Aufgabe 3: Abbiegen?

Teilnahme-Id: 53280

Bearbeiter/-in dieser Aufgabe:
Raphael Gaedtke

10. April 2020

Inhaltsverzeichnis

1 Lösungsidee

Definition 1 *Es sei der Weg A der Weg durch den Graphen, der die geringste Anzahl an Abbiegevorgängen hat. Analog steht der Weg S für den Weg, der am kürzesten ist (gibt es für einen der beiden Wege mehrere Möglichkeiten, wird eine beliebige davon ausgewählt). Der Weg K soll dann den ausgegebenen Weg bezeichnen. Es kann auch*

$$A = K, \tag{1}$$

$$S = K \tag{2}$$

oder sogar

$$A = S = K \tag{3}$$

gelten. Ein Weg ist durch eine (geordnete) Menge von Knoten bestimmt (s. u.).
Für die Anzahl der Abbiegevorgänge auf den drei Wegen werden die Platzhalter a, s und k gewählt. Es muss $a, s, k \in \mathbb{N}$ gelten.

1.1 Modellierung der Straßenkarte und der Wege

Die Straßenkarte soll als gewichteteter, ungerichteter Graph modelliert werden. Als Ecken des Graphens fungieren die Kreuzungen auf der Straßenkarte, die Kanten stellen die Verbindungstraßen dieser Kreuzungen dar. Die Anzahl der Ecken soll mit n bezeichnet werden.
Jede Kreuzung ist eindeutig durch ihre Koordinaten bestimmt. Durch diese Koordinaten lassen sich auch die Längen der die Kreuzungen verbindenden Straßen berechnen.
Der Graph wird gespeichert als eine Menge von nummerierten Ecken und eine Gewichtungsmatrix der Größe $n \times n$, die die Kanten und deren Länge festhält.
In dieser Matrix geben Zeilen- und Spaltennummerierung für die Liste der Knoten die Indizes der Knoten, die die Endpunkte einer Kante sind, an. Eingetragen wird dann die Länge der entsprechenden Kante. Existiert zwischen zwei Knoten keine Kante, wird -1 in die Matrix eingetragen.
Wege durch den Graphen werden als eine Liste von Knoten gespeichert. Obwohl ein „Weg“ üblicherweise als eine Liste von Kanten definiert ist, ist das hier kein Problem, weil es keine Mehrfachkanten geben soll.

1.2 Finden des kürzesten Weges

Um Bilal einen Weg vorzuschlagen, müssen der Weg S und insbesondere dessen Länge bekannt sein. Ohne diese Größe kann die maximale Länge des auszugebenden Weges, der sich aus der Länge des Weges S und der durch den Nutzer vorgegebenen, maximalen Verlängerung berechnen lässt, überhaupt nicht bestimmt werden.
Zur Auffindung des kürzesten Weges wird der Dijkstra-Algorithmus verwendet, der mit verhältnismäßig geringem Rechenaufwand den Weg zwischen einem vorgegebenen Start- und einem Zielknoten zurückliefert, auf dem die Summe der Kantengewichte (diese werden hier als „Kosten“ bezeichnet) am geringsten ist. Als Kantengewichte dienen hier die zuvor aus den Koordinaten der Punkte berechneten Längen der Straßen.

Der Dijkstra-Algorithmus berechnet dann von Knoten zu Knoten die momentan günstigste Route, wobei er auch Verbesserungen vornehmen kann. Dies tut er, bis er alle Knoten besucht hat und kein besserer Weg mehr gefunden werden kann.
Zur Durchführung des Algorithmus wird eine Warteschlange verwendet, in der sich zu Beginn nur der Startknoten befindet. Für jeden Knoten werden außerdem die momentan geringsten Kosten gespeichert, am Anfang hat der Startknoten die Kosten 0 (der Weg vom Startknoten zum Startknoten kann keine Länge haben) und alle anderen die Knoten vorerst unendliche Kosten.
Bis die Warteschlange leer ist, werden nun immer wieder folgende Schritte durchgeführt:

1. Der vorderste Knoten wird aus der Warteschlange genommen (Der erste Knoten, der so untersucht wird, muss also der Startknoten sein).

2. Dann werden alle Nachbarknoten des ausgewählten Knotens betrachtet und für jeden dieser Knoten überprüft, ob die Summe der Kosten des aus der Warteschlange genommen Knotens und der Kosten der Verbindungskante geringer als die bisher für den Knoten gespeicherten Kosten sind.

3. Ist dieses Bedingung erfüllt, werden die Kosten des entsprechenden Nachbarknotens auf den neuen Wert gesetzt.

4. Dann wird noch überprüft, ob sich der aktualisierte Knoten in der Warteschlange befindet (dabei ist es unerheblich, ob er sich schon einmal dort befunden hat).

5. Falls dies der Fall ist, wird der Knoten in die Warteschlange eingefügt. Auf diese Weise können auch Verbesserungen vorgenommen werden, weil so die Nachbarknoten des aktualisierten Knotens ebenfalls geringere Kosten erhalten.

Sobald die Warteschlange leer ist, wurden für jeden Knoten die minimalen Kosten vom Startknoten aus berechnet. Das Ergebnis bilden dann die Kosten des Zielknotens.
Mithilfe des Dijkstra-Algorithmus werden allerdings auch die den Kosten zugehörigen Wegen ermittelt, wenn der Computer sich für jeden Knoten den bisher günstigsten Weg merkt und dies ebenfalls entsprechend aktualisiert.
Für den weiterem Programmverlauf wird in diesem Fall auch der Weg S selbst benötigt. Durch ihn wird auch die Zahl s bestimmt.

1.3 Finden des Weges mit der geringsten Zahl an Abbiegevorgängen

Auch ein Verfahren zur Bestimmung des Weges A soll implementiert werden. Dessen Kenntnis selbst ist zwar nicht nötig, das Verfahren zu seiner Bestimmung wird aber im weiteren Verlauf genutzt. Dafür wird der Dijkstra-Algorithmus für das Finden des kürzesten Weges adaptiert.
Bei der Verwendung des in Kapitel 1.2 beschriebenen Algorithmus ergibt sich allerdings das Problem, dass die Anzahl der Abbiegevorgänge für einen bestimmten Weg nicht als fixe Kosten einer Kante gespeichert werden können, da das Entstehen eines Abbiegevorgangs beim Betreten einer neuen Kante auch von der zuletzt genutzen Kante abhängt.

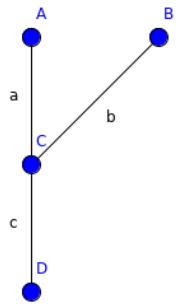

Abbildung 1: Ob beim Betreten von Kante c vom Knoten C aus abgebogen werden muss, hängt davon ab, ob zuvor Kante a oder Kante b benutz wurde. Kante c kann also keine fixen Kosten haben.

Also müssen die Kosten bei jedem neuen Knoten auf einem anderen Weg berechnet werden. Hierzu wird für jeden Weg auch die zuletzt genutzte Kante gespeichert (diese ist implizit auch gegeben, wenn der jeweils günstigste Weg gespeichert wird, weil die letzte Kante dann durch den vorletzten und den letzten Knoten dieses Weges bestimmt wird).
Bei der Berechnung der Kosten für einen Knoten können dann zwei Fälle auftreten:

1. Von den verschiedenen Kanten, über die die Ecke erreicht werden kann, soll zunächst nur eine zu den bis dahin geringsten Kosten führen. In diesem Fall muss auch nur diese Kante gespeichert werden, auch wenn eine andere Kante dazu führen würde, dass an dieser Ecke für eine weiterführende Kante nicht abgebogen werden muss. Dies würde zwar die Kosten um 1 erhöhen, die andere Kante hat aber onehin schon eine Kostendifferenz von mindestens 1.

2. Im anderen Fall haben mehrere Kanten gleichzeitig die bis dahin geringsten Kosten. In diesem Fall müssen alle diese Kanten gespeichert werden. Beim Betrachten des Punktes wird dabei diejenige unter den gespeicherten Kanten ausgewählt, die die geringsten Kosten beim weiteren Abbiegen verursacht. Macht die Wahl der entsprechenden Kante keinen Unterschied, wird zufällig eine ausgewählt.

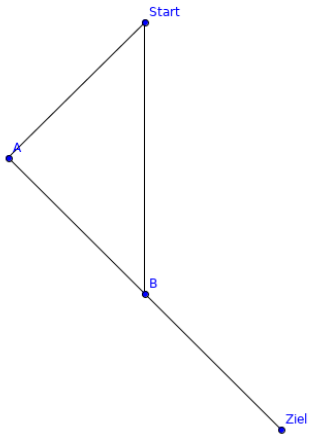

Abbildung 2: In diesem Graphen kann Kreuzung B über zwei verschiedene Kanten erreicht werden, von denen die Kante vom Startknoten aus die geringsten Kosten hat. Obwohl die Kante von Kreuzung A aus die geringsten Abbiegekosten bei B verursachen würde, kann die andere Kante gespeichert werden, weil die Abbiegekosten dann um 1 steigen würde, wobei 1 die Mindestdifferenz zu allen anderen möglichen Kanten darstellt.

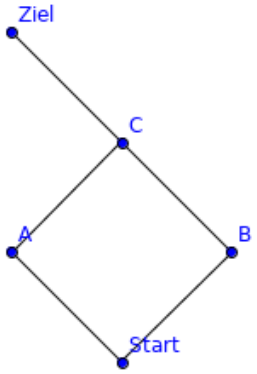

Abbildung 3: Kreuzung C kann in diesem Graphen von zwei Kanten erreicht werden, nämlich von den Kreuzungen A und B aus. Von Kreuzung B aus kommend, muss zum Erreichen des Ziels allerdings einmal weniger abgebogen werden, also müssen zunächst beide Kanten gespeichert werden.

Diese Unterscheidungen werden bei jedem Erreichen einer Kreuzung gemacht, ansonsten unterscheidet sich der Algorithmus nicht von dem in Kapitel 1.2 beschriebenen.
Auf dem so gefundenen Weg muss mindestens einmal abgebogen werden, sonst ist er bereits der kürzeste Weg. Existieren zwischen zwei Punkten mehrere in der Anzahl der Abbiegevorgänge gleichwertige Wege, wird derjenige ausgegeben, der zufälligerweise zuerst gespeichert wurde.

1.4 Finden des auszugebenden Weges

Nachdem nun die Länge des kürzesten Weges und die Wege S und A bekannt sind, soll nun ein Weg, der mit möglichst wenigen Abbiegevorgängen die maximale Weglänge unterschreitet, mit möglichst geringem Rechenaufwand gefunden werden.
Für die Anzahl der Abbiegevorgänge k dieses Weges K muss

$$a \leq k \leq s \tag{4}$$

gelten. Damit ist eine obere Schranke für k gegeben.
Der Computer müsste also alle Wege, auf denen es bis zu $s-1$ Abbiegevorgänge gibt, berechnen. Von ihnen wird dann derjenige ausgwählt, der mit der geringsten Anzahl von Abbiegevorgängen die maximale Weglänge unterschreitet. Findet sich kein Weg, der den Zielknoten mit $s-1$ oder weniger Abbiegevorgängen erreicht, wird der Weg S ausgegeben, da s dann die minimale Anzahl an Abbiegevorgängen ist und der Weg S auf jeden Fall der kürzeste Weg mit dieser Anzahl an Abbiegevorgängen sein muss, weil er insgesamt der kürzeste Weg ist.

1.5 Bearbeitung des Graphen

Das Ausprobieren aller Wege mit bis zu $s-1$ Abbiegevorgängen, wie es in Kapitel 1.4 beschrieben ist, würde immer noch viel Rechenaufwand erfordern, weil die Wege von Kreuzung zu Kreuzung berechnet werden müssten und die Zahl der auf einem solchen Weg liegenden Kreuzungen sehr hoch ist. Außerdem muss beim Hinzufügen einer Kreuzung zu einem Weg immer wieder berechnet werden, wie viele Abbiegevorgänge auf dem Weg zu ihr liegen, weil nicht bekannt ist, zwischen welchen Kreuzungen abgebogen werden muss, und zwischen welchen nicht.
Um diesen Rechenaufwand deutlich zu verringern, soll der die Straßenkarte darstellende Graph so bearbeitet werdem, dass bei jedem Schritt abgebogen werden *muss*, wobei die Länge der Wegstrecken zwischen den Kreuzungen und alle möglichen Wege erhalten bleiben sollen. Damit muss der Computer nur noch Wege mit einer Länge von bis zu s Schritten ausprobieren und auch das Zählen von Abbiegevorgängen ist nicht mehr nötig, weil diese Anzahl einfach durch die Anzahl der bisher besuchten Knoten gegeben ist.

Um dieses Ziel zu erreichen, müssen bei der bestehenden Knotenmenge Kanten entfernt und dafür andere Kanten hinzugefügt werden. Die Prozedur zur Bearbeitung des Graphen soll an folgendem Beispielgraphen veranschaulicht werden:

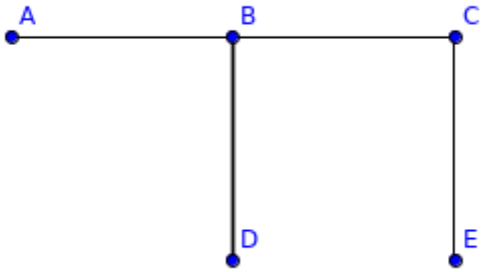

Abbildung 4: Dieser Beispielgraph hat 5 Knoten, von denen drei (nämlich die Knoten A, B und C) so in einer Reihe liegen, dass Bilal von A nach B und von dort nach C fahren kann, ohne abzubiegen. Zwei dieser drei Knoten sind noch adjazent zu je einer weiteren Kante, für die auf jeden Fall abgebogen werden muss.

Zunächst wird überprüft, ob es Knoten gibt, die nicht nebeneinanderliegen, zwischen denen aber nicht zwangsläufig abgebogen werden muss. Um dies zu überprüfen, kommt die in Kapitel 1.3 beschriebende Dijkstra-Adaption zum Einsatz. Diesem wird jedes Punktepaar des Graphen einmal als Start- und Zielknoten übergeben. Hat der zurückgelieferte Wert dann keine Abbiegevorgänge, bilden diese Teile oder Endpunkte einer Punktekette, auf der Bilal geradeaus fahren könnte.
Nicht überprüft werden hierbei Knotenpaare, die einander adjazent sind.

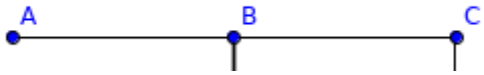

Abbildung 5: Die aus dem Graph in Abbildung 4 separierte Punktekette

Die Kanten dieser Kette werden entfernt, um die Punktekette zu unterbrechen. Damit ist es nicht mehr möglich, in mehreren Schritten ohne Abbiegevorgang von einem Punkt dieser Kette zu einem anderen zu gelangen.

Abbildung 6: Die Kanten der Knotenkette wurden entfernt.

Dann werden dem Graphen Kanten hinzugefügt, um alle vorher möglichen Wege auch im resultierenden, bearbeiteten Graphen zu erhalten.
Dieses Problem könnte auch gelöst werden, indem die an einer Kette beteiligten Knoten dupliziert würden. Jeder der Duplikate eines Knotens hätte die gleichen Koordinaten und außerhablb der Kette die gleichen Kanten, jedes der Duplikate wäre aber mit einem anderen Knoten aus der Kette verbunden.
Dieses Vorgehen ist allerdings nicht praktikabel, weil die Anzahl der Knoten damit stark erhöht werden und somit viel zu viel Speicherplatz verbrauchen würde. Deswegen sollen alle Knoten wie im Originalgraphen bestehen bleiben.
Um ohne Ketten alle Wege ermöglichen zu können, muss für die Kanten eine zweite Gewichtungsmatrix eingeführt werden. Diese kann für jede Kante eine Anzahl von Abbiegevorgängen als zusätzliche Kosten speichern.
Die Kette im Graphen wird also ersetzt, indem jeder Knoten der Kette mit jedem zu einem anderen Knoten der adjazenten verbunden und das Gewicht in der zweiten Gewichtungsmatrix auf 1 gesetzt wird.
Auf diese Weise können auch zwei (oder mehr) Ketten behandelt werden, die einen gemeinsamen Punkt haben. In diesem Fall sind die resultierenden Gewichte in der Gewichtungsmatrix je nach Anzahl der verbundenen Ketten einfach länger.
Bei diesem Verfahren kann die Anzahl der Knoten nicht erhöht werden und die Anzahl der Kanten den Wert n^2 nicht übersteigen. Damit ist die Größe des resultierenden Graphen von vorneherein limitiert.
Beim Ausprobieren der verschiedenen Wege durch den Graphen werden die in der zweiten Matrix mit einem Gewicht größer 0 versehenen Wege als mehrere implizit gemachte Schritte behandelt. Die Länge der Kanten zwischen im Originalgraphen nicht vorhandenen Kanten ist die Summe der vorher an diesem Weg beteiligten Kanten.

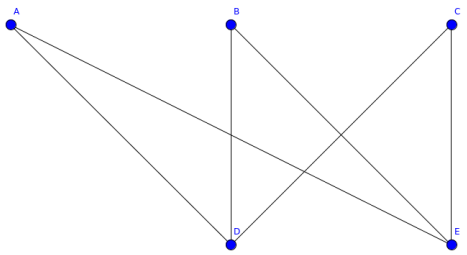

Abbildung 7: Der fertig bearbeitete Graph aus Abbildung 4. Alle im Bild diagonal verlaufenden Kanten haben in der zweiten Gewichtungsmatrix einen Wert von 1.

1.6 Weitere Optimierungen

Ist der Graph wie in Abbildung 7 bearbeitet, werden alle Wege mit bis zu s Schritten durch den Graph ausprobiert (Ein „Schritt“ ist ein Wechsel von einem Knoten zu einem ihm adjazenten Knoten). Sobald dabei der Zielknoten erreicht ist, wird der Prozess abgebrochen und der entsprechende Weg zum Zielknoten ausgegeben.

Dieses Verfahren zur Berechnung des Weges K kann allerdings noch optimiert werden. Für eine Verkürzung der Rechenzeit wird das in Kapitel 1.3 beschriebene Verfahren zur Berechnung des Weges mit der geringsten Anzahl an abbiegevorgängen zwischen zwei Knoten verwendet. Dieses wird vor dem Ausprobieren möglicher Wege für alle Knoten außer dem Start- und dem Zielknoten verwendet. Von diesen wird jeder einmal als Startknoten genutzt, und auf diese Weise die geringste Anzahl an Abbiegevorgängen zwischen diesem Knoten und dem Zielknoten berechnet. Wird ein Punkt dann an einem Punkt des Verfahrens mit einer gewissen Anzahl an Schritten erreicht, während die Summe dieser Schritte und des so für diesen Punkt gespeicherten Wert größer als s ist, wird der betrachtete Weg sofort aussortiert. Damit wird die Berechnung uninteressanter Wege bereits frühzeitiger abgebrochen.

2 Umsetzung

Die Lösungsidee wird in C++ umgesetzt.

2.1 Modellierung, Eingabe und Speicherung

Im Programm wird der Graph, der durch eine Menge von Punkten und eine Gewichtungsmatrix modelliert werden soll, in einem Container des Typs „vector“, der die Knoten aufnimmt, und einem zweidimensionalen Array des Datentyps „double“ für die Gewichtungsmatrix gespeichert. Für die Matrix ist die Speicherung in einem Array unproblematisch, da auf dieser Matrix keine Operationen durchgeführt werden, sie bleibt genau wie der Container für die Knoten bzw. Kreuzungen nach dem Einlesen der Daten aus der Eingabedatei unverändert.
Der Container „nodes“, speichert für die Knoten Variablen des Datentyps „crossing“, der eigens für die Kreuzungen neu definiert wird. Dieser ist als „struct“ definiert und hat zwei Membervariablen, die den zwei Koordinaten einer Kreuzung zugeodrnet werden.
Die Wege durch den Graphen werden ebenfalls als Container des Typs „vector“ gespeichert, die allerdings Integer und nicht Variablen des Datentyps für die Kreuzungen aufnehmen. Jeder Integer repräsentiert dabei den Knoten mit dem entsprechenden Index im „nodes“-Container.
Beim Einlesen der Daten ergibt sich das Problem, dass die Anzahl der Punkte, die auch die Größe des Arrays für die Kanten bestimmt, nicht zu Beginn bekannt ist. Deswegen wird statt der Matrix für die Kanten zunächst ein Container des Typs „vector“ verwendet, der Variablen vom Typ „bool“ speichert, die nur angeben, ob zwischen zwei Punkten eine Kante existiert. Bei jeder Kante wird überprüft, ob die Punkte, zwischen denen sich befindet, bereits in diesem Container gespeichert sind. Ist dies nicht der Fall, werden die Punkte im Container „nodes“ gespeichert und der Vektor für die Kanten entsprechend vergrößert, bevor die Existenz der Kante eingetragen wird. Nach dem Beenden des Einlesens wird dieser Container dann in ein zweidimensionales Array überführt, das dann auch die jeweiligen Kantenlängen speichert.

2.2 Umsetzung des Dijkstra-Algorithmus

Beide Ausführungen des Dijkstra-Algorithmus (die ursprüngliche Version unter Berücksichtigung der Kantengewichte und die Adaption zum Finden des Weges mit der geringsten Anzahl von Abbiegevorgängenn) werden in einer Funktion ausgeführt, die die Länge des gefundenen Weges zurückliefert und den Weg selbst via Zeiger speichert.
In diesen Funktionen werden die Warteschleife als Container und die Kosten der Knoten als Array lokal gespeichert, wobei jeder Eintrag des Arrays für die Kosten des Knotens mit dem gleichen Index im „nodes“-Container.
Eine while-Schleife wird dann ausgeführt, bis sich kein Knoten mehr in der Warteschlange befindet. In dieser Schleife wird zunächst der jeweils erste Knoten aus der Warteschlange genommen, die anderen Knoten um einen Platz nach vorne verschoben und das letzte Element gelöscht. Für die ausgewählte Kreuzung wird dann eine for-Schleife ausgeführt, die für jeden Knoten überprüft, ob eine Kante zwischen

diesem und dem aus der Warteschlange entnommen existiert.
Ist dies der Fall, werden die Kosten des resultierenden Weges über die betrachtete Kante mithilfe einer entsprechenden Funktion berechnet. Sind diese geringer als die bisher für diese Kreuzung gespeicherten, werden diese aktualisiert und der neue Weg für den weiteren Knoten gespeichert. Für die Speicherung der Wege wird ein Array von Containern lokal definiert, das, genau wie das Array für die Kosten, jeweils den Weg für den Knoten mit dem gleichen Index speichert. Zum Schluss werden die Kosten des Weges zum Zielknoten und der dazugehörige Weg zurückgeliefert.
Um dies auch für den Weg mit der geringsten Anzahl der Abbiegevorgänge zu adaptieren, muss der ursprüngliche Vorgang angepasst werden: Zur Berechnung der Kosten muss eine andere Funktion verwendet werden, die die Anzahl der Abbiegevorgänge für einen bestimmten Weg bestimmt. In jedem Schritt muss also ein potenziell besserer Weg erstellt werden, den diese Funktion dann überprüft.
Die dies bestimmende Funktion betrachtet damit alle direkt aufeinanderfolgende Kantenpaare des betrachteten Weges und vergleicht deren Neigung zur X-Achse. Ist diese für ein direkt aufeinanderfolgendes Kantenpaar unterschiedlich, wird die Anzahl der Abbiegevorgänge in einem Zähler um 1 erhöht. Nach dem Abgleichen aller direkt aufeinanderfolgenden Kantenpaare wird dann der resultierende Wert des Zählers zurückgegeben.
Schlussendlich müssen für jeden Knoten eventuell mehrere Wege gespeichert werden können. An die Stelle des Arrays, das in der Originalfunktion die Wege in Containern gespeichert hat, tritt hier ein Array, das Container von Containern, die Wege repräsentieren, ablegt. Dafür muss dann auch die vormalig einfache for-Schleife zu einer verschachtelten for-Schleife erweitert werden, die dann alle möglichen Wege bis zum ersten Knoten aus der Warteschlange durchgeht.
Existieren am Ende mehrere gleichwertige Wege zum Zielknoten wird der zufällig erste von ihnen übergeben.

2.3 Bearbeiten des Graphen

Nach dem Einlesen der Daten aus der Eingabedatei wird der Graph wiederum in einer eigenen Funktion bearbeitet. Die Kanten des bearbeiteten Graphen werden in einem eigenen Array, also einer zweiten Gewichtungsmatrix gespeichert. Hinzu kommt noch eine Matrix des Typs int, die die Anzahl der Abbiegevorgänge bei einzelnen Kanten in der neuen Kantenmatrix speichert.
In der besagten Funktion werden zunächst alle Knotenpaare des Graphen durchgegangen und festgestellt, ob sie in einer gemeinsamen Kette von Knoten liegen, ohne Nachbarn zu sein. Dafür wird durch die Dijkstra-Adaption für den Weg mit der geringsten Anzahl an Abbiegevorgängen der Weg mit der geringsten Anzahl an Abbiegevorgängen zwischen diesen beiden Knoten festgestellt und überprüft, ob der zurückgelieferte Weg Abbiegevorgänge hat.
Ist dies nicht der Fall, wurde eine Kette mit einer Mindestlänge von drei Knoten gefunden. Um die komplette Kette aufzufinden, werden zunächst alle Knoten auf dem kürzesten Weg zwischen den beiden überprüften Knoten hinzugefügt. Danach wird nach Knoten gesucht, die nach auf der durch die Kette definierten Gerade liegen und mit den vorläufigen Endknoten der Kette verbunden sind. Sind alle diese Knoten gefunden, wird die Kette in einem Container gespeichert.
Für jedes Knotenpaar der Kette wird zudem in einem bool-Array markiert, dass dieses Knotenpaar nicht mehr überprüft werden muss. So werden Knotenpaare nicht unnötigerweise überprüft und die gleiche Kette kann nicht mehrmals „gefunden“ werden.
Im nächsten Schritt werden dann alle Kanten der Kette aus der zu überarbeiteten Kantenmatrix entfernt und durch Kanten, die einen Kettenknoten mit den Nachbarn anderer Kettenknoten verbinden, ersetzt.
Um einen Weg im bearbeiteten Graphen später wieder auf einen Weg des Originalgraphens zurückführen zu können, wird während dieses Verfahrens noch ein zweidimensionales Array von Containern initialisiert. Dieses Array „bridges“ speichert die Wege des Originalgraphen, die den Kanten im bearbeiteten Graphen entsprechen.

2.4 Finden des auszugebenden Weges

Der für Bilal zu empfehlende Weg wird dann in einer rekursiven Funtion namens „find_fitting()“ aufgefunden. Diese ruft sich selbt auf, bis ein Weg zum Zielknoten gefunden ist oder eine durch die Anzahl der Abbiegevorgänge auf dem kürzesten Weg vorgegebenen Anzahl an Funktionsaufrufen erreicht hat.
Tritt der letztere Fall ein, wird der kürzeste Weg zurückgeliefert, im anderen Fall der jeweilige Weg zum Zielknoten.
Um die Wege speichern und erweitern zu können, werden zwei Container verwendet: Einer speichert die Wege selbst und einer die Anzahl der Schritte für diese Wege (diese kann variieren, weil einige Kanten einen

Abbiegevorgang „beinhalten“ - dies ist in der Matrix für die Anzahl der Abbiegevorgänge gespeichert). Erweitert werden dann nur die Wege mit der dem aktuellen Funktionsaufruf entsprechenden Anzahl an Schritten, die anderen haben durch das Nutzen einer Kante, die einen Abbiegevorgang „beinhaltet“ eine höhere Anzahl an Schritten und werden unbearbeitet an den nächsten Funktionsaufruf weitergeleitet. Bevor der Weg ausgegeben wird, werden dann noch die im Array „bridges“ gespeicherten Wegen hinzugefügt, um einen Weg im Originalgraphen zu erhalten.

3 Beispiele

Ausgabe für die Datei abbiegen0.txt mit einer maximalen Verlängerung von 10 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 10%.
Der kürzeste Weg hat eine Länge von 5.82843.
Der ausgegeben Weg darf also eine maximale Länge von 6.41127 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 7 und 1 Abbiegevorgänge.
Der kürzeste Weg hat 3 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(0,1)
(1,1)
(2,2)
(3,3)
(4,3)

Ausgabe für die Datei abbiegen0.txt mit einer maximalen Verlängerung von 15 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 15%.
Der kürzeste Weg hat eine Länge von 5.82843.
Der ausgegeben Weg darf also eine maximale Länge von 6.70269 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 7 und 1 Abbiegevorgänge.
Der kürzeste Weg hat 3 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(0,1)
(0,2)
(1,3)
(2,3)
(3,3)
(4,3)

Ausgabe für die Datei abbiegen0.txt mit einer maximalen Verlängerung von 30 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 30%.
Der kürzeste Weg hat eine Länge von 5.82843.
Der ausgegeben Weg darf also eine maximale Länge von 7.57696 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 7 und 1 Abbiegevorgänge.
Der kürzeste Weg hat 3 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(0,1)
(0,2)
(0,3)

(1,3)
(2,3)
(3,3)
(4,3)

Ausgabe für die Datei abbiegen1.txt mit einer maximalen Verlängerung von 10 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 10%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 18.8347 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 19.1224 und 5 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(2,0)
(1,1)
(2,1)
(2,0)
(7,2)
(14,3)
(14,2)
(14,1)
(14,0)

Ausgabe für die Datei abbiegen1.txt mit einer maximalen Verlängerung von 15 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 15%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 19.6908 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 19.1224 und 5 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(2,0)
(1,1)
(2,1)
(2,0)
(7,2)
(14,3)
(14,2)
(14,1)
(14,0)

Ausgabe für die Datei abbiegen1.txt mit einer maximalen Verlängerung von 30 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 30%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 22.2591 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 19.1224 und 5 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,1)
(2,2)
(2,0)
(3,3)
(2,0)
(10,4)
(14,3)
(14,2)
(14,1)
(14,0)

Ausgabe für die Datei abbiegen2.txt mit einer maximalen Verlängerung von 10 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 10%.
Der kürzeste Weg hat eine Länge von 10.8863.
Der ausgegeben Weg darf also eine maximale Länge von 11.975 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 15.9443 und 3 Abbiegevorgänge.
Der kürzeste Weg hat 6 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,1)
(2,0)
(2,2)
(8,2)
(9,1)
(9,0)

Ausgabe für die Datei abbiegen2.txt mit einer maximalen Verlängerung von 15 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 15%.
Der kürzeste Weg hat eine Länge von 10.8863.
Der ausgegeben Weg darf also eine maximale Länge von 12.5193 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 15.9443 und 3 Abbiegevorgänge.
Der kürzeste Weg hat 6 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,1)
(2,0)
(2,2)
(8,3)
(9,5)
(9,4)
(9,3)
(9,2)
(9,1)
(9,0)

Ausgabe für die Datei abbiegen2.txt mit einer maximalen Verlängerung von 30 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 30%.
Der kürzeste Weg hat eine Länge von 10.8863.
Der ausgegeben Weg darf also eine maximale Länge von 14.1523 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 15.9443 und 3 Abbiegevorgänge.
Der kürzeste Weg hat 6 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(0,1)
(2,0)
(0,2)
(8,3)
(9,5)
(9,4)
(9,3)
(9,2)
(9,1)
(9,0)

Ausgabe für die Datei abbiegen3.txt mit einer maximalen Verlängerung von 10 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 10%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 18.8347 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 17.8863 und 4 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,0)
(2,0)
(2,0)
(5,1)
(2,0)
(7,2)
(14,3)
(14,2)
(14,1)
(14,0)

Ausgabe für die Datei abbiegen3.txt mit einer maximalen Verlängerung von 15 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 15%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 19.6908 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 17.8863 und 4 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,0)
(2,0)

(2,0)
(5,1)
(2,0)
(7,2)
(14,3)
(14,2)
(14,1)
(14,0)

Ausgabe für die Datei abbiegen3.txt mit einer maximalen Verlängerung von 30 %:

Die durch den Nutzer gegebene maximale Verlängerung beträgt 30%.
Der kürzeste Weg hat eine Länge von 17.1224.
Der ausgegeben Weg darf also eine maximale Länge von 22.2591 haben.

Der Weg mit der geringsten Anzahl an Abbiegevorgängen hat eine Länge von 17.8863 und 4 Abbiegevorgänge.
Der kürzeste Weg hat 8 Abbiegevorgänge.

Beim für Bilal empfohlenen Weg werden folgende Knoten in dieser Reihenfolge besucht:
(0,0)
(1,0)
(2,0)
(2,0)
(5,1)
(2,0)
(7,2)
(14,3)
(14,2)
(14,1)
(14,0)

4 Quellcode

Die main-Funktion spiegelt die groben Programmschritte wieder:

```
int main()
{
    vector<crossing> nodes; //array for the nodes of the graph
    double** edges; //2D-array for the edges of the graph
    int number_of_edges; //Number of edges
    crossing start; //This node is the start node
    crossing goal; //The goal node
    int x_max = 0;
    int y_max = 0; //The biggest coordinates
    initialize_graph(&nodes, edges, &start, &goal, &number_of_edges, &x_max, &y_max);
    //the variables above are initialized now!
    //Evaluating the shortest way:
    vector<int> shortest_way (0);
    double length_of_shortest_way = dijkstra_shortest(&shortest_way, edges, &nodes, start, goal);
    int shortest_way_number_of_turning_processess = number_of_turning_processes(&shortest_way,
    &nodes);
    //The shortest way is now evaluated!

    //Preparing the graph for the next steps:
    double** edges_prepared; //Stores the new version of the first matrix
    double** matrix_turning_processes;//stores the second matrix of weights
    vector<int>** bridges; //The way for the abbreviations in the matrix for the turning processes
    prepare_graph(edges, &nodes, edges_prepared, matrix_turning_processes,
    index_of_crossing(goal, &nodes), index_of_crossing(start, &nodes), x_max, y_max, bridges);

    int tp_minimal[nodes.size()];
    //Stores the minimal number of turning processes from every node to the goal node
```

```
    //Initialize the array above:
    for(unsigned int n = 0; n < nodes.size(); n++)
    {
        vector<int> way_for_one_point (0);
        dijkstra_minimal_turning(&way_for_one_point, edges, &nodes, nodes[n], goal);
        tp_minimal[n] = number_of_turning_processes(&way_for_one_point, &nodes);
    }
    //No, we are ready for trying the ways through the graph!!!
    //Ask the user for the maximal extension:
    double maximal_length_way; //maximal length of the way in units of length
    int maximal_extension_procent; //maximal extension as given by the user
    cout<<"Geben Sie die maximale Verlaengerung des Weges in Prozent an: ";
    cin>>maximal_extension_procent;
    maximal_length_way = length_of_shortest_way * (maximal_extension_procent+100)/100;
    //Every necessary information is given

    vector<int> resulting_way; //This is the way to give out
    vector<vector<int>> ways (0, vector<int>(0)); //Dummy for the function below
    vector<int> start_of_ways (0);
    start_of_ways.push_back(index_of_crossing(start, &nodes));
    ways.push_back(start_of_ways); //Has initialized the vector "ways"
    vector<int> number_steps_way (1); //Also a dummy for "find_fitting"
    number_steps_way[0] = 0;
    resulting_way = find_fitting_way(matrix_turning_processes, 0,
    shortest_way_number_of_turning_processess, &ways, &shortest_way, edges_prepared, &nodes,
    &number_steps_way, &maximal_length_way, goal);

    //Output of the results:
    cout<<"Geben Sie den Namen der Ausgabedatei an: ";
    string output_file_name;
    cin>>output_file_name;
    output_to_file(output_file_name, &resulting_way, edges, &nodes, &shortest_way,
    maximal_length_way, length_of_shortest_way, maximal_extension_procent, start, goal, bridges);
    return 0;
}
```

Eine der beiden Versionen des Dijkstra-Algorithmus:

```
double dijkstra_minimal_turning(vector<int>* way, double** edges, vector<crossing>* nodes,
crossing starting_node, crossing goal_node)
{
    //This function determines the shortest way on which Bilal has to turn as little as possible
    //It returns the lengths of the way and stores the way itself via pointer
    //All in all, it is the same algorithm as implemented in the function for the shortest way
    //Only the calculation of the costs differs and more ways has to be stored and handled
    vector<int> queue_dijkstra (0);
    //This is needed for the Dijkstra-algorithm as described in the documentation
    int costs[nodes->size()]; //Stores the costs for every node
    int index_starting_node = index_of_crossing(starting_node, nodes);
    //Initialize the array costs:
    for(unsigned int n = 0; n < nodes->size(); n++)
    {
        costs[n] = 1000000000; //Higher values are not expected
    }
    //Now, the way has to be stored as a vector of vectors for every node.
    vector<vector<int>> ways[nodes->size()];
    for(unsigned int n = 0; n < nodes->size(); n++)
    {
        ways[n].erase(ways[n].begin(), ways[n].end()); //Clear the vector
    }
    //Put the starting node in the way of the starting node:
    ways[index_starting_node].push_back(vector<int> (0));
    ways[index_starting_node][0].push_back(index_starting_node);


    costs[index_starting_node] = 0; //costs for starting node have to be 0
    queue_dijkstra.push_back(index_starting_node); //Put the starting node in the queue
    vector<int>::iterator it; //Used for erasing elements in the container
    //Preparation is done, now start the process:
    while(!queue_dijkstra.empty())
    {
        //Take the first node in the queue:
```

```
        crossing first_node;
        first_node.x = (*nodes)[queue_dijkstra[0]].x;
        first_node.y = (*nodes)[queue_dijkstra[0]].y;
        //pushing all of the nodes one place forward:
        for(unsigned int n = 0; n < queue_dijkstra.size()-1; n++)
        {
            queue_dijkstra[n] = queue_dijkstra[n+1];
        }
        it = queue_dijkstra.end(); //Sets the iterator on the last element of the vector
        it--;
        queue_dijkstra.erase(it); //Erase the last (unneeded) element

        for(unsigned int c = 0; c < ways[index_of_crossing(first_node, nodes)].size(); c++)
        {
            //Going through the ways to this point - handling each one as it's own
            for(unsigned int n = 0; n < nodes->size(); n++)
            {
                //Going through the edges to which the node is belonging:
                bool visited_as_last_node = false;
                for(unsigned int b = 0;
                  b < ways[index_of_crossing(first_node, nodes)].size(); b++)
                {
                    if(ways[index_of_crossing(first_node, nodes)][b]
                    [ways[index_of_crossing(first_node, nodes)][b].size()-2] == n)
                    {
                        visited_as_last_node = true;
                    }
                }
                if(edges[index_of_crossing(first_node, nodes)][n] > -1 && !visited_as_last_node)
                {
                    //An edge to the node n is existing and n is not visited yet in this way
                    //Evaluating the new costs for a potential way:
                    vector<int> potential_way(0);
                    for(unsigned int a = 0;
                      a < ways[index_of_crossing(first_node, nodes)][c].size(); a++)
                    {
                        potential_way.push_back(ways[index_of_crossing(first_node, nodes)][c][a]);
                    }
                    potential_way.push_back(n); //The potential new way is initialized now!
                    if(number_of_turning_processes(&potential_way, nodes) < costs[n])
                    {
                        //Costs of the way above the actual
                        //edge are lower than the old costs of the node
                        costs[n] = number_of_turning_processes(&potential_way, nodes);
                        //Actualise the costs
                        if(!crossing_in_queue((*nodes)[n], nodes, &queue_dijkstra))
                        {
                            //The node is not in the queue
                            queue_dijkstra.push_back(index_of_crossing((*nodes)[n], nodes));
                        }
                        //Actualize the way:
                        ways[n].erase(ways[n].begin(), ways[n].end());
                        ways[n].push_back(potential_way);
                    }
                    else if(number_of_turning_processes(&potential_way, nodes) == costs[n])
                    {
                        //Both edges have same costs
                        //Add the way to the vector of ways:
                        ways[n].push_back(potential_way);
                    }
                }
            }
        }
    }

    //Now, the costs for every node is evaluated!
    //Store the way:
    way->erase(way->begin(), way->end());
    for(unsigned int n = 0; n < ways[index_of_crossing(goal_node, nodes)][0].size(); n++)
    {
        way->push_back(ways[index_of_crossing(goal_node, nodes)][0][n]);
    }
    //Return the length of the way:
```

```
108     return length_of_way(edges, nodes, &ways[index_of_crossing(goal_node, nodes)][0]);
}
```

Die rekursive Funktion zum Auffinden des Lösungsweges:

```
vector<int> find_fitting_way(double** matrix_turning_processes, int recent_step,
int number_steps_max, vector<vector<int>>* ways, vector<int>* shortest_way,
double** edges, vector<crossing>* nodes, vector<int>* number_steps_way,
double* length_resulting_way_max, crossing goal_node)
{
    //At first, we look for a way leading to the goal node in the given ways:
    for(unsigned int n = 0; n < ways->size(); n++)
    {
        if((*number_steps_way)[n] == recent_step)
        {
            //Right number of steps
            if((*ways)[n][(*ways)[n].size()-1] == index_of_crossing(goal_node, nodes) &&
            length_of_way(edges, nodes, &(*ways)[n]) < *length_resulting_way_max)
            {
                //We have found the goal!
                return (*ways)[n];
            }
        }
    }

    vector<int> way_to_return(0);
    if(recent_step < number_steps_max)
    {
        //The ways are not too long now!
        vector<vector<int>> new_ways (0); //vector for the next ways
        vector<int> new_number_steps (0); //vector for the number of steps in the vector above
        for(unsigned int n = 0; n < ways->size(); n++)
        {
            //Go through the ways
            if((*number_steps_way)[n] <= recent_step)
            {
                //The numbers of steps to this node is small enough, so we can add new steps
                for(unsigned int a = 0; a < nodes->size(); a++)
                {
                    if(edges[(*ways)[n][(*ways)[n].size()-1]][a] != -1 &&
                    a != (*ways)[n].size()-2)
                    {
                        //An edge exists and is not the last visited!
                        vector<int> way_to_add = (*ways)[n]; //The way to add
                        way_to_add.push_back(a);
                        new_ways.push_back(way_to_add);
                        new_number_steps.push_back((*number_steps_way)[n] +
                        matrix_turning_processes[(*ways)[n][(*ways)[n].size()-1]][a] + 1);
                    }
                }
            }
            else
            {
                //The way has to be handled in one of the next steps
                new_ways.push_back((*ways)[n]);
                new_number_steps.push_back((*number_steps_way)[n]);
            }
        }
        //We have produced the new ways, but no one reaches the goal node
        //Reaching the next step:
        way_to_return = find_fitting_way(matrix_turning_processes, recent_step+1,
        number_steps_max, &new_ways, shortest_way, edges, nodes, &new_number_steps,
        length_resulting_way_max, goal_node);
    }
    else
    {
        //The shortest way has to be returned,
        //no way with a smaller number of turning processes is short enough
        return *shortest_way;
    }
    return way_to_return;
```

```
67 }
```