knoten war, als str repräsentieren
    """
    if j1 < j2:
        return '{}_{}'.format(j1, j2)
    return '{}_{}'.format(j2, j1)

def get_roadnode(n):
    """
    Die Knoten des node-based-graphs ermitteln, zwischen denen der Knoten n des edge-
    ↪ based-graphs liegt
    """
    return tuple(map(int, n.split("_")))

def build_graph(junctions, roads, source_junction, target_junction):
    """
    Den node-based-graph aus den Kreuzungen junctions, den Straßen roads, dem Start
    ↪ source_junction und dem Ziel target_junction in einen edge-based-graph umwandeln
    """
    nodes = dict()
    edges = collections.defaultdict(set)

    for source in roads:
        for target in roads[source]:
            nodes[format_roadnode(source, target)] = distance(junctions[source],
    ↪ junctions[target])

    for junction in junctions:
        for comb in itertools.combinations(roads[junction], 2):
            pos0 = junctions[comb[0]]
            posj = junctions[junction]
            pos1 = junctions[comb[1]]
            s = straight(pos0, posj, pos1)
            r0 = format_roadnode(comb[0], junction)
            r1 = format_roadnode(comb[1], junction)
            edges[r0].add((r1, s))
            edges[r1].add((r0, s))


    sources = set()
    for next_junction in roads[source_junction]:
        sources.add(format_roadnode(source_junction, next_junction))

    targets = set()
    for last_junction in roads[target_junction]:
        targets.add(format_roadnode(target_junction, last_junction))


    return nodes, edges, sources, targets

def dijkstra(nodes, edges, sources, targets, number_turns=None):
    """
    Den Dijkstra-Shortest-Path-Algorithmus mit der maximalen Anzahl an Abbiegevorgängen
    ↪ number_turns auf dem gegebenen edge-based-graph ausführen (siehe Dokumentation)
    """
    pq = queue.PriorityQueue()
    for source in sources:
        pq.put((0, source, [source], 0))
```

```
88      discovered = {}
90      while True:
            if pq.empty():
92              return None, None, None
            prio, node, prev, turns = pq.get()
94          if node in discovered and turns >= discovered[node]:
                continue
96
            discovered[node] = turns
98          for edge in edges[node]:
                prio_new = prio + nodes[edge[0]]
100             if not edge[1]:
                    if turns == number_turns:
102                     continue
104             pq.put((prio_new, edge[0], prev + [edge[0]], turns + (1 if not edge[1] else
        ↪ 0)))
106         if node in targets:
                break
108
        distance = 0
110     for step in prev:
            distance += nodes[step]
112
        if number_turns is not None and turns != number_turns:
114         return None, None, None
116     return prev, distance, turns
118 def parse_tuple(t):
        """
120     Einen tuple aus ints aus einem str einlesen
        """
122     return tuple(map(int, t.replace('(', '').replace(')', '').split(",")))
124 def parse_input(file):
        """
126     Die Eingabedatei einlesen und in einen node-based-graph umwandeln
        """
128     with open(file) as f:
            lines = f.read().split("\n")
130         start_coords = parse_tuple(lines[1])
            end_coords = parse_tuple(lines[2])
132         lines_roads = lines[3:]
134         junctions = {}
            roads = defaultdict(set)
136         junction_to_id = {}
            i = 0
138         for line_road in lines_roads:
                if line_road != "":
140                 line_road_split = line_road.split("␣")
                    a = parse_tuple(line_road_split[0])
142                 b = parse_tuple(line_road_split[1])
                    if a not in junction_to_id:
144                     junction_to_id[a] = i
                        junctions[i] = a
146                     i += 1
                    if b not in junction_to_id:
148                     junction_to_id[b] = i
                        junctions[i] = b
150                     i += 1
                    id_a = junction_to_id[a]
152                 id_b = junction_to_id[b]
                    roads[id_a].add(id_b)
154                 roads[id_b].add(id_a)
            start = junction_to_id[start_coords]
156         end = junction_to_id[end_coords]
        return junctions, roads, start, end
158
    def visualize(junctions, roads, start, end):
```

```
160     ...

162 if __name__ == '__main__':
        ...
164
        junctions, roads, source, target = parse_input(sys.argv[1])
166     visualize(junctions, roads, source, target)
        nodes, edges, sources, targets = build_graph(junctions, roads, source, target)
168     min_path, min_distance, max_turns = dijkstra(nodes, edges, sources, targets)
        if min_path is None:
170         print("Error: Cannot find any path from source to target in road network!")
            exit(1)
172
        less_turns = 0
174     while True:
            path, distance, turns = dijkstra(nodes, edges, sources, targets, number_turns=max
        ↪ _turns-less_turns)
176
            if path is None or distance > min_distance * (1 + tolerance/100):
178             break

180         res_path, res_distance, res_turns = path, distance, turns
            less_turns += 1
182
        path = [source]
184     for road in res_path:
            j1, j2 = get_roadnode(road)
186         if path[-1] == j1:
                path.append(j2)
188         else:
                path.append(j1)
190
        print("Distance", round(res_distance, 2))
192
        if res_turns == 1:
194         print(res_turns, "turn")
        else:
196         print(res_turns, "turns")
        print(' -> '.join(map(lambda j: str(junctions[j]), path)))
```

tikz/code.py

Literatur

[1] Robert Geisberger und Christian Vetter. „Efficient Routing in Road Networks with Turn Costs“. In: *International Symposium on Experimental Algorithms*. 2011. URL: https://algo2.iti.kit.edu/download/urn_ch.pdf.