

Aufgabe 3: Abbiegen

Teilnahme-Id: 52586

Bearbeiter dieser Aufgabe:
Michal Boron

April 2020

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Definitionen	1
1.2	Graph G'	2
1.3	Yen-Algorithmus	4
1.4	Laufzeit	5
1.5	Erweiterungen	8
1.5.1	Start und Ziel an verschiedenen Stellen	8
1.5.2	Nichtganzzahlige Koordinaten	8
1.5.3	Kreuzungen von Straßen	8
1.5.4	Genauigkeit	8
2	Umsetzung	9
2.1	Die Klasse <code>Graph</code>	9
2.2	Die Klasse <code>Dijkstra</code>	10
2.3	Die Klasse <code>Yen</code>	11
3	Beispiele	13
3.1	Beispiel 1 (BWINF)	13
3.2	Beispiel 2 (BWINF)	14
3.3	Beispiel 3 (BWINF)	16
3.4	Beispiel 4	18
3.5	Beispiel 5	18
3.6	Beispiel 6	19
3.7	Beispiel 7	19
3.8	Beispiel 8	20
3.9	Beispiel 2 (erweitert)	21
4	Quellcode	23
5	Pseudocode aus Wikipedia	27

1 Lösungsidee

1.1 Definitionen

Gegeben seien eine zweidimensionale Ebene, eine Menge V von Punkten auf dieser Ebene, eine Menge E von Straßen zwischen den Punkten, ein Startpunkt s und ein Zielpunkt z .

Jeder *Punkt* $p(x, y)$ besitzt zwei ganzzahlige Koordinaten x und y , die die Lage des Punktes auf der Ebene bestimmen. Eine *Straße* $b(v_1, v_2)$ besteht aus einem Paar von zwei Punkten v_1 und v_2 aus der

Menge V . Als eine *Länge* $l(b)$ (auch *Entfernung*) einer Straße $b(v, w)$ bezeichnet man den euklidischen Abstand zwischen zwei Punkten v und w .

Unter einer *Abbiegung* versteht man eine Situation, in der drei Punkte g, h und i aus der Menge V zwei verschiedene Straßen $b(g, h)$ und $c(g, i)$ aus der Menge E bilden. Eine solche Abbiegung bezeichnen wir als *eine Abbiegung von b zu c* oder als *eine Abbiegung von c zu b* . Darüber hinaus besitzt eine Abbiegung einen Wert, der dem Zustand entspricht, ob die Punkte g, h, i kollinear sind (0 für kollinear, 1 für nicht kollinear). Eine Abbiegung mit Wert 1, nenne ich auch *1-Abbiegung*. Eine Abbiegung mit Wert 0 nenne ich *0-Abbiegung*.

Mit Hilfe der obengenannten Mengen V und E lässt sich ein ungerichteter Graph $G(V, E)$ mit gewichteten Kanten bilden. Jede Kante k ist stets mit zwei Knoten – p, q – verbunden. Als das Gewicht einer Kante (p, q) gilt die Länge $l(b)$ der Straße $b(p, q)$. Kennzeichnen wir noch jeweils einen Start- und Zielknoten, die s und z entsprechen, so erhalten wir einen Graphen, in dem

Beobachtung 1 *die Länge des Weges im Graphen G vom Startknoten zum Zielknoten gleich der Summe aller Längen der Straßen auf dem Weg ist.*

Dank dieser Beobachtung können wir feststellen, dass wir den kürzesten Weg vom Startknoten zum Zielknoten finden können, wenn wir den Dijkstra-Algorithmus laufen lassen. Dieser Wert – die kürzeste gesamte Länge vom Startknoten zum Zielknoten – wird dementsprechend zu einem Maßstab, einem Prototypen, der uns später in unseren weiteren Betrachtungen helfen wird. Die Länge des kürzesten Weges nennen wir ω .¹

Wir fügen noch zwei Begriffe hinzu. Wir betrachten eine Straße $b(p, q)$. Die anderen Straßen, die auch mit den Punkten p und q verbunden sind, nennen wir *die benachbarten Straßen* der Straße b . Die Punkte, die die benachbarten Straßen zusammen mit p bilden, nennen wir dementsprechend *die benachbarten Punkte* des Punkts p (siehe Abb. 1).

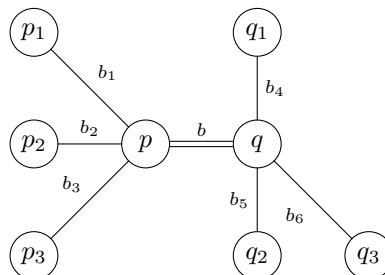


Abbildung 1: Benachbarte Straßen: alle Straßen b_1, b_2, \dots, b_6 nennen wir *die benachbarten Straßen* der Straße b . Benachbarte Punkte: als *die benachbarten Punkte* des Punkts p gelten p_1, p_2, p_3, q . Die benachbarten Punkte von q sind: q_1, q_2, q_3, p .

Beobachtung 2 *Außerdem findet man immer in den Beispielen so einen Weg zum Ziel, der stets von einer Straße $a(p_x, p_y), q(q_x, q_y)$ zu einer andern Straße $b(q(q_x, q_y), r(r_x, r_y))$ führt, wobei stets $p_x \leq r_x$.*

Die letzte Bemerkung ist sehr wichtig und gilt demzufolge als eine führende Beobachtung in meinen weiteren Betrachtungen.

1.2 Graph G'

Nun legen wir eine neue Menge fest: C . Wir nehmen jeweils eine Straße aus der Menge E , bestimmen ihre benachbarten Straßen und fügen diejenigen in C ein, die die Bedingung in der Beobachtung 2 erfüllen. Jede Abbiegung besteht *de facto* aus drei Punkten. Einer der Punkte ist gemeinsam für beide Straßen, aus denen eine Abbiegung entstanden ist. Jetzt vergleichen wir die x -Koordinaten der zwei übrigen Punkte. Angenommen haben wir eine Abbiegung von einer Straße $a(p, q)$ zu einer anderen Straße $b(q, r)$. Wenn

¹Um möglichst verständlich zu formulieren, benutzen wir künftig für «Pfad in G » stets den Begriff «Weg in G ». Im Graphen G' (s. Abschnitt 1.2) bleiben wir – zur besseren Unterscheidung – bei «Pfad in G' ».

die x -Koordinate des Punkts p nicht kleiner ist als die x -Koordinate des Punkts r , dann fügen wir eine Abbiegung von b zu a mit einem entsprechenden Wert ein. Wenn aber die x -Koordinate des Punkts p nicht größer ist als die x -Koordinate des Punkts r , dann fügen wir eine Abbiegung von a zu b mit einem entsprechenden Wert ein. Einfacher formuliert, nehmen wir keine Straßen, die nach „hinten“ leiten. (s. Abb. 2 und Abb. 3)

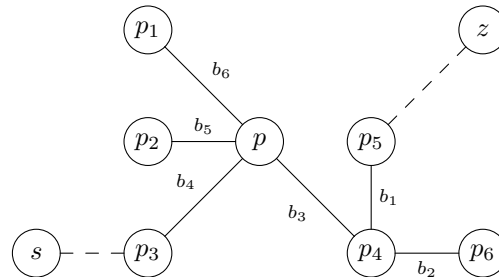


Abbildung 2: Graph G : Von der Straße b_3 kann man nicht zu den Straßen: b_4, b_5, b_6 abbiegen, aber zu den Straßen b_1 und b_2 ist es erlaubt. s und z dienen hier nur für die Orientierung, in welche Richtung die Traversierung verläuft.

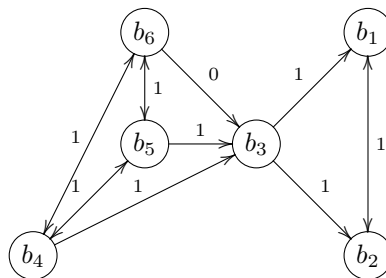


Abbildung 3: Graph G' , der anhand der Abb. 2 entstanden ist, mit Abbiegungen als Kanten und Straßen als Knoten.

Nun bilden wir einen gewichteten, gerichteten Graphen $G'(E, C)$, in dem die Straßen als Knoten und die Abbiegungen als Kanten verwendet werden. Die Kanten sind gerichtet aus einem wichtigen Grund: um die Endlosschleifen zu vermeiden. Die Gewichte an den Kanten entsprechen dem Zustand, ob wir in jeweiliger Situation von einer Straße $a(p, q)$ zu einer Straße $b(q, r)$ abbiegen müssen oder nicht. Dieser boolsche Wert wird dadurch bestimmt, dass wir prüfen, ob alle Punkte p, q, r kollinear sind, das heißt, auf einer Geraden liegen.

Definieren wir dazu zwei neue Punkte, die wir *Pivotpunkte* nennen. Die Lage der Punkte ist beliebig und über meine Platzierung der beiden Punkte lesen Sie weiter im Abschnitt „Umsetzung“.

Den *Pivotpunkt des Startpunkts* verbinde ich mit dem Startpunkt durch eine Straße. Das gleiche gilt für den Zielpunkt: ich verbinde ihn mit dem *Pivotpunkt des Zielpunkts*. Die beiden Straßen füge ich in den Graphen G' ein und verbinde sie durch Kanten stets mit Gewicht 0 (diese Straßen sind ja kein Teil der ursprünglichen Aufgabe) mit jeder Straße, die aus dem Startpunkt oder zum Zielpunkt führt. Die Straßen, die vom Pivotpunkt zum Startpunkt oder vom Zielpunkt zum Pivotpunkt führen, nennen wir *Pivotstraßen*.

Die Erstellung der Pivotstraßen erleichtert die Aufgabe insofern, dass wir jedes Mal die Start- und Zielknoten in G' nicht zusätzlich bestimmen.

Nun können wir auf eine wichtige Bemerkung kommen.

Beobachtung 3 Die Länge eines Pfades P in G' vom Startknoten (Start-Pivotstraße) zum Zielknoten (Ziel-Pivotstraße), also die Summe S der Kantengewichte von P , entspricht der Anzahl der 1-Abbiegungen in P . Die Zahl S bedeutet auch, dass wir im entsprechenden Weg W von s zu z S -mal abbiegen müssen.

Daraus kann man auch eine andere Schlussfolgerung ziehen:

Beobachtung 4 Die Länge des kürzesten Pfades in G' vom Startknoten (Start-Pivotstraße) zum Zielknoten (Ziel-Pivotstraße) entspricht der minimalen Anzahl der 1-Abbiegungen vom Startknoten zum Zielknoten im Graphen G' .

1.3 Yen-Algorithmus

Direkt aus der Beobachtung 4 ergibt sich die Lösung der Aufgabe. Nach der Aufgabenstellung müssen wir den besten Weg in G finden, bei dem wir am wenigsten abbiegen und dessen Länge sich in Rahmen einer maximalen prozentualen Verlängerung von ω befindet. Diese maximale Länge nennen wir ψ . Den Prozentwert, der eingegeben wird, nennen wir ξ .

Um dem Problem zu begegnen, bediene ich mich des Yen-Algorithmus, der die k -kürzesten Pfade findet. Seine Beschreibung und ein vollständiger Pseudocode sind im englischsprachigen Wikipedia-Artikel² (s. unten, Seite 27) zu finden. Genau dieses Pseudocodes bediente ich mich bei der Implementierung der Aufgabe. Ich lasse den Algorithmus auf dem Graph G' laufen, weil, wie wir schon feststellten, die Länge eines Pfades vom Startknoten zum Zielknoten in G' der Anzahl der Abbiegungen des entsprechenden Pfades in G gleich ist. Ich erkläre im nächsten Abschnitt, wie der Algorithmus funktioniert, aber es ist sehr empfehlenswert, bevor man weiterliest, mindestens einen kurzen Blick auf den Pseudocode zu werfen, um eine Idee zu bekommen, was der Algorithmus macht.

Ich veränderte und erweiterte aber die Implementierung im Pseudocode an einigen Stellen. Vor allem baute ich dieses im Pseudocode dargestellte Programm zu einer Funktion um, die einen nächsten kürzesten Pfad generiert. Sie hilft mir, weil ich dadurch keine feste Zahl k angeben muss. Ich generiere stets einen neuen Pfad nach Bedarf (mehr dazu im folgenden Abschnitt).

Ganz am Anfang lasse ich den Dijkstra-Algorithmus auf dem Graphen G' laufen, um den Pfad mit der niedrigsten Anzahl an Abbiegungen zu finden (vergl. Pseudocode auf Seite 27, Z. 3). Ich speichere gleichzeitig auch den *spurNode* (vergl. Z. 13) dieses Pfades, heißt, den Startknoten in G' . Der *spurNode* ist ein Knoten aus einem vorherigen Pfad (Z. 13), an dem wir die Kanten im Graphen G' entfernen (Z. 18–22). Diesen besten Pfad füge ich nicht zur Liste A , wie das im Pseudocode steht, sondern zu B . Liste A enthält alle fertigen k -kürzesten Pfade und Liste B enthält die Kandidatenpfade. Die letztere wird nach jeder neuen Generierung eines Pfades sortiert, sodass das erste Element stets einem nächsten kürzesten Pfad entspricht (vergl. Z. 51).

Danach beginnt meine Funktion, die einen nächsten kürzesten Pfad generiert. Ich nehme den ersten Pfad aus der Kandidatenliste B . In dieser Betrachtung bezeichne ich ihn als *den aktuellen Pfad*.

Danach füge ich diesen Pfad in die Liste A ein (vergl. Z. 53). Ich entferne natürlich auch den aktuellen Pfad in B .

Ich rufe den gespeicherten *spurNode* dieses Pfades ab (vergl. Z. 13). Ich bilde einen *rootPath* vom Startknoten zum *spurNode*, wie im Pseudocode (vergl. Z. 16). Ein *rootPath* ist demzufolge ein Teilpfad des aktuellen Pfades vom Startknoten zum *spurNode*.

Danach iteriere ich durch alle Pfade in A und vergleiche, ob der Anfang des jeweiligen Pfades mit *rootPath* übereinstimmt. Wenn ja, dann entferne ich im Graphen G' die nächste Kante nach dem Teilpfad, der mit *rootPath* übereinstimmt (wie im Pseudocode, vergl. Z. 18–22). Man kann auch bemerken, dass wir diejenige Kante entfernen, die zwischen dem *spurNode* und dem nächsten Knoten im iterierten Pfad liegt.

Danach entferne ich im Graphen G' alle Knoten und Kanten, die im aktuellen Pfad auftreten, wie im Pseudocode (vergl. Z. 24–25).

Im Pseudocode, in der Zeile 28, wird es nicht genauer bestimmt, wie man den *spurPath* bzw. den *totalPath* bestimmt. Ich tue das auf folgender Weise. Bei jeder Generierung eines Pfades (Z. 8) wird ein nächster *spurNode* bestimmt (Z. 13). Die Reihenfolge, in der dies bestimmt wird, ist vom Anfang des vorherigen Pfades zum vorletzten Knoten in diesem Pfad (Z. 8). Aus diesem Grund entschied ich mich in meiner Implementierung dafür, dass ich beginne, neue Verbindungen (nach dem Entfernen) im Graphen G' zur Bestimmung des nächsten kürzesten Pfades (Z. 28) vom Zielknoten zu finden.

Nachdem entsprechende Kanten und Knoten entfernt worden sind (Z. 18–25), lasse ich den Dijkstra-Algorithmus vom Zielknoten laufen. Dieser Algorithmus liefert aber keinen neuen Pfad. Es werden nur die Verbindungen zwischen den Knoten im Graphen G' aktualisiert. Es entsteht *de facto* ein Baum mit einer Wurzel als Zielknoten (=Ziel-Pivotstraße).

Danach iteriere ich durch den aktuellen Pfad vom vorletzten Knoten (dem Knoten vor dem Zielknoten) zum *spurNode*. Jeden iterierten Knoten nenne ich hier n_i .

²https://en.wikipedia.org/wiki/Yen%27s_algorithm#Pseudocode (Zugang 15.03.2020)

Am Anfang stelle ich den Knoten n_i in den Graphen zurück. Da der Graph G' gerichtet ist, nehme ich alle Kanten, die aus n_i führen (Richtung Wurzel zeigen) und prüfe, ob ein Gewicht an einer Kante zwischen n_i und dem anderen Knoten kleiner ist als das jetzige zwischen einem anderen Knoten. Es kann auch sein, dass eine Kante vom vorherigen Pfad entfernt wurde und dann ist der Knoten n_i noch nicht erreicht. In beiden Fällen wird an dieser Stelle die beste neue Kante gewählt. Ich nenne einen solchen Vorgang eine *Aktualisierung*. Wenn ein Knoten, der aus n_i führt, aktualisiert wurde, dann forme ich einen neuen *spurPath* (vergl. Z. 28), weil ich weiß, dass ein neuer Pfad entstand.

Wenn ich einen neuen Pfad fand, aktualisiere ich die Gewichte für die Knoten, die zu dem Knoten n_i führen, also diejenigen, die sich eine Stufe niedriger im Baum befinden als n_i . Diese Aktualisierung hilft uns schon beim Knoten n_{i+1} .

Gleich danach folgt noch eine Iteration vom aktuellen Pfad. Dies erfolgt bis zum Knoten n_i . Hier werden die Gewichte an den Kanten im aktuellen Pfad bis zu n_i aufaddiert. Dieser Wert wird gleich danach zur Länge des neuen *spurPath* addiert und wird zur Länge des neuen *totalPath*.

Danach formen wir aus dem Teilpfad des aktuellen Pfades vom Startpunkt zum n_i und dem *spurPath* den *totalPath* (vergl. Z. 31). Anschließend wird überprüft, ob ein solcher neu entstandener *totalPath* sich schon in der Kandidatenliste befindet. Wenn nicht, dann wird er eingefügt (vergl. 33–34). Am Ende erfolgt die Zurückstellung der Kante zwischen n_i und dem nächsten Knoten im aktuellen Pfad in den Graphen G' .

Anhand des Yen-Algorithmus generieren wir die k -kürzesten Pfade P_1, P_2, \dots, P_k in G' von der Start-Pivotstraße zur Ziel-Pivotstraße. Die Länge eines solchen Pfades ist die Summe seiner Kantengewichte (also der Nullen und Einsen). Zu jedem $P_i, i = 1 \dots k$, in G' betrachten wir den zugehörigen Weg W_i im Graphen G , der aus allen Straßen besteht, die die Knoten des Pfades P_i bilden. Dem Weg W_i in G ordnen wir seine (euklidische) Länge γ , d.h. die Summe der Längen aller Straßen von W_i , zu.

Dabei wird noch ein anderer, wichtiger Aspekt beachtet. Da die Knoten im Graphen G' Straßen sind, bedeutet das, dass es sein kann, dass ein Punkt auf der Ebene mehr als dreimal besucht wird. Eine Straße besitzt zwei Punkte. Im Dijkstra-Algorithmus in G' wird nicht beachtet, ob ein Punkt, sondern ob ein Knoten bereits besucht wurde. Das bedeutet, dass es Traversierungen geben kann, in denen ein Punkt dreimal besucht wurde (s. Abb. 4). Aus diesem Grund werden alle solchen Pfade an dieser Stelle aus unserer Betrachtung ausgeschlossen. Im Programm werden sie einfach übersprungen.

Wir vergleichen γ mit ψ . Wenn γ größer ist als ψ , dann müssen wir einen neuen Pfad generieren und erneut vergleichen. Wenn aber γ nicht größer ist als ψ , fanden wir den besten Pfad, da diese Pfade, die ich generiere, die besten k -kürzesten Pfade sind, das heißt Pfade mit der niedrigsten Anzahl von Abbiegungen. Der beste Pfad ist dementsprechend unser Ergebnis.

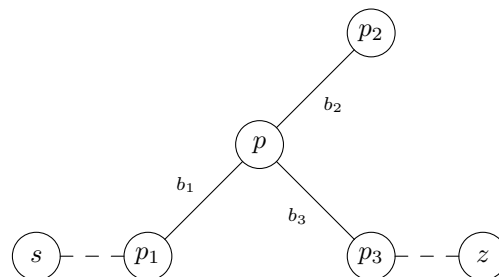


Abbildung 4: p_2 und p_3 besitzen dieselben x -Koordinaten. Im Graphen G' existieren folgende Abbiegungen: von b_1 zu b_2 (mit Wert 0) und von b_1 zu b_3 (mit Wert 1), aber auch von b_2 zu b_3 und andersherum (jeweils mit Wert 1). Das bedeutet, dass es eine Traversierung geben kann, die von b_1 zu b_2 und danach von b_2 zu b_3 verläuft. Dementsprechend werden alle Straßen b_1 , b_2 und b_3 besucht. Aus dem Grund wird auch der Punkt p dreimal besucht.

Über die Idee, Korrektheit und Laufzeit des Yen-Algorithmus kann man mehr in den zitierten Artikeln lesen: Yen 1970 [1], Yen 1971 [2].

1.4 Laufzeit

Die Laufzeit können wir grundsätzlich in drei Phasen aufteilen: Bildung des Graphen G , Bildung des Graphen G' und Generierung der Pfade.

s – Anzahl der Straßen

p – Anzahl der Punkte

k – Anzahl der generierten Pfade

Graph G :

- für das Einlesen der Eingabe: $O(s)$. In der Datei gibt es s Straßen, die eingelesen werden müssen.
- Vorbereitung der Mengen V und E : $O(p + s)$. Jedem Punkt und jeder Straße werden Indizes zugeordnet: $O(p + s)$. Jedem Punkt werden seine Nachbarn zugeordnet: $O(s)$.
- Bildung des Graphen G : $O(s)$. Alle Kanten (Straßen) werden eingefügt.
- Dijkstra auf dem Graphen $G - \omega$ wird bestimmt: $O(p^2)$ (worst-case).

$$O(s) + O(p + s) + O(s) + O(p^2) = O(p^2 + 3s + p) \in O(p^2 + p + s) \in O(p^2 + p), s \text{ wird weggelassen, denn } s < p.$$

Graph G' :

- für das Einlesen der Eingabe: $O(s)$. In der Datei gibt es s Straßen, die eingelesen werden müssen.
- Vorbereitung der Mengen V und E : $O(p + s)$. Jedem Punkt und jeder Straße werden Indizes zugeordnet: $O(p + s)$. Jedem Punkt werden seine Nachbarn zugeordnet: $O(s)$. Die Pivotpunkte werden in $O(1)$ erstellt.
- Vorbereitung der Menge C : $O(s)$. Die Menge mit allen Straßen wird iteriert ($O(s)$) und bei jeder Straße werden noch die Mengen von den benachbarten Punkten jedes Punkts dieser Straße iteriert. Jede Straße hat in der Regel nicht mehr als 10 Abbiegungen (nach den BWINF-Beispielen). Am Ende bekommen wir eine Liste mit Mengen, in denen sich die benachbarten Straßen befinden, also die Mengen mit Abbiegungen: $O(10 * s)$.
- Bildung des Graphen G' : $O(s)$. Alle Kanten (Abbiegungen) werden eingefügt.

$$O(s) + O(p + s) + O(s) + O(s) = O(4s + p) \in O(p + s)$$

In der folgenden Abschätzung berufe ich mich auf konkreten Zeilen im Pseudocode aus Wikipedia.
(s. Seite 27)

Yen-Algorithmus:

- Dijkstra zur Bestimmung des kürzesten Pfades: $O(s^2)$ (worst-case), Zeile 3.

Bei jeder Generierung eines Pfades wird ein Pfad aus der Kandidatenmenge genommen. Den nenne ich hier *der betrachtete Pfad*. Ab dem nächsten Punkt beginnt die Schleife, die k mal iteriert wird. l entspricht der Anzahl der Kanten des betrachteten Pfades in G' (nicht der Länge).

- Entfernen der Kanten: $O(k * l)$. Hier wird Liste A mit allen bisherigen Ergebnissen iteriert. Diese Liste hat die Länge k . In jeder Iteration erfolgt noch eine Iteration von *rootPath*, der maximal so lang sein kann, wie fast der ganze Pfad (außer dem Zielknoten): $O(l)$, Zeilen 18–22.
- Entfernen der Kanten und Knoten, die im betrachteten Pfad auftreten: $O(l)$, Zeilen 24f.
- Dijkstra auf dem Graphen G' mit entfernten Kanten und Knoten (der Baum): $O(s^2)$ (worst-case), Zeile 28.
- Aktualisierung der Gewichte und Zurückstellung der Kanten und Knoten: $O(c * l^2)$ (worst-case)
Der betrachtete Pfad wird iteriert: $O(l)$. Jeden iterierte Knoten nenne ich n_i .
 - Aktualisierung der Kanten, die aus n_i führen: $O(l)$. Hier werden einfach die Knoten, die aus n_i führen, iteriert. Diese Anzahl beträgt nicht mehr als 10, wie schon beschrieben. Danach folgt noch eine Bildung des neuen *spurPath*: $O(l)$, Zeilen 28–39..

- Aktualisierung der Kanten, die zu n_i führen: $O(10 * 10 * l) = O(l * c)$, wobei $c = \text{const.}$ ³
Ein Knoten hat auch in der Regel nicht mehr als 10 Knoten, die zu ihm führen.
Hier wird durch die Knoten, die zu n_i führen, iteriert und wenn ein Knoten aktualisiert wird, wird durch seine Knoten, die zu ihm führen, iteriert usw. Maximal kann man l solche Wiederholungen durchführen, bis man den *spurNode* erreicht.
- Bestimmung der Länge des *spurNode*: $O(l)$. Der betrachtete Pfad wird iteriert bis zum n_i und jedes Mal wird das Gewicht an der Kante aufaddiert: $O(l)$.

$$O(s^2) + k(O(k * l) + O(l) + O(s^2) + O(cl^2)) = O(k(k * l + cl^2 + s^2 + l) + s^2) \in O(k^2l + ckl^2 + ks^2)$$

Nun addieren wir alles zusammen:

$$\begin{aligned} O(p^2 + p) + O(p + s) + O(k^2l + ckl^2 + ks^2) &= O(k^2l + ckl^2 + ks^2 + p^2 + 2p + s) \\ O(k^2l + ckl^2 + ks^2 + p^2 + 2p + s) &\in O(k(kl + s^2 + cl^2) + p^2) \end{aligned}$$

³ Da die Anzahl von Punkten und Straßen in den Beispielen sich auch in diesem Zahlbereich befindet, dürfen wir diese Konstante c nicht einfach aus unserer Betrachtung weglassen.

1.5 Erweiterungen

1.5.1 Start und Ziel an verschiedenen Stellen

In allen verfügbaren Beispielen, die auf der BWINF-Webseite vorhanden sind, entspricht die x -Koordinate des Zielpunkts stets der letzten x -Koordinate der gegebenen Matrix. Der Startpunkt liegt auch stets im Punkt $(0,0)$. In meinem Programm ist es auch erlaubt, Startpunkte und Zielpunkte an andere Stellen zu setzen. (s. dazu das Beispiel 3.6.)

1.5.2 Nichtganzzahlige Koordinaten

In den auf der Website von BWINF vorhandenen Beispielen sind alle Koordinaten nur ganzzahlig und positiv. In meinem Programm können sowohl ganzzahlige als auch rationale Koordinaten verwendet werden. (s. dazu das Beispiel 3.7.)

1.5.3 Kreuzungen von Straßen

In keinem BWINF-Beispiel kreuzt sich eine Straße mit einer anderen. Unter einer Kreuzung verstehe ich einen Zustand, in dem zwei Straßen $a(o, p)$ und $b(q, r)$ im Graphen G existieren und die Strecke op einen Schnittpunkt mit der Strecke qr hat, der gleichzeitig keiner der Punkte o, p, q oder r ist. In meinem Programm sind solche Kreuzungen erlaubt. (s. dazu das Beispiel 3.8.)

1.5.4 Genauigkeit

Als eine weitere Erweiterung der Aufgabenstellung biete ich dem Benutzer meines Programms an, die Genauigkeit einer Suche zu wählen. Es gibt zwei Varianten: eine *approximierte Suche* und eine *genaue Suche*. Bei der genauen Variante wird im Yen-Algorithmus bei der Generierung eines Pfades γ mit ψ verglichen und daraus wird entschieden, ob wir einen weiteren Pfad generieren müssen.

In der approximierten Variante wird γ zu einem Prozentwert δ umgewandelt, wobei 100% ω entspricht. Danach wird δ auf Einer gerundet. Nun wird im Yen-Algorithmus bei der Generierung eines neuen Pfades ξ mit δ verglichen, ob $\delta > \xi$. Wenn ja, müssen wir einen nächsten Pfad generieren. Wenn nicht, dann fanden wir den besten approximierten Pfad. (s. dazu das Beispiel 3.9.) Das ist ein gutes Beispiel dafür, dass wir dadurch Zeit sparen können und das Ergebnis sogar besser ist.

Außerdem funktioniert auch das Ergebnis des in der Aufgabenstellung gelösten Beispiels nur unter der Voraussetzung, dass wir γ runden.

Literatur

- [1] Yen, Jin Y. *An Algorithm for Finding Shortest Routes From All Source Nodes to a Given Destination in General Networks*. Quarterly of Applied Mathematics 27, 526-530, 1970. <https://doi.org/10.1090/qam/253822>.
- [2] Yen, Jin Y. *Finding the K Shortest Loopless Paths in a Network*. Management Science, USA, 1971. <https://doi.org/10.1287/mnsc.17.11.712>.

2 Umsetzung

Das Programm ist in die drei wichtigsten Klassen unterteilt: **Graph**, **Dijkstra**, **Yen**.

2.1 Die Klasse Graph

Die Klasse **Graph** behandelt den Bau von beiden Graphen G und G' , die in der Implementierung durch eine Variable **type** (**type 1** ist G und **type 2** G') unterschieden werden.

Wir deklarieren einen Graphen, stellen den Typ ein und lesen aus der Textdatei die Punkte und Straßen ein. Einen Punkt speichere ich auf folgender Weise: **pair<double, double>**. Da ich erlaube, dass die Koordinaten von Punkten rational sind, entschied ich mich aus dem Grund für **Double**, obwohl in der Aufgabenstellung die Koordinaten nur ganzzahlig sind. Eine Straße wird ganz einfach als **pair<pair<double, double>, pair<double, double>>** gespeichert. Diese vielleicht nicht ganz komfortable Schreibweise dient nur für das Einlesen der Daten aus der Textdatei und für die Unwandlung und Bearbeitung der Daten.

Jedem Punkt und jeder Straße ordne ich einen internen Index zu. Alle Punkte und Straßen füge ich in entsprechende **map** ein. Ich erstelle immer zwei Map-Container jeder Art: einen Map-Container mit Indizes als Schlüssel und einen anderen mit dem entsprechenden Datentypen als Schlüssel. Die Elemente der Map-Container werden stets mit der eingebauten Funktion **find()** gefunden, die in der Zeit von $O(\log n)$ läuft. Diese Zeit wird in der Laufzeit des Algorithmus vernachlässigt.

Wie schon beschrieben, beim Bau des Graphen Typ 1 suche ich bei jedem Punkt **p** nach allen Straßen, die mit diesem Punkt verbunden sind. Gleichzeitig messe ich die Entfernung **dist** zwischen den beiden Punkten, die eine solche Straße bilden. Danach füge ich den Index des anderen Punktes dieser Straße mit der Länge **dist** als ein Paar **pair<int, double>** in die Menge der Nachbarn von **p** ein.

Beim Bau des Graphen Typ 2 erfolgt das Gleiche, aber diesmal müssen noch die Abbiegungen bestimmt werden.

Wir nehmen jedes Mal eine Straße. Eine Straße besitzt zwei Punkte a und b . Für jeden Punkt bestimmen wir bereits alle seine Nachbarn. Wir nehmen den Punkt a und iterieren durch die Menge seiner Nachbarn. Nun prüfen wir, ob Punkte a , b und der Nachbarpunkt p kollinear sind. Wir speichern diesen boolschen Wert als **currTurn**. Jetzt müssen wir nur prüfen, ob einer der Punkte ein Pivotpunkt ist. Wenn ja, dann fügen wir zur Menge von Nachbarn der Start-Pivotstraße den Index der anderen Straße, die aus den zwei übrig gebliebenen Punkten besteht, bzw. zur Menge der anderen Straße, die aus den zwei übrig gebliebenen Punkten besteht, die Ziel-Pivotstraße ein. Diese Informationen werden als **pair<double, double>** gespeichert. In beiden Fällen ist der Wert der Abbiegung 0 und als zweiter Wert in **pair<double, double>** gilt genau diese Zahl.

Wenn aber keiner der Punkte ein Pivotpunkt ist, dann müssen wir die x -Koordinaten von a und p vergleichen.

Ist $a_x \leq p_x$, dann fügen wir zur Menge der Straße (a, b) den Index der Straße (b, p) mit dem Wert **currTurn** ein.

Ist $a_x \geq p_x$, dann fügen wir zur Menge der Straße (b, p) den Index der Straße (a, b) mit dem Wert **currTurn** ein.

Danach wiederholen wir das Gleiche für den Punkt b . Für das Einfügen der Indizes der entsprechenden Straßen vertauscht man nur die Variablen a mit b .

Am Ende bekommen wir eine **map**, in der jedem internen Straße-Index eine Menge mit Nachbarstraßen und deren entsprechende Abbiegungen zugeordnet ist.

Um die größte Funktionalität meines Programms zu erhalten, entschied mich dafür, dass ich die Pivotpunkte außerhalb der Matrix platziere. Dem Start-Pivotpunkt gebe ich folgende Koordinaten: $(-1, 0)$ und dem Ziel-Pivotpunkt die Koordinaten: $(-2, 0)$.

In beiden Graphen werden Knoten als Zeiger der Klasse **Node** gespeichert. Diese Klasse besteht aus zwei Werten: dem internen Index und einem Gewicht (Entfernung vom Startknoten im Graphen; genutzt bei Yen). Auch werden Pfade als Zeiger der Klasse **Path** dargestellt. Ein Objekt von **Path** beinhaltet die Länge des gesamten Pfads, das gesamte Gewicht an den Kanten des Pfads und den Vektor von Zeigern von **Node**, der alle Knoten im Pfad enthält. Die Verwendung von Zeigern hat zum Ziel eine verbesserte Laufzeit und eine effiziente Arbeitsspeicherverwaltung. Die Kanten werden in einer **map<int, double>** gespeichert, die den internen Index einer Kante und das entsprechende Gewicht enthält.

Ein wichtiges Element der Implementierung stellen zwei Map-Container dar, die ich **inNodes** und

`outNodes` nenne. Diese erhalten als Schlüssel einen Knotenindex und als Wert eine Menge mit allen Nachbarn, die entsprechend aus (engl. *out*) oder zu (engl. *in*) dem Knoten führen. Diese sind die Grundlagen der Funktionsweise des Dijkstra-Algorithmus in meiner Implementierung.

2.2 Die Klasse Dijkstra

In der Klasse `Dijkstra` läuft ein ganz normaler Dijkstra-Algorithmus. Ich verwende eine Vorrangwarteschlange. Die Entfernungen vom Start zum jedem Knoten werden mit Hilfe einer `map<Node*, double>` gespeichert. Sie nenne ich `startToDistance`. Eine andere `map`, die einen Vorgänger jedes Knotens im Pfad enthält, nenne ich `map<Node*, Node*> previousNode`. Eine Menge `visNodes` enthält alle bereits besuchten Knoten.

Die Methode `Dijkstra::determineConnections(Node* start, Node* end, bool direction)` stellt Vorgänger von Knoten in `previousNode` ein und bestimmt die Entfernung jedes Knotens vom Ursprung `start` in `startToDistance`. Das ist die Grundlage der Funktionsweise des Dijkstra-Algorithmus. Diese Methode wird auch zur Traversierung des Graphen G' vom Zielknoten im Yen-Algorithmus verwendet. Aus diesem Grund beschreibt `direction`, in welche Richtung die Traversierung verläuft (1 für die Traversierung vom Startknoten zum Zielknoten, 0 für andersherum).

Als `startNode` und `endNode` bezeichnet werden die Knoten, von dem wir anfangen und an dem wir enden. Zuerst wird überprüft, in welche Richtung traversiert wird. Wenn `direction` gleich 1 ist, werden `start` und `end` entsprechend `startNode` und `endNode`. Andernfalls wird die Reihenfolge getauscht.

Der Knoten `startNode` wird in die Vorrangwarteschlange eingefügt. Es beginnt eine Schleife, die läuft, bis es keine weiteren Knoten in der Vorrangwarteschlange gibt. Jeden iterierten Knoten aus der Vorrangwarteschlange nenne ich hier `currNode`.

In der Schleife wird zuerst der erste Knoten in der Schlange abgerufen: `currNode`. Dieses Element wird auch gleichzeitig aus der Schleife gelöscht. Wir prüfen, ob `currNode` gleich `endNode` ist, ob wir die Schleife nicht jetzt beenden sollen.

`currNode` wird als besucht in `visNodes` gekennzeichnet. Jetzt wird entweder die Menge `outNodes` des Knotens `currNode`, wenn `direction` 1 ist, oder `inNodes` des Knotens `currNode`, wenn `direction` 0 ist, als `neighborNodes` abgerufen. Wir iterieren durch diese Menge. Jeden iterierten Nachbarn des Knotens `currNode` bezeichne ich hier als n_i .

Wir prüfen, ob n_i noch nicht besucht wurde. Wir ordnen `currNode` seine Entfernung vom Ursprung aus `startToDistance` zu. Wenn ihm noch keine Entfernung zuzuordnet war, ordnen wir ihm den Wert `DBL_MAX` (ein Ersatz für Unendlichkeit). Die Entfernung vom Ursprung von `currNode` speichern wir als `distance`. Wir vergrößern `distance` um das Gewicht an der Kante zwischen n_i und `currNode`. Wir suchen im Map-Container `startToDistance` nach der Entfernung von n_i und speichern sie als d .

Wenn d nicht gefunden wird (Knoten nicht erreicht) oder wenn $d > \text{distance}$, heißt das, wir fanden eine bessere Verbindung von `currNode` zu n_i . Der Wert von n_i in `startToDistance` wird als `distance` gespeichert. `currNode` wird zum Vorgänger von n_i und diese Information wird in `previousNode` gespeichert. In der Klasse `Node` von n_i wird das Gewicht als `distance` eingestellt. Danach prüfen wir, ob n_i sich in der Vorrangwarteschlange befindet. Wenn ja, dann entfernen wir ihn. Es kann ja sein, dass es eine veraltete Version von n_i in der Vorrangwarteschlange gibt. Danach fügen wir n_i in die Vorrangwarteschlange ein.

Die folgenden zwei Methoden helfen uns im Yen-Algorithmus, die Pfade zu aktualisieren, wenn Kanten und Knoten entfernt werden.

Die Methode `Dijkstra::updateOutDistance(Node* n)` funktioniert auf folgender Weise. Wir rufen die Menge `outNodes` des Knotens n ab. Wir prüfen, ob es bereits einen Pfad zu n gibt, indem wir prüfen, ob dieser Knoten einen Wert im Map-Container `startToDistance` hat. Wenn kein Pfad gefunden wurde, dann fügen wir diesen Knoten mit dem Wert `DBL_MAX` (ein Ersatz für Unendlichkeit) in diese `map` ein. Dann iterieren wir durch alle Knoten in der Menge `outNodes` des Knotens n . Wir suchen im Map-Container `startToDistance` nach der Entfernung vom Ursprung des Nachbarn und speichern sie als `currDistance`. Falls zu einem Knoten kein Pfad gefunden wurde, bekommt diese Variable den Wert `DBL_MAX`. Danach wird `currDistance` bei jedem Nachbarn um die Kante, die n und der Nachbar bilden, vergrößert. Wenn `currDistance` kleiner ist als die Entfernung vom Ursprung von n , wird diese Entfernung aktualisiert.

Wenn die Entfernung vom Ursprung von n überhaupt aktualisiert wurde, erstellen wir einen neuen Pfad. Wir fangen mit dem Knoten n an und setzen mit dem Wert von n im Map-Container `previousPath` fort.

Dann ist der nächste Knoten der Vorgänger von n . So setzen wir fort, bis wir auf einen Knoten kommen, der keinen Vorgänger hat. Die Länge des ganzen Pfades ist natürlich die aktualisierte Entfernung vom Ursprung des Knotens n .

Die andere Methode, die die Knoten aktualisiert, ist `Dijkstra::updateInDistance(Node* n)`. In dieser Methode befindet sich eine Liste von Knoten, die zu aktualisieren sind. Als ersten Knoten fügen wir den Knoten n ein. Dann läuft eine Schleife, in der alle Elemente von dieser Liste iteriert werden.

Bei jedem Knoten i , der sich in der Liste befindet, wird seine Entfernung vom Ursprung `currDistance` anhand des Map-Containers `startToDistance` gefunden. Dann wird die Menge `inNodes` des Knotens i abgerufen. Wir iterieren durch diese Menge. Hier wird für jeden Nachbarn q seine Entfernung vom Ursprung aus dem Map-Container `startToDistance` abgelesen. Diese Information wird in der Variable `neighborDistance` gespeichert. Ähnlich wie in der vorherigen Methode: Falls zu einem Knoten kein Pfad gefunden wurde, bekommt diese Variable den Wert `DBL_MAX`. Hier erstellen wir auch eine andere neue Variable `updatedDistance`, die gleich `currDistance` und dem Gewicht an der Kante zwischen i und q ist.

Wenn `updatedDistance` kleiner ist als `neighborDistance`, dann wird die Entfernung vom Ursprung von q aktualisiert, der Vorgänger von q wird i und der Knoten q wird in die Liste von Knoten, die zu aktualisieren sind, eingefügt.

Die Methode `Dijkstra::getReversedTraversal` initialisiert die Funktion `determineConnections`, aber die Variable `direction` ist stets 0. Der Startknoten wird aber auch nicht definiert, sodass die Funktion für jeden `spurNode` im Yen-Algorithmus geeignet ist. Wir traversieren vom Zielknoten solange, wie es noch Knoten in der Vorrangsschlange gibt, wir sind durch keinen Zielknoten (=Startknoten) begrenzt. Diese Funktion wird im Yen-Algorithmus benutzt, um neue Verbindungen zwischen Knoten im Graphen G' mit entfernten Knoten und Kanten zu finden. Anhand dieser Funktion entsteht der erwähnte Baum.

2.3 Die Klasse Yen

Wie schon im ersten Kapitel beschrieben, basiert meine Implementierung des Yen-Algorithmus sehr stark auf dem Pseudocode im entsprechenden englischsprachigen Wikipedia-Artikel.

In dieser Klasse befinden sich eine Liste `A`, die die fertigen Ergebnisse enthält, eine Vorrangswarteschlange `B`, in der sich die Kandidatenpfade befinden und ein Map-Container `pathsToSpurNode`, der einem Pfad seinen `spurNode` zuordnet.

Am Anfang in der Methode `preprocess()` wird der kürzeste Pfad bestimmt. Ein Dijkstra-Algorithmus läuft auf dem Graphen G' . Der gefundene Pfad wird in `B` eingefügt. Als `spurNode` gilt für diesen Pfad der Startknoten. Diese Information wird in `pathsToSpurNode` gespeichert.

Danach folgt die Methode `generatePath()`. Diese Methode, wie der Name sagt, generiert einen nächsten kürzesten Pfad. Zuerst wird der erste Pfad aus der Kandidatenliste genommen und wir nennen ihn `currPath`. Er wird auch gleichzeitig aus der `B` entfernt. Wir fügen ihn in `A` ein. Wir rufen den `spurNode` von `currPath` aus dem Map-Container ab. Der `rootPath` wird vom Startknoten zu `spurNode` gebildet. Jetzt erfolgt eine Iteration der Liste `A` (`currPath` ist aber ausgeschlossen).

Jedes Mal wird ein neuer Pfad `currSubPathRootPath` gebildet. Dieser Pfad ist ein Teilpfad des iterierten Pfades aus der Liste `A` vom Startknoten zu `spurNode`. Dann wird verglichen, ob `currSubPathRootPath` gleich `rootPath` ist. Wenn ja, dann entfernen wir aus dem Graphen G' die folgende Kante nach `currSubPathRootPath` im iterierten Pfad aus der Liste `A`.

Danach werden alle Kanten und Knoten im `currPath` außer dem Zielknoten entfernt.

An dieser Stelle lasse ich den Dijkstra-Algorithmus in Form von `Dijkstra::getReversedTraversal()` laufen.

Danach beginne ich den `currPath` vom vorletzten Knoten zum `spurNode` zu iterieren. Den iterierten Knoten nenne ich `currNode`. Ich stelle ihn gleich in den Graphen zurück. Nun wird die Methode `updateOutDistance()` an den `currNode` angewendet. Diese Methode liefert auch einen neuen `spurPath`, den ich hier aber `newPath` nenne.

Wenn `newPath` geliefert wird (es muss ja nicht unbedingt sein), lasse ich noch die Methode `updateInDistance()` für den `currNode` auf dem Graphen laufen.

Danach erfolgt noch eine Iteration von `currPath` vom Startknoten zum `currNode`. Es werden die Ge-

wichte an den Kanten des `currPath` zu `totalWeight` zusammenaddiert. Hier wird auch der Teilpfad des `currPath` vom Startknoten zum `currPath` zu `spurPath` kopiert.

Gleich nach dieser Iteration erfolgt Kopieren des `newPath` am Ende des `spurPath`. `totalWeight` wird um die Länge von `newPath` vergrößert. Es entsteht gleich danach `totalPath`, der gleich `spurPath` ist und die Länge von `totalWeight` besitzt.

Es wird überprüft, ob `totalPath` schon in `pathsToSpurNode` existiert. Wenn nicht, wird er in `B` eingefügt. Er wird auch in `pathsToSpurNode` mit `currNode` als *spurNode* eingefügt.

Danach erfolgt Zurückstellung der Kante zwischen `currNode` und dem nächsten Knoten im `currPath`. Am Ende rufen wir noch die Länge der Kante zwischen `currNode` und dem nächsten Knoten im `currPath` ab und addieren dazu die Entfernung vom Ursprung dieses nächsten Knotens aus `getReversedTraversal()`.

Wir speichern diese Summe als `edgeWeight`. Falls Entfernung vom Ursprung des `currNode` aus `getReversedTraversal()` größer ist als `edgeWeight`, müssen wir diese Entfernung in `getReversedTraversal()` aktualisieren und als `edgeWeight` speichern. In `getReversedTraversal()` wird `nextNode` zum Vorgänger von `currNode`. In diesem Falle läuft nochmal die Funktion `Dijkstra::updateInDistance()` für den Knoten `currNode`.

Letzendlich wird `currPath` als der nächste kürzeste Pfad ausgegeben.

Ich füge den Code des Yen-Algorithmus, sowie die Implementierungen der beiden Methoden `Dijkstra::updateOutDistance()` und `Dijkstra::updateInDistance()` im Abschnitt „Quellcode“ bei.

3 Beispiele

Als eine Erweiterung der Aufgabenstellung generiere ich Grafiken, die den gefundenen Pfad abbilden. In den dargestellten Grafiken stehen der blaue Punkt für den Startpunkt und der grüne Punkt für den Zielpunkt. Der rote Pfad bezeichnet den Pfad mit der niedrigsten Anzahl von Abbiegungen.

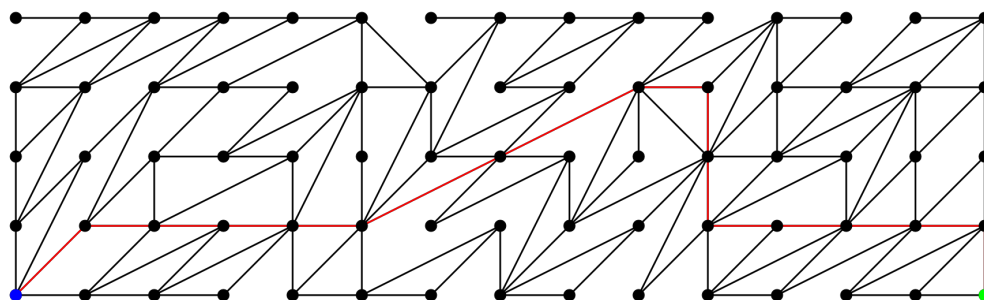
Einige Pfade wiederholen sich in manchen Prozentbegrenzungen. Das passiert aus dem Grund, dass ich aus der Menge der Pfade mit derselben Anzahl von Abbiegungen immer den kürzesten generierten Pfad nehme.

Außerdem präsentiere ich als eine nächste Erweiterung der Aufgabenstellung die Pfade, bei denen die Anzahl von Abbiegungen minimal ist, wenn in keiner von den Prozentbegrenzungen eine solche minimale Anzahl auftritt. Zusätzlich präsentiere ich zum Vergleich auch den kürzesten Pfad, also den Pfad mit der Länge ω , mit der entsprechenden Anzahl von Abbiegungen, wenn er nicht auftritt.

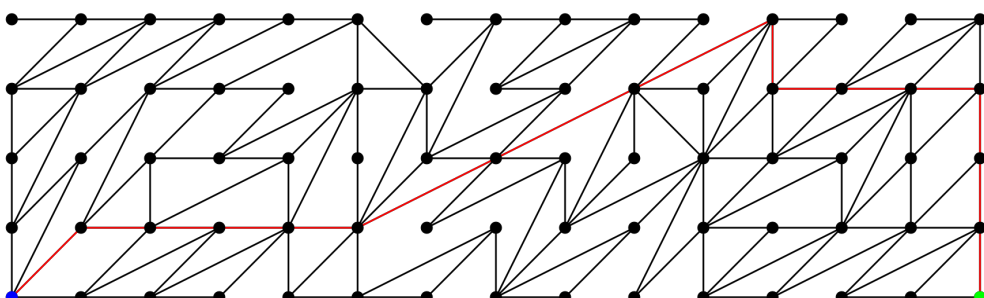
3.1 Beispiel 1 (BWINF)

Textdatei: abbiegen1.txt

- 110%
 - Abbiegen: **6**
 - Länge des Pfades: **17,8863**
 - Länge des Pfades prozentual: **104,462%**
 - Punkte: (0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (10, 2), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (14, 0)

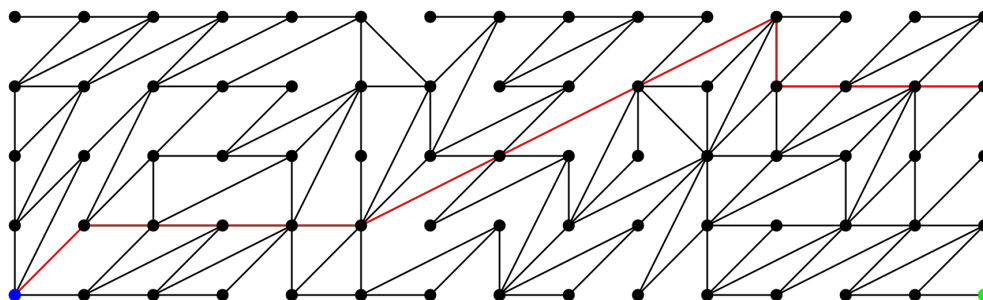


- 115%
 - Abbiegen: **5** (das beste Ergebnis)
 - Länge des Pfades: **19,1224**
 - Länge des Pfades prozentual: **111,6%**
 - Punkte: (0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (11, 4), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)

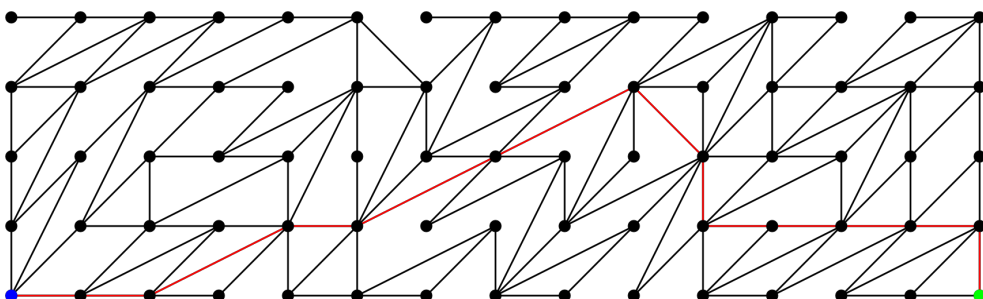


- 130%
 - Abbiegen: **5** (das beste Ergebnis)
 - Länge des Pfades: **19,1224**

- Länge des Pfades prozentual: **111,6%**
- Punkte: (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)



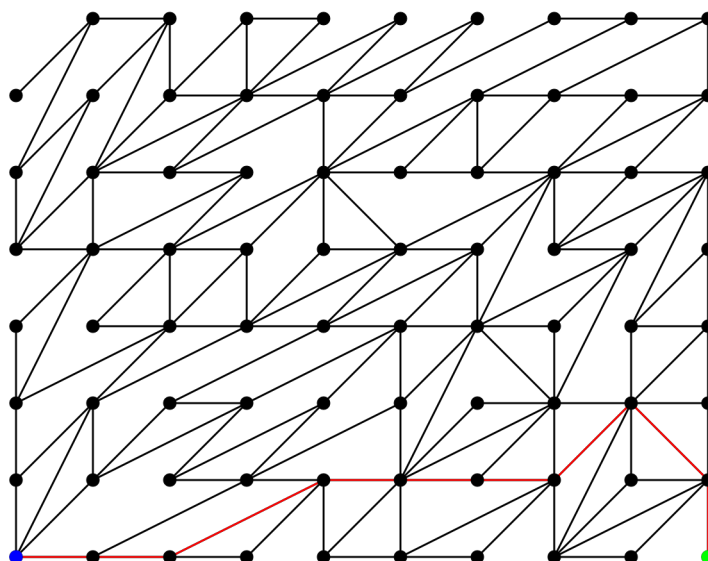
- der kürzeste Pfad
 - Abbiegen: 7
 - Länge des Pfades: 17,1224



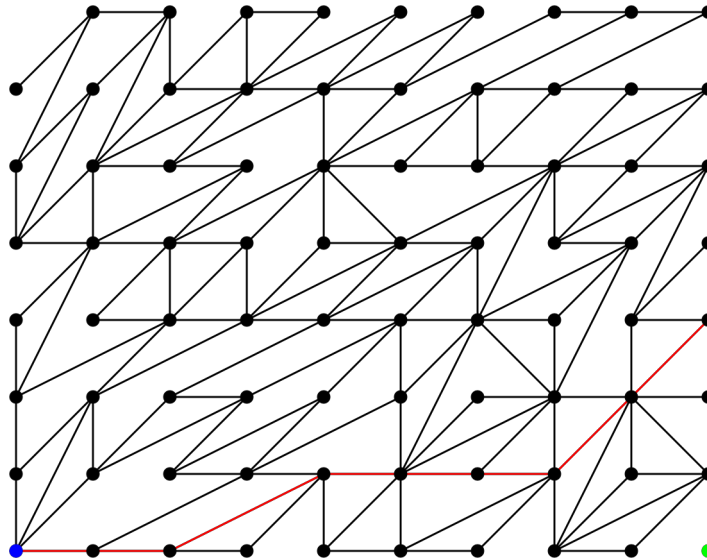
3.2 Beispiel 2 (BWINF)

Textdatei: abbiegen2.txt

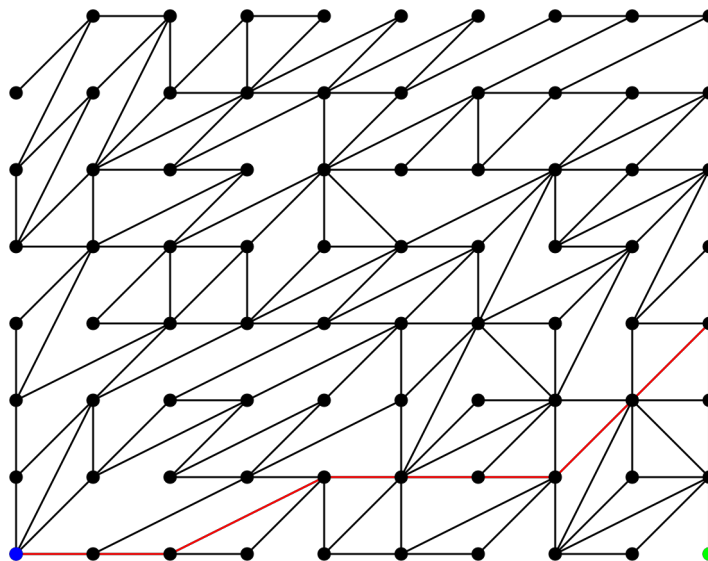
- 110%
 - Abbiegen: **5**
 - Länge des Pfades: **11,0645**
 - Länge des Pfades prozentual: **101,636%**
 - Punkte: (0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 1), (9, 0)



- 115%
 - Abbiegen: 4
 - Länge des Pfades: **13,0645**
 - Länge des Pfades prozentual: **120,008%**
 - Punkte: (0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 3), (9, 2), (9, 1), (9, 0)

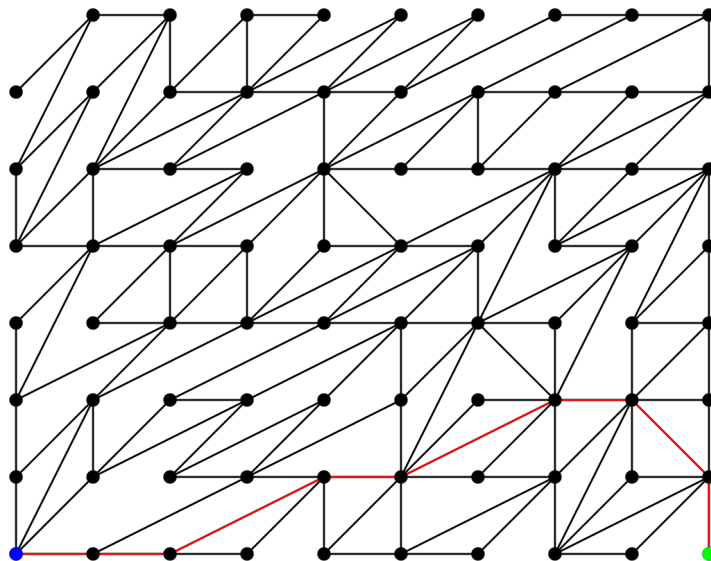


- 130%
 - Abbiegen: 4
 - Länge des Pfades: **13,0645**
 - Länge des Pfades prozentual: **120,008%**
 - Punkte: (0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 3), (9, 2), (9, 1), (9, 0)

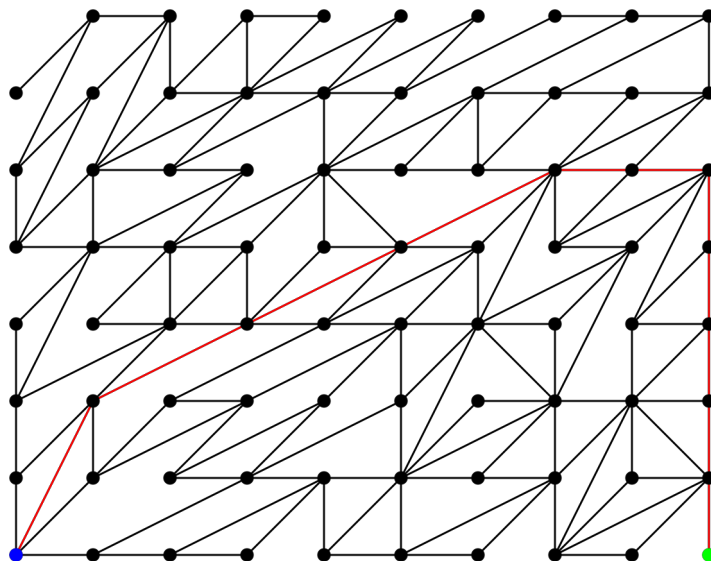


- der kürzeste Pfad
 - Abbiegen: 6

- Länge des Pfades: 10,8863



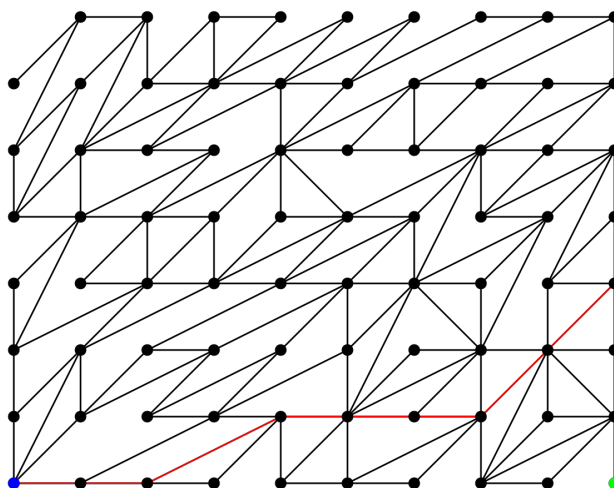
- der Pfad mit der niedrigsten Anzahl von Abbiegungen
 - Abbiegen: 3
 - Länge des Pfades: 15,9443
 - Länge des Pfades prozentual: 146,461%



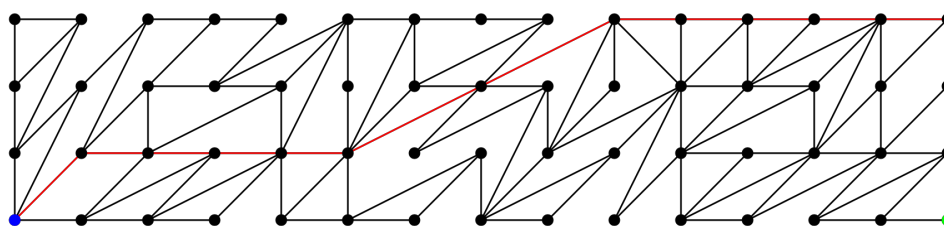
3.3 Beispiel 3 (BWINF)

Textdatei: abbiegen3.txt

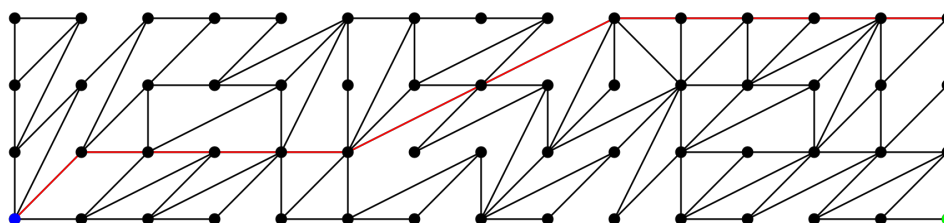
- 110%
 - Abbiegen: 4 (das beste Ergebnis)
 - Länge des Pfades: **17,8863**
 - Länge des Pfades prozentual: **104,462%**
 - Punkte: (0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)



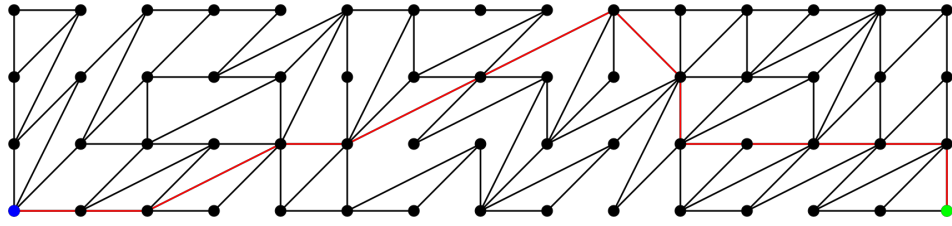
- 115%
 - Abbiegen: 4
 - Länge des Pfades: **17,8863**
 - Länge des Pfades prozentual: **104,462%**
 - Punkte: (0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)



- 130%
 - Abbiegen: 4
 - Länge des Pfades: **17,8863**
 - Länge des Pfades prozentual: **104,462%**
 - Punkte: (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)



- der kürzeste Pfad
 - Abbiegen: 7
 - Länge des Pfades: 17,1224

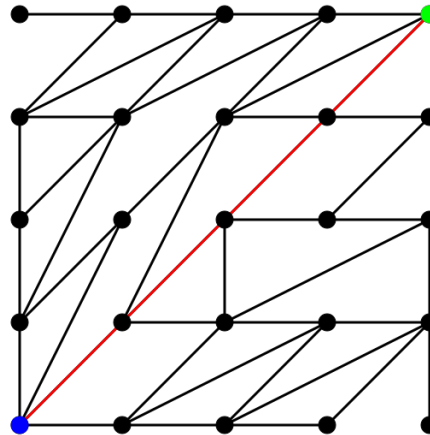


3.4 Beispiel 4

Textdatei: `abbiegen4.txt`

Besonderheit: der Pfad mit der niedrigsten Anzahl von Abbiegungen ist gleich 0, der Pfad verläuft nur über den Diagonalen

- 110%
 - Abbiegen: **0** (das beste Ergebnis)
 - Länge des Pfades: **5,65685**
 - Länge des Pfades prozentual: **100%**
 - Punkte: **(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)**



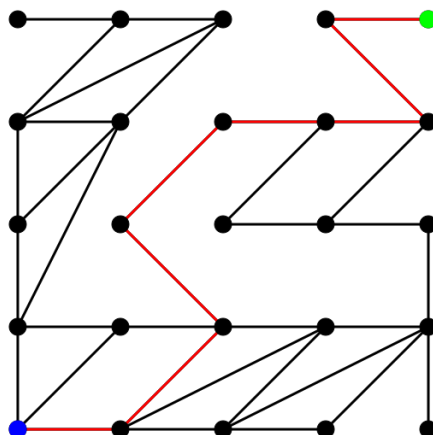
- 115%: das Gleiche wie bei 110%
- 130%: das Gleiche wie bei 110%

3.5 Beispiel 5

Textdatei: `abbiegen5.txt`

Besonderheit: sehr viele Abbiegungen im besten Pfad, das Ziel ist isoliert vom Rest des Graphen, vom Punkt (4, 3) zum (3, 4) macht der Algorithmus einen Schritt zurück in Bezug auf die x -Achse (s. Lösungsidee)

- 110%
 - Abbiegen: **6** (das beste Ergebnis)
 - Länge des Pfades: **9,65685**
 - Länge des Pfades prozentual: **100%**
 - Punkte: **(0, 0), (1, 0), (2, 1), (1, 2), (2, 3), (3, 3), (4, 3), (3, 4), (4, 4)**



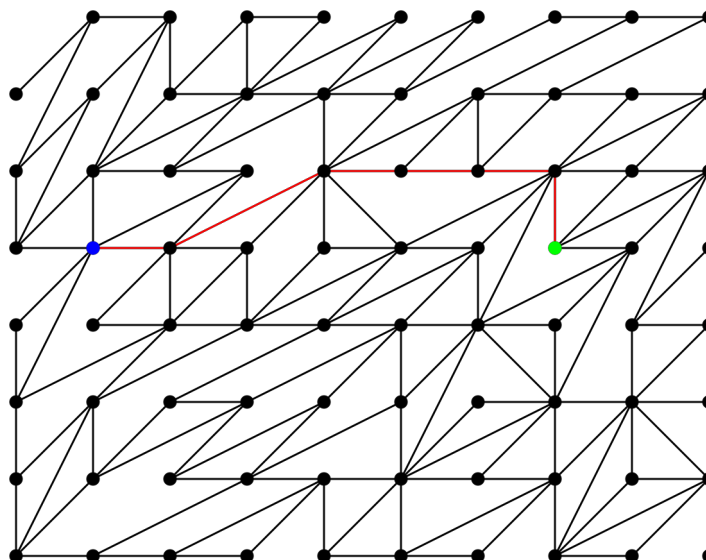
- 115%: das Gleiche wie bei 110%
- 130%: das Gleiche wie bei 110%

3.6 Beispiel 6

Textdatei: `abbiegen6.txt`

Besonderheit: Da Ziel befindet sich nicht „am Ende“ der Matrix

- 110%
 - Abbiegen: **3** (das beste Ergebnis)
 - Länge des Pfades: **7.23607**
 - Länge des Pfades prozentual: **100%**
 - Punkte: (1, 4), (2, 4), (4, 5), (5, 5), (6, 5), (7, 5), (7, 4)



- 115%: das Gleiche wie bei 110%
- 130%: das Gleiche wie bei 110%

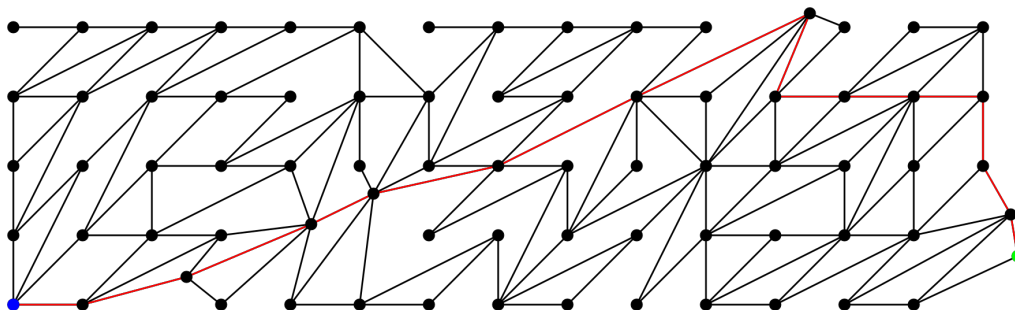
3.7 Beispiel 7

Textdatei: `abbiegen7.txt`

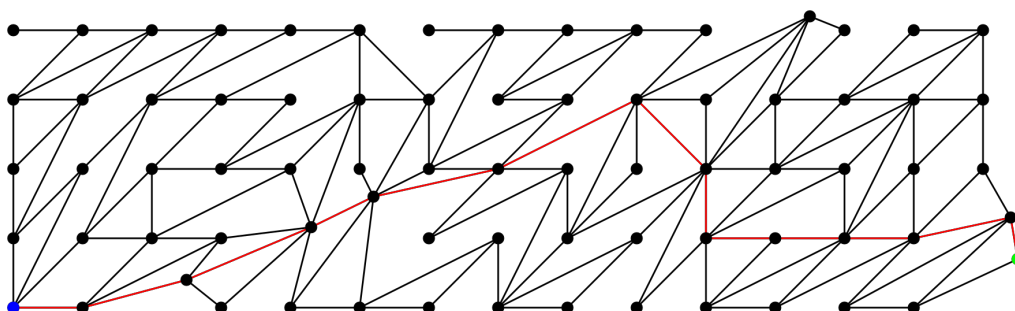
Besonderheit: nichtganzzahlige Koordinaten

- 110%

- Abbiegen: **3** (das beste Ergebnis)
- Länge des Pfades: **13,4721**
- Länge des Pfades prozentual: **106,497%**
- Punkte: **(0, 0), (1, 0), (2.5, 0.4), (4.3, 1.16), (5.2, 1.6), (7, 2), (9, 3), (11.5, 4.2), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14.4, 1.3), (14.5, 0.7)**



- 115%: das Gleiche wie bei 110%
- 130%: das Gleiche wie bei 110%
- der kürzeste Pfad
 - Abbiegen: 3
 - Länge des Pfades: 12,6503

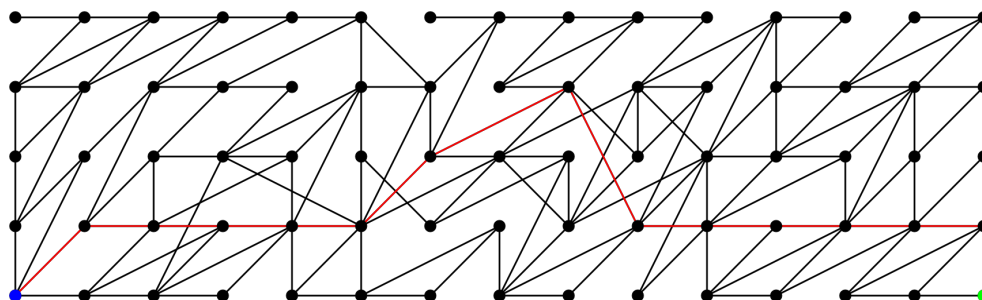


3.8 Beispiel 8

Textdatei: abbiegen8.txt

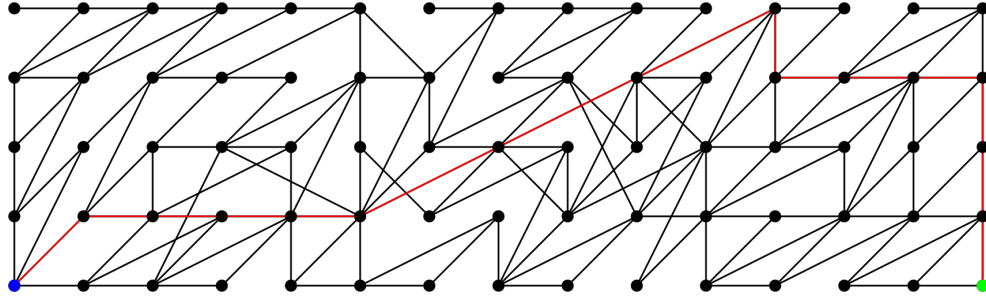
Besonderheit: Straßen kreuzen sich

- 110%
 - Abbiegen: **6**
 - Länge des Pfades: **17,3006**
 - Länge des Pfades prozentual: **101,04%**
 - Punkte: **(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 2), (8, 3), (9, 1), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (14, 0)**

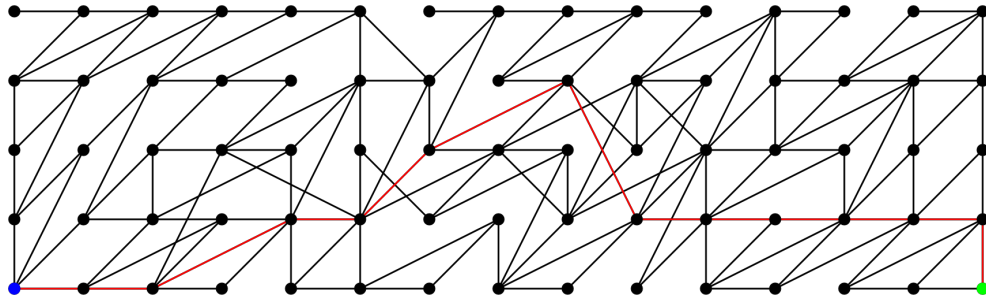


- 115%

- Abbiegen: **5** (das beste Ergebnis)
- Länge des Pfades: **19,1224**
- Länge des Pfades prozentual: **111,681%**
- Punkte: **(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)**



- 130%: das Gleiche wie bei 115%
- der kürzeste Pfad
 - Abbiegen: 7
 - Länge des Pfades: 17,1224

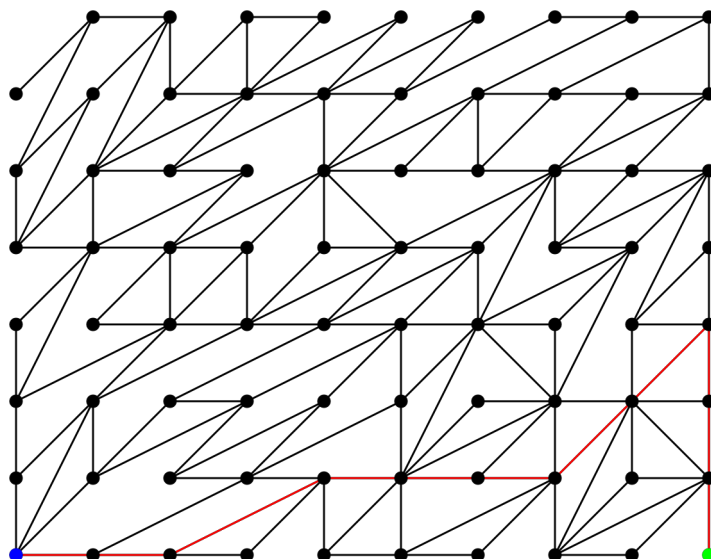


3.9 Beispiel 2 (erweitert)

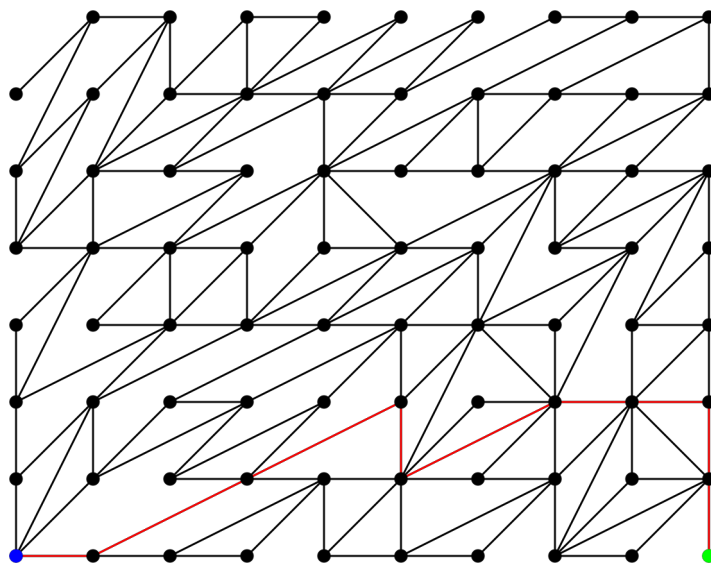
Textdatei: `abbiegen2.txt`

Besonderheit: Anwendung der Möglichkeit der Wahl der Genauigkeit

- 120% (approximierte Suche)
 - Abbiegen: **4**
 - Länge des Pfades: **13,0645**
 - Länge des Pfades prozentual: **120%** (eigentlich 120,008%)
 - Punkte: **(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 3), (9, 2), (9, 1), (9, 0)**



- 120% (genaue Suche)
 - Abbiegen: **5**
 - Länge des Pfades: **12,7082**
 - Länge des Pfades prozentual: **116,735%**
 - Punkte: (0, 0), (1, 0), (3, 1), (5, 2), (5, 1), (7, 2), (8, 2), (9, 2), (9, 1), (9, 0)



4 Quellcode

Hier füge ich Implementierungen von zwei Methoden bei: den Yen-Algorithmus und den Algorithmus, der die Kanten im Graphen G' (Typ 2) erstellt.

Die Zeilenangaben beziehen sich auf die Zeilen im Pseudocode aus Wikipedia auf Seite 27.

```

1  //ein "naechster" Pfad des Yen-Algorithmus wird
3  //hier bestimmt
   Path* generatePath()
5  {
   //An dieser Stelle wurde bereits der kuerzeste Pfad im Graphen gefunden
7   //und wurde in die Menge B (Kandidatenliste) eingefuegt

9   //der erste Pfad aus der Kandidatenliste wird betrachtet [Z.53]
   Path* currPath = *(B.begin());
11  int currPathLen = currPath->getLength();

13  //der aktuelle Pfad wird in die Ergebnissenlisten eingefuegt
   A.pb(currPath);

15  //der spurNode (Abzweigungsknoten) gilt als Knoten, ab dem
17  //wir den Pfad veraendern
   Node* spurNode = pathsToSpurNodes.find(currPath)->second;

19  //der Teilpfad bis zum spurNode
21  vector<Node*> rootPath;
   currPath->subPath(rootPath, spurNode);

23  //wir beginnen, entsprechende Knoten und Kanten zu entfernen [Z.18-22]
25  for (int i=0; i<A.size()-1; ++i)
   {
27     //jeder fertige Pfad (ausser dem, der wir gerade einfuegten)
       //aus der Liste A wird untersucht, ob er den Teilpfad beinhaltet
29     Path* currAPath = A[i];
       vector<Node*> currSubPathRootPath;

31     //wir bestimmen den Teilpfad
33     //wenn es nicht geht, setzen wir mit dem naechsten fertigen Pfad fort
       if (!currAPath->subPath(currSubPathRootPath, spurNode)) continue;

35     //wenn die Laengen der Teilpfade nicht uebereinstimmen,
37     //setzen wir mit dem naechsten fertigen Pfad fort
       if (rootPath.size() != currSubPathRootPath.size()) continue;

39     //hier wird geprueft,
41     //ob die Pfade dieselbe Knoten an allen Stellen beinhalten
       bool same = true;
43     for (int i=0; i<rootPath.size(); ++i)
       {
45         if (rootPath[i] != currSubPathRootPath[i])
           {
47             same = false;
               break;
49         }
       }
51     if (!same) continue;

53     //wir suchen nach dem folgenden Knoten im aktuellen Pfad,
       //um die Kante, die ihn und den spurNode verbindet
55     //danach zu entfernen
       Node* nextNode = currAPath->getNode(rootPath.size()+1);
57     G->removeEdge(mp(spurNode->getID(), nextNode->getID()));
   }

59  //wir entfernen Kanten und Knoten im aktuellen Pfad [Z. 24f.]
61  for(int i=0; i<currPathLen-1; ++i)
   {
63     G->removeNode(currPath->getNode(i)->getID());
       G->removeEdge(mp(currPath->getNode(i)->getID(), currPath->getNode(i+1)->getID()));
65   }

67  //wir lassen den Dijkstra-Algorithmus uaf

```

```

69 //dem Graphen mit entfernten Kanten und Knoten laufen
//Es es entsteht der Baum
//[Start (s. Ende unten) Z. 28]
71 Dijkstra rDijkstra(G); //der Baum
rDijkstra.getReversedTraversal(endNode);
73
//wir koennen nun die entfernten Kanten und Knoten
75 //auf dem aktuellen Pfad zurueckstellen
bool done = false;
77 for(int i = currPathLen-2; i >= 0 && !done; i--)
{
79     //Zurueckstellung jedes Knotens im aktuellen Pfad
Node* currNode = currPath->getNode(i);
81     G->restoreNode(currNode->getID());

83     //wir sollen ueberpruefen, ob wir in der naechsten Iteration
//aufhoeren sollen, die Kanten und Knoten zurueckzustellen
85     if (currNode->getID() == spurNode->getID())
        done = true;
87

//die Gewichte an den Kanten aus der Menge der outNodes des
89 //aktuellen Knotens werden aktualisiert
//und es entsteht ein Teilpfad vom aktuellen Knoten
91 //bis zum Knoten der bereits einen Vorgaenger hat [Z.28]
Path* newPath = rDijkstra.updateOutDistance(currNode);
93

//wenn ein solcher Pfad entsteht
95 if (newPath != NULL)
{
97     //die gesamte Anzahl an Abbiegungen im aktuellen Pfad
double totalWeight = 0;
99     //die inNodes werden um die entfernten Kanten und Knoten aktualisiert
rDijkstra.updateInDistance(currNode);
101

//der aktuelle Pfad (currPfad) wird zu spurPath kopiert
103 //bis auf den aktuellen Knoten, den wir entfernten
vector<Node*> spurPath;
105 for (int j=0; j<currPathLen; ++j)
{
107     Node* n = currPath->getNode(j);
if (n->getID() != currNode->getID())
109     {
        totalWeight +=
111         G->getRemovedEdgeWeight(currPath->getNode(j),
currPath->getNode(j+1));
113

        spurPath.pb(n);
115     }
else
117     break;
}

119 //newPath wird am Ende von spurPath eingefuegt
121 for (int j = 0; j < newPath->getLength(); ++j)
    spurPath.pb(newPath->getNode(j));
123

//Erstellung eines neuen Kandidaten [Ende (s. Start oben) Z.28]
125 totalPath = new Path(spurPath, totalWeight+newPath->getWeight());

127 //wir stellen sicher, ob es sich genau so
//einen Pfad noch nicht in der Kandidatenliste gibt [Z.33f.]
129 if (pathsToSpurNodes.find(totalPath) == pathsToSpurNodes.end())
{
131     B.insert(totalPath);
pathsToSpurNodes[totalPath] = currNode;
133 }
}

135 //Zurueckstellung jeder Kante im aktuellen Pfad [Z.39]
137 Node* nextNode = currPath->getNode(i+1);
G->restoreEdge(mp(currNode->getID(), nextNode->getID()));
139

//es kann sein, dass das Gewicht an der Kante sich aenderte,

```



```

141     //muessen wir es entsprechend aktualisieren
double edgeWeight = G->getEdgeWeight(currNode, nextNode)
143     + rDijkstra.getStartDistance(nextNode);

145     //Wenn das Gewicht sich veraenderte, muessen
//wir die Entfernung, den Vorgaenger und die
147     //inNodes vom aktuellen Knoten im Baum aktualisieren
if (rDijkstra.getStartDistance(currNode) > edgeWeight)
149     {
        rDijkstra.setStartDistance(currNode, edgeWeight);
151        rDijkstra.setPreviousNode(currNode, nextNode);
        rDijkstra.updateInDistance(currNode);
153    }
}

155
//wir stellen sicher, dass keine
157 //Kanten und Knoten in den Vektoren
//uebrig blieben [Z.38f]
159 G->purgeRemovedEdges();
G->purgeRemovedNodes();

161
//der erste Pfad in der Kandidatenliste wird entfernt [Z. 56]
163 B.erase(B.begin());

165 return currPath;
}

167

169 //Die Funktion aktualisiert die Gewichte (Entfernungen)
//der Knoten, die aus dem Knoten n fuehren
171 Path* Dijkstra::updateOutDistance(Node* n)
{
173     //wir stellen das Geiwcht als den maximalen Wert von double ein
//Indikator dafuer, dass mindestens eine Entfernung veraendert wurde
175     double weight = DBL_MAX;

177     //wir rufen die outNodes des aktuellen Knotens ab
set<Node*>* outNeighbors = new set<Node*>();
179     G->getOutNeighbors(n, *outNeighbors);

181     //wir suchen nach der Entfernung vom Ursprung des aktuellen Knotens
map<Node*, double>::iterator it = startToDistance.find(n);
183     //wenn es ihn nicht gibt, wird er mit dem maximalen Wert
//von double als Entfernung eingefuegt
185     if (it == startToDistance.end())
        it = (startToDistance.insert(mp(n, DBL_MAX))).first;
187

//wir iterieren durch alle Nachbarn, die aus dem aktuellen Knoten fuehren
189 for (set<Node*>::const_iterator it2 = outNeighbors->begin(); it2 != outNeighbors->end(); ++it2)
{
191     //wir stellen die Entfernung des Nachbarn ein
//wenn es ihn nicht gibt, wird der maximale Wert von double zugeordnet
193     map<Node*, double>::const_iterator it3 = startToDistance.find(*it2);
double currDistance = it3 == startToDistance.end() ? DBL_MAX : it3->second;

195
//die aktuelle Entfernung ist um das Gewicht an der Kante
197 //zwischen dem aktuellen Nachbarn und dem aktuellen Knoten vergroesst
currDistance += G->getEdgeWeight(n, *it2);

199
//wir aktualisieren die Entfernung vom Ursprung von n,
//falls sie kleiner ist
201 if (it->second > currDistance)
203 {
        startToDistance[n] = currDistance;
205        previousNode[n] = it3->first;
        weight = currDistance;
207    }
}

209
//wenn die Entfernung von n aktualisiert wurde,
//erstellen wir einen neuen Pfad
211 Path* p = NULL;
213 if (weight < DBL_MAX)

```

```

{
215     vector<Node*> v;

217     //wir fangen mit dem aktuellen Knoten an
    v.pb(n);

219     map<Node*, Node*>::const_iterator it = previousNode.find(n);

221     //alle Knoten, die einen Vorgaenger haben, werden
223     //in den neuen Pfad eingefuegt
    while(it != previousNode.end())
225     {
        v.pb(it->second);
227         it = previousNode.find(it->second);
    }

229     //weight gilt hier als die Gesamtentfernung des Pfades
231     p = new Path(v, weight);
}
233 return p;
}

235 //Die Funktion aktualisiert die Gewichte (Entfernungen)
237 //der Knoten, die zum Knoten n fuehren
void Dijkstra::updateInDistance(Node* n)
239 {
    vector<Node*> v;
241     v.pb(n);

243     while(!v.empty())
    {
245         Node* currNode = *(v.begin());
        v.erase(v.begin());

247         //die aktuelle Entfernung ist die Entfernung des aktuellen Knotens
249         double currDistance = startToDistance[currNode];

251         //wir rufen die inNodes des aktuellen Knotens ab
        set<Node*> inNeighbors;
253         G->getInNeighbors(currNode, inNeighbors);

255         //wir iterieren durch alle Nachbarn, die zum aktuellen Knoten fuehren
        for (set<Node*>::const_iterator it = inNeighbors.begin(); it != inNeighbors.end(); ++it)
257         {
            //wir stellen die Entfernung des Nachbarns ein
            //wenn es ihn nicht gibt, wird der maximale Wert von double zugeordnet
259            map<Node*, double>::const_iterator it1 = startToDistance.find(*it);
            double neighborDistance = startToDistance.end() == it1 ? DBL_MAX : it1->second;

263            //wir aktualisieren die Gesamtentfernung, falls sie kleiner ist
            //als die Entfernung des Nachbarns
            double updatedDistance = currDistance + G->getEdgeWeight(*it, currNode);
265            if (neighborDistance > updatedDistance)
            {
267                startToDistance[*it] = updatedDistance;
                previousNode[*it] = currNode;

271                //wir fuegen den Nachbarn in den Vektor ein,
                //um auf ihn danach Veraenderungen durchzufuehren
273                v.pb(*it);
            }
        }
275     }
}
277 }

```

abbiegen.m

5 Pseudocode aus Wikipedia

Der mehrmals zitierte Pseudocode aus dem englischen Wikipedia-Artikel über den Yen-Algorithmus.

Quelle: https://en.wikipedia.org/wiki/Yen%27s_algorithm#Pseudocode, Zugang 15.03.2020

```

1 function YenKSP(Graph, source, sink, K):
    // Determine the shortest path from the source to the sink.
3   A[0] = Dijkstra(Graph, source, sink);
    // Initialize the set to store the potential kth shortest path.
5   B = [];

11  for k from 1 to K:
    // The spur node ranges from the first node to the
    // next to last node in the previous k-shortest path.
13   for i from 0 to size(A[k - 1]) - 2:

15     // Spur node is retrieved from the previous k-shortest path, k - 1.
    spurNode = A[k-1].node(i);
    // The sequence of nodes from the source to the
    // spur node of the previous k-shortest path.
    rootPath = A[k-1].nodes(0, i);

17     for each path p in A:
19       if rootPath == p.nodes(0, i):
        // Remove the links that are part of the previous
21         // shortest paths which share the same root path.
        remove p.edge(i, i + 1) from Graph;

23     for each node rootPathNode in rootPath:
25       remove rootPathNode from Graph;

27     // Calculate the spur path from the spur node to the sink.
    spurPath = Dijkstra(Graph, spurNode, sink);

29     // Entire path is made up of the root path and spur path.
    totalPath = rootPath + spurPath;
    // Add the potential k-shortest path to the heap.
31     if (totalPath not in B):
33       B.append(totalPath);

35     // Add back the edges and nodes that were removed
    // from the graph.
37     restore edges to Graph;
    restore nodes in rootPath to Graph;

39   if B is empty:
    // This handles the case of there being no spur paths,
    // or no spur paths left.
    // This could happen if the spur paths have already
41     // been exhausted (added to A),
    // or there are no spur paths at all - such as when both
43     // the source and sink vertices
    // lie along a "dead end".
45     break;

    // Sort the potential k-shortest paths by cost.
51   B.sort();
    // Add the lowest cost path becomes the k-shortest path.
53   A[k] = B[0];
    // In fact we should rather use shift since we are
    // removing the first element
55   B.pop();

57 return A

```