Aufgabe 3: Abbiegen

Teilnahme-Id: 55174

Bearbeiter-/in dieser Aufgabe:
Nils Weißer

Inhaltsverzeichnis

1 Lösungsidee

1.1 Theoretische Beschreibung der Aufgabenstellung

Das in der Aufgabenstellung vorgestellte Problem ist ein klassischer Fall von Wegfindung. Das Ziel ist es einen Wegfindungsalgorithmus zu erstellen, der bei Angabe der Kanten zwischen den Knoten, sowie Start- und Zielpunkt, nach bestimmten Kriterien den optimalen Weg findet. Die beiden Parameter, die ein Rolle spielen, sind die Richtungswechselanzahl und die Streckenlänge. Zunächst wird nach dem Weg mit der kleinstmöglichen Streckenlänge gesucht und die Richtungswechselanzahl ist zweitrangig. Durch eine Benutzereingabe kann jedoch eine maximale Verlängerung der Streckenlänge bestimmt werden, sodass in diesem Bereich der Weg mit der minimalen Richtungswechselanzahl der optimale Weg ist. Weitere Einschränkungen oder Bedingungen, wie zum Beispiel Hindernisse, gibt es nicht.

1.2 Zerlegung in Teilaufgaben

Zunächst sollen alle möglichen Wege gefunden werden, das heißt Wege vom Start- bis zum Zielpunkt über die Kanten.

Dann wird die Streckenlänge sowie die Richtungswechselanzahl der Wege berechnet. Die Streckenlänge der einzelnen Teilstrecken, bzw. Kanten, werden mit dem Satz des Pythagoras berechnet. Wenn die beiden Knoten der Kante $P(x_1|\, y_1)$ und $Q(x_2|\, y_2)$ sind, ist die Formel für die Berechnung der Länge der Teilstrecke:

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

Für die Berechnung der Richtungswechselanzahl, berechnet man zunächst den Winkel der Teilstrecke und vergleicht ihn dann mit der vorherigen Teilstrecke. Wenn die Winkel identisch sind, wurde die Richtung nicht gewechselt, wenn sie nicht identisch sind, wird die Richtungswechselanzahl um 1 erhöht. Zur Berechnung des Winkels einer Kante benutzen wir den Arkustangens und erhalten somit einen Wert zwischen -π und π. Die Formel lautet:

$$\alpha = \arctan\left(\frac{x_2 - x_1}{y_2 - y_1}\right)$$

Wenn alle Wege gefunden wurden und ihre Parameter berechnet wurden, werden diese nach den in 1.1 vorgestellten Kriterien verglichen, um am Ende den optimalen Weg zu finden.

1.3 Der Algorithmus

Der Algorithmus basiert auf einem Bruteforce-Verfahren. Alle möglichen Wege werden ausprobiert, um am Ende den optimalen Weg herauszusuchen. Der große Vorteil dieses Algorithmus ist, dass immer der optimale Weg, und nicht nur eine Annäherung oder der „wahrscheinlich optimale Weg“, gefunden wird. Das Bruteforce-Verfahren ist zu 100% genau und auch bei Sonderfällen stimmt das Ergebnis.

Bei dem Bruteforce-Verfahren wird bei dem Startpunkt angefangen und zu allen benachbarten Knoten gegangen. Von diesen Knoten aus, wird wieder zu allen benachbarten Knoten gegangen. Diese Prozedur wird immer weiter fortgesetzt. Wenn man dies als Grafik darstellen würde, hätte man eine Struktur, die einem Baumdiagramm gleicht. Damit dieser Vorgang funktioniert, müssen zunächst zwei Regeln aufgestellt werden, um ein Ergebnis zu erreichen und um Endlosschleifen zu verhindern.

1. Ein Weg wird nicht weiter verfolgt, wenn der Zielpunkt erreicht wird.

2. Ein Weg wird nicht weiter verfolgt, wenn ein Knoten erreicht wird, der bereits im Laufe des selben Weges erreicht wurde.

Da bei diesem Verfahren jedoch wirklich alle Wege gesucht werden, ist die Laufzeit dementsprechend sehr hoch. Bei einigen der Testbeispiele wäre es vermutlich schneller, wenn sich Bilal einen unterirdischen Tunnel graben würde, der direkt zum Zielpunkt führt. Daher werden Heuristiken eingeführt, um die Laufzeit zu verbessern:

1. Ein Weg wird nicht weiter verfolgt, wenn ein Knoten erreicht wird, der bereits auf einem anderen Weg erreicht wurde und er dafür weniger Richtungswechsel und Distanz gebraucht hat.

2. Ein Weg wird nicht weiter verfolgt, wenn die bis jetzt erreichte Richtungswechselanzahl und zurückgelegte Streckenlänge über den Parametern des momentan besten Weges, der bereits vollendet ist, liegt.

An der Fehlerfreiheit des Ergebnisses ändert sich jedoch nichts, da nur Wege vorzeitig beendet werden, die nicht mehr zum optimalen Weg werden können.

1.4 Laufzeitanalyse

Bei dem Algorithmus ohne Heuristiken gibt es keinen Worst- oder Best-Case, da jeder Weg durchsucht wird und nicht vorzeitig abgebrochen wird. Die Laufzeit des Algorithmus ohne Optimierung durch die Heuristiken wäre O(|V| + |E|), wobei |V| für die Anzahl der Knoten und |E| für die Anzahl der Kanten steht. Da durch die Heuristiken aber nur noch ein kleiner Teil aller Möglichkeiten aus-

probiert wird, hat diese Laufzeitüberlegung keine Aussagekraft. Bei dem optimierten Verfahren ist es schwierig eine generelle Berechnung der Laufzeit zu erstellen, da die Laufzeit stark vom Anwendungsbeispiel abhängt.

Bei Netzen mit einem sehr kleinen Datensatz, also mit sehr wenigen Knoten, ist die Laufzeit niedriger, wenn man ein reines Bruteforce-Verfahren benutzt, da die Laufzeit bei wenigen Knoten noch sehr niedrig ist und die Laufzeit der Heuristiken im Vergleich höher ist. Wenn die Anzahl der Knoten und Kanten jedoch steigt, werden die Heuristiken immer wichtiger, um die Laufzeiten niedrig zu halten.

Zur Ermittlung der Laufzeiten werden diese im Programmablauf bestimmt. Hiermit können die Laufzeitoptimierungen durch die Heuristiken praktisch bewiesen werden.

1.5 Sonderfälle

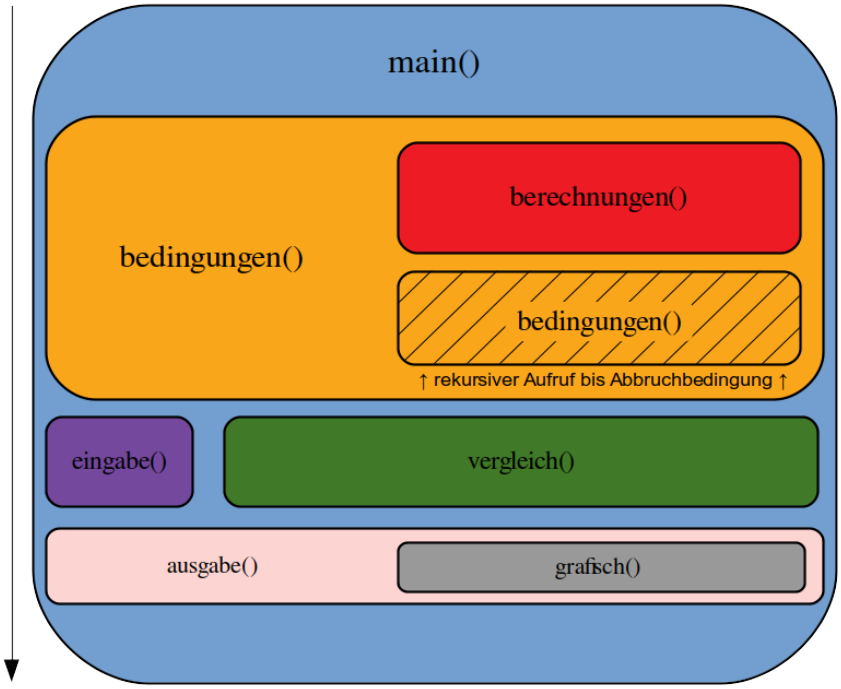
Folgende Sonderfälle können auftreten:

1. Es gibt keinen Weg, der den Start- und den Zielpunkt verbindet.

2. Es gibt zwei oder mehr Wege, die eine identische Richtungswechselanzahl und Streckenlänge aufweisen.

Diese Sonderfälle sollen erkannt, und dementsprechend behandelt werden.

2 Umsetzung



Diese Abbildung zeigt eine Übersicht der Python3-Implementierung des Algorithmus. In dieser Übersicht werden alle Funktion als Rechtecke mit verschiedenen Farben dargestellt. Der zeitliche Ablauf des Programms entspricht dem Pfeil an der linken Seite der Abbildung. Wenn eine Funktion innerhalb einer anderen Funktion geschrieben ist, wird die innere Funktion von der äußeren aufgerufen. Außerdem steht die Größe der Rechtecke in Zusammenhang mit der Relevanz der Funktionen. Wenn ein Rechteck besonders groß ist, ist die Funktion wichtiger als wenn ein Rechteck eher klein ist.

Das Programm beginnt mit dem Einlesen der Datei, die die Anzahl der Strecken, den Start- sowie Zielpunkt und die Koordinaten aller Strecken enthält. Dabei werden Variablen (`anzahl_str`, `start`, `ziel`, `liste_koords`) mit den Elementen der Datei initialisiert und direkt zu sinnvollen Typen umgewandelt um sie weiterverwenden zu können.

Aus der Liste der Kanten `liste_koords[]` werden alle Knoten extrahiert und in der Liste `alle_knoten[]` gespeichert. Dann werden in einem Dictionary `knotenliste{}`, mit Hilfe einer verschachtelten for-Schleife, alle Knoten mit ihren möglichen Nachfolgeknoten gespeichert.

In der Funktion main() werden zunächst die globalen Variablen `moeglichkeiten[]` und `niedrigste_kurv_und_dist[]` deklariert. Dann wird die Funktion bedingungen() aufgerufen.

Die Funktion bedingungen() überprüft zunächst, ob die Koordinaten des übergebenen Knoten den Koordinaten des Zielknotens entsprechen. Ist dies der Fall werden zunächst die Berechnungen mit der Funktion berechnungen() durchgeführt, die gleich noch einmal genauer erläutert werden. Wenn die Richtungswechselanzahl und die Streckenlänge dieser Gesamtstrecke nun unter dem momentanem Minimum dieser Werte der vollendeten Strecken liegen, wird die Variable `niedrigste_kurv_und_dist[]` auf diese Werte gesetzt. Zudem wird die Gesamtstrecke mit ihren Knoten sowie ihrer Richtungswechselanzahl und ihrer Streckenlänge zur Liste `moeglichkeiten[]` hinzugefügt. Wenn der Knoten jedoch nicht der Zielknoten ist, wird als nächstes überprüft, ob der Knoten bereits in der lokalen Liste `fliste[]` zu finden ist. In der Liste `fliste[]` befinden sich die Attribute, wie z.B. die Koordinaten, der vorherigen Knoten. Dies würde bedeuten, dass der Knoten im Laufe des Weges bereits erreicht wurde. Somit kann die Funktion abgebrochen werden. Wenn dies nicht der Fall ist, wird als nächstes versucht auf den letzten Eintrag der lokalen Liste `fliste[]` zuzugreifen. Im letzten Eintrag sind die Koordinaten des vorherigen Knoten, die Richtungswechselanzahl und die Streckenlänge bis zum vorherigen Knoten, sowie der Winkel der vorherigen Strecke enthalten. Diese Werte werden nun der Funktion bedingungen() übergeben. Wenn das Programm nicht auf den letzten Eintrag zugreifen kann, weil er nicht existiert, wird ein IndexError zurückgegeben. Dieser IndexError wird mit einer Ausnahmebehandlung (try-except) aufgefangen. Wenn dieser Fehler auftritt, bedeutet es, dass der momentane Knoten

der Startpunkt ist und es somit keinen vorherigen Knoten geben kann. Dann kann die Funktion berechnungen() nicht ausgeführt werden.

Die Funktion berechnungen() berechnet zunächst die Distanz zwischen dem vorherigen Knoten und dem aktuellen Knoten. Dies wird mit dem Satz des Pythagoras durchgeführt. Für die Berechnung der Wurzel wurde das Python-Standardmodul math importiert. Dann wird mit Hilfe des Arkustangens der Winkel der Strecke zwischen dem vorherigen und aktuellen Knoten berechnet. Auch für diesen Vorgang wird das Modul math verwendet. Wenn es keine Änderung, entweder zwischen den x- oder den y-Koordinaten der beiden Knoten, gibt, kann die Formel nicht angewandt werden, da man durch 0 teilen würde. Somit wird der Winkel bei diesen beiden Sonderfällen manuell bestimmt. Wenn der Winkel der aktuellen Strecke nicht dem Winkel der vorherigen Strecke entspricht, wird der Integer `kurven` um 1 erhöht. Um Rundungsfehler auszugleichen, überprüft das Programm ob die Differenz der beiden Winkel größer als 0,0001 ist. Dann wird die Variable `strecke` um die Länge der aktuellen Strecke erhöht. In dieser Variable ist die Streckenlänge vom Start bis zu dem aktuellen Knoten gespeichert. Als nächstes wird überprüft ob dieser Knoten bereits von einem anderen Weg erreicht wurde. Wenn dies nicht der Fall ist, wird die Richtungswechselanzahl und die Streckenlänge des momentanen Knoten in dem Dictionary `kurv_und_dist{}` an der Stelle des Knoten gespeichert. Wenn es bereits Werte an dieser Stelle gibt, werden die bereits gespeicherten Werte mit den neuen Werten verglichen. Wenn die Richtungswechselanzahl und die Streckenlänge des aktuellen Knoten unter den gespeicherten Werten liegt, ersetzen sie die im Dictionary `kurv_und_dist{}` gespeicherten Werte. Dann werden die Koordinaten des Knoten, die Richtungswechselanzahl und die Streckenlänge an dieser Stelle, sowie der Winkel der Strecke zurückgegeben und in der Variable `berechnetes[]` gespeichert. Wenn beide neuen Werte über den alten Werten liegen, wird None zurückgegeben und die Funktion bedingungen() abgebrochen. Die Liste `berechnetes[]` wird dann in der lokalen Liste `fliste[]` gespeichert. Wenn die Funktion berechnungen() nicht aufgerufen werden konnte, da es noch keinen vorherigen Knoten gab, werden die Koordinaten des Startpunkts, 0 für die Richtungswechselanzahl, 0 für die Streckenlänge und None für den Winkel in der Liste `fliste[]` gespeichert.

Zurück in der Funktion bedingungen() wird überprüft ob die Richtungswechselanzahl und die Streckenlänge über den Werten, die in der Liste `niedrigste_kurv_und_dist[]` liegen. In dieser Liste sind die momentan niedrigste Richtungswechselanzahl und niedrigste Streckenlänge der bereits vollendeten Wege gespeichert. Wenn dieser boolesche Test TRUE zurückgibt, wird die Funktion abgebrochen.

Wenn keine dieser Abbruchbedingungen eintritt, wird die Funktion bedingungen() in einer for-Schleife, so oft aufgerufen, wie es Werte für den aktuellen Knoten in dem Dictionary knotenliste{} gibt,

also die Anzahl der nachfolgenden Knoten. Die Nachfolgeknoten sind dann die Übergabewerte für die Funktion bedingungen(), sowie die lokale Liste `fliste[]`. Durch diese for-Schleife wird die Funktion bedingungen() also in der selben Funktion aufgerufen. Dadurch ist diese Funktion eine rekursive Funktion.

Wenn die globale Liste `moeglichkeiten[]` leer ist, das heißt keine Strecke gefunden wurde, wird dies dem Benutzer mit einem print-Befehl signalisiert. Wenn die Liste nicht leer ist, wird zunächst durch die Funktion eingabe() der Integer `max_abweichung[]` vom Benutzer abgefragt und dann die Funktion vergleich() aufgerufen.

In der Funktion vergleich() wird zunächst mit Hilfe einer for-Schleife in der Liste `moeglichkeiten[]` iterativ die minimale Streckenlänge aller gefundenen Wege gesucht und in der Variable `min_strecke` gespeichert. Außerdem werden in der Liste `vergleiche[]` die Richtungswechselanzahl und Streckenlänge aller Wege gespeichert. Dann wird die Abweichung in Prozent der Streckenlänge jedes Weges im Vergleich zu der minimalen Streckenlänge mit einer weiteren for-Schleife berechnet. Wenn diese Prozentzahl unter der Variable `max_abweichung` liegt, werden die Attribute des Weges sowie der Index in der Liste `vergleiche[]` des Weges, in der Liste `unter_prozent[]` gespeichert. Weiterhin wird mit der BuiltIn-Funktion min() die minimale Richtungswechselanzahl der Wege in der Liste `unter_prozent[]` gesucht und in der Variable `min_kurv` gespeichert. Nun werden aus der Liste `unter_prozent[]` die Wege mit einer Richtungswechselanzahl, die `min_kurv` entspricht, extrahiert und miteinander in Bezug auf die Streckenlänge verglichen. Dieser Vorgang wird mit einer for-Schleife und verschachtelten if-Abfragen erledigt. Zuletzt gibt die Funktion die Koordinaten, sowie die Richtungswechselanzahl und Streckenlänge des Weges mit der niedrigsten Streckenlänge bei dem Vergleich zurück. Diese Werte werden in der globalen Liste `beste_strecke[]` gespeichert.

Die Attribute dieser Liste werden der Funktion ausgabe() übergeben, die daraus nun eine Ausgabe in Textform mit einem print-Befehl erstellt, sowie, falls das Modul matplotlib installiert ist, fragt ob eine grafische Ausgabe erwünscht ist. Wenn der Benutzer eine grafische Ausgabe fordert, wird die Funktion grafisch() aufgerufen.

In der Funktion grafisch() werden zunächst alle Strecken geplottet. Dann wird der übergebene Weg in grün geplottet. Zuletzt wird der Start- und der Zielknoten rot markiert und die Grafik wird aufgerufen.

3 Erweiterungsausblicke

3.1 Bilal in den Alpen

Eine interessante Erweiterung der Aufgabenstellung wäre, anstatt eines 2D-Koordinatensystems, ein 3D-Koordinatensystem zu benutzen. Bilal würde in einer bergigen Landschaft wohnen und müsste die z-Koordinate auch mit einberechnen. Bilal fährt nicht gerne bergauf. Bergab fährt er jedoch sehr gerne. Man könnte also den Höhenunterschied als dritten Parameter, neben der Richtungswechselanzahl und der Streckenlänge, benutzen, um den optimalen Weg zu finden. Zum Beispiel könnte Bilal eine höhere Richtungswechselanzahl in Kauf nehmen, wenn er dafür weniger bergauf fahren müsste.

3.2 Bilal hat Gegenwind

Ein weitere Erweiterung wäre das Einbringen von Wind. Durch eine Benutzereingabe könnte die Windrichtung angegeben werden. Bilal bevorzugt Strecken, die einen hohen Rückenwind-, bzw. niedrigen Gegenwindanteil haben. Zum Beispiel könnte Bilal eine höhere Richtungswechselanzahl in Kauf nehmen, wenn er dafür weniger Teilstrecken mit direktem Gegenwind, bzw. mehr Teilstrecken mit Rückenwind hat.

4 Beispiele

4.1 Einführung

Hier sind die vier vorgegebenen Beispiele sowie die beiden 1.5 dargestellten Sonderfälle enthalten. Die Ausgabe in Textform beinhaltet die vom Benutzer angegebene maximale Verlängerung, die Verlängerung im Vergleich zum kürzesten Weg, die Anzahl der Richtungswechsel, die Streckenlänge, sowie die Koordinaten der Knoten, über die der Weg führt.

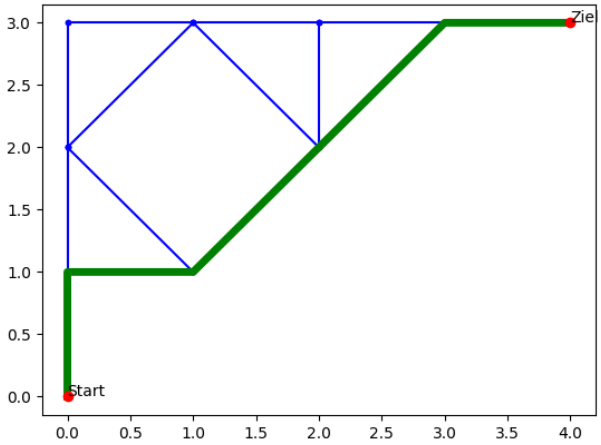
Die angegeben Laufzeit bezieht sich ausschließlich auf den Programmteil, der den Algorithmus ausführt. Das Einlesen der Datei, die Eingabe und die Ausgabe sind nicht enthalten. Die Laufzeit wurde mit Python BuiltIn-Modul „time“ berechnet. Das Programm wurde auf einem Laptop mit dem CPU Intel Core i5-3317U ausgeführt. Dieser Prozessor hat 4 Kerne à 1.70GHz und erschien vor ca. 8 Jahren.

Die grafischen Ausgaben wurden mit Hilfe der Programmbibliothek „matplotlib“ in Python erstellt.

Zu jedem Beispiel wird zunächst der kürzeste Weg gezeigt, das heißt der Weg mit einer maximalen Verlängerung von 0%. Zuletzt wird der Weg mit der minimalen Richtungswechselanzahl gezeigt, also einer maximalen Verlängerung von 100%. Wenn es Wege gibt, die zwischen diesen bei-

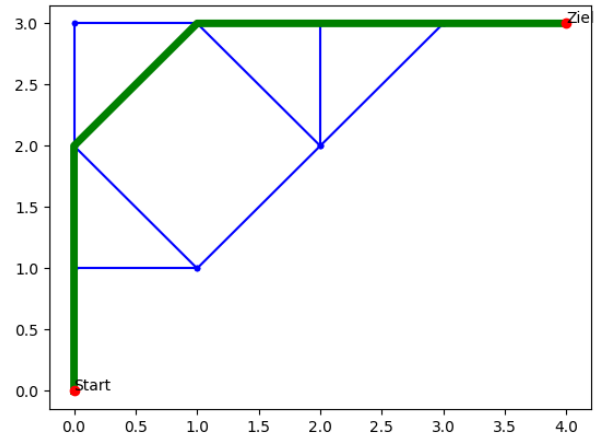
den extremen liegen, werden sie ebenfalls gezeigt. Die Programmausgabe wird immer unter der grafischen Ausgabe notiert.

4.2 Beispiel 1



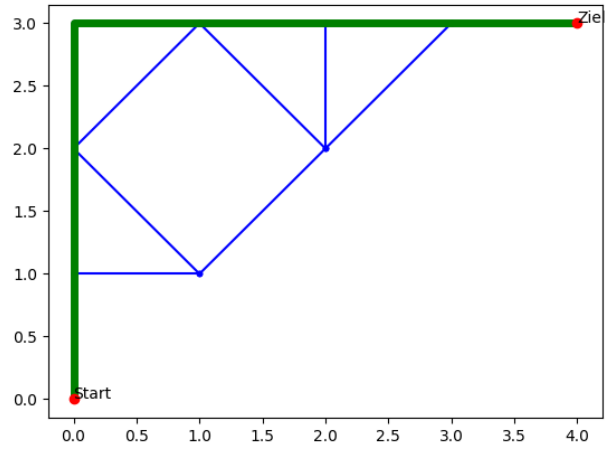

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verlängerung von
 0% mit einer Verlängerung von 0.0%, 3 Richtungswechsel/n und der Streckenlänge
5.82842712474619 geht über die Punkte: [[0.0, 0.0], [0.0, 1.0], [1.0, 1.0], [2.0
, 2.0], [3.0, 3.0], [4.0, 3.0]]
Laufzeit: 0.0014109611511230469 Sekunden
```

==========



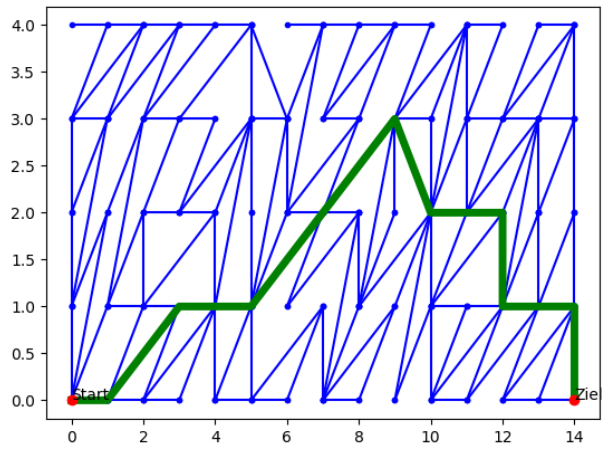

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 15% mit einer Verlängerung von 10.050506338833458%, 2 Richtun
gswechsel/n und der Streckenlänge 6.414213562373095 geht über die Punkt
e: [[0.0, 0.0], [0.0, 1.0], [0.0, 2.0], [1.0, 3.0], [2.0, 3.0], [3.0, 3
.0], [4.0, 3.0]]
Laufzeit: 0.002167224884033203 Sekunden
```

==========



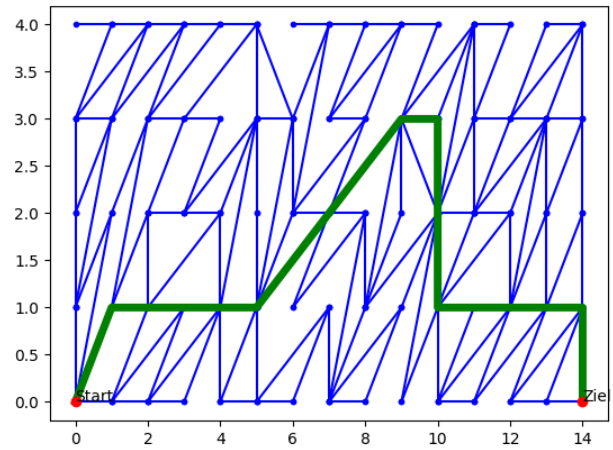

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 100% mit einer Verlängerung von 20.10101267766693%, 1 Richtun
gswechsel/n und der Streckenlänge 7.0 geht über die Punkte: [[0.0, 0.0]
, [0.0, 1.0], [0.0, 2.0], [0.0, 3.0], [1.0, 3.0], [2.0, 3.0], [3.0, 3.0
], [4.0, 3.0]]
Laufzeit: 0.00199127197265625 Sekunden
```

4.3 Beispiel 2



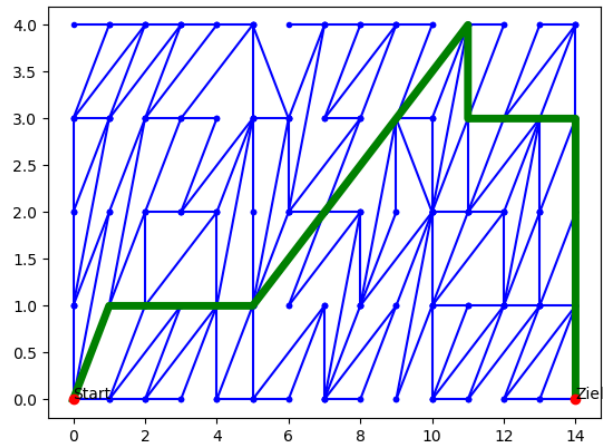

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 0% mit einer Verlängerung von 0.0%, 8 Richtungswechsel/n und
der Streckenlänge 17.122417494872465 geht über die Punkte: [[0.0, 0.0],
 [1.0, 0.0], [3.0, 1.0], [4.0, 1.0], [5.0, 1.0], [7.0, 2.0], [9.0, 3.0]
, [10.0, 2.0], [11.0, 2.0], [12.0, 2.0], [12.0, 1.0], [13.0, 1.0], [14.
0, 1.0], [14.0, 0.0]]
Laufzeit: 0.10791587829589844 Sekunden
```

==========



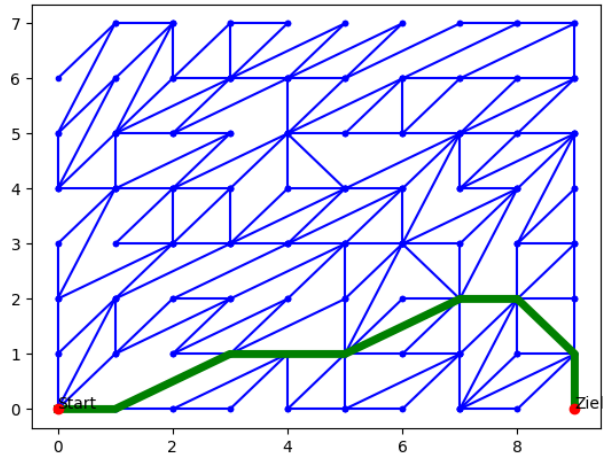

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 5% mit einer Verlängerung von 4.461589741804744%, 6 Richtungs
wechsel/n und der Streckenlänge 17.886349517372675 geht über die Punkte
: [[0.0, 0.0], [1.0, 1.0], [2.0, 1.0], [3.0, 1.0], [4.0, 1.0], [5.0, 1.
0], [7.0, 2.0], [9.0, 3.0], [10.0, 3.0], [10.0, 2.0], [10.0, 1.0], [11.
0, 1.0], [12.0, 1.0], [13.0, 1.0], [14.0, 1.0], [14.0, 0.0]]
Laufzeit: 0.10784578323364258 Sekunden
```

==========



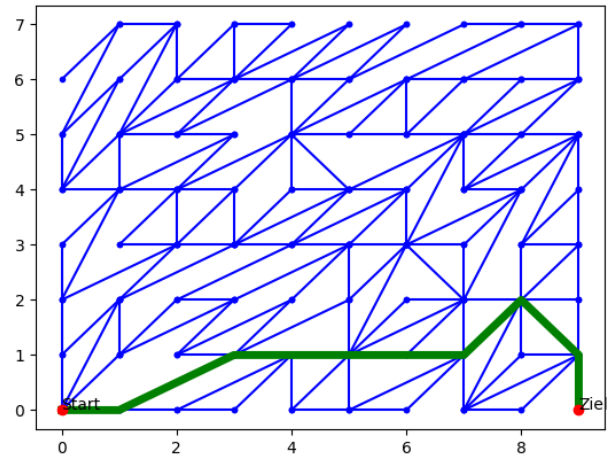

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 100% mit einer Verlängerung von 11.680593587902678%, 5 Richtu
ngswechsel/n und der Streckenlänge 19.122417494872465 geht über die Pun
kte: [[0.0, 0.0], [1.0, 1.0], [2.0, 1.0], [3.0, 1.0], [4.0, 1.0], [5.0,
 1.0], [7.0, 2.0], [9.0, 3.0], [11.0, 4.0], [11.0, 3.0], [12.0, 3.0], [
13.0, 3.0], [14.0, 3.0], [14.0, 2.0], [14.0, 1.0], [14.0, 0.0]]
Laufzeit: 0.10869193077087402 Sekunden
```

4.4 Beispiel 3



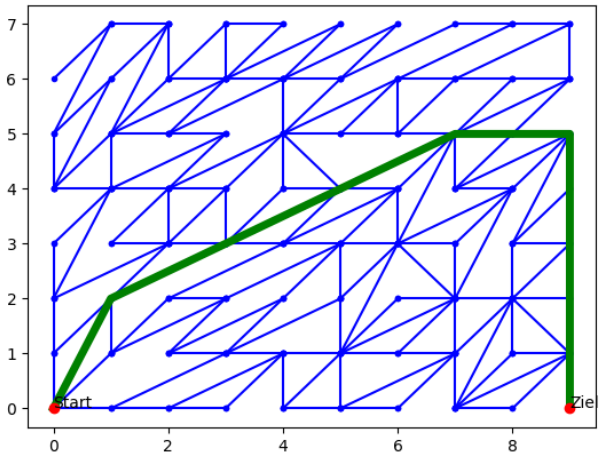

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 0% mit einer Verlängerung von 0.0%, 6 Richtungswechsel/n und
der Streckenlänge 10.886349517372675 geht über die Punkte: [[0.0, 0.0],
 [1.0, 0.0], [3.0, 1.0], [4.0, 1.0], [5.0, 1.0], [7.0, 2.0], [8.0, 2.0]
, [9.0, 1.0], [9.0, 0.0]]
Laufzeit: 0.060756921768188 48 Sekunden
```

==========



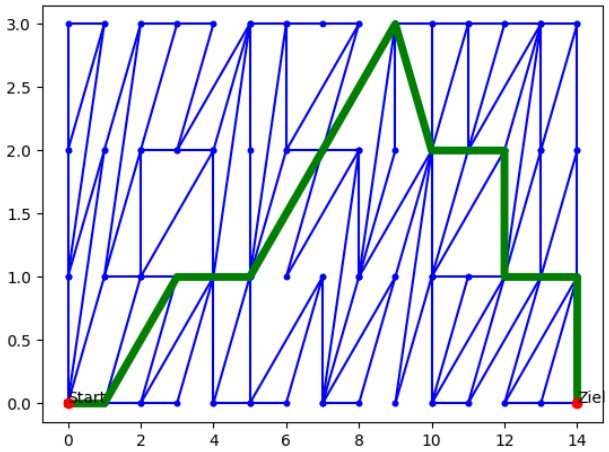

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 5% mit einer Verlängerung von 1.636412505303241%, 5 Richtungs
wechsel/n und der Streckenlänge 11.064495102245981 geht über die Punkte
: [[0.0, 0.0], [1.0, 0.0], [3.0, 1.0], [4.0, 1.0], [5.0, 1.0], [6.0, 1.
0], [7.0, 1.0], [8.0, 2.0], [9.0, 1.0], [9.0, 0.0]]
Laufzeit: 0.061337232589721 68 Sekunden
```

==========



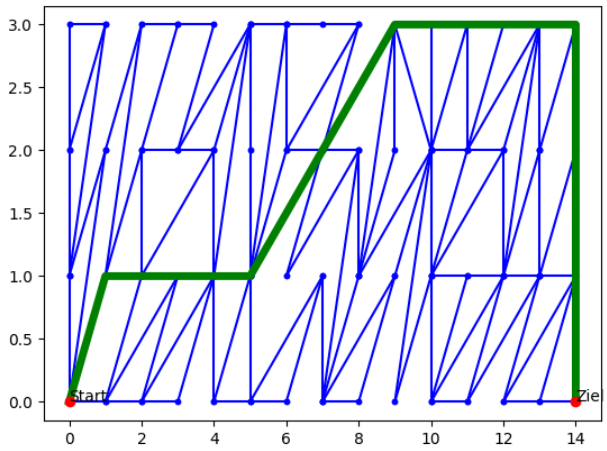

```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 100% mit einer Verlängerung von 46.46114277843955%, 3 Richtun
gswechsel/n und der Streckenlänge 15.94427190999916 geht über die Punkt
e: [[0.0, 0.0], [1.0, 2.0], [3.0, 3.0], [5.0, 4.0], [7.0, 5.0], [8.0, 5
.0], [9.0, 5.0], [9.0, 4.0], [9.0, 3.0], [9.0, 2.0], [9.0, 1.0], [9.0,
0.0]]
Laufzeit: 0.06116914749145508 Sekunden
```

4.5 Beispiel 4




```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 0% mit einer Verlängerung von 0.0%, 8 Richtungswechsel/n und
der Streckenlänge 17.122417494872465 geht über die Punkte: [[0.0, 0.0],
 [1.0, 0.0], [3.0, 1.0], [4.0, 1.0], [5.0, 1.0], [7.0, 2.0], [9.0, 3.0]
, [10.0, 2.0], [11.0, 2.0], [12.0, 2.0], [12.0, 1.0], [13.0, 1.0], [14.
0, 1.0], [14.0, 0.0]]
Laufzeit: 0.056160688400268555 Sekunden
```

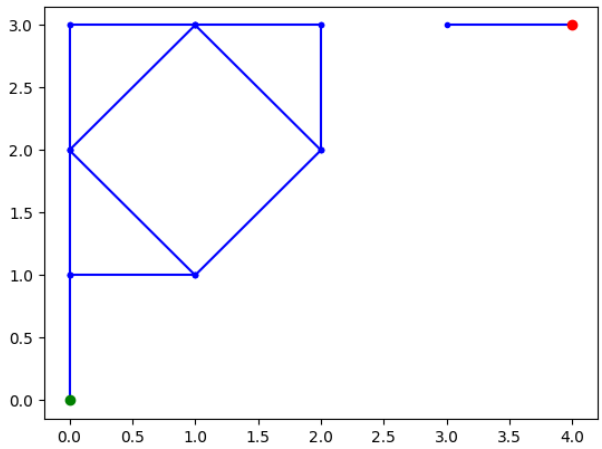
==========




```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 100% mit einer Verlängerung von 4.461589741804744%, 4 Richtun
gswechsel/n und der Streckenlänge 17.886349517372675 geht über die Punk
te: [[0.0, 0.0], [1.0, 1.0], [2.0, 1.0], [3.0, 1.0], [4.0, 1.0], [5.0,
1.0], [7.0, 2.0], [9.0, 3.0], [10.0, 3.0], [11.0, 3.0], [12.0, 3.0], [1
3.0, 3.0], [14.0, 3.0], [14.0, 2.0], [14.0, 1.0], [14.0, 0.0]]
Laufzeit: 0.060801982879638 67 Sekunden
```

4.6 Sonderfall 1

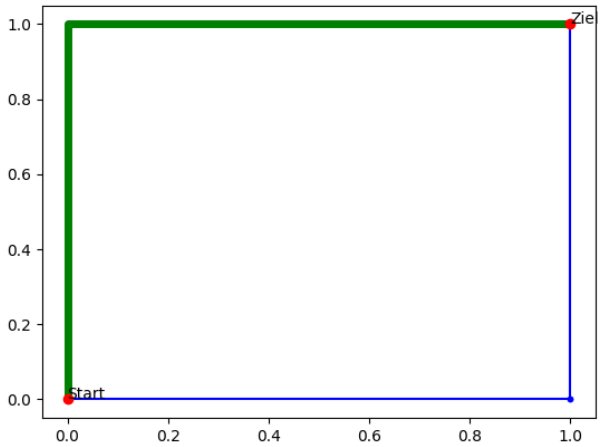
In diesem Fall wird ein Straßennetz behandelt, das zu keinem Ergebnis führt.



```
Keine moegliche Strecke gefunden
```

4.7 Sonderfall 2

In diesem Fall wird ein Straßennetz behandelt, das zwei Wege hat, die genau die gleiche Streckenlänge und Richtungswechselanzahl hat. Das Programm gibt einen der Wege aus.




```
Der Weg mit den wenigsten Richtungswechseln bei einer maximalen Verläng
erung von 100% mit einer Verlängerung von 0.0%, 1 Richtungswechsel/n un
d der Streckenlänge 2.0 geht über die Punkte: [[0.0, 0.0], [0.0, 1.0],
[1.0, 1.0]]
Laufzeit: 0.0004088878631591797 Sekunden
```

5 Quelltext

Hier ist einerseits eine Variablenliste abgedruckt, sowie der Programmcode.

5.1 Variablenliste

	Variablenname	Variablentyp	Variablenbeschreibung
Globale Variablen:			
	pfad	string	Pfad der einzulesenden Datei
	anzahl_str	integer	Anzahl der Strecken
	start	list	x- und y-Koordinate des Startpunkts
	ziel	list	x- und y-Koordinate des Ziels
	liste_koords	list	beinhaltet alle Streckenkoordinaten
	alle_knoten	list	beinhaltet die Koordinaten aller Knoten
	knotenliste	dictionary	beinhaltet alle Knoten mit ihren möglichen Nachfolgeknoten
	kurv_und_dist	dictionary	beinhaltet alle Knoten mit ihren niedrigsten Kurvenanzahlen und Distanzen
	knoten_davor	list	Koordinaten des vorherigen Knoten
	moeglichkeiten	list	beinhaltet alle vollendete Wege
	niedrigste_kurv_und_dist	list	beinhaltet die niedrigste Kurvenanzahl und niedrigste Distanz, die im Ziel erreicht wurde
	beste_strecke	list	Koordinaten sowie Attribute des besten Weges
	max_abweichung	integer	die vom Benutzer eingegebene maximale Verlängerung in Prozent
Lokale Variablen:			
berechnungen()	ber_knoten	list	Koordinaten des aktuellen Knoten
	y_alt	float	y-Koordinate des vorherigen Knoten
	x_neu	float	x-Koordinate des aktuellen Knoten
	y_neu	float	y-Koordinate des aktuellen Knoten
	diff_x	float	Differenz zwischen x_neu und x_alt
	diff_y	float	Differenz zwischen y_neu und y_alt
	dist	float	Distanz zwischen vorherigem und aktuellen Knoten
	winkel	float	Winkel der Strecke zwischen vorherigem und aktuellen Knoten
	winkel_davor	float	Winkel der vorherigen Strecke
	kurven	integer	Anzahl der bereits erfolgten Richtungswechsel
	strecke	float	die bereits zurückgelegte Distanz vom Startpunkt bis zum aktuellen Knoten
bedingungen()	liste_davor	list	beinhaltet Attribute des vorherigen Knoten (wert_davor, kurven, strecke, winkel_davor)
	knoten	list	Koordinaten des aktuellen Knoten
	fliste	list	lokale Liste mit den Attributen aller vorherigen Knoten
	berechnetes	list	Rückgabewerte der Funktion berechnungen()
vergleich()	min_strecke	float	minimale Distanz aller möglichen Wege
	vergleiche	list	Kurvenanzahl, Distanz und Verlängerung in Prozent aller möglichen Wege
	prozent	float	Prozentzahl der Verlängerung jedes Weges
	unter_prozent	list	Kurvenanzahl, Distanz und Verlängerung in Prozent aller Wege, die unter der maximalen Verlängerung liegen mit Index
	min_kurv	integer	minimale Kurvenanzahl aller unter der maximalen Verlängerung liegenden Wege
	kleinste_str	float	niedrigste Distanz aller Wege die unter der maximalen Verlängerung liegen und die minimale Kurvenanzahl haben
	min_kurv_moegl	list	Kurvenanzahl, Distanz, Verlängerung in Prozent und Index des optimalsten Weges

Die lokalen Variablen der Funktionen ausgabe(), grafisch() und eingabe() werden da sie nicht zum Hauptprogramm gehören, nicht aufgeführt.

5.2 Programmcode

```python
#!/usr/bin/python
# -*- coding: utf-8 -*-
import math
import time
try:
    import matplotlib.pyplot as plt
    matplot = True
except ModuleNotFoundError:
    matplot = False

pfad = "abbiegen0.txt"

#Einlesen der Datei und Werte in bestimmtes Format umwandeln
with open(pfad) as datei:
    anzahl_str = int(datei.readline())
    start = datei.readline()[1:-2].split(",")
    start = [float(elem) for elem in start]
    ziel = datei.readline()[1:-2].split(",")
    ziel = [float(elem) for elem in ziel]
    liste_koords = datei.readlines()
for i in range(len(liste_koords)):
    liste_koords[i] = liste_koords[i][:-1].split(" ")
    liste_koords[i][0] = liste_koords[i][0][1:-1].split(",")
    liste_koords[i][0] = [float(elem) for elem in liste_koords[i][0]]
    liste_koords[i][1] = liste_koords[i][1][1:-1].split(",")
    liste_koords[i][1] = [float(elem) for elem in liste_koords[i][1]]
#alle Knoten werden in eine Liste geschrieben
alle_knoten = [list(x) for x in set(
            [tuple(e) for e in [*list(x[0] for x in liste_koords),
            *list(x[1] for x in liste_koords)]])]

#Dictionary in dem jeder Knoten mit seinen möglichen
#weiterführenden Knoten aufgelistet wird
#Zeit stoppen
zeit = time.time()
knotenliste = {}
kurv_und_dist = {}
for elem in alle_knoten:
    knotenliste[str(elem)] = []
    kurv_und_dist[str(elem)] = []
    for f in range(len(liste_koords)):
        if liste_koords[f][0] == elem:
            koord = liste_koords[f][1]
            knotenliste[str(elem)].append(koord)
        elif liste_koords[f][1] == elem:
            koord = liste_koords[f][0]
            knotenliste[str(elem)].append(koord)

def berechnungen(ber_knoten, knoten_davor, kurven, strecke, winkel_davor):
    x_alt, y_alt = knoten_davor[0], knoten_davor[1]
    x_neu, y_neu = ber_knoten[0], ber_knoten[1]
    diff_x, diff_y = x_neu - x_alt, y_neu - y_alt
    #Distanz zwischen Knoten davor und jetzigem Knoten errechnen
    dist = math.sqrt(diff_x**2 + diff_y**2)
    #Winkel der Strecke zwischen Knoten davor und jetzigem Knoten errechnen
    if diff_x == 0.0:
        winkel = 1.57
    elif diff_y == 0.0:
        winkel = 0.0
    else:
        winkel = math.atan(diff_y/diff_x)
```

```
62      if winkel_davor != None:
63          #wenn Winkel ungleich ist --> kurven um 1 erhöhen
64          if abs(winkel_davor - winkel) > 0.0001:
65              kurven += 1
66      #bereits zurückgelegte Strecke errechnen
67      strecke = strecke + dist
68      #Dictionary mit allen Knoten und ihrer niedrigsten
69      #Richtungswechselanzahl und Streckenlänge
70      if not kurv_und_dist[str(ber_knoten)]:
71          kurv_und_dist[str(ber_knoten)].append(kurven)
72          kurv_und_dist[str(ber_knoten)].append(strecke)
73          return [ber_knoten, kurven, strecke, winkel]
74      else:
75          #wenn der jetzige Punkt im Dictionary bereits einen Eintrag
76          #hat in dem Richtungswechselanzahl
77          #und Streckenlänge niedriger ist gibt die Funktion None zurück
78          if kurven >= kurv_und_dist[str(ber_knoten)][0]:
79              if strecke >= kurv_und_dist[str(ber_knoten)][1]:
80                  return None
81          #wenn Richtungswechselanzahl und Streckenlänge höher,
82          #durch neue Werte ersetzen und Attribute zurückgeben
83          else:
84              kurv_und_dist[str(ber_knoten)][0] = kurven
85              kurv_und_dist[str(ber_knoten)][1] = strecke
86              return [ber_knoten, kurven, strecke, winkel]

88  def bedingungen(knoten, fliste):
89      #wenn bei Ziel angekommen, return
90      if knoten == ziel:
91          liste_davor = fliste[-1]
92          #Strecke etc. wird berechnet
93          berechnetes = berechnungen(knoten, liste_davor[0],
94                                     liste_davor[1], liste_davor[2],
95                                     liste_davor[3])
96          #wenn None zurückgegeben, return
97          if berechnetes == None:
98              return
99          #niedrigste Anzahl von Richtungswechseln und Streckenlänge
100         #bei Ende der Strecke
101         global niedrigste_kurv_und_dist
102         if niedrigste_kurv_und_dist == []:
103             niedrigste_kurv_und_dist = [berechnetes[1],
104                                         berechnetes[2]]
105         elif berechnetes[1] <= niedrigste_kurv_und_dist[0]:
106             if berechnetes[2] <= niedrigste_kurv_und_dist[1]:
107                 niedrigste_kurv_und_dist = [berechnetes[1],
108                                             berechnetes[2]]
109
110         fliste.append(berechnetes)
111         #Weg zur globalen Liste moeglichkeiten hinzugefügen
112         moeglichkeiten.append(fliste)
113         return
```

```
    #wenn der Knoten bereits in der lokalen Liste fliste ist, return
    if knoten in [elem[0] for elem in fliste]:
        return
    try:
        #Versuch auf Eintrag davor in der lokalen Liste zuzugreifen
        liste_davor = fliste[-1]
        #Strecke etc. wird berechnet
        berechnetes = berechnungen(knoten, liste_davor[0], liste_davor[1],
                                   liste_davor[2], liste_davor[3])
        #wenn None zurückgegeben, return
        if berechnetes == None:
            return
        #wenn Richtungswechselanzahl und Streckenlänge des jetzigen Punktes
        #höher ist als diese Werte eines schon vollendeten Weges, return
        try:
            if berechnetes[1] >= niedrigste_kurv_und_dist[0]:
                if berechnetes[2] >= niedrigste_kurv_und_dist[1]:
                    return
        except IndexError:
            pass
        fliste.append(berechnetes)
    #wenn es keinen Eintrag davor in der lokalen Liste gibt, erster Eintrag
    #--> Kurven = 0, Strecke = 0, Winkel nicht definiert
    except IndexError:
        fliste.append([knoten, 0, 0, None])
    #Rekursion, zu allen moeglichen naechsten Knoten gehen
    for elem in knotenliste[str(knoten)]:
        bedingungen(elem, fliste[:])
    return

def vergleich():
    vergleiche = []
    #kleinste Gesamtstrecke suchen
    min_strecke = 0
    for i in range(len(moeglichkeiten)):
        if min_strecke:
            if moeglichkeiten[i][-1][2] < min_strecke:
                min_strecke = moeglichkeiten[i][-1][2]
        else:
            min_strecke = moeglichkeiten[i][-1][2]

        ende = [moeglichkeiten[i][-1][1], moeglichkeiten[i][-1][2]]
        vergleiche.append(ende)
    #Prozentzahl der Verlängerung des Weges für jede Strecke berechnen
    for j in range(len(vergleiche)):
        prozent = vergleiche[j][1]/min_strecke*100-100
        vergleiche[j].append(prozent)
    unter_prozent = []
    #alle Wege die unter der maximalen Verlängerung liegen
    #in Liste mit Index
    for k in range(len(vergleiche)):
        if vergleiche[k][2] <= max_abweichung:
            unter_prozent.append([vergleiche[k], k])
    #minimale Richtungswechselanzahl bei allen moeglichen
    #unter der maximalen Abweichung suchen
    min_kurv = min([unter_prozent[k][0][0] for k in range(len(unter_prozent))])
```

```
    #die Wege mit der minimalen Richtungswechselanzahl extrahieren und
    #den Weg mit der niedrigsten Streckenlänge bestimmen
    kleinste_str = 0
    for elem in unter_prozent:
        if elem[0][0] == min_kurv:
            if kleinste_str:
                if elem[0][1] < kleinste_str:
                    kleinste_str = elem[0][1]
                    min_kurv_moegl = elem
            else:
                kleinste_str = elem[0][1]
                min_kurv_moegl = elem
    #Koordinaten sowie Richtungswechselanzahl, Streckenlänge und
    #Verlängerung des "besten" Weges zurückgeben
    return [moeglichkeiten[min_kurv_moegl[1]], min_kurv_moegl[0][0],
            min_kurv_moegl[0][1], min_kurv_moegl[0][2]]

def ausgabe(weg, kurven, strecke, verlaengerung, laufzeit):
    #alle Koordinaten in richtiger Reihenfolge in Liste
    k_liste = []
    for elem in weg:
        k_liste.append(elem[0])
    print("Der Weg mit den wenigsten Richtungswechseln bei einer " +
        "maximalen Verlängerung von " + str(max_abweichung) +
        "% mit einer Verlängerung von " + str(verlaengerung) +
        "%, " + str(kurven) + " Richtungswechsel/n und der Streckenlänge "+
        str(strecke) + " geht über die Punkte: " + str(k_liste))
    print("Laufzeit: " + str(laufzeit) + " Sekunden")
    #wenn matplotlib installiert ist, grafische Ausgabe anbieten
    if matplot == False:
        print("Grafische Ausgabe nicht möglich, da das Modul matplotlib"+
            "nicht installiert ist")
    else:
        grafische = input("Grafische Ausgabe?[J/n]")
        if grafische == "J" or grafische == "j":
            grafisch(beste_strecke[0])

def grafisch(weg):
    #Grundnetzstruktur plotten
    for i in range(len(liste_koords)):
        plt.plot([liste_koords[i][0][0], liste_koords[i][1][0]],
                 [liste_koords[i][0][1], liste_koords[i][1][1]],
                 marker=".", color="blue")
    #die Strecke des "besten" Weges plotten
    x = []
    y = []
    for elem in weg:
        x.append(elem[0][0])
        y.append(elem[0][1])
    plt.plot(x,y, color="green", linewidth=5)
    #Start und Ziel plotten
    plt.plot(start[0], start[1], "o", color="red")
    plt.text(start[0], start[1], "Start")
    plt.plot(ziel[0], ziel[1], "o", color="red")
    plt.text(ziel[0], ziel[1], "Ziel")
    #Grafik aufrufen
    plt.show()
```

```
230 def eingabe():
231     while True:
232         try:
233             eingabe = int(input("Maximale Abweichung in Prozent bitte " +
234                 "als Integer eingeben (für den schnellsten Weg 0 eingeben)"))
235             return eingabe
236             break
237         except ValueError:
238             print("Bitte geben Sie einen Integer ein")
239 def main():
240     global moeglichkeiten
241     moeglichkeiten = []
242     global niedrigste_kurv_und_dist
243     niedrigste_kurv_und_dist = []
244     #Aufrufen der Funktionen
245     bedingungen(start, [])
246     #wenn Moeglichkeiten gefunden wurden, beste Strecke suchen und ausgeben
247     if moeglichkeiten != []:
248         global zeit
249         zeit = time.time() - zeit
250         global max_abweichung
251         max_abweichung = eingabe()
252         zeit2 = time.time()
253         global beste_strecke
254         beste_strecke = vergleich()
255         zeit += time.time()-zeit2
256         ausgabe(beste_strecke[0], beste_strecke[1], beste_strecke[2],
257                 beste_strecke[3], zeit)
258     else:
259         print("Keine moegliche Strecke gefunden")
260 main()
```