Aufgabe 3: Abbiegen

Teilnahme-Id: 52493

Bearbeiter/-in dieser Aufgabe:
Christoph Waffler

20. April 2020

Inhaltsverzeichnis

Lösungsidee

Anmerkung: In der Dokumentation wird eine Abfolge von Knoten sowohl als Weg als auch als Pfad angegeben. Bei dieser Aufgabe sind alle Knoten der erstellten Wege unterschiedlich und können somit auch als Pfad bezeichnet werden.
Ebenfalls kann die Gewichtung einer Kante mit Kosten bezeichnet werden.

Ziel dieser Aufgabe ist es, von dem kürzesten Weg aus einen Weg mit gegebener, maximaler prozentualen Verlängerung zu finden, bei welchem Bilal so wenig wie möglich abbiegen muss.

Die Lösungsidee kann in folgende Schritte zusammengefasst werden:

1. Erzeuge einen ungerichteten, gewichteten Graphen *G* aus dem gegebenen Straßennetz. Die Kreuzungen der Straßen sind die Knoten von *G* und die dazugehörigen Straßen sind die Kanten von *G*.
2. Finde mithilfe des A*-Algorithmus den kürzesten Weg vom gegebenen Startpunkt zum Zielpunkt.
3. Berechne aus der gegebenen prozentualen Abweichung, die maximale Länge *l* des neuen Wegs.
4. Finde den Weg *s* mit maximaler Länge *l* mithilfe einer Tiefensuche in *G*. Im Weg *s* soll so wenig wie möglich abgebogen werden müssen.
   → Gebe diesen Weg *s* aus.
5. Erweiterung: Finde den Weg *d* mit uneingeschränkter Länge, in dem am wenigsten abgebogen werden muss.

Die gefundenen Wege sollen mit ihren beinhaltenen Knoten in der Konsole ausgegeben werden. Eine Visualisierung erfolgt ebenfalls mithilfe eines GUIs.

Erstellung des Graphen

Die Straßen sind im Format des BwInf als direkte Verbindung zwischen zwei Kreuzungen dargestellt. Von den beiden Kreuzungen ist die x- und y-Koordinate gegeben.
Jede Straße *k* zwischen den beiden Kreuzungen *P, P'* wird zum Graph als Kante hinzugefügt. Die Gewichtung der Kante ist der euklidische Abstand zwischen den beiden Kreuzungen *P* und *P'*, welche die jeweiligen Knoten der Kante darstellen. Die Kanten sollen in beide Richtungen befahren werden können.
Somit ergibt sich ein gewichteter, ungerichteter Graph.

Ermittlung des kürzesten Wegs

Die Berechnung des kürzesten Wegs *w* im Graphen *G* ist für diese Aufgabe essenziell, da von der Länge des Wegs *w* mithilfe der maximalen Verlängerung die Länge *l* berechnet werden soll. Die Länge *l* gibt die maximale Länge des Wegs *s* an, in welchem so wenig wie möglich abgebogen werden soll.

Der kürzeste Weg *w* wird mithilfe des A*-Algorithmus berechnet, da dieser im Gegensatz zum Dijkstra-Algorithmus durch Verwendung einer Heuristik eine zielgerichtete Suche ermöglicht und somit seine Laufzeit verringert.[1]

Der Dijkstra-Algorithmus besitzt eine Komplexität von $O(n^2)$ bei einem Graph mit *n* Knoten[2], welche vom A*-Algorithmus nur im worst-case erreicht werden könnte[1].

Beim A*-Algorithmus werden immer diejenigen Knoten zuerst untersucht, die wahrscheinlich schnellstmöglich zum Ziel führen. Jedem bekannten Knoten wird dabei ein bestimmter *f*-Wert zugeordnet, mit welchem abgeschätzt werden kann, wie lang der Pfad vom Start- zum Zielknoten über diesen betrachteten Knoten wäre.

---

[1] *Quelle: A*-Algorithmus - https://de.wikipedia.org*
[2] *Quelle: Dijkstra-Algorithmus - https://de.wikipedia.org*

Der *f*-Wert setzt sich aus der Summe des *g*- und des *h*-Werts zusammen.

Der *g*-Wert gibt an, wie hoch die bisherigen Kosten vom Start- zum aktuellen Knoten sind.
Der *h*-Wert gibt mithilfe einer heuristischen Funktion die geschätzten Kosten vom aktuell betrachteten Knoten zum Zielknoten an. Als heuristische Funktion wird in dieser Aufgabe die euklidische Distanz verwendet, da diese die Luftlinie darstellt und die darin vorkommenden Knoten mit einem geringeren Wert näher am Zielkoten sind.

Somit gilt für den aktuell betrachteten Knoten *k*: $f(k) = g(k) + h(k)$

Der A*-Algorithmus wählt somit immer den Knoten mit dem besten *f*-Wert aus, wodurch diese Suche auch als „best first search“ bezeichnet werden kann.[3]

Berechnung der maximalen Länge des neuen Wegs *s*

Aus der Länge *x* des kürzesten Wegs *w* wird mithilfe der eingegebenen prozentualen Verlängerung *p* die maximale Länge *l* des neuen Wegs *s* berechnet.
Die Länge *l* ergibt sich aus der Summe von *x* und *x* multipliziert mit *p/100*: $l = x + x * (\frac{p}{100})$

Im Weg *s* mit der Länge *l* soll nun ein Weg mit der geringen Anzahl an Abbiegevorgängen wie möglich gefunden werden.

Finden des Wegs *s* mit geringster Anzahl an Abbiegevorgängen

Ziel der Tiefensuche ist es, den optimalsten Pfad f_{opt} im Graph *G* zu finden, der möglichst wenige Abbiegevorgänge beinhaltet und dabei die Länge *l* nicht überschreitet. Start- und Zielknoten des Wegs sollen die gegebenen Start- und Zielpunkte der Straßenkarte sein.
Folgende Schritte werden bei dieser Tiefensuche ausgeführt:

1. Füge den aktuellen Knoten *k* zum aktuellen Pfad *f*.
2. Prüfe, ob die Länge von *f* geringer als die maximale Länge *l* ist.
   Fahre nur mit der Tiefensuche fort, wenn dies wahr ist.
3. Prüfe, ob die Anzahl der Abbiegevorgänge des aktuellen Pfads *f* kleiner (oder gleich) der Anzahl der Abbiegevorgänge des bisher optimalsten Pfads f_{opt} ist.
   Fahre nur mit der Tiefensuche fort, wenn dies wahr ist.
4. Ist der aktuelle Knoten *k* gleich dem Zielknoten, so wurde ein neuer optimaler Pfad mit weniger Abbiegevorgängen gefunden. Setzte f_{opt} auf *f*.
5. Heuristik: Sortiere die Nachfolgeknoten *N* anhand ihrer Entfernung zum Zielknoten aufsteigend an, sodass zuerst diejenigen Nachfolgeknoten besucht werden, die näher am Zielknoten liegen.
   Rufe für die geordneten Nachfolgeknoten *N* die Tiefensuche rekursiv auf und übergebe dabei den aktuell optimalen Pfad f_{opt} weiter.

Ist die Tiefensuche beendet, so ist f_{opt} der bestmöglichste Pfad im Graph *G*, d.h., dass f_{opt} derjenige Pfad ist, der die geringste Anzahl an Abbiegevorgängen hat und dabei die maximale Länge *l* nicht überschreitet.

Dieser Pfad f_{opt} soll ausgegeben werden.

Anmerkung zur Heuristik:
Die Heuristik ist, wie bereits erwähnt, so gewählt, dass jene Nachbarknoten zuerst bearbeitet werden, die nahe am Zielknoten liegen. So wird systematisch die Länge des Pfades verlängert und dabei der optimalste Pfad gefunden.

[3] *Quelle: Difference and advantages between Dijkstra & A star - https://stackoverflow.com/*

Erweiterung: uneingeschränkter Pfad mit geringster Anzahl an Abbiegevorgängen

Als Erweiterung soll optional gewählt werden können, ob das Programm zusätzlich denjenigen Pfad in G ermitteln soll, der die wenigsten Abbiegevorgänge beinhaltet. Dieser Pfad ist nicht längenbegrenzt wie f_{opt}.
So könnte Bilal bei bestem Wetter entspannt diesen Weg wählen, wo er am wenigsten abbiegen muss.

Dieser Pfad wird mit $f_{minAbbiegen}$ bezeichnet.
Start- und Zielknoten von $f_{minAbbiegen}$ sollen die gegebenen Start- und Zielpunkte der Straßenkarte sein.

Die Ermittlung von $f_{minAbbiegen}$ wird ebenfalls mit einer ähnlichen Tiefensuche wie für f_{opt} durchgeführt. Die einzige Anpassung ist, dass die Tiefensuche nicht durch eine maximale Länge eingegrenzt wird.
Ebenfalls wird die Tiefensuche durch eine Heuristik optimiert.
Die Heuristik wird so gewählt, dass derjenige Nachbarknoten n ausgewählt wird, bei dem die Anzahl der Abbiegevorgänge in $f_{minAbbiegen}$ zusammen n am geringsten ist. Durch die Verwendung einer Heuristik wird die Laufzeit erheblich verringert.

Dadurch ergeben sich folgende Schritte für die rekursive Tiefensuche von $f_{minAbbiegen}$:

1. Füge den aktuellen Knoten k zum aktuellen Pfad f.
2. Prüfe, ob die Anzahl der Abbiegevorgänge des aktuellen Pfads f kleiner (oder gleich) der Anzahl der Abbiegevorgänge des bisher besten Pfads $f_{minAbbiegen}$ ist.
   Fahre nur mit der Tiefensuche fort, wenn dies wahr ist.
3. Ist der aktuelle Knoten k gleich dem Zielknoten, so wurde ein neuer Pfad mit geringerer Anzahl an Abbiegevorgängen gefunden. Setze $f_{minAbbiegen}$ auf f.
4. Rufe für die Nachbarknoten N die heuristische Funktion auf, welche N aufsteigend sortiert, sodass die Tiefensuche mit dem Nachbarknoten n fortgefahren wird, der den geringsten Wert besitzt.
5. Rufe die Tiefensuche für n_{best}, $n_{best} \in N$, rekursiv auf und übergebe dabei den aktuell besten Pfad $f_{minAbbiegen}$.

Umsetzung

Die Lösungsidee wird objektorientiert in Python implementiert.
(Anmerkung: Die hier in der Umsetzung verwendete Bezeichnung für Variablen kann für eine bessere Übersichtlichkeit von der Bezeichnung der Variablen im Quellcode abweichen.)

Die zwei relevantesten Klassen sind die Klasse Graph und die Klasse Berechnungen.
In der Klasse Graph wird die Datenstruktur eines Graphs verwaltet und die Klasse Berechnungen die jeweiligen Pfade.
Die Eingabe der Straßenkarte mit dazugehöriger Verlängerung und die Auswahl, ob die Erweiterung zusätzlich berechnet werden soll, erfolgt ebenso wie die Visualisierung der Pfade und Straßenkarte in einer eigenen Klasse.

Datenstruktur Graph

Die Datenstruktur Graph wird in Form einer Adjazenzliste implementiert. Der Graph soll dabei ungerichtet und kantengewichtet sein.
Die Adjazenzliste speichert in einem Dictionary alle Startknoten als Key und die Zielknoten mit deren dazugehörigen Gewichtungen in einer Liste als Item des Keys. Die Zielknoten und die Gewichtung der jeweiligen Kante wird dabei ebenfalls als in einem eigenen Dictionary gespeichert, indem der Zielknoten als Key fungiert und der Betrag der Gewichtung als dazugehöriges Item.
Die Adjazenzliste könnte zum Beispiel dann so aussehen:

```
{
        (1, 1):   [
                        { (1, 2): 1 }
                  ],
        (1, 2):   [
                        { (1, 1): 1}
                  ]
}
```

Hier stehen {…} für ein Dictionary und […] für eine Liste, (x, y) für die Koordinaten eines Knotens.
Das oben gezeigte Beispiel zeigt, dass von (1, 1) eine Kante zu (1, 2) und von (1, 2) eine Kante zu (1, 1) verläuft, welche jeweils mit 1 gewichtet sind.

Im Folgenden werden die Methoden der Klasse Graph kurz dargestellt.

Mithilfe der Methode *add_Kante(knoten1, knoten2, gewichtung)* wird zur Adjazenzliste eine neue Kante einmal mit *knoten1* als Startknoten und *knoten2* als Zielknoten und einmal mit *knoten2* als Startknoten und *knoten1* als Zielknoten hinzugefügt. Die Gewichtung der beiden Kanten ist der gegebene Wert *gewichtung*.

Durch die Methode *__getitem__(key)* kann mithilfe des „Eckigen-Klammern-Operators“ [] die Adjazenzliste indiziert werden.

Durch die Verwendung des ‚@property‘-Operators vor der Methode *knoten()* kann die Methode als Eigenschaft des Graphen verwendet und aufgerufen werden.
In dieser Methode wird ein Set aus den gesamten Keys der Adjazenzliste zurückgegeben. Dieses Set enthält somit alle Knoten des Graphen.

Klasse Berechnungen

Die Klasse Berechnungen ist jene Klasse, in der alles Grundlegende bezüglich der Aufgabenstellung ermittelt wird.
Einige Methoden und Verfahren werden im Folgenden erklärt, da die Main-Methode sich auf diese aufbaut.

Initialisierung

Zu Beginn wird die Klasse Berechnungen vom Eingabefenster aufgerufen und erhält eine Liste mit Strings, welche die Eingabe enthalten (z.B. ['14', '(0,0)', '(4,3)', '(0,0)', '(0,1)', '(0,1)', '(0,2)', '(0,2)', '(0,3)', …]).
Das erste Element dieser Liste ist die Anzahl der Straßen. Das zweite und dritte sind der Start- bzw. Zielpunkt von Bilals Weg. Danach folgen die Straßen, die durch zwei Koordinatenpaare (x, y) dargestellt werden.
Alle Daten werden so eingelesen und zum Datentyp *int* konvertiert.
Die Straßen werden alle zusammen in *self.liste_verbindungen* gespeichert.

Die Klasse Berechnungen erhält ebenfalls vom Benutzer, wie lange die maximale Verlängerung in Prozent sein darf, und ob die Erweiterung (unbegrenzter Weg mit min. Abbiegen) mitberechnet werden soll.

In der Initialisierungsmethode *__init__(...)* wird am Schluss die main-Methode aufgerufen.

Methoden zur Berechnung der Längen

Mithilfe der Methode *berechneLänge(knoten1, knoten2)* wird die euklidische Distanz zwischen dem ersten und dem zweiten Knoten berechnet. Dabei wird der Satz des Pythagoras verwendet.
Für die Punkte *P(x1, y1)* und *P‘(x2, y2)* würde diese Distanz wie folgt berechnet werden:

$$distanz = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

In der Methode *berechneLängePfad(pfad)* wird die Längenbestimmung auf einen Pfad erweitert, dessen Länge durch Aufrufen der Methode *berechneLänge(…)* berechnet wird.
So wird für jeden Punkt *P* von *pfad* (außer dem ersten) der Abstand *s* zu seinem Vorgänger *P'* über die Methode *berechneLänge(P, P')*. Alle Abstände *s* werden addiert, sodass die Summe die Länge von *pfad* beschreibt.
Diese Summe wird zurückgegeben.

Methoden zur Bestimmung der Anzahl der Abbiegevorgänge

Die Methode *berechneSteigungKante(kante)* ermittelt zuerst Δx und Δy aus den beiden Knoten von *kante*.
Die Steigung ergibt sich dann durch den Quotienten aus $\Delta y \,/\Delta x$. Falls $\Delta x = 0$ ist, so wird der Steigung der Wert unendlich zugewiesen.
Die Steigung ist der Rückgabewert der Methode.

Die Methode *berechneAnzahlAbbiegenPfad(pfad)* berechnet im gegebenen Pfad *pfad* die Anzahl der notwendigen Abbiegevorgänge.

Jeder Punkt *P* (außer dem ersten) von *pfad* stellt zusammen mit seinem Vorgänger *P'* eine Kante *m* dar, von der die Steigung mithilfe der Methode *berechneSteigungKante(m)* berechnet wird. Alle Steigungen werden in der Liste *q* gespeichert.

Nun wird über die Liste *q* iteriert und jedes Mal, wenn sich die Steigung ändert, wird der Zähler *a* um eins erhöht.
Der Zähler *a* gibt am Schluss die Anzahl der Abbiegevorgänge an und ist der Rückgabewert dieser Methode.

A*-Algorithmus für den kürzesten Weg

Die Methode *astar(start, ziel, graph)* soll im gegebenen Graph *graph* vom Startknoten *start* zum Zielknoten *ziel* den kürzesten Weg berechnen.
Die Schätzfunktion *heuristik_astar(…)* berechnet den *h*-Wert, indem die Methode *berechneLänge(…)* für den aktuellen Knoten und dem Zielknoten aufgerufen wird.

Wie bereits in der Lösungsidee erwähnt, setzt sich der *f*-Wert aus $f = g + h$ zusammen.

Zu Beginn der Methode *astar* werden zwei leere Dictionary f_{Dict} und g_{Dict} erstellt, in denen die *f*- und *g*-Werte für die besuchten Knoten gespeichert werden. Der *g*-Wert für den Startknoten *start* ist zu Beginn 0 und der *f*-Wert von *start* besteht daher nur noch aus dem heuristischen Wert, der mithilfe der Methode *heuristik_astar(…)* berechnet wird.

Noch offene und bereits geschlossene Knoten werden in jeweils einem Set *offene_knoten* und *geschlossene_knoten* gespeichert. Zu Beginn wird der Startknoten *start* zu *offene_knoten* hinzugefügt.
Der Verlauf des Wegs kann - falls der Zielknoten erreicht wird - mithilfe des Dictionary *gekommen_von* zurückverfolgt werden.

Mit einer while-Schleife werden folgende Anweisungen solange ausgeführt, bis keine Knoten mehr in *offene_knoten* sind:

- Wähle mithilfe einer for-Schleife denjenigen Knoten von *offene_knoten* aus, der den geringsten *f*-Wert besitzt. Dies wird mithilfe von f_{Dict} realisiert. Dieser Knoten ist nun der aktuelle Knoten *k*.
- Abbruchbedingung der while-Schleife: Ist der aktuelle Knoten *k* gleich dem Zielknoten *ziel*, so wird der Pfad rückwärtsgegangen, indem mit einer while-Schliefe über das Dictionary *gekommen_von* iteriert wird und die jeweiligen Vorgänger zur Liste *p* hinzugefügt werden.
  Die Liste *p* wird umgekehrt, damit der Zielknoten an letzter und der Startknoten an erster Stelle ist.

Die Liste *p* wird mit der Länge *l* des kürzesten Wegs (*p*) zurückgegeben. Die Länge *l* ist der *f*-Wert des Zielknoten *ziel* (hier identisch mit dem aktuellen Knoten *k*).

- Ist die Abbruchbedingung nicht eingetreten, so markiere den aktuellen Knoten *k* als geschlossen. Daher wird *k* von *offene_knoten* entfernt und zu *geschlossene_knoten* hinzugefügt.

- Aktualisiere anschließend die *g*-Werte für die Nachbarknoten *N*, welche die Nachfolgeknoten des aktuellen Knoten *k* sind.
  Mithilfe einer for-Schleife wird für jeden Nachbarknoten *n, n ∈ N*, zuerst überprüft, ob dieser bereits ausgeschöpft ist und daher *n* in *geschlossene_knoten* ist.
  Ist dies der Fall, so wird durch ‚*continue*' mit dem nächsten Nachbarknoten die for-Schleife fortgesetzt.

  Wenn nicht, so wird für *n* ein neuer *g*-Wert berechnet, der mit *g_wert_kandidat* bezeichnet wird.
  Ist *n* noch nicht in *offene_knoten* gespeichert, so wird er zu *offene_knoten* hinzugefügt.

  Anschließend wird überprüft, ob der neue Wert *g_wert_kandidat* schlechter (=größer) ist, als der vorher gefundene.
  Ist dies der Fall, so wird durch ‚*continue*' mit dem nächsten Element die for-Schleife fortgesetzt, da der bessere *g*-Wert behalten werden sollte.

  Wenn nicht, so wird der *g*-Wert von *n* im Dictionary g_{Dict} auf den Wert *g_wert_kandidat* gesetzt. Der Vorgänger von *n* wird im Dictionary *gekommen_von* auf den aktuellen Knoten gesetzt.

  Nun wird der Abstand zum Zielknoten mithilfe der Methode *heuristik_astar(...)* geschätzt.
  Der neue *f*-Wert von *n* wird in f_{Dict} aktualisiert, der sich aus der Summe des eben berechneten heuristischen Werts und des *g*-Werts in g_{Dict} ergibt.

Endet die while-Schleife ohne die Abbruchbedingung zu erreichen, so wurde kein Weg gefunden und es wird *None* zurückgegeben.

Methode zum Finden eines optimalen Wegs mit gegebener Länge

Die rekursive Methode *optimalsterPfad(graph, start, ziel, max_länge, pfad=[], bisher_optimalster_pfad=None, länge_bisherOptimalsterPfad=math.inf, anzahl_abbiegen_bisherOptimalsterPfad=math.inf)* berechnet im gegebenen Graphen mithilfe des Verfahrens der Tiefensuche den optimalsten Weg. Dieser Weg hat maximal die Länge *max_länge* und hat so wenig Abbiegevorgänge wie möglich.

Kurze Erklärung der Parameter:

- *graph*: Datenstruktur Graph, der die Straßenkarte speichert
- *start*: aktueller Knoten, der zu *pfad* hinzugefügt werden soll
- *ziel*: Zielknoten, der am Ende erreicht werden soll
- *max_länge*: Länge des optimalsten Wegs, die mithilfe der prozentualen Verlängerung aus der Länge des kürzesten Wegs berechnet wurde
- *pfad*: Liste zum Speichern des aktuellen Wegs (mit Knoten des Graphs)
- *bisher_optimalster_pfad*: zu Beginn ‚*None*', soll später den bisher besten Pfad beinhalten
- *länge_bisherOptimalsterPfad*: zu Beginn ‚unendlich', soll später die Länge von *bisher_optimalster_pfad* speichern
- *anzahl_abbiegen_bisherOptimalsterPfad*: zu Beginn ‚unendlich', soll später die Anzahl der Abbiegevorgänge von *bisher_optimalster_pfad* speichern.

Zu Beginn der Methode *optimalsterPfad* wird der aktuelle Knoten *start* zu *pfad* hinzugefügt.
Anschließend wird die Länge *länge_aktueller_pfad* mithilfe der Methode *berechneLängePfad(pfad)* berechnet.

Dann wird überprüft, ob die berechnete Länge *länge_aktueller_pfad* kleiner bzw. gleich der maximalen Länge *max_länge* ist. Es wird nur mit der Tiefensuche fortgefahren, wenn dies wahr ist.

Danach wird die Anzahl der Abbiegevorgänge von *pfad* berechnet. Auch hier wird die Tiefensuche mit dem aktuellen Pfad *pfad* nur fortgesetzt, wenn diese Anzahl kleiner (oder gleich) dem Wert *anzahl_abbiegen_bisherOptimalsterPfad* ist.

Jetzt wird überprüft, ob der aktuelle Knoten *start* gleich dem Zielknoten *ziel* ist.

- Ist die Anzahl der Abbiegevorgänge kleiner als *anzahl_abbiegen_bisherOptimalsterPfad* so wird der *bisher_optimalster_pfad* wie folgt aktualisiert, da der bisher beste gefundene Pfad nun der aktuelle Pfad *pfad* ist.
  Der *bisher_optimalster_pfad* wird auf *pfad* gesetzt, der Variablen *länge_bisherOptimalsterPfad* wird der Wert *länge_aktueller_pfad* zugewiesen und die Variable *anzahl_abbiegen_bisherOptimalsterPfad* speichert nun die Anzahl der Abbiegevorgänge von *pfad*.
- Ist die Anzahl der Abbiegevorgänge gleich dem Wert *anzahl_abbiegen_bisherOptimalsterPfad* so muss die Länge *länge_aktueller_pfad* kleiner als *länge_bisherOptimalsterPfad* sein, damit *bisher_optimalster_pfad* wie oben genannt aktualisiert wird.

Die Nachbarknoten *N* des aktuellen Knoten *start* werden mithilfe der Methode *berechneLänge(...)* nach ihrem Abstand zum Zielknoten aufsteigend sortiert.
Anschließend werden mithilfe einer for-Schleife zuerst für nahe am Zielknoten liegende Nachbarknoten n, $n \in N$, die Methode *optimalsterPfad* rekursiv aufgerufen, falls *n* noch nicht in *pfad* ist. Somit wird *n* für den Parameter *start* übergeben.

Rückgabewerte der Methode *optimalsterPfad* sind der gefundene optimalste Pfad *bisher_optimalster_pfad*, dessen Länge *länge_bisherOptimalsterPfad* und dessen Anzahl an Abbiegevorgängen *anzahl_abbiegen_bisherOptimalsterPfad*.

Erweiterung: Methode zum Finden des Pfades mit minimaler Anzahl an Abbiegevorgängen ohne Längenbegrenzung

Die Methode *minAbbiegenPfad(graph, start, ziel, pfad=[], bisher_minAbbiegen_pfad=math.inf, anzahl_abbiegen_bisherMinAbbiegen_pfad=math.inf)* berechnet ebenfalls mithilfe der Tiefensuche den Pfad mit der minimalen Anzahl an Abbiegevorgängen. Diesmal ist die Länge nicht begrenzt.

Kurze Erklärung der Parameter:

- *graph*: Datenstruktur Graph, der die Straßenkarte speichert
- *start*: Startknoten, der zu *pfad* hinzugefügt wird
- *ziel*: Zielknoten, der am Ende erreicht werden soll
- *pfad*: Liste zum Speichern des aktuellen Wegs (mit Knoten des Graphs)
  Wird kein Pfad beim Aufruf mitgegeben, so wird eine leere Liste erstellt.
- *bisher_minAbbiegen_pfad*: zu Beginn ‚*None*‘, speichert den Pfad, in welchem bisher am wenigsten häufig abgebogen werden muss
- *anzahl_abbiegen_bisherMinAbbiegen_pfad*: speichert die Anzahl der Abbiegevorgänge des Pfads *bisher_minAbbiegen_pfad*

Zu Beginn der Methode *minAbbiegenPfad* wird der aktuelle Knoten *start* zu *pfad* hinzugefügt.
Anschließend wird die Anzahl der Abbiegevorgänge im aktuellen Pfad berechnet, welche kleiner als *anzahl_abbiegen_bisherMinAbbiegen_pfad* sein muss, um mit dem aktuellen Pfad fortfahren zu können.

Nun wird überprüft, ob der Zielknoten erreicht wurde und somit der aktuelle Knoten *start* und der Zielknoten *ziel* gleich sind.
Ist dies der Fall, so ist der aktuelle Pfad *pfad* der neue ‚beste' Pfad mit geringster Anzahl an Abbiegevorgängen. Somit wird der *bisher_minAbbiegen_pfad* auf *pfad* gesetzt und die Variable *anzahl_abbiegen_bisherMinAbbiegen_pfad* speichert die Anzahl der Abbiegevorgänge von *pfad*.

Anschließend werden die Nachbarknoten *N* bezüglich ihrem heuristischem Wert, der mithilfe der Methode *heuristik_dfs(...)* berechnet wird, aufsteigend sortiert.
Die Methode *heuristik_dfs(knoten, aktueller_pfad)* ermittelt, wie oft im gegebenen Pfad *aktueller_pfad* zusammen mit *knoten* abgebogen werden müsste. Diese Anzahl ist der heuristische Wert.

So wird zuerst derjenige Nachbarknoten *n, n* ϵ *N*, besucht, der den geringsten heuristischen Wert hat und somit die Anzahl der Abbiegevorgänge in *pfad* möglichst gering bleibt.

Die Methode *minAbbiegenPfad* wird rekursiv aufgerufen, wobei nun der Knoten *n* als aktueller Knoten *start* übergeben wird, falls *n* noch nicht in *pfad* ist.

Rückgabewerte der Methode *minAbbiegenPfad* sind der gefundene beste Pfad *bisher_minAbbiegen_pfad* und dessen Anzahl an Abbiegevorgänge *anzahl_abbiegen_bisherMinAbbiegen_pfad*.

Main-Methode

Zu Beginn der Methode *main()* wird ein neues Objekt *G* der Klasse Graph initiiert und jede Kante in der Liste *self.liste_verbindungen* zu *G* hinzugefügt. Die Länge der jeweiligen Kante und somit deren Gewichtung wird durch die Methode *berechneLänge(...)* ermittelt.

Darauffolgend wird mithilfe des A*-Algorithmus der kürzeste Weg in *G* vom gegebenen Startpunkt zum gegebenen Zielpunkt berechnet. Dabei wird die Methode *astar(...)* aufgerufen.

Die aufgerufene Methode *astar* gibt neben dem kürzesten Pfad auch dessen Länge *l* zurück, womit nun die maximale Länge berechnet werden kann.
Die maximale Länge wird berechnet, indem Summe von $l + verlängerung/100$ gebildet wird. Die Verlängerung hat der Benutzer in Prozent eingegeben und wird daher noch in Dezimalschreibweise umgewandelt.

Nun kann die Methode *optimalsterPfad(...)* mit der eben berechneten maximalen Länge aufgerufen werden. Diese Methode gibt den für Bilal optimalen Pfad zurück.
Ebenfalls wird noch der zufahrende Umweg im Vergleich zum kürzesten Weg berechnet.

Obwohl bei einer maximalen Verlängerung von 0 % der berechnete kürzeste Weg in Frage kommen würde, wird die Methode *optimalsterPfad(...)* trotzdem aufgerufen, da es sein könnte, dass zwar der kürzeste Weg gefunden wurde, aber es nicht sicher ist, ob es nicht noch weitere kürzeste Pfade mit einer geringeren Anzahl an Abbiegevorgängen gibt.

Falls die Erweiterung aktiviert wurde, dass neben diesem optimalen längenbegrenzten Pfad auch der unbegrenzte Pfad mit der geringsten Anzahl an Abbiegevorgängen berechnet werden soll, wird die Methode *minAbbiegenPfad(...)* ebenfalls noch aufgerufen.
Auch hier wird der zufahrende Umweg im Vergleich zum optimalen Pfad (längenbegrenzt) und zum kürzesten Pfad berechnet.

Eine Visualisierung erfolgt durch das Erzeugen eines Objekts der Klasse *Koordinatensystem*, welche mithilfe der Bibliothek *matplotlib* die Straßenkarte anschaulich darstellt.
Der kürzeste Weg wird dabei blau, der optimale Weg gelb und der Weg der Erweiterung pink dargestellt.
(Eine genaue Beschreibung der Funktionsweise der Visualisierung der Straßenkarte ist dem Quellcode des Programms zu entnehmen).

Eine Konsolenausgabe mit den einzelnen Punkten des kürzesten Wegs, des optimalen Wegs und eventuell des Wegs der aktivierten Erweiterung erfolgt ebenfalls. In allen drei Fällen werden die Anzahl der Abbiegevorgänge ebenfalls dargestellt sowie der zu fahrende Umweg für den optimalen Weg und den Weg der Erweiterung auch ausgegeben.

Eine Messung der Laufzeit erfolgt durch Stoppen der Zeit mithilfe des Moduls *time* am Anfang und am Ende der Methode *main()*. Die Laufzeit wird am Ende auch ausgegeben.

Anmerkung bei der Visualisierung:
Falls sich bspw. der kürzeste Weg mit dem optimalsten Weg überlappt, so wird stets der optimale Weg (gelb) zu sehen sein, da dieser als letzter gezeichnet wird.

Beispiele

Aus Platzgründen und für eine bessere Übersichtlichkeit ist der kürzeste Weg nur bei 10 % Verlängerung zu sehen, da bei den anderen Verlängerungen der kürzeste Weg derselbe bleibt und daher hier für die Dokumentation entfernt wurde.

Die Screenshots des GUI der jeweiligen Verlängerungen ist nach den Textausgaben des Beispiels zu sehen.

Beispiel ‚abbiegen0.txt' der BwInf-Website

bei 10 % Verlängerung

> Kürzester Weg: [(0, 0), (0, 1), (1, 1), (2, 2), (3, 3), (4, 3)]
>> Länge kürzester Weg: 5.82842712474619
>> Anzahl Abbiegen kürzester Weg: 3

> Optimaler Weg: [(0, 0), (0, 1), (1, 1), (2, 2), (3, 3), (4, 3)]
>> Länge optimaler Weg: 5.82842712474619
>> Anzahl Abbiegen optimaler Weg: 3
>>> Umweg im Vergleich zum kürzesten Weg: 0.0;
Maximal mögliche Länge war 6.411269837220809

>> Laufzeit des Programms: 0.6354389190673828 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 15 % Verlängerung

(kürzester Weg ... siehe bei 10 %)

> Optimaler Weg: [(0, 0), (0, 1), (0, 2), (1, 3), (2, 3), (3, 3), (4, 3)]
>> Länge optimaler Weg: 6.414213562373095
>> Anzahl Abbiegen optimaler Weg: 2
>>> Umweg im Vergleich zum kürzesten Weg: 0.5857864376269051; Maximal mögliche Länge war 6.702691193458119

>> Laufzeit des Programms: 0.2276463508605957 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 30 % Verlängerung

(kürzester Weg ... bei 10 %)

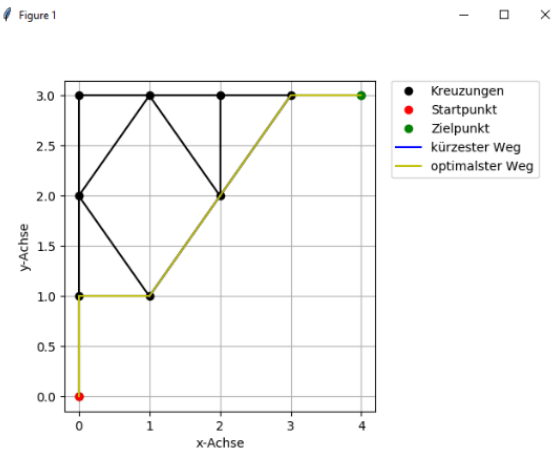
> Optimaler Weg: [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3)]
>> Länge optimaler Weg: 7.0
>> Anzahl Abbiegen optimaler Weg: 1
>>> Umweg im Vergleich zum kürzesten Weg: 1.1715728752538102; Maximal mögliche Länge war 7.5769552621700464

>> Laufzeit des Programms: 0.2509956359863281 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

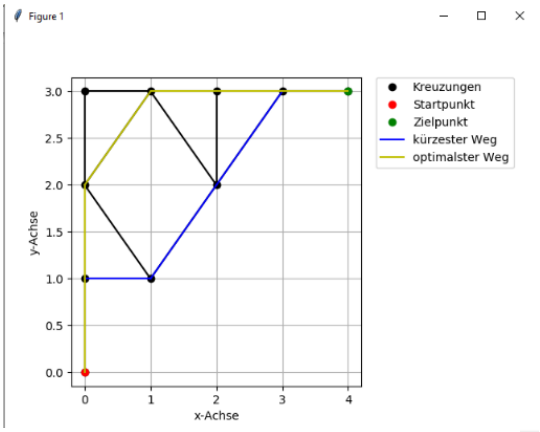
mit Erweiterung:
(kürzester Weg ... siehe bei 10 %)
(optimalster Weg ... siehe bei 30 %)

Erweiterung: Berechnung des Wegs mit geringster Anzahl an Abbiegevorgängen:
> 'min. Abbiegen Weg': [(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3)]
>> Länge von 'min. Abbiegen Weg': 7.0
>> Anzahl Abbiegen: 1
>>> Umweg im Vergleich zum optimalsten Weg: 0.0
>>> Umweg im Vergleich zum kürzesten Weg: 1.1715728752538102

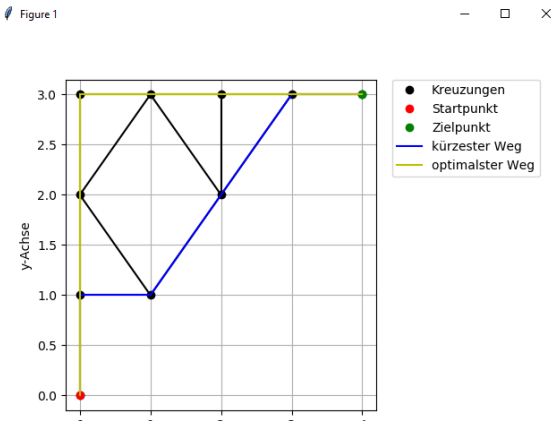
>> Laufzeit des Programms: 0.22898101806640625 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)



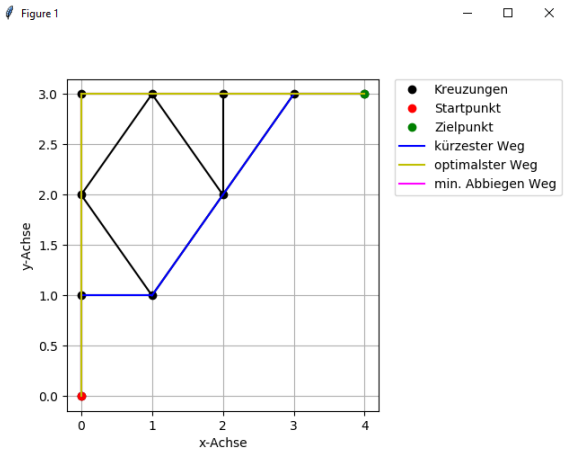
GUI bei 10% Verlängerung des Beispiels ‚abbiegen0.txt'



GUI bei 15% Verlängerung des Beispiels ‚abbiegen0.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen0.txt'



GUI bei 30 % Verlängerung des Beispiels ‚abbiegen0.txt' mit Erweiterung
(nicht sichtbar, da dies derselbe Weg wie der optimalste Weg ist und somit überdeckt wird)

Beispiel ‚abbiegen1.txt' der BwInf-Website
bei 10% Verlängerung

> Kürzester Weg: [(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (7, 2), (9, 3), (10, 2), (11, 2), (12, 2), (12, 1), (13, 1), (14, 1), (14, 0)]
>> Länge kürzester Weg: 17.122417494872465
>> Anzahl Abbiegen kürzester Weg: 8

> Optimaler Weg: [(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)]
>> Länge optimaler Weg: 17.30056307974577
>> Anzahl Abbiegen optimaler Weg: 6
>>> Umweg im Vergleich zum kürzesten Weg: 0.178145584873306; Maximal mögliche Länge war 18.834659244359713

>> Laufzeit des Programms: 2.8615520000457764 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 15 % Verlängerung
(kürzester Weg ... siehe bei 10%)

>> Länge kürzester Weg: 17.122417494872465
>> Anzahl Abbiegen kürzester Weg: 8

> Optimaler Weg: [(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)]
>> Länge optimaler Weg: 19.122417494872465
>> Anzahl Abbiegen optimaler Weg: 5
>>> Umweg im Vergleich zum kürzesten Weg: 2.0; Maximal mögliche Länge war 19.690780119103334

>> Laufzeit des Programms: 1.201186180114746 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 30 % Verlängerung:
(kürzester Weg ... siehe bei 10%)

> Optimaler Weg: [(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)]
>> Länge optimaler Weg: 19.122417494872465
>> Anzahl Abbiegen optimaler Weg: 5
>>> Umweg im Vergleich zum kürzesten Weg: 2.0; Maximal mögliche Länge war 22.259142743334206

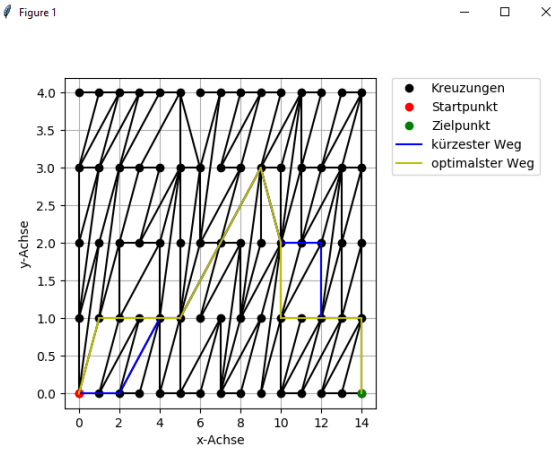
>> Laufzeit des Programms: 1.2441883087158203 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit Erweiterung
(kürzester Weg ... siehe bei 10%)
(optimaler Weg ... siehe bei 30%)

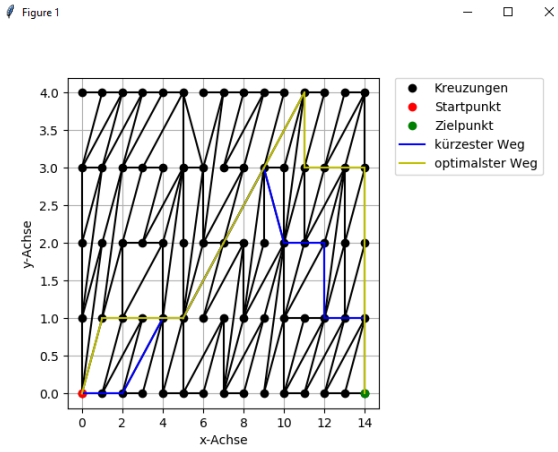
Erweiterung: Berechnung des Wegs mit geringster Anzahl an Abbiegevorgängen:

> 'min. Abbiegen Weg': [(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)]
>> Länge von 'min. Abbiegen Weg': 19.122417494872465
>> Anzahl Abbiegen: 5
>>> Umweg im Vergleich zum optimalsten Weg: 0.0
>>> Umweg im Vergleich zum kürzesten Weg: 2.0

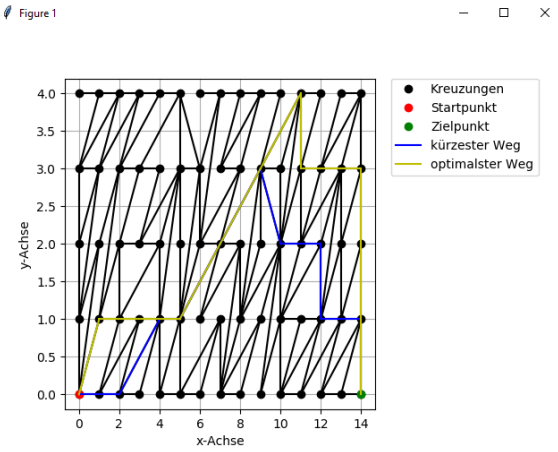
>> Laufzeit des Programms: 2.2000036239624023 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)



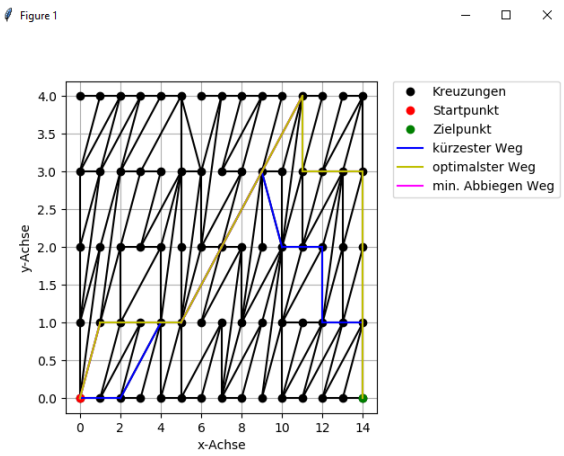
GUI bei 10% Verlängerung des Beispiels ‚abbiegen1.txt'



GUI bei 15% Verlängerung des Beispiels ‚abbiegen1.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen1.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen1.txt' mit dem Weg der Erweiterung (nicht sichtbar, da der optimale Weg und dieser hier identisch sind)

Beispiel ‚abbiegen2.txt' der BwInf-Website

mit 10 % Verlängerung

> Kürzester Weg: [(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (7, 2), (8, 2), (9, 1), (9, 0)]
>> Länge kürzester Weg: 10.886349517372675
>> Anzahl Abbiegen kürzester Weg: 6

> Optimaler Weg: [(0, 0), (1, 0), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 1), (9, 0)]
>> Länge optimaler Weg: 11.064495102245981
>> Anzahl Abbiegen optimaler Weg: 5
>>> Umweg im Vergleich zum kürzesten Weg: 0.178145584873306; Maximal mögliche Länge war 11.974984469109943

>> Laufzeit des Programms: 0.9033563137054443 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit 15 % Verlängerung

(kürzester Weg ... siehe bei 10 % Verlängerung)

> Optimaler Weg: [(0, 0), (1, 0), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 1), (9, 0)]
>> Länge optimaler Weg: 11.064495102245981
>> Anzahl Abbiegen optimaler Weg: 5
>>> Umweg im Vergleich zum kürzesten Weg: 0.178145584873306; Maximal mögliche Länge war 12.519301944978576

>> Laufzeit des Programms: 1.0044758319854736 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit 30 % Verlängerung

(kürzester Weg ... siehe bei 10 % Verlängerung)

> Optimaler Weg: [(0, 0), (1, 0), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 3), (9, 2), (9, 1), (9, 0)]
>> Länge optimaler Weg: 13.064495102245981
>> Anzahl Abbiegen optimaler Weg: 4
>>> Umweg im Vergleich zum kürzesten Weg: 2.178145584873306; Maximal mögliche Länge war 14.152254372584478

>> Laufzeit des Programms: 0.735907793045044 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

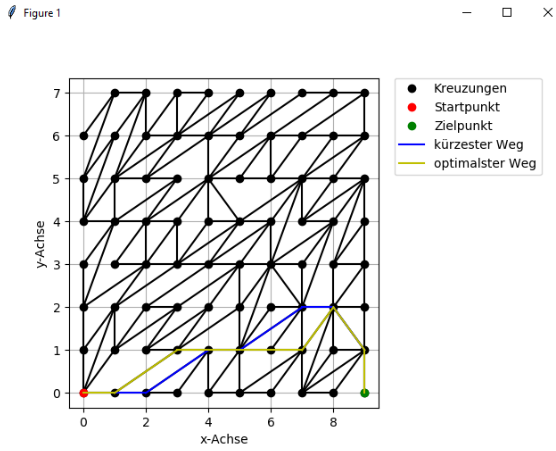
mit Erweiterung

(kürzester Weg ... siehe bei 10 %)
(optimaler Weg ... siehe bei 30 %)

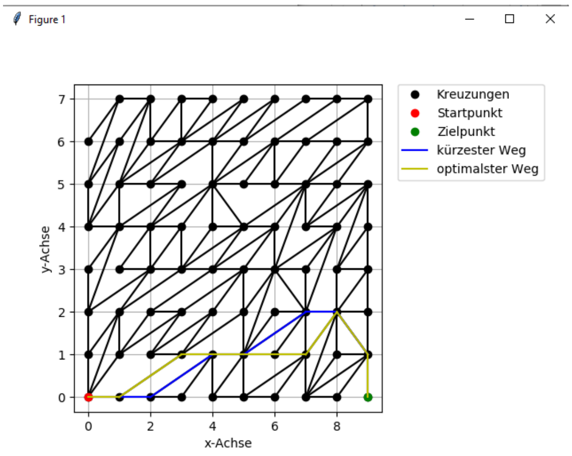
Erweiterung: Berechnung des Wegs mit geringster Anzahl an Abbiegevorgängen:
> 'min. Abbiegen Weg': [(0, 0), (0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 5), (6, 5), (7, 5), (8, 5), (9, 5), (9, 4), (9, 3), (9, 2), (9, 1), (9, 0)]
>> Länge von 'min. Abbiegen Weg': 16.65685424949238
>> Anzahl Abbiegen: 3
>>> Umweg im Vergleich zum optimalsten Weg: 3.5923591472463983
>>> Umweg im Vergleich zum kürzesten Weg: 5.770504732119704

>> Laufzeit des Programms: 0.8239922523498535 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

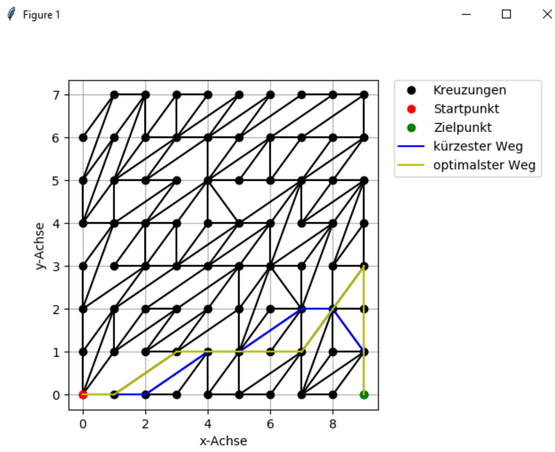
Screenshots der jeweiligen GUIs



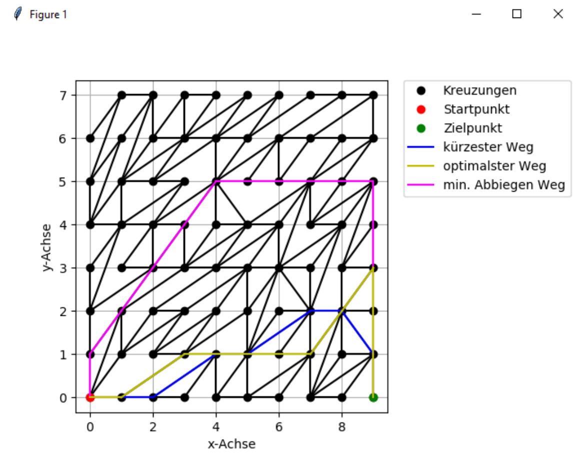
GUI bei 30% Verlängerung des Beispiels ‚abbiegen2.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen2.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen2.txt'



GUI bei 30% Verlängerung des Beispiels ‚abbiegen2.txt' mit dem Weg der Erweiterung (pink eingezeichnet)

Beispiel ‚abbiegen3.txt'

mit 10 % Verlängerung

> Kürzester Weg: [(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (7, 2), (9, 3), (10, 2), (11, 2), (12, 2), (12, 1), (13, 1), (14, 1), (14, 0)]
>> Länge kürzester Weg: 17.122417494872465
>> Anzahl Abbiegen kürzester Weg: 8

> Optimaler Weg: [(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)]
>> Länge optimaler Weg: 17.886349517372675
>> Anzahl Abbiegen optimaler Weg: 4
>>> Umweg im Vergleich zum kürzesten Weg: 0.7639320225002102; Maximal mögliche Länge war 18.834659244359713

>> Laufzeit des Programms: 0.49481892585754395 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit 15 % Verlängerung

(kürzester Weg ... siehe bei 10 %)

> Optimaler Weg: [(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)]
>> Länge optimaler Weg: 17.886349517372675
>> Anzahl Abbiegen optimaler Weg: 4
>>> Umweg im Vergleich zum kürzesten Weg: 0.7639320225002102; Maximal mögliche Länge war 19.690780119103334

>> Laufzeit des Programms: 0.5031797885894775 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit 30 % Verlängerung

(kürzester Weg ... siehe bei 10 %)

> Optimaler Weg: [(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)]
>> Länge optimaler Weg: 17.886349517372675
>> Anzahl Abbiegen optimaler Weg: 4
>>> Umweg im Vergleich zum kürzesten Weg: 0.7639320225002102; Maximal mögliche Länge war 22.259142743334206

>> Laufzeit des Programms: 0.504692792892456 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

mit Erweiterung

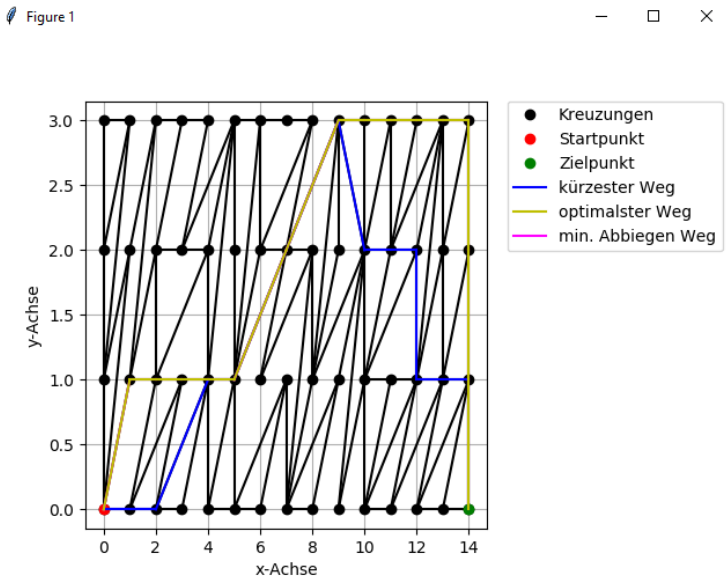
(kürzester Weg ... siehe bei 10 %)
(optimaler Weg ... siehe bei 30 %)

Erweiterung: Berechnung des Wegs mit geringster Anzahl an Abbiegevorgängen:
> 'min. Abbiegen Weg': [(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)]

>> Länge von 'min. Abbiegen Weg': 17.886349517372675
>> Anzahl Abbiegen: 4
>>> Umweg im Vergleich zum optimalsten Weg: 0.0
>>> Umweg im Vergleich zum kürzesten Weg: 0.7639320225002102

>> Laufzeit des Programms: 0.7000889778137207 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

Hier bei Beispiel ‚abbiegen3.txt' wurde der Weg mit geringster Anzahl an Abbiegevorgängen, der überhaupt möglich ist, bereits bei 10 % Verlängerung gefunden.
Die gezeichneten Straßenkarten der unterschiedlichen Verlängerungen unterscheiden sich nicht, daher wird ein Screenshot der 30% Verlängerung mit Erweiterung (nicht sichtbar) eingefügt.

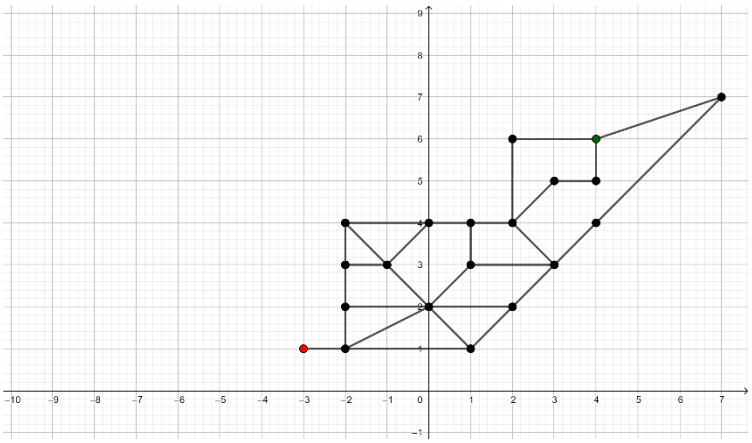


GUI des Beispiels ‚abbiegen3.txt'
hier mit 30 % und mit Erweiterung

Eigenes Beispiel (mit negativen Zahlen)

30

(-3,1)	(-2,2) (-2,3)
(4,6)	(-1,3) (-2,3)
(-2,1) (1,1)	(-1,3) (0,4)
(-2,1) (-2,2)	(0,4) (1,4)
(-2,2) (0,2)	(-2,3) (-2,4)
(0,2) (1,3)	(-2,4) (0,4)
(0,2) (1,1)	(0,2) (-1,3)
(1,1) (2,2)	(-1,3) (-2,4)
(2,2) (3,3)	(3,3) (4,4)
(3,3) (2,4)	(4,4) (7,7)
(1,3) (1,4)	(7,7) (4,6)
(1,4) (2,4)	(2,4) (3,5)
(1,3) (3,3)	(2,4) (2,6)
(-2,1) (0,2)	(2,6) (4,6)
(0,2) (2,2)	(3,5) (4,5)
(-3,1) (-2,1)	(4,5) (4,6)



Screenshot aus Geogebra mit der Straßenkarte

bei 10 % Verlängerung

> Kürzester Weg: [(-3, 1), (-2, 1), (0, 2), (1, 3), (1, 4), (2, 4), (3, 5), (4, 5), (4, 6)]
>> Länge kürzester Weg: 10.06449510224598
>> Anzahl Abbiegen kürzester Weg: 7

> Optimaler Weg: [(-3, 1), (-2, 1), (0, 2), (1, 3), (1, 4), (2, 4), (2, 6), (4, 6)]
>> Länge optimaler Weg: 10.650281539872886
>> Anzahl Abbiegen optimaler Weg: 6
>>> Umweg im Vergleich zum kürzesten Weg: 0.585786437626906; Maximal mögliche Länge war 11.070944612470578

>> Laufzeit des Programms: 0.2514188289642334 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 15 % Verlängerung

(kürzester Weg ... siehe bei 10 %)

> Optimaler Weg: [(-3, 1), (-2, 1), (-2, 2), (-2, 3), (-2, 4), (0, 4), (1, 4), (2, 4), (3, 5), (4, 5), (4, 6)]
>> Länge optimaler Weg: 11.414213562373096
>> Anzahl Abbiegen optimaler Weg: 5
>>> Umweg im Vergleich zum kürzesten Weg: 1.3497184601271162; Maximal mögliche Länge war 11.574169367582876

>> Laufzeit des Programms: 0.25887036323547363 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

bei 30 % Verlängerung

(kürzester Weg ... siehe bei 10 %)

> Optimaler Weg: [(-3, 1), (-2, 1), (-2, 2), (-2, 3), (-2, 4), (0, 4), (1, 4), (2, 4), (2, 6), (4, 6)]
>> Länge optimaler Weg: 12.0

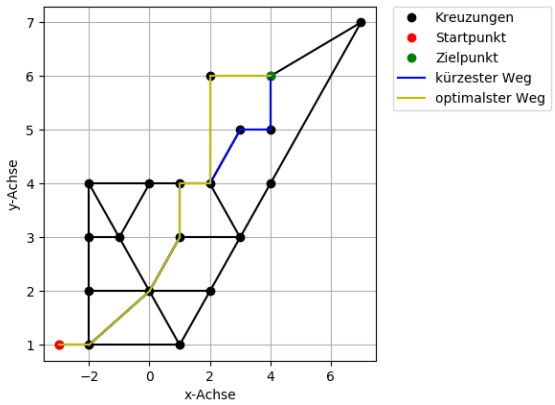
>> Anzahl Abbiegen optimaler Weg: 4
>>> Umweg im Vergleich zum kürzesten Weg: 1.9355048977540203; Maximal mögliche Länge war 13.083843632919773

>> Laufzeit des Programms: 0.25300025939941406 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)

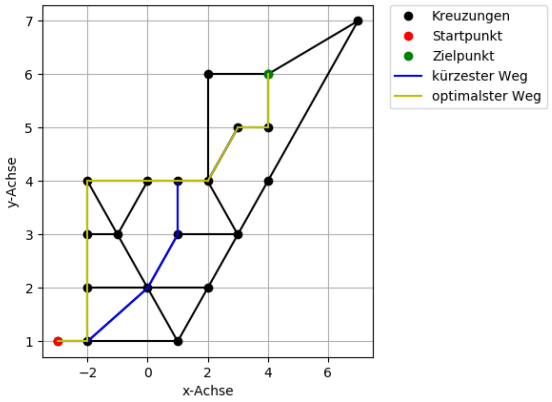
<u>mit Erweiterung</u>
(kürzester Weg ... siehe bei 10 %)
(optimaler Weg ... siehe bei 30 %)

Erweiterung: Berechnung des Wegs mit geringster Anzahl an Abbiegevorgängen:
> 'min. Abbiegen Weg': [(-3, 1), (-2, 1), (1, 1), (2, 2), (3, 3), (4, 4), (7, 7), (4, 6)]
>> Länge von 'min. Abbiegen Weg': 15.647559034406951
>> Anzahl Abbiegen: 2
>>> Umweg im Vergleich zum optimalsten Weg: 3.6475590344069513
>>> Umweg im Vergleich zum kürzesten Weg: 5.583063932160972

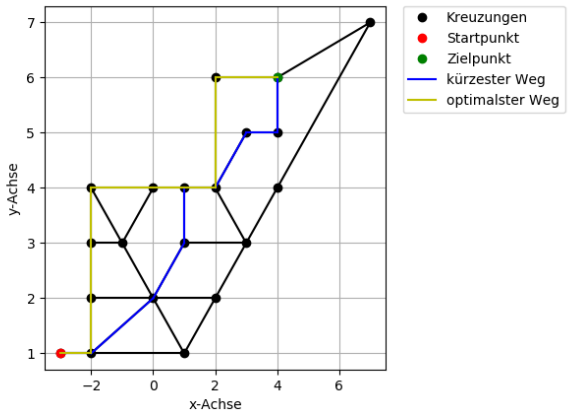
>> Laufzeit des Programms: 0.2650899887084961 Sekunden
(Start der Zeitmessung bei Aufruf der main-Methode)



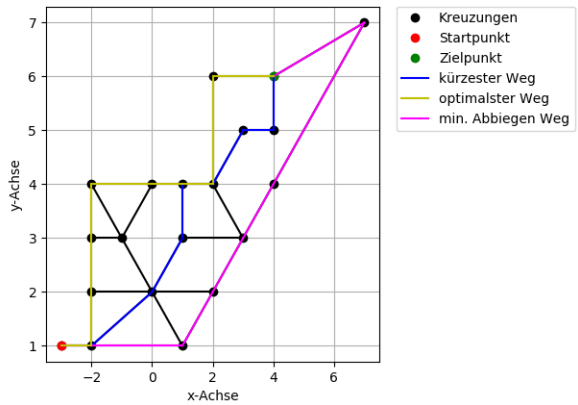
eigenes Beispiel mit 10 % Verlängerung



eigenes Beispiel mit 15 % Verlängerung



eigenes Beispiel mit 30 % Verlängerung



eigenes Beispiel mit 30 % und mit Erweiterung (pink)

Komplexität und Laufzeitanalyse

Ohne Verwendung einer Heuristik für die Suche des optimalsten Wegs würde der Algorithmus in der Tiefensuche im schlimmsten Fall zuerst alle zu langen Wege vom Start- zum Zielpunkt besuchen und die Tiefensuche würde erst schrittweise einen immer kürzeren Weg auswählen

So würden im worst-case zuerst alle einfachen Pfade vom Start- zum Zielpunkt berechnet werden. Die Anzahl dieser einfachen Pfade hängt von der Anzahl n der Knoten im Graph ab. Es gibt somit $n!$ verschiedene Möglichkeiten mit einem einfachen Pfad vom Start- zum Zielpunkt zu gelangen.

Ein naiver Algorithmus hat folglich eine Laufzeit von $O(n!)$.

So hat der Tiefensuche-Algorithmus für den optimalen Pfad bei 15% Verlängerung des Beispiels ‚abbiegen1.txt' ohne Heuristik eine Laufzeit von ca. 22,2 Sekunden, was keineswegs ideal ist.

Durch die Heuristik in der Tiefensuche werden zuerst jene Knoten besucht, die sich nahe am Zielpunkt befinden. Dadurch nähert sich die Tiefensuche vom kürzeren Wegen ausgehend dem optimalen Weg an.
So kann die Laufzeit drastisch gesenkt werden und das eben genannte Beispiel benötigt nur noch ca. 1,22 Sekunden, was sehr akzeptabel ist.

Quellcode

Methoden zur Bestimmung der Anzahl der Abbiegevorgänge

```
def berechneAnzahlAbbiegenPfad(self, pfad: list):
    """ Berechnung der Anzahl, wie häufig auf dem gegebenen Pfad (Liste von Punkte
n) abgebogen werden muss.

    Args:
        pfad (list): Pfad

    Returns:
        float. Anzahl der Abbiegevorgänge im gegebenen Pfad
    """

    # Liste mit allen Steigungen der gegebenen Strecken wird berechnet
    steigungen = []
    for i in range(len(pfad)):
        if i > 0:
            steigungen.append(
                self.berechneSteigungKante((pfad[i-1], pfad[i]))
            )

    # Für jeden Änderung des Betrags der aktuellen Steigerung wird der Zähler um 1
 erhöht

    # die Anzahl der Abbiegevorgänge beträgt zu Beginn 0
    anzahl_abbiegen = 0

    for i in range(len(steigungen)):

        # die Steigung der aktuellen Strecke wird zugewiesen
        aktuell = steigungen[i]

        # falls es eine Strecke vor der aktuellen Strecke gibt
        if i > 0:
            # die Steigung der Strecke vor der aktuellen Strecke wird zugewiesen
            zuvor = steigungen[i-1]

            # wenn die beiden zugewiesenen Steigungen unterschiedlich sind,
            # wird der Zähler erhöht
            if aktuell != zuvor:
                anzahl_abbiegen += 1

    return anzahl_abbiegen

def berechneSteigungKante(self, kante):
    """berechnet die Steigung der eingegebenen Kante."""

    """ Methode zur Berechnung der aktuellen Steigung in der gegebenen Kante

    Args:
        kante: Liste aus zwei Punkten (Knoten)

    Returns:
        float. Steigung der Strecke der beiden Punkte

    """
```

```
    (x1, y1), (x2, y2) = kante

    # Differenz der y-Koordinaten des Start- und Zielpunktes der gegebenen Kante
    y_diff = y2 - y1

    # Differenz der x-Koordinaten des Start- und Zielpunktes der gegebenen Kante
    x_diff = x2 - x1
    try:
        steigung = y_diff / x_diff

    # falls keine Veränderung in x-
Richtung vorliegt, wird der Steigung die 'Zahl' unendlich zugewiesen
    except ZeroDivisionError:
        steigung = math.inf

    return steigung
```

A*-Algorithmus für kürzesten Weg

```
def astar(self, start, ziel, graph):
    """ A*-Algorithmus zur Berechnung des kürzesten Weg.

    Im gegebenen Graph soll vom Start- zum Zielknoten der kürzeste Weg berechnet w
erden.
    Dabei wird eine Schätzfunktion verwendet, um das Verfahren zu optimieren, inde
m der Abstand zum Zielknoten berechnet wird.
    Als Schätzfunktion wird die Methode heuristik_astar() verwendet, welche die Lu
ftlinie berechnet.

    Args:
        start (tuple): Startknoten
        ziel (tuple): Zielknoten
        graph (Graph): Graph als Datenstruktur

    Returns:
        list. Pfad mit dem Weg vom Start- zum Zielknoten
        float. Kosten für diesen Weg
    """
    # Tatsächliche Kosten zu jedem Knoten vom Startknoten aus
    G = {}

    # Geschätzte Kosten vom Start zum Ende über die Knoten
    F = {}

    # Initialisierung der Startwerte
    G[start] = 0
    F[start] = self.heuristik_astar(start, ziel)

    # offene und geschlossene Knoten werden in einem Set gespeichert
    geschlossene_knoten = set()
    # zu Beginn wird der Startknoten zu den offenen Knoten hinzugefügt
    offene_knoten = set([start])
    # Verlauf des Wegs wir in gekommen_von gespeichert
    gekommen_von = {}

    # Solange noch ein Knoten offen ist:
    while len(offene_knoten) > 0:
```

```
        # Wähle den Knoten von der offenen Liste aus, der den geringsten F-
Wert besitzen
        aktueller_knoten = None
        aktueller_F_wert = None

        for knoten in offene_knoten:
            if aktueller_knoten is None or F[knoten] < aktueller_F_wert:
                aktueller_F_wert = F[knoten]
                aktueller_knoten = knoten

        # Überprüfe, ob der Zielknoten erreicht wurde
        if aktueller_knoten == ziel:

            # falls ja, wird die Route rückwärts gegangen
            pfad = [aktueller_knoten]
            while aktueller_knoten in gekommen_von:
                aktueller_knoten = gekommen_von[aktueller_knoten]
                pfad.append(aktueller_knoten)

            # Pfad wird umgekehrt
            pfad.reverse()

            # der Pfad wird mit der Länge zurückgegeben
            return pfad, F[ziel]  # Fertig!

        # Markiere den aktuellen Knoten als geschlossen
        offene_knoten.remove(aktueller_knoten)
        geschlossene_knoten.add(aktueller_knoten)

        # Aktualisierung der Werte für die Nachbarknoten neben dem aktuellen Knote
n
        for item in graph[aktueller_knoten]:
            nachbar_knoten, gewichtung = item[0]

            if nachbar_knoten in geschlossene_knoten:
                # dieser Knoten wurde bereits ausgeschöpft
                continue

            g_wert_kandidat = G[aktueller_knoten] + gewichtung

            # falls der Nachbarknoten noch nicht offen ist,
            # wird er als offener gespeichert
            if nachbar_knoten not in offene_knoten:
                offene_knoten.add(nachbar_knoten)

            elif g_wert_kandidat >= G[nachbar_knoten]:
                # Wenn der G-Wert schlechter als der vorher gefundene ist
                continue

            # G-Wert wird angepasst
            gekommen_von[nachbar_knoten] = aktueller_knoten
            G[nachbar_knoten] = g_wert_kandidat

            # Abstand zum Zielknoten wird geschätzt
            H = self.heuristik_astar(nachbar_knoten, ziel)
            F[nachbar_knoten] = G[nachbar_knoten] + H

    # Falls kein Weg gefunden wurde wird None zurückgegeben
    return None
```

```
def heuristik_astar(self, knoten1, knoten2):
    """ Methode als heuristische Funktion im A*-Algorithmus

    Es wird der euklidische Abstand (Luftlinie) zwischen den beiden gegebenen Knot
en berechnet.

    Args:
        knoten1: Startpunkt
        knoten2: Zielpunkt

    Returns:
        float. euklidische Distanz zwischen Start- und Zielpunkt

    """
    # Methode berechneLänge wird verwendet
    return self.berechneLänge(knoten1, knoten2)
```

Methoden zum Finden des optimalen Wegs mit gegebener Länge

```
def optimalsterPfad(self, graph, start: tuple, ziel: tuple, max_länge: int, pfad=[
], bisher_optimalster_pfad=None, länge_bisherOptimalsterPfad=math.inf, anzahl_abbi
egen_bisherOptimalsterPfad=math.inf):
    """ rekursive Methode zur Tiefensuche im gegebenen Graph

    Dabei wird die Tiefensuche abgebrochen, falls der aktuelle Pfad länger als max
imal erlaubt ist.
    Ebenfalls wird die Tiefensuche abgebrochen, falls im aktuellen Pfad öfters abg
ebogen wird als der bisher optimalste Pfad (bisher beste)

    Args:
        graph (Graph): Datenstruktur Graph zum Verwalten der Straßen
        start (tuple): Startknoten
        ziel (tuple): Zielknoten
        max_länge (int): maximale Länge des optimalen Pfads, wurde berechnet aus d
er maximalen Verlängerung (Eingabe des Benutzers)
        pfad (list): aktueller Pfad, zu dem der Startknoten hinzugefügt wird
        bisher_optimalster_pfad (list): zu Beginn 'None'. Falls der aktuelle Pfad
den Zielknoten erreicht und nicht davor abgebrochen wird,
                                        gilt dieser Pfad als bisher optimalster Pf
ad.
        länge_bisherOptimalsterPfad (float): zu Beginn 'unendlich'. Wird ein bishe
r_optimalster_pfad gefunden, so wird die Länge dieses Pfades aktualisiert.
        anzahl_abbiegen_bisherOptimalsterPfad (float): zu Beginn 'unendlich'. Wird
 ein bisher_optimalster_pfad gefunden, so wird die Anzahl der Abbiegevorgänge dies
es Pfades aktualisiert.

    Returns:
        list. der bisher_optimalster_pfad ist zum Schluss der Methode der beste op
timalste Pfad und wird zurückgegeben
        float. Länge des besten optimalsten Pfades wird zurückgegeben
        float. Anzahl der Abbiegevorgänge des optimalesten Pfades wird zurückgegeb
en.
    """
    # Startknoten wird zum aktuellen Pfad hinzugefügt
    pfad.append(start)

    # Länge des aktuellen Pfads wird berechnet
    länge_aktueller_pfad = self.berechneLängePfad(pfad)
```

```
    # Länge des aktuellen Pfads darf nicht länger als die maximale Länge sein
    if länge_aktueller_pfad <= max_länge:

        # die Anzahl der Abbiegevorgänge des aktuellen Pfads wird berechnet
        anzahl_abbiegen_aktueller_pfad = self.berechneAnzahlAbbiegenPfad(pfad)

        # die Anzahl der Abbiegevorgänge des aktuellen Pfads sollen weniger sein a
ls beim bisher optimalsten Pfad
        if anzahl_abbiegen_aktueller_pfad <= anzahl_abbiegen_bisherOptimalsterPfad
:

            # falls der aktuelle Knoten der Zielknoten ist
            if start == ziel:

                # Sonderfall: Anzahl Abbiegevorgänge des aktuellen und bisher opti
malsten sind gleich,
                # so wird der kürzere weiter verwendet
                if anzahl_abbiegen_bisherOptimalsterPfad == anzahl_abbiegen_aktuel
ler_pfad:

                    if länge_aktueller_pfad < länge_bisherOptimalsterPfad:
                        bisher_optimalster_pfad = pfad
                        länge_bisherOptimalsterPfad = länge_aktueller_pfad
                        anzahl_abbiegen_bisherOptimalsterPfad = anzahl_abbiegen_ak
tueller_pfad

                # die Eigenschaften des bisher optimalsten Pfades werden aktualisi
ert,
                # da der aktuelle Pfad besser ist
                else:
                    bisher_optimalster_pfad = pfad
                    länge_bisherOptimalsterPfad = länge_aktueller_pfad
                    anzahl_abbiegen_bisherOptimalsterPfad = anzahl_abbiegen_aktuel
ler_pfad


            # Die Nachfolgeknoten werden anhand ihrem Abstand zum Zielknoten aufst
eigend sortiert
            sortierte_liste_nachfolger = sorted(graph[start],
                                                key=lambda item: self.berechneLäng
e(item[0][0], self.zielpunkt))

            for nachfolger_item in sortierte_liste_nachfolger:

                knoten, gewichtung = nachfolger_item[0]

                # rekursiver Aufruf mit den Nachfolgerknoten
                if knoten not in pfad:
                    bisher_optimalster_pfad, länge_bisherOptimalsterPfad, anzahl_a
bbiegen_bisherOptimalsterPfad = self.optimalsterPfad(graph, knoten, ziel, max_läng
e,

                                                         pfad.copy(), bisher_optimals
ter_pfad, länge_bisherOptimalsterPfad, anzahl_abbiegen_bisherOptimalsterPfad)

    # der in diesem Teilgraphen optimalster Pfad wird zusammen mit seinen Eigensch
aften zurückgegeben.
    return bisher_optimalster_pfad, länge_bisherOptimalsterPfad, anzahl_abbiegen_b
isherOptimalsterPfad
```

Erweiterung: Methoden zur Bestimmung des Wegs mit min. Anzahl an Abbiegevorgängen ohne Längenbegrenzung

```
def heuristik_dfs(self, knoten: tuple, aktueller_pfad: list):
    """ Funktion zur Optimierung der Tiefensuche der Methode minAbbiegenPfad()

    Hierfür wird berechnet, wie oft man abbiegen müsste, falls der gegebene Knoten
 im aktuellen Pfad sein würde
    Diese Anzahl der Abbiegevorgänge im neuen Pfad wird zurückgegeben.

    Args:
        knoten (tuple): Knoten, der zu aktueller_pfad hinzugefügt wird
        aktueller_pfad (list): Ausgangspfad

    Returns:
        int. Anzahl der Abbiegevorgänge im neu entstandenen Pfad

    """
    # Falls überhaupt bereits Knoten im aktuellen Pfad sind
    if aktueller_pfad:
        neuer_pfad = aktueller_pfad.copy()
        neuer_pfad.append(knoten)

        return self.berechneAnzahlAbbiegenPfad(neuer_pfad)

    # sind keine Knoten im aktuellen Pfad, so wird 0 zurückgegeben
    else:
        return 0

def minAbbiegenPfad(self, graph, start: tuple, ziel: tuple, pfad=[], bisher_minAbb
iegen_pfad=None, anzahl_abbiegen_bisherMinAbbiegen_pfad=math.inf):
    """ Methode zum Berechnen des Pfades mit minimalen Abbiegen ohne Eingrenzung

    Ziel dieser Methode ist es, denjenigen Pfad zu finden, in dem am wenigsten Mal
 abgebogen werden muss.
    Dabei wird der aktuelle Pfad verworfen, wenn man in diesem öfters abbiegen mus
s, als in dem bisher 'besten' Pfad (bisher_minAbbiegen_pfad).

    Args:
        graph (Graph): Datenstruktur Graph zum Verwalten der Straßen
        start (tuple): Startknoten
        ziel (tuple): Zielknoten
        pfad (list): aktueller Pfad, zu dem der Startknoten hinzugefügt wird

        bisher_optimalster_pfad (list): zu Beginn 'None'.
            Falls der aktuelle Pfad den Zielknoten erreicht und nicht davor abgebr
ochen wird,
            gilt dieser Pfad als bisher optimalster Pfad.

        anzahl_abbiegen_bisherOptimalsterPfad (float): zu Beginn 'unendlich'.
            Wird ein bisher_optimalster_pfad gefunden, so wird die Anzahl der Abbi
egevorgänge dieses Pfades,
            also anzahl_abbiegen_bisherOptimalsterPfad, aktualisiert.

    Returns:
        list. der bisher_optimalster_pfad ist zum Schluss der Methode der beste op
timalste Pfad und wird zurückgegeben
```

```
        float. Anzahl der Abbiegevorgänge des optimalesten Pfades wird zurückgegeb
en.
    """

    # Aktueller Startknoten wird zum aktuellen Knoten hinzugefügt
    pfad.append(start)

    # Die Anzahl der Abbiegevorgänge im aktuellen Pfad werden berechnet
    anzahl_abbiegen_aktueller_pfad = self.berechneAnzahlAbbiegenPfad(pfad)

    # Falls diese Anzahl geringer als die Anzahl der Abbiegevorgänge des bisher 'b
esten' Pfades ist,
    # kann mit diesem aktuellen Pfad fortgefahren werden.
    if anzahl_abbiegen_aktueller_pfad < anzahl_abbiegen_bisherMinAbbiegen_pfad:

        # falls der aktuelle Knoten der Zielknoten ist
        if start == ziel:

            # aktueller Pfad ist nun der neue 'beste' Pfad mit geringster Anzahl a
n Abbiegevorgängen
            bisher_minAbbiegen_pfad = pfad
            anzahl_abbiegen_bisherMinAbbiegen_pfad = anzahl_abbiegen_aktueller_pfa
d

        # mithilfe der heuristischen Funktion heuristik_dfs wird für jedem Nachbar
knoten ein Wert bestimmt
        # der beste Nachbarknoten ist derjenige, mit dem geringsten heuristischen
Wert
        sortierte_liste_nachfolger = sorted(graph[start],
                                            key=lambda item: self.heuristik_dfs(it
em[0][0], pfad))

        for nachfolger_item in sortierte_liste_nachfolger:
            nachfolger, gewichtung = nachfolger_item[0]


            if nachfolger not in pfad:

                # die Methode wird rekursiv aufgerufen
                bisher_minAbbiegen_pfad, anzahl_abbiegen_bisherMinAbbiegen_pfad =
self.minAbbiegenPfad(graph, nachfolger,

                     ziel, pfad.copy(), bisher_minAbbiegen_pfad, anzahl_abbiegen_
bisherMinAbbiegen_pfad)

    return bisher_minAbbiegen_pfad, anzahl_abbiegen_bisherMinAbbiegen_pfad
```

Main-Methode

```
def main(self):
    """ Hauptmethode der Klasse und wird von __init__() aufgerufen """

    # Zeitmessung wird gestartet
    start_zeit = time.time()

    # ein Graph-Objekt wird erzeugt
    # dieser Graph besteht aus Kanten, welche wiederum aus zwei Knoten bestehe
n
    # die Länge einer Kante dient als Gewichtung
```

```
    self.graph = Graph()
    for kante in self.liste_verbindungen:
        strecke = self.berechneLänge(*kante)
        self.graph.add_Kante(*kante, strecke)

    # Initialisierung des Koordinatensystem zum Zeigen der Straßenkarte
    self.koordinatensystem = Koordinatensystem(self.erweiterung_minAbbiegen)
    self.koordinatensystem.zeichneStraßenkarte(
        self.startpunkt,
        self.zielpunkt,
        liste_knoten=self.graph.knoten,
        liste_verbindungen=self.liste_verbindungen
    )

    # Kürzester Weg vom Start- zum Zielpunkt und dessen Länge wird berechnet
    # A*-Algorithmus wird dabei verwendet
    kürzester_weg, länge_kürzester_weg = self.astar(self.startpunkt, self.ziel
punkt, self.graph)
    anzahl_abbiegen_kürzester_weg = self.berechneAnzahlAbbiegenPfad(kürzester_
weg)

    # der kürzeste Weg wird blau im Koordinatensystem dargestellt
    self.koordinatensystem.zeichnePfad(kürzester_weg, 'b-')


    # maximale Länge wird mithilfe der gegebenen maximalen Verlängerung (in %)
 berechnet
    maximale_länge = länge_kürzester_weg + \
        länge_kürzester_weg*(self.maximale_verlängerung/100)

    # optimaler Pfad wird ermittelt
    optimaler_pfad, länge_optimaler_pfad, anzahl_abbiegen_optimaler_pfad = sel
f.optimalsterPfad(self.graph, self.startpunkt, self.zielpunkt, maximale_länge)

    umweg_optimaler_imVergleich_kürzester = länge_optimaler_pfad - länge_kürze
ster_weg


    if self.erweiterung_minAbbiegen:
        #### Erweiterung
        # Pfad mit minimalstem Abbiegen (ohne Eingrenzung) wird berechnet
        minAbbiegen_pfad, anzahl_abbiegen_minAbbiegen_pfad = self.minAbbiegenP
fad(self.graph, self.startpunkt, self.zielpunkt)

        # Länge dieses Pfades
        länge_minAbbiegen_pfad = self.berechneLängePfad(minAbbiegen_pfad)

        # Umweg im Vergleich zum optimalen Pfad
        umweg_minAbbiegen_imVergleich_optimaler = länge_minAbbiegen_pfad - län
ge_optimaler_pfad

        # Umweg im Vergleich zum kürzesten Pfad
        umweg_minAbbiegen_imVergleich_kürzester = länge_minAbbiegen_pfad - län
ge_kürzester_weg

        # min Abbiegen-Pfad wird gezeichnet
```

```
        self.koordinatensystem.zeichnePfad(minAbbiegen_pfad, 'fuchsia')


    # optimaler Pfad wird gezeichnet am Schluss gezeichnet
    self.koordinatensystem.zeichnePfad(optimaler_pfad, 'y-')

    # Zeitmessung wird beendet
    ende_zeit = time.time()


    #### Konsolenausgaben
    # Kürzester
    print(f"> Kürzester Weg: {kürzester_weg}")
    print(f" >> Länge kürzester Weg: {länge_kürzester_weg}")
    print(f" >> Anzahl Abbiegen kürzester Weg: {anzahl_abbiegen_kürzester_weg}
")

    # Optimaler
    print(f"\n> Optimaler Weg: {optimaler_pfad}")
    print(f" >> Länge optimaler Weg: {länge_optimaler_pfad}")
    print(f" >> Anzahl Abbiegen optimaler Weg: {anzahl_abbiegen_optimaler_pfad
}")
    print(f" >>> Umweg im Vergleich zum kürzesten Weg: {umweg_optimaler_imVerg
leich_kürzester}; Maximal mögliche Länge war {maximale_länge}")

    if self.erweiterung_minAbbiegen:
        ### Erweiterung
        # Min Abbiegen
        print(f"\nErweiterung: Berechnung des Wegs mit geringster Anzahl an Ab
biegevorgängen:")
        print(f"> 'min. Abbiegen Weg': {minAbbiegen_pfad}")
        print(f" >> Länge von 'min. Abbiegen Weg': {länge_minAbbiegen_pfad}")
        print(f" >> Anzahl Abbiegen: {anzahl_abbiegen_minAbbiegen_pfad}")
        print(f" >>> Umweg im Vergleich zum optimalsten Weg: {umweg_minAbbiege
n_imVergleich_optimaler}")
        print(f" >>> Umweg im Vergleich zum kürzesten Weg: {umweg_minAbbiegen_
imVergleich_kürzester}")

    # Zeitmessung wird ausgegeben
    print(f"\n>> Laufzeit des Programms: {ende_zeit-
start_zeit} Sekunden \n (Start der Zeitmessung bei Aufruf der main-Methode)")

    self.koordinatensystem.show()
```