Aufgabe 3: Abbiegen

Teilnahme-Id: 52586

Bearbeiter dieser Aufgabe:
Michal Boron

27. März 2020

Inhaltsverzeichnis

1 Lösungsidee

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 Abbiegung bezeichnen wir als *eine Abbiegung von b zu c*. 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).

Mit Hilfe der obengenannten Mengen V und E lässt sich ein gewichteter, 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 Pfades im Graphen G vom Startknoten zum Zielknoten gleich der Summe aller Längen der Straßen auf dem Pfad ist.*

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

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).

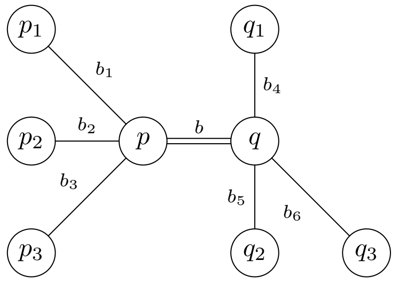


Abbildung 1: Benachbarte Straßen: alle Straßen $b_1, b_2, ..., b_6$ nennen wir *die benachbarten Straßen* der Straße b. Benachbarte Punkte: als *die benachbarten Punkte* des Punkts p gelten p_1, p_2, p_3, q. *Die benachbarten Punkte* von q sind: q_1, q_2, q_3, p.

Beobachtung 2 *In allen verfügbaren Beispielen, die auf der BWINF-Webseite vorhanden sind, entspricht die x-Koordinate des Zielpunkts stets der letzen x-Koordinate der gegebenen Matrix.*

Beobachtung 3 *Außerdem findet man immer in den Beispielen so einen Pfad zum Ziel, der stets von einer Straße $a(p(p_x, p_y), q(q_x, q_y))$ zu einer andern Straße $b(q(q_x, q_y), r(r_x, r_y))$ führt, wobei stets $p_x \le r_x$.*

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

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 3 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–Kooridnaten der zwei übrigen Punkte. Angenommen haben wir eine Abbiegung von einer Straße $a(p, q)$ zu einer anderen Straße $b(q, r)$. Wenn 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)

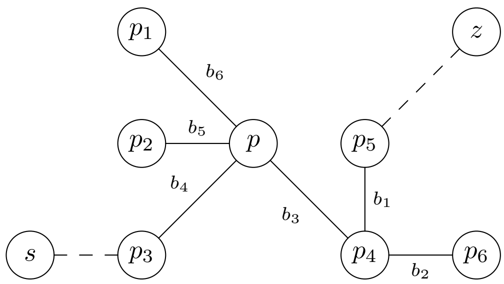


Abbildung 2: Graph G: Von der Straße b_3 kann man nicht zu den Straßen: b_4, b_5, b_6 abbiegen, aber zu den Straßen b_1 und b_2 ist es erlaubt. s und z dienen hier nur für die Orientierung, in welche Richtung die Traversierung verläuft.

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

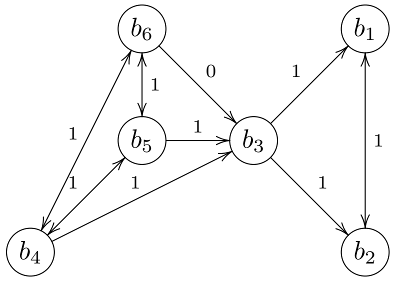


Abbildung 3: Graph G', der anhand der Abb. 2 entstanden ist, mit Abbiegungen als Kanten und Straßen als Knoten.

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 erleichert 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 4 *Die Länge des Pfades in G' vom Startknoten (Start-Pivotstraße) zum Zielknoten (Ziel-Pivotstraße) entspricht der Anzahl der Abbiegungen auf dem Pfad von s zu z.*

Daraus kann man auch eine andere Schlussfolgerung ziehen:

Beobachtung 5 *Die Länge des kürzesten Pfades in G' vom Startknoten (Start-Pivotstraße) zum Zielknoten (Ziel-Pivotstraße) entspricht der minimalen Anzahl der Abbiegungen auf dem Pfades von s zu z.*

Direkt aus der Beobachtung 5 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.
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-Artikel[1] (s. unten) zu finden. Genau dieses Pseudocodes bediente ich mich bei der Implentierung der Aufgabe.

Anhand des Yen–Algorithmus generieren wir die k–kürzesten Pfade $P_1, P_2, ..., P_k$ 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 $P_i, i = 1...k$, 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, dass es sein kann, dass ein Punkt auf der Ebene mehr als dreimal besucht wird. Eine Straße besitzt zwei Punkte. Im Dijkstra–Algorithmus 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

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

generiere, die besten k–kürzesten Pfade sind, heißt, Pfade mit der niedrigsten Anzahl von Abbiegungen. Der beste Pfad ist dementsprechend unser Ergebnis.

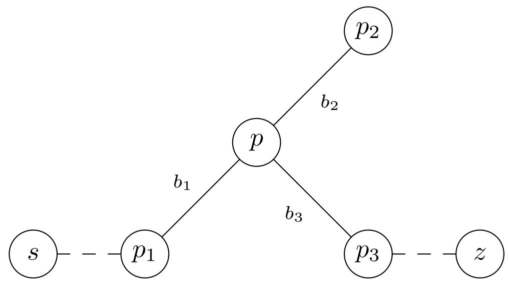


Abbildung 4: p_2 und p_3 besitzen dieselben x–Koordinaten. Im Graphen G' existieren folgende Abbiegungen: von b_1 zu b_2 (mit Wert 0) und von b_1 zu b_3 (mit Wert 1), aber auch von b_2 zu b_3 und andersherum (jeweils mit Wert 1). Das bedeutet, dass es eine Traversierung geben kann, die von b_1 zu b_2 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].

1.1 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
a – Anzahl der allen möglichen Abbiegungen
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: $O(p^2)$ (worst-case).

$O(s) + O(p + s) + O(s) + O(p^2) = O(p^2 + 3s + p) \in O(p^2 + p + s) \in O(p^2 + p)$

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: $O(a)$. 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. Am Ende bekommen wir eine Liste mit Mengen, in denen sich die benachbarten Straßen befinden, also die Mengen mit Abbiegungen: $O(a)$.
- Bildung des Graphen G': $O(a)$. Alle Kanten (Abbiegungen) werden eingefügt.

$O(s) + O(p + s) + O(s) + O(a) + O(a) = O(2a + p + 3s) \in O(a + p + s)$

Yen-Algorithmus:

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

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 Länge des betrachteten Pfades.

- Entfernen der Kanten: $O(k * l)$. Hier wird der Vektor mit allen bisherigen Ergebnissen iteriert. In jeder Iteration erfolgt noch eine Iteration von $rootPath$ (s. Pseudocode, unten): $O(l)$.
- Entfernen der Kanten und Knoten, die im betrachteten Pfad auftreten: $O(l)$.
- Dijkstra auf dem umgekehrten Graphen (s. Umsetzung): $O(s^2)$ (worst-case).
- Aktualisierung der Gewichte und Zurückstellung der Kanten und Knoten: $O(l^2 * n^2)$ (worst-case), wobei n der Anzahl der Nachbarn von einem Knoten entspricht, die in der Regel nicht größer als 5 ist.
  Hier wird der betrachtete Pfad iteriert: $O(l)$. Bei jedem Knoten werden die Knoten und Kanten, die zu diesem Knoten führen, aktualisiert: $O(n)$. Danach werden auch die Knoten und Kanten, die aus dem Knoten führen, aktualisiert: $O(l * n)$. Dabei wird auch ein neuer Pfadkandidat gefunden.

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

Nun addieren wir alles zusammen.

$$O(p^2 + p + s) + O(a + p + s) + O(k^2 l + ckl^2 + s^2) = O(k^2 l + ckl^2 + p^2 + s^2 + a + 2p + 2s)$$
$$O(k^2 l + ckl^2 + p^2 + s^2 + a + 2p + 2s) \in O(k^2 l + ckl^2 + p^2 + s^2 + d) \quad c = n^2, d = a + 2p + 2s$$

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`.

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<int, int>`. Da die Koordinaten von allen Punkten ganzzahlig (und nichtnegativ) sind, entschied ich mich aus dem Grund für Integer. Eine Straße wird ganz einfach als `pair< pair<int,int>, pair<int,int> >` 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 entsprecheden 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 bestimmten 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<int, int>` gespeichert. In beiden Fällen ist der Wert der Abbiegung 0 und als zweiter Wert in `pair<int, int>` gilt genau diese Zahl.
Wenn aber keiner der Punkte ein Pivotpunkt ist, dann müssen wir die x–Koordinaten von a und p vergleichen.
$a_x \leq p_x$, dann fügen wir zur Menge der Straße (a, b) den Index der Straße (b, p) mit dem Wert `currTurn` ein.
$a_x \geq p_x$, 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.

Aufgrund der Beobachtungen 2 und 3 entschied ich mich dafür, dass ich die Pivotpunkte auf folgender Weise platziere: dem Start-Pivotpunkt gebe ich dieselbe y–Koordinate wie die des Startpunkts und als x–Koordinate gebe ich die Zahl um 1 kleiner als die x–Koordinate des Startpunkts. Mit dem Ziel-Pivotpunkt tue ich etwas Ähnliches und die y–Koordinate schreibe ich vom Zielpunkt ab und die x–Koordinate vergröße ich, im Vergleich zum Zielpunkt, um 1.
Natürlich kann man sofort ein Problem bemerken, in dem einer der Pivotpunkte an derselben Stelle platziert ist wie ein anderer, normaler Punkt aus der Eingabe. Dieses Problem tritt in unseren Beispielen gar nicht auf, was durch die erwähnten Beobachtungen unterstützt ist. Außerdem kann man dem Problem einfach begegnen, in dem man den Pivotpunkten nichtganzzahlige Koordinaten zuordnet.

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.

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 enthält, nenne ich `map<Node*, Node*> previousNode`. 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 wird 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 Urspung 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. Ählich 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.

Ein wichtiger Punkt in dieser Klasse ist die Möglichkeit, eine umgekehrte Traversierung laufen zu lassen. Sie erfolgt vom Zielknoten zum Startknoten. Dieses Element wird beim Yen–Algorithmus ausgenutzt, um Pfade im Graphen zu aktualisieren, wenn Kanten und Knoten entfernt wurden. Aus den aktualisierten Kanten und Knoten wird ein neuer Pfad mit Hilfe der Methode `Dijkstra::updateOutDistance()` gefunden, der im Pseudocode des Yen–Algorithmus als *spurPath* beizeichnet wird.

Wie schon beschrieben, basiert meine Implementierung des Yen–Algorithmus sehr stark auf dem Pseudocode im entsprechenden englischsprachigen Wikipedia–Artikel. Praktisch ist die einzige Änderung/Erweiterung nur der Fakt, dass ich einen umgekehren Graphen benutze. Was auch nicht erwähnt wurde, sind die beiden Methoden `Dijkstra::updateInDistance()` und `Dijkstra::updateOutDistance()`, die die Entfernungen in den Pfaden aktualisieren, um neue Pfade generieren zu können. Ich füge den Code des Yen-Algorithmus, sowie die Implementierungen der beiden Methoden im Abschnitt „Quellcode” bei.

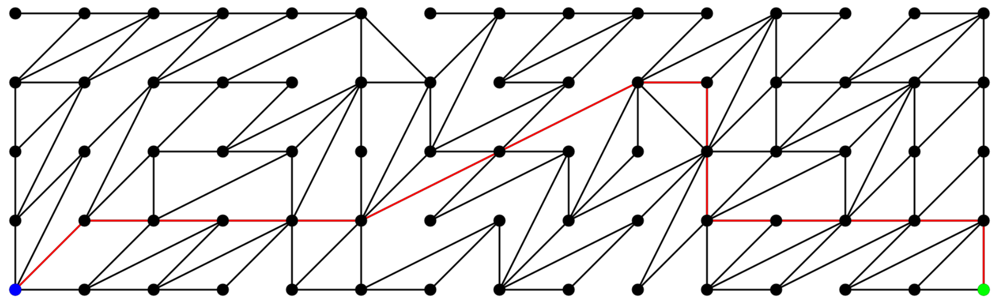
3 Beispiele

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 Pfad nehme.

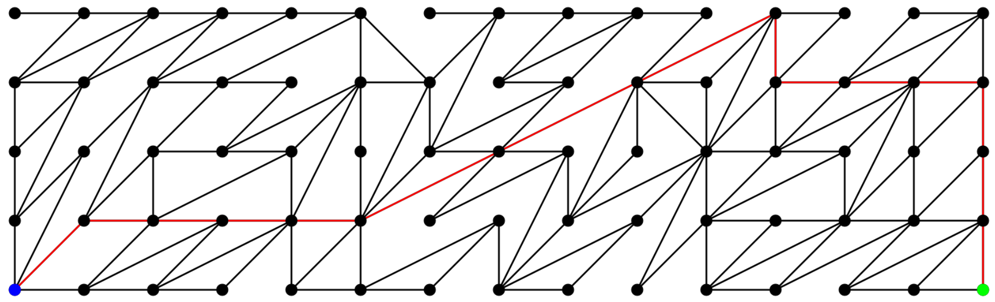
3.1 Beispiel 1 (BWINF)

Textdatei: `abbiegen1.txt`

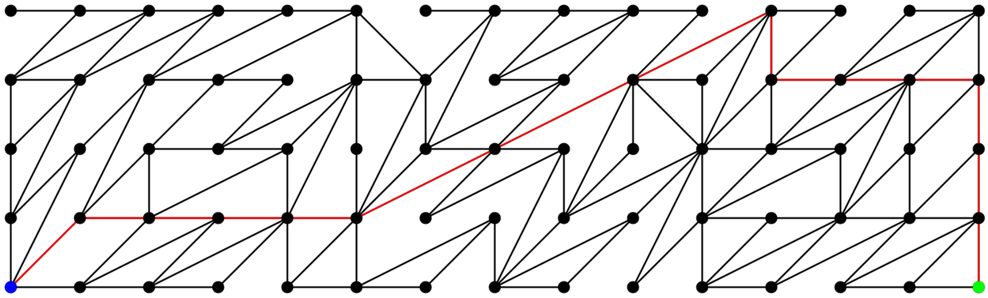
- 110%
  - Abbiegen: **6**
  - Länge des Pfades: **17,8863**
  - Länge des Pfades prozentual: **104,462%**
  - Punkte: **(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (10, 2), (10, 1), (11, 1), (12, 1), (13, 1), (14, 1), (14, 0)**



- 115%
  - Abbiegen: **5**
  - 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)**



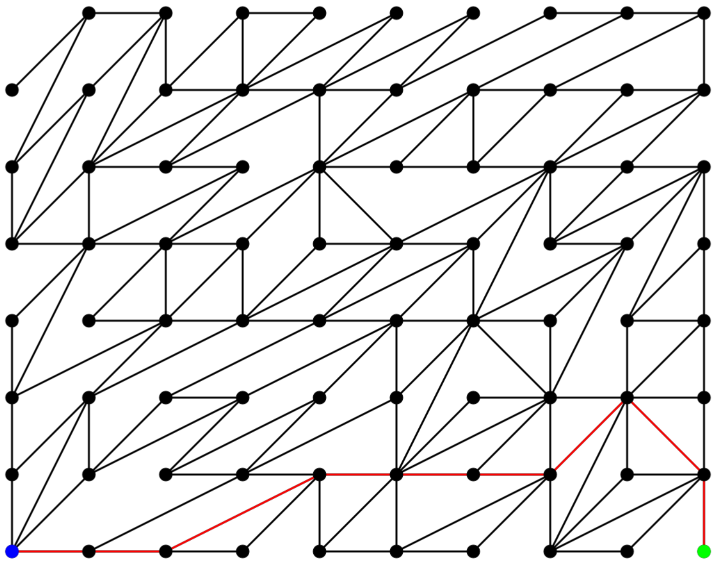
- 130%
  - Abbiegen: **5**
  - Länge des Pfades: **19,1224**
  - Länge des Pfades prozentual: **111,6%**
  - Punkte: **(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (11, 4), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)**



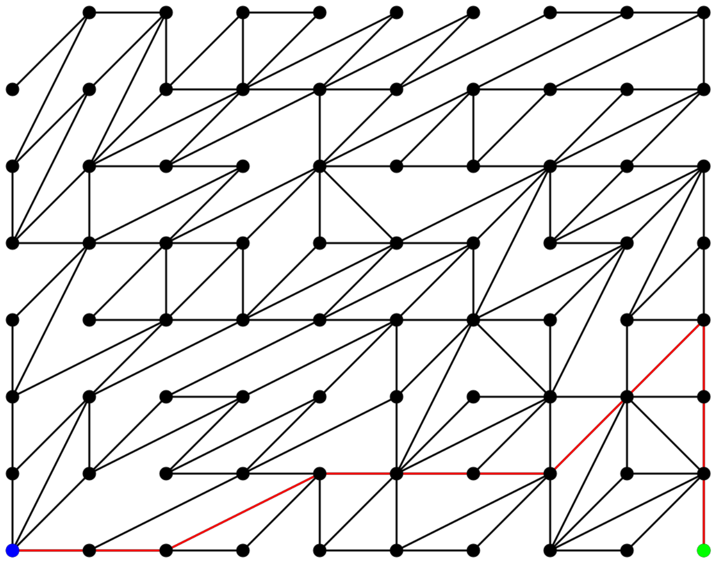
3.2 Beispiel 2 (BWINF)

Textdatei: `abbiegen2.txt`

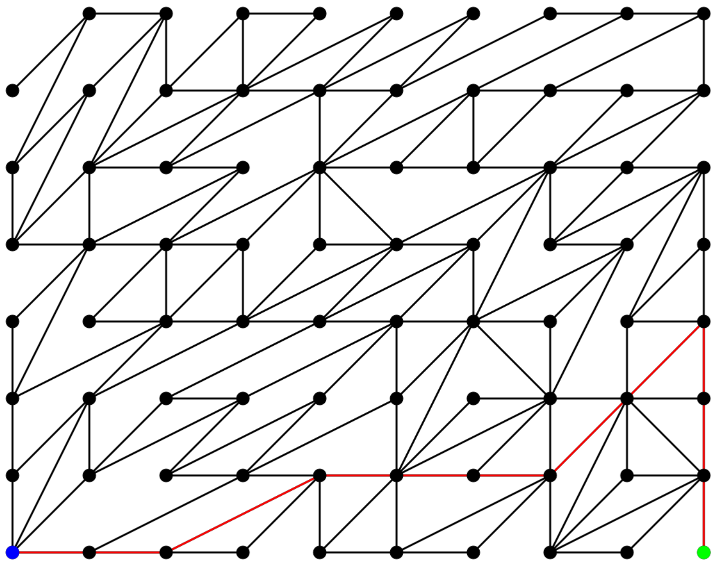
- 110%
  - Abbiegen: **5**
  - Länge des Pfades: **11,0645**
  - Länge des Pfades prozentual: **101,636%**
  - Punkte: **(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 1), (9, 0)**



- 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)**



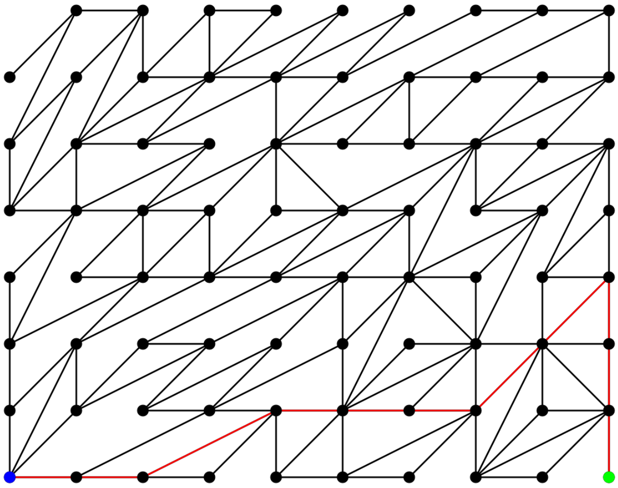
- 130%
  - Abbiegen: **4**
  - Länge des Pfades: **13,0645**
  - Länge des Pfades prozentual: **120,008%**
  - Punkte: **(0, 0), (1, 0), (2, 0), (4, 1), (5, 1), (6, 1), (7, 1), (8, 2), (9, 3), (9, 2), (9, 1), (9, 0)**



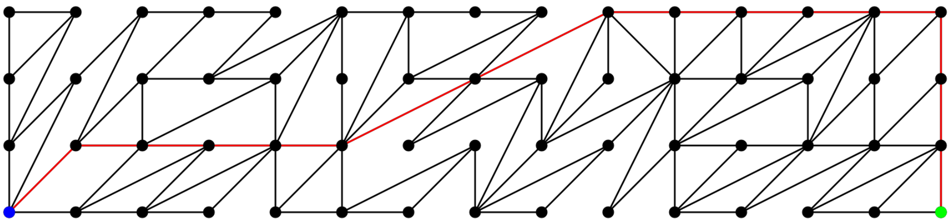
3.3 Beispiel 3 (BWINF)

Textdatei: `abbiegen3.txt`

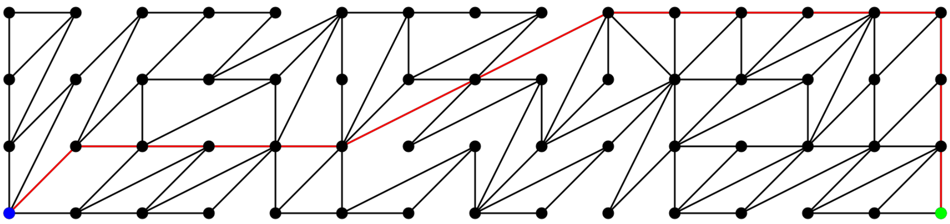
- 110%
  - Abbiegen: **4**
  - Länge des Pfades: **17,8863**
  - Länge des Pfades prozentual: **104,462%**
  - Punkte: **(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (7, 2), (9, 3), (10, 3), (11, 3), (12, 3), (13, 3), (14, 3), (14, 2), (14, 1), (14, 0)**



- 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)**

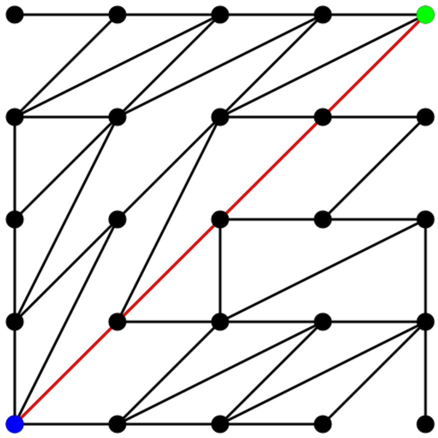


3.4 Beispiel 4

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**

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

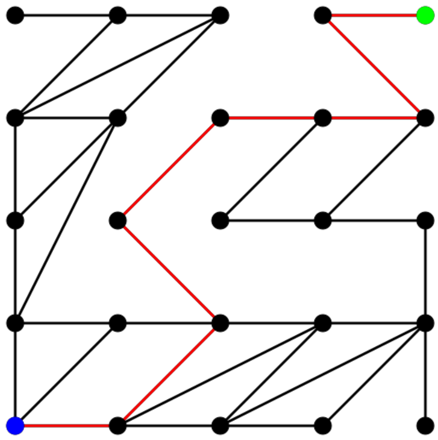


- 115%: das Gleiche wie bei 110%
- 130%: das Gleiche wie bei 110%

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**
  - 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%

4 Quellcode

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

```

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

  //der erste Pfad aus der Kandidatenliste wird betrachtet
  Path* currPath = *(B.begin());
  int currPathLen = currPath->getLength();

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

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

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

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

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

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

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

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

  //wir entfernen Kanten und Knoten im aktuellen Pfad
  for(int i=0; i<currPathLen-1; ++i)
  {
    G->removeNode(currPath->getNode(i)->getID());
    G->removeEdge(mp(currPath->getNode(i)->getID(), currPath->getNode(i+1)->getID()));
  }

  //wir lassen den Dijkstra-Algorithmus
  //auf einen umgekehrten Graphen laufen,
```

```
//um in den veraenderten Graphen
//neue Pfade zu finden
Dijkstra rGraph(G);
rGraph.getReversedGraph(endNode);

//wir koennen nun die entfernten Kanten und Knoten
//auf dem aktuellen Pfad zurueckstellen
bool done = false;
for(int i = currPathLen-2; i >= 0 && !done; i--)
{
  //Zurueckstellung jedes Knotens im aktuellen Pfad
  Node* currNode = currPath->getNode(i);
  G->restoreNode(currNode->getID());

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

  //die Gewichte an den Kanten aus der Menge der inNodes des
  //aktuellen Knotens werden aktualisiert
  //und es entsteht ein Teilpfad vom aktuellen Knoten
  //bis zum Knoten der bereits einen Vorgaenger hat
  Path* totalPath = rGraph.updateOutDistance(currNode);

  if (totalPath != NULL)
  {
    pathNum++;

    //die gesamte Anzahl an Abbiegungen im aktuellen Pfad
    double totalWeight = 0;
    //die inNodes werden um die entfernten Kanten und Knoten aktualisiert
    rGraph.updateInDistance(currNode);

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

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

    //totalPath wird am Ende von prePath eingefuegt
    for (int j = 0; j < totalPath->getLength(); ++j)
      spurPath.pb(totalPath->getNode(j));

    //Erstellung eines neuen Kandidaten
    totalPath = new Path(spurPath, totalWeight+totalPath->getWeight());

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

  //Zurueckstellung jeder Kante im aktuellen Pfad
  Node* nextNode = currPath->getNode(i+1);
  G->restoreEdge(mp(currNode->getID(), nextNode->getID()));

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

```
    //muessen wir es entprechend aktualisieren
    double edgeWeight = G->getEdgeWeight(currNode, nextNode)
      + rGraph.getStartDistance(nextNode);

    //Wenn das Gewicht sich veraenderte, muessen
    //wir die Entfernung, den Vorgaenger und die
    //inNodes vom aktuellen Knoten aktualisieren
    if (rGraph.getStartDistance(currNode) > edgeWeight)
    {
      rGraph.setStartDistance(currNode, edgeWeight);
      rGraph.setPreviousNode(currNode, nextNode);
      rGraph.updateInDistance(currNode);
    }
  }

  //wir stellen sicher, dass keine
  //Kanten und Knoten in den Vektoren
  //uebrig blieben
  G->purgeRemovedEdges();
  G->purgeRemovedNodes();

  //der erste Pfad in der Kandidatenliste wird entfernt
  B.erase(B.begin());

  return currPath;
}


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

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

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

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

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

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

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

```
  {
    vector<Node*> v;

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

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

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

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

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

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

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

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

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

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

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

abbiegen.m

5 Pseudocode aus Wikipedia

Das 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

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

   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.
       for i from 0 to size(A[k - 1]) - 2:

           // 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);

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

           for each node rootPathNode in rootPath except spurNode:
               remove rootPathNode from Graph;

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

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

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

       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
           // been exhausted (added to A),
           // or there are no spur paths at all - such as when both
           // the source and sink vertices
           // lie along a "dead end".
           break;
       // Sort the potential k-shortest paths by cost.
       B.sort();
       // Add the lowest cost path becomes the k-shortest path.
       A[k] = B[0];
       // In fact we should rather use shift since we are
       // removing the first element
       B.pop();

   return A
```