Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Verwaltung und Indizierung von Bewegungstrajektorien

Niklas Owens

Studiengang: Informatik

Prüfer/in: Prof. Dr. Stefan Funke

Betreuer/in: Prof. Dr. Stefan Funke

Beginn am: 10. Juli 2017

Beendet am: 10. Januar 2018

CR-Nummer: G.2.2

Kurzfassung

Mittels GPS aufgezeichnete Bewegungstrajektorien sind eine wichtige Informationsquelle zur Kartenerstellung für das Geoinformationssystem OpenStreetMap. Dennoch gestaltet sich aufgrund der hohen Anzahl der Zugriff auf Bewegungstrajektorien einer Region schwierig. Die vorliegende Bachelorarbeit stellt eine Datenstruktur zur Verwaltung und Indizierung von Bewegungstrajektorien vor. Sie basiert auf Contracion Hierarchies, einer Datenstruktur zur Beschleunigung der Berechnung von kürzesten Wegen, und erweitert diese, sodass gezielt Bewegungstrajektorien innerhalb einer Rechtecksregion identifiziert werden können. In einem experimentellen Vergleich mit einem naiven Ansatz konnte gezeigt werden, dass sich der Zugriff auf Bewegungstrajektorien mithilfe der vorgestellten Datenstruktur deutlich effizienter gestaltet.

Inhaltsverzeichnis

1 Einleitung **13**

2 Grundlagen und Definitionen **15**
2.1 Graphentheorie 15
2.2 Dijkstra-Algorithmus 16
2.3 Contraction Hierarchies 17

3 Eine Datenstruktur zur Verwaltung von Pfaden **19**
3.1 Problemstellung 19
3.2 Zerlegung in abschnittweise kürzeste Pfade 19
3.3 Invertierter Index 20
3.4 Datenstruktur 21
3.5 Anfragen 23

4 Experimente und Evaluation **25**
4.1 Testgraphen 25
4.2 Erstellung von Testpfaden 25
4.3 Laufzeit von Anfragen 27

5 Zusammenfassung und Ausblick **31**

Literaturverzeichnis **33**

Abbildungsverzeichnis

2.1 Gerichteter Graph . . . 15
2.2 Ungerichteter Graph . . . 15
2.3 Knotenkontraktion . . . 17

3.1 Zerlegung eines Pfad in kürzeste Pfadabschnitte . . . 20
3.2 Kantenrahmen . . . 22
3.3 Knotenrahmen . . . 22
3.4 Schnitt mit Anfragerechteck . . . 24

4.1 Gitter zum Finden von Pfaden . . . 26

Tabellenverzeichnis

4.1 Testgraphen . . . 25
4.2 Pfade für Mecklenburg-Vorpommern . . . 26
4.3 Pfade für Stuttgart . . . 27
4.4 Pfade für Baden-Württemberg . . . 27
4.5 Laufzeiten für MV . . . 28
4.6 Laufzeiten für ST . . . 28
4.7 Laufzeiten für BW . . . 28
4.8 Ergebnismengen für MV . . . 29
4.9 Ergebnismengen für ST . . . 29
4.10 Ergebnismengen für BW . . . 29

Verzeichnis der Algorithmen

2.1 Dijkstra Algorithmus 16
2.2 CH-Konstruktion 18
2.3 Knotenkontraktion 18

3.1 Zerlegung in abschnittweise kürzeste Pfade 20
3.2 Modifizierte binäre Suche 21
3.3 Potenzielle Kanten finden 23
3.4 Kanten untersuchen 24

1 Einleitung

Das Projekt OpenStreetMap [OSMP17] sammelt Geodaten mit dem Ziel, eine frei zugängliche Weltkarte zu schaffen. Ein wichtiges Werkzeug zur Modellierung von Straßen und Wegen stellen hierbei neben Luftbildern mittels GPS aufgezeichnete Bewegungstrajektorien dar. Trotz ihrer Wichtigkeit ist der Zugriff auf Bewegungstrajektorien einer Region derzeit nur schwer möglich. Dies liegt daran, dass es sich um sehr viele Bewegungstrajektorien handelt. So wurden nach aktuellem Stand 450,000,000 Trajektorien hochgeladen [OSMD17]. Diese Bachelorarbeit stellt einen Algorithmus zur komprimierten Speicherung sowie eine Datenstruktur zur effizienten Verwaltung und Indizierung der Bewegegungstrajektorien vor.

Es existieren bereits Methoden zur Komprimierung von Trajektorien, wie beispielsweise PRESS [SSZZ14]. Bei diesem Komprimierungsverfahren wird davon ausgegangen, dass die Trajektorien als Sequenz von GPS-Positionen vorliegen. Diese Arbeit setzt allerdings voraus, dass alle zu verwaltenden Bewegungstrajektorien durch Map-Matching bereits auf die Kanten eines Straßengraphen abgebildet wurden. Eine Übersicht zu verschiedenen Map-Matching-Algorithmen kann in [QON07] gefunden werden. Die Datenstruktur REAPER [FSS17] von Funke et al. ähnelt der in dieser Arbeit vorgestellten Datenstruktur. Sie basiert ebenfalls auf Contraction Hierarchies und wird zum Rendern von Karten verwendet.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen und Definitionen: Dieses Kapitel beschreibt die nötigen Grundlagen zu Graphentheorie und Contraction Hierarchies.

Kapitel 3 – Eine Datenstruktur zur Verwaltung von Pfaden: In diesem Kapitel wird ein Ansatz beschrieben, mit dem Bewegungstrajektorien verwaltet werden können.

Kapitel 4 – Experimente und Evaluation: Hier wird die Effizienz der Datenstruktur experimentell mit einem naiven Verfahren verglichen.

Kapitel 5 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und geht auf mögliche weitere Arbeiten ein.

2 Grundlagen und Definitionen

2.1 Graphentheorie

Straßennetzwerke können als gerichtete, positiv gewichtete Graphen dargestellt werden. Ein positiv gewichteter Graph $G(V, E, c)$ besteht aus einer endlichen Knotenmenge V, einer Menge von Kanten $E \subseteq V \times V$, sowie einer Kostenfunktion $c\colon E \to \mathbb{R}^+$. Jede Kante $e = (u, v) \in E$ eines gerichteten Graphen ist ein geordnetes Paar aus dem Startknoten source$(e) = u$ und dem Zielknoten target$(e) = v$. Ein Graph ist ungerichtet, wenn für jede Kante $(u, v) \in E$ eine Kante $(v, u) \in E$ mit $c(v, u) = c(u, v)$ existiert. Zwei Knoten $u, v \in V$ eines gerichteten Graphen sind adjazent, wenn $(u, v) \in E$. Eine Kante $e = (u, v)$ ist inzident mit u, v und allen Kanten, die mit u und v inzident sind.

Ein Pfad $\pi = \langle e_1, e_2, ..., e_k \rangle$ zwischen zwei Knoten u und v ist eine Kantensequenz, sodass source$(e_1) = u$, target$(e_k) = v$, source$(e_{i+1}) =$ target(e_i) für $0 < i < k$, und $e \in E$ für alle $e \in \pi$. Die Kosten eines Pfades sind die Summe seiner Kantenkosten, also $\Sigma_{i=1}^{k-1} c(e_i)$. Die minimalen Kosten um von einem Knoten u aus einen Knoten v zu erreichen, werden als Distanz $d(u, v)$ bezeichnet. Ist v von u aus nicht erreichbar, so ist $d(u, v) = \infty$. Ein Pfad π von u nach v ist also ein kürzester Pfad, wenn $\Sigma_{i=1}^{k-1} c(e_i) = d(u, v)$.

Ein ungerichteter Graph ist schwach zusammenhängend, falls es für alle Knotenpaare $u, v \in V$ einen Pfad von u nach v gibt. Wenn man alle Kanten in einem gerichteten Graphen durch ungerichtete Kanten ersetzt und der entstandene ungerichtete Graph zusammenhängend ist, so ist der gerichtete Graph schwach zusammenhängend. Ein Beispiel für einen gerichteten Graphen wird in Abbildung 2.1 gezeigt, der entsprechende ungerichtete Graph in Abbildung 2.2.

Für das in dieser Arbeit vorgestellte Verfahren wird weiterhin davon ausgegangen, dass der Graph im $\mathbb{R}^2$ eingebettet vorliegt. Eine Einbettung $\Phi\colon V \to \mathbb{R}^2$ weist jedem Knoten des Graphens

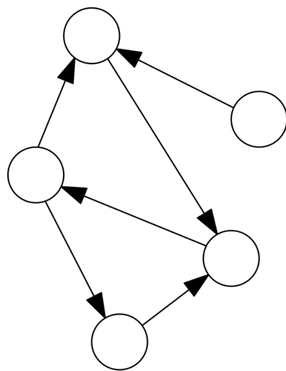

Abbildung 2.1: Gerichteter Graph

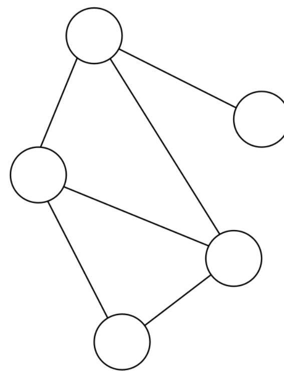

Abbildung 2.2: Ungerichteter Graph

Algorithmus 2.1 Dijkstra Algorithmus

```
function DIJKSTRA(Graph G(V, E, c), Node s ∈ V)
    for all v ∈ V \ {s} do
        d[v] ← ∞
        predecessor[v] ← null
    end for
    d[s] ← 0
    predecessor[s] ← s
    for all v ∈ V do
        PQ.insert(v, d[v])
    end for
    while PQ ≠ ∅ do
        u ← PQ.popMin()
        for all e = (u, v) ∈ E do
            if d[u] + c(u, v) < d[v] then
                d[v] ← d[u] + c(u, v)
                predecessor[v] ← e
                PQ.decreaseKey(v, d[v])
            end if
        end for
    end while
    return d, predecessors
end function
```

Koordinaten in der Ebene zu. Die Kanten des Graphen werden dann als Strecken zwischen ihren Knoten dargestellt.

2.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus [Dij59] ist ein Algorithmus, der für einen gegebenen positiv gewichteten Graphen $G = (V, E)$ und einen Knoten $s \in V$ die Distanz $d(s, v)$ von s zu allen Knoten $v \in V$ berechnet. Dazu wird für jeden Knoten v seine vorläufige Distanz $d[v]$ gespeichert. Zu Beginn ist $d[v] = \infty$ für alle Knoten $v \in V \setminus \{s\}$, für s selbst gilt $d[s] = 0$. In jedem Schritt des Algorithmus wird der Knoten $u \in Q$ mit geringstem $d[u]$ als besucht markiert und seine ausgehenden Kanten $e = (u, v) \in E$ *relaxiert*. Bei der Relaxierung einer Kante $e = (u, v)$ wird überprüft, ob $d[u] + c(u, v)$ kleiner als $d[v]$ ist. Ist dies der Fall, so wird $d[u]$ und die vorläufige Distanz des Knotens in der Prioritätswarteschlange auf die neue, niedrigere Distanz gesetzt. Um immer den Knoten mit der geringsten Distanz zu s wählen zu können, werden die noch unbesuchten Knoten in einer Prioritätswarteschlange verwaltet.

Zusätzlich zu den Distanzen kann der Dijkstra-Algorithmus auch die kürzesten Pfade selbst ausgeben. Dazu muss für jeden Knoten $v \in V$ die Kante, die auf dem Pfad von s nach v zu ihm führt, gespeichert werden.

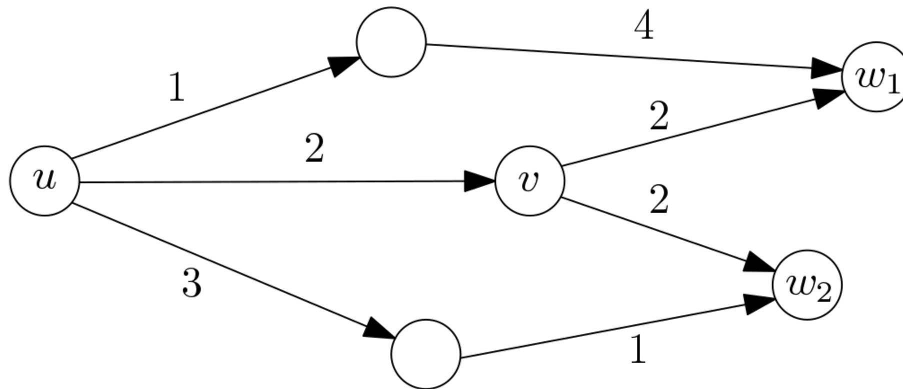

Abbildung 2.3: Bei der Kontraktion des Knotens v muss dem Graphen eine neue Kante (u, w_1) mit $c(u, w_1) = 4$ hinzugefügt werden. Eine Kante (u, w_2) ist nicht nötig, da ein anderer Pfad von u nach w_2 mit gleichen Kosten existiert.

2.3 Contraction Hierarchies

Contraction Hierarchies [GSSD08] sind ein Ansatz zur Beschleunigung von kürzesten-Weg-Anfragen, der berücksichtigt, dass Knoten von unterschiedlicher Wichtigkeit sind. Dazu werden zunächst in einer Vorberechnungsphase dem Graphen zusätzliche Kanten, die kürzeste Wege zwischen wichtigen Knoten repräsentieren, hinzugefügt. Durch diese Shortcuts, die große Sequenzen von Kanten repräsentieren können, müssen bei der Berechnung von kürzesten Wegen deutlich weniger Kanten untersucht werden.

2.3.1 Vorberechnung

In der Vorberechnungsphase wird zunächst eine Reihenfolge gewählt, die die Wichtigkeit der Knoten wiederspiegelt. Anschließend werden anhand dieser Reihenfolge alle Knoten nach aufsteigender Wichtigkeit kontrahiert.

Bei der Kontraktion eines Knotens v wird dieser aus dem Graphen entfernt, ohne dass dabei kürzeste-Weg-Distanzen zwischen verbleibenden Knoten beeinträchtigt werden. Dazu wird paarweise für alle Vorgängerknoten u mit $(u, v) \in E$ und Nachfolgerknoten w mit $(v, w) \in E$ überprüft, ob der Pfad $\langle u, v, w \rangle$ der einzige kürzeste Pfad zwischen u und w ist. Ist dies der Fall, so muss eine neue Kante (u, w) mit $c(u, w) = c(u, v) + c(v, w)$ in den Graphen eingefügt werden. Dadurch wird sichergestellt, dass sämtliche kürzeste-Wege-Distanzen erhalten bleiben. Abbildung 2.3 illustriert die Kontraktion eines Knotens.

2.3.2 Kürzeste Wege mit Contraction Hierarchies

Um von einem Knoten s den kürzesten Pfad zu einem Knoten t zu finden, wird sowohl von s als auch t eine Variante von Dijkstras Algorithmus ausgeführt. Der von s startende Dijkstra relaxiert beim Besuch eines Knotens v ausschließlich Kanten $e = (v, w) \in E$ mit $l(v) < l(w)$. Durch diese kürzeste-Wege-Suche wird nicht zwingend $d(s, t)$ berechnet, beispielsweise wenn $l(s) > l(t)$. Darum wird analog dazu von t ein rückwärts gerichteter Dijkstra gestartet, der für einen Knoten

Algorithmus 2.2 CH-Konstruktion

```
function PREPROCESSING(Graph G(V, E, c), Contraction Order η)
    i ← 0
    for all v ∈ V ordered by η do
        G ←NODE CONTRACTION(G(V, E, c), v)
        level(v) = i
        i ← i + 1
    end for
    return G(V, E, c)
end function
```

Algorithmus 2.3 Knotenkontraktion

```
function NODE CONTRACTION(Graph G(V, E, c), Node v ∈ V)
    for all u with (u, v) ∈ E do
        for all w with (v, w) ∈ E do
            if ⟨u, v, w⟩ is unique shortest path between u and w then
                E ← E ∪ (u, w)
                c(u, w) ← c(u, v) + c(v, w)
            end if
        end for
    end for
    return G(V, E, c)
end function
```

v nur Kanten $e = (u, v) \in E$ mit $l(u) > l(v)$ berücksichtigt. Falls ein kürzester Pfad von s nach t existiert, so verläuft dieser über einen Knoten v der von beiden Dijkstras besucht wurde. Von allen von beiden Dijkstras besuchten Knoten muss also der Knoten v gefunden werden, der $d(s, v) + d(v, t)$ minimiert. Der kürzeste Pfad von s nach t setzt sich dann aus den kürzesten Pfaden von s nach v und von v nach t zusammen. Um aus dem gefundenen Pfad einen Pfad zu gewinnen, der ausschließlich Kanten des Ausgangsgraphen enthält, müssen alle vorkommenden Shortcuts rekursiv entpackt werden.

3 Eine Datenstruktur zur Verwaltung von Pfaden

3.1 Problemstellung

Gegeben sei eine Menge von Trajektorien, die bereits auf eine Menge von Pfaden Π auf einem gerichteten, gewichteten und zusammenhängenden Straßengraphen $G(V, E)$ mit einer Kostenfunktion $c\colon E \to \mathbb{R}^+$ sowie einer Einbettung $\Phi : V \to \mathbb{R}^2$ abgebildet wurden. Gesucht ist eine Datenstruktur gesucht, die es ermöglicht, für ein beliebiges Koordinatenrechteck $Q = [x_1, x_2] \times [y_1, y_2]$ alle Pfade $\pi \in \Pi$ zu identifizieren, die durch Q verlaufen.

3.2 Zerlegung in abschnittweise kürzeste Pfade

Zunächst wird für den Graphen G seine Contraction Hierarchy $G'(V, E \cup E')$ mit der Levelfunktion $l\colon V \to \mathbb{N}_0$, berechnet. Die Kantenmenge E umfasst die Kanten des Ausgangsgraphens $G(V, E)$, während E' die durch die Contraction Hierarchy entstandenen *Shortcuts* enthält. Die zu verwaltenden Pfade bestehen ausschließlich aus Kanten aus E. Shortcuts sind kürzeste Pfade zwischen Knoten und repräsentieren teilweise sehr lange Kantensequenzen. Sie können daher dazu genutzt werden, die zu verwaltenden Pfade in eine komprimiertere Darstellung zu bringen, indem Abschnitte der Pfade, für die Shortcuts existieren, durch diese Shortcuts ersetzt werden. Damit ein Pfad $\pi = \langle e_1, ..., e_k \rangle \in \Pi$ in diese Darstellung gebracht werden kann, muss dieser also auf Kantensequenzen untersucht werden, die kürzeste Wege darstellen. Wurde ein kürzester Teilabschnitt eines Pfades identifiziert, so kann dieser durch Shortcuts ersetzt werden, die die Kanten auf diesem Abschnitt repräsentieren. Die entsprechenden Shortcuts werden durch eine kürzeste-Wege-Suche vom Anfangsknoten des Abschnitts zu seinem Endknoten gefunden. Um von einem Startknoten source(e_i) mit $e_i \in \pi$ aus den bezüglich der Kantenanzahl längsten Abschnitt mit minimalen Kosten zu finden, wird eine modifizierte binäre Suche gestartet. Diese sucht nach dem höchsten Index $j \in \{i, ..., k\}$, sodass $\langle e_i, ..., e_j \rangle$ ein kürzester Pfad von source(e_i) nach target(e_j) ist. Für jeden Vergleichsschritt muss also eine kürzeste-Weg-Anfrage berechnet werden. Zu beachten ist hierbei, dass durch die bereits vorhandene Contraction Hierarchy kürzeste Wege deutlich schneller berechnet werden können.

Abbildung 3.1 zeigt ein Beispiel für die Zerlegung eines Pfades in kürzesten Teilabschnitte. Der mit durchgezogenen Linien dargestellte Pfad $\langle (v_1, v_2), (v_2, v_3), ..., (v_5, v_6), (v_6, v_7) \rangle$ soll komprimiert werden. Dazu müssen Kantensequenzen, die kürzeste Wege sind, durch ihre entsprechenden Shortcuts ersetzt werden. Die Kanten $(v_1, v_2), (v_2, v_3), (v_3, v_4)$ bilden einen kürzesten Weg von v_1 nach v_4 und werden daher durch (v_1, v_4) repräsentiert. Die komprimierte Darstellung des Pfades ist dann $\langle (v_1, v_4), (v_4, v_5), (v_5, v_7) \rangle$.

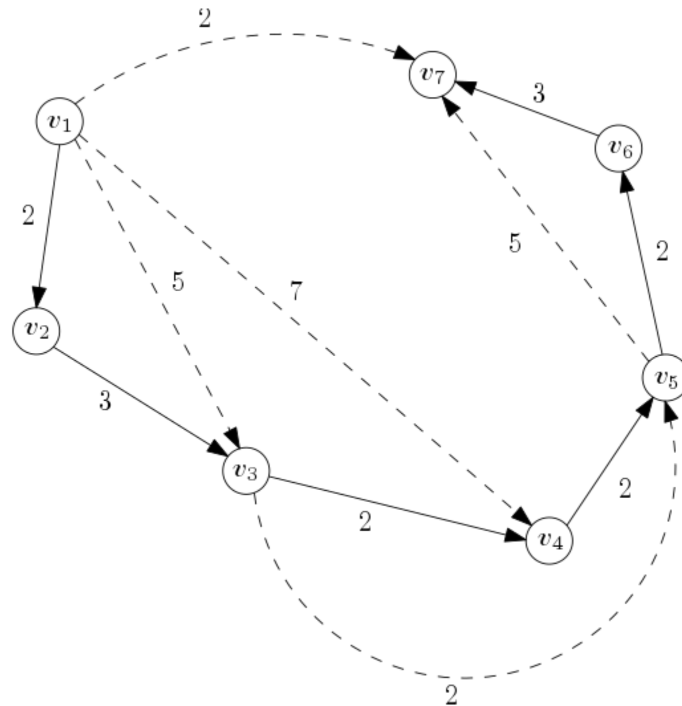

Abbildung 3.1: Zerlegung eines Pfad in kürzeste Pfadabschnitte

Algorithmus 3.1 Zerlegung in abschnittweise kürzeste Pfade

function DECOMPOSITION(Path $\pi = \langle e_1, ..., e_k \rangle$)
  $\pi' \leftarrow \langle\rangle$
  $i \leftarrow 1$
  while $i < n$ **do**
    $tmp \leftarrow i$
    $i \leftarrow$ BINARY SEARCH(π, i)
    $\pi'.append$(shortest path(source(e_{tmp}),target(e_i))
  end while
  return π'
end function

3.3 Invertierter Index

Damit nicht für jede Pfade ermittelt werden muss, ob er durch Q verläuft, wird ein invertierter Index erstellt. Dieser enthält für jede Kante, die in Π vorkommt, eine Liste, die alle Pfade aufzählt, die über die diese Kante verlaufen. Dadurch ist es ausreichend für jede Kante in Π lediglich einmal zu untersuchen ob diese Q schneidet und gegebenenfalls alle Pfade auszugeben, die in der Liste enthalten sind.

Algorithmus 3.2 Modifizierte binäre Suche

function BINARY SEARCH(Path $\pi = \langle e_1, ..., e_{n-1} \rangle = \langle v_1, ..., v_n \rangle, i$)
  $start \leftarrow i$
  $end \leftarrow n$
  $result, mid$
  while $start <= end$ **do**
    $mid \leftarrow \frac{start+end}{2}$
    if $\sum_{j=i}^{mid-1} c(e_i) = d(v_i, v_{mid})$ **then**
      $result \leftarrow mid$
      $start \leftarrow mid + 1$
    else
      $end \leftarrow mid - 1$
    end if
  end while
  return $result$
end function

3.4 Datenstruktur

Um nicht für alle im invertierten Index vorkommenden Kanten ermitteln zu müssen, ob sie das Anfragerechteck Q schneiden, wird eine Datenstruktur aufgebaut, die es ermöglicht, Kanten, die nicht in Q liegen können, herauszufiltern. Diese Datenstruktur erweitert die Contraction Hierarchy, indem alle Knoten und Kanten mit Rechtecken assoziiert werden, die alle von ihnen erreichbaren Knoten und Kanten von niedriger Wichtigkeit vollständig im $\mathbb{R}^2$ umfassen. Diese Rechtecke werden im weiteren *Rahmen* genannt. Die Wichtigkeit der Knoten ist bereits durch die Levelfunktion l gegeben, die, wie in Kapitel 2 beschrieben, durch die Kontraktionsreihenfolge der Knoten definiert ist. Da Knoten mit einem hohen Level später kontrahiert wurden, sind sie meist mit vielen Shortcuts inzident. Sie sind also allgemein wichtiger für kürzeste Pfade.

Kantenrahmen

Zunächst wird mit jeder Kante $e = (u, v)$ ein rechteckiger Rahmen R assoziiert. Hierbei müssen zwei Fälle unterschieden werden. Falls es sich bei e um eine Originalkante des Graphen handelt, bildet ihr Rahmen ein minimales Rechteck um die inzidenten Knoten u und v. Andernfalls, also wenn $e \in E'$, umfasst der Rahmen die Rahmen der Kanten, die e repräsentiert.

Knotenrahmen

Außerdem wird auch jeder Knoten mit einem Rahmen assoziiert. Der Rahmen eines Knotens v umfasst alle Knoten mit niedrigerem Level als v, die auf ungerichteten Kanten von v aus erreichbar sind, und die Rahmen aller zu diesen Knoten inzidenten Kanten. Der Rahmen des Knotens mit dem höchstem Level umfasst also den gesamten Graphen, während die Rahmen von Knoten mit minimalem Level nur den Knoten selbst enthalten.

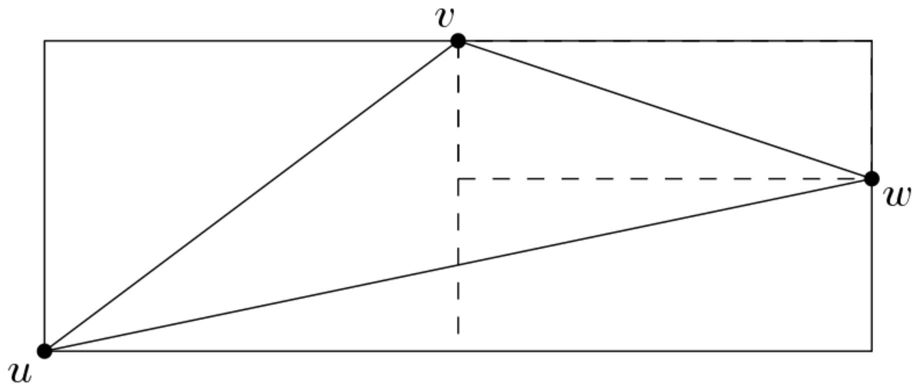

Abbildung 3.2: Da es sich bei der Kante (u, w) um einen Shortcut handelt, umfasst ihr Rahmen die Rahmen der von ihr überbrückten Kanten (u, v) und (v, w).

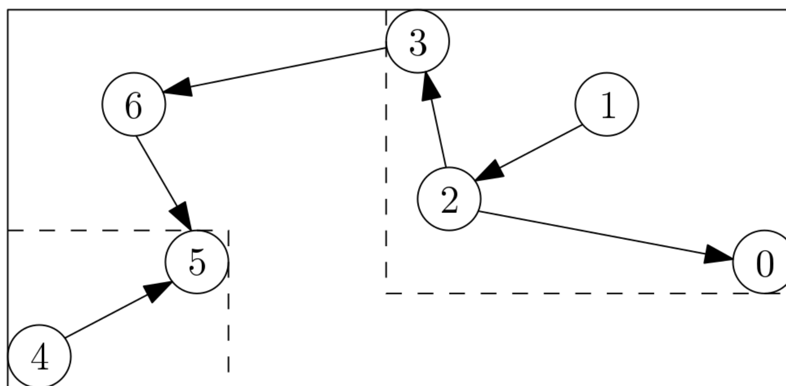

Abbildung 3.3: Der Rahmen des höchstleveligen Knotens ist ein minimales Rechteck um den Graphen. In gestrichelten Linien die Rahmen der adjazenten Knoten mit niedrigerem Level.

Durch diese Hierarchisierung der Graphkomponenten können Anfragen deutlich beschleunigt werden, da für Teile des Graphen, die nicht im Anfragerechteck liegen, lediglich der Rahmen des Knotens mit dem höchsten Level auf Schnitt mit dem Rechteck untersucht werden muss. Schneidet er es nicht, so können auch von ihm aus erreichbare, unwichtigere Knoten und ihre inzidenten Kanten nicht im Anfragerechteck liegen, da ihre assoziierte Rahmen vollständig von ihm umfasst werden.

Anzumerken ist, dass der Rahmen eines Knotens keine Shortcuts umfasst, die über ihn verlaufen, da die Endpunkte dieser Shortcuts sie bereits umfassen.

Bei der Konstruktion der Rahmen ist es hilfreich für alle Knoten Rahmen zu erstellen, die zunächst nur adjazente Knoten mit niedrigerem Level und die Rahmen deren indizidenten Kanten umfassen. Ausgehend von diesen Rahmen kann dann in aufsteigender Levelreihenfolge der Rahmen eines jeden Knotens v einfach als minimales Rechteck um die bereits erstellten Rahmen seiner adjazenten Knoten mit niedrigerem Level konstruiert werden.

Bestimmung relevanter Knoten

Weiterhin wird für jeden Knoten gespeichert, ob er für die zu verwaltenden Pfade relevant ist. Dies ist der Fall wenn der Knoten mit einer Kante aus Π inzident ist, oder von ihm aus andere relevante Knoten mit niedrigerem Level auf ungerichteten und, bezüglich des Knotenlevels, streng monoton sinkenden Kanten erreichbar sind.

3.5 Anfragen

Die beschriebene Datenstruktur ermöglicht es Kanten, die potenziell durch das Anfragerechteck verlaufen, zu identifizieren. Durch Traversierung der Datenstruktur können Knoten gefunden werden, die sowohl mit mindestens einer Kante in Π inzident sind, als auch mit einem Rahmen assoziiert sind der das Anfragerechteck schneidet. Dazu wird zunächst die Datenstruktur vom Knoten mit den höchsten Level aus auf ungerichteten Kanten abwärts traversiert. Hierbei werden immer nur Knoten besucht, deren Rahmen das Anfragerechteck potenziell schneiden können, und die ein niedrigeres Level als der zuvor besuchte Knoten besitzen. Falls der Rahmen eines besuchten Knotens das Anfragerechteck tatsächlich schneidet und relevant ist, so werden all seine inzidenten Kanten zurückgegeben.

Algorithmus 3.3 Potenzielle Kanten finden

```
function FIND EDGES(v ∈ V, Result)
    if R(v) intersects Q and relevant[v] then
        for all e = {v, w} with l(w) < l(v) do
            Result ← Result ∪ e
            FIND EDGES(w, Result)
        end for
    end if
    return Result
end function
```

Unter den durch Traversierung der Datenstruktur gefundenen Kanten sind garantiert alle Kanten die durch das Anfragerechteck verlaufen. Dennoch können sich unter ihnen auch Kanten befinden, die es nicht tun. Bei diesen Kanten müssen zwei Fälle unterschieden werden. Zum einen kann es sich bei ihnen um Kanten handeln, deren Rahmen das Anfragerechteck Q nicht schneiden. Um diesen Fall auszuschließen genügt es also, jede ausgegebene Kante darauf zu untersuchen, ob ihr assoziierter Rahmen Q schneidet. Ist dies der Fall und handelt es sich bei der Kante um eine Originalkante des Graphen, so durchläuft die Kante Q. Handelt es sich bei der Kante aber um einen Shortcut, besteht die Möglichkeit, dass das Anfragerechteck den assoziierten Rahmen des Shortcuts schneidet, aber nicht die Rahmen der von ihm überbrückten Kanten. Abbildung 3.4 zeigt ein Beispiel für einen Shortcut, bei dem dies der Fall ist.

Gefundene Shortcuts müssen also weiter daraufhin untersucht werden, ob die Rahmen der von ihnen repräsentierten Kanten das Anfragerechteck tatsächlich schneiden. Um diese Shortcuts nicht vollständig entpacken und sämtliche überbrückten Kanten untersuchen zu müssen, bietet es sich an die Shortcuts rekursiv zu untersuchen. Schneidet der Rahmen einer Kante $e = (u, w)$ das Anfragerechteck nicht, kann die Untersuchung beendet werden, da auch auch die Rahmen von

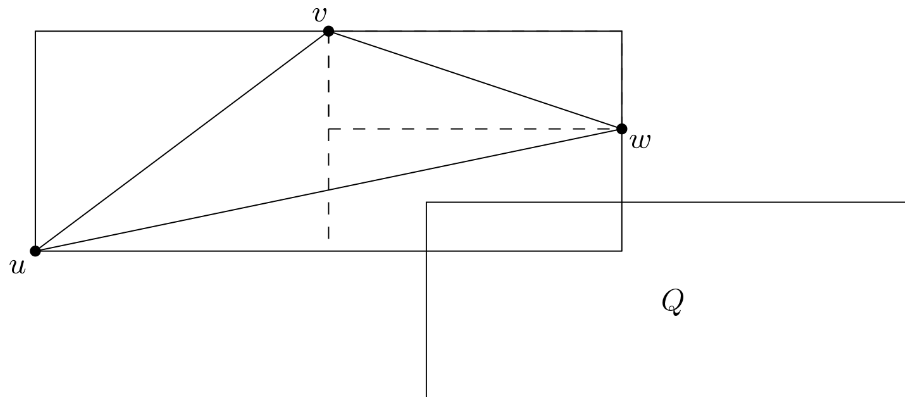

Abbildung 3.4: Q schneidet zwar den Rahmen des Shortcuts (u, w), aber nicht die der Kanten die (u, w) repräsentiert.

Algorithmus 3.4 Kanten untersuchen

```
function EXAMINE EDGE(e = (u, w))
    if visited[e] then
        return intersection[e]
    end if
    visited[e] ← True
    if R(e) intersects Q then
        if e ∈ E then
            intersection[e] ← True
            return True
        else if u or w in Q then
            intersection[e] ← True
            return True
        else
            return EXAMINE EDGE(e = (u, v))
            return EXAMINE EDGE(e = (v, w))
        end if
    else
        intersection[e] ← False
        return False
    end if
end function
```

eventuell von e überbrückten Kanten das Anfragerechteck nicht schneiden können. Schneiden sich $R(e)$ und Q, müssen zwei Fälle unterschieden werden. Handelt es sich bei e um eine Originalkante des Graphen, also $e \in E$, oder liegen u und w in Q, so liegt die Kante definitiv in Q und es können alle im invertierten Index mit e assoziierten Pfade ausgegeben werden. Falls $e \in E'$ müssen die von e überbrückten Kanten (u, v) und (v, w) ebenfalls wie beschrieben untersucht werden. Um in mehreren Pfaden auftretende Kanten nicht mehrmals untersuchen zu müssen, wird das Ergebnis der Untersuchung jeder Kante gespeichert.

4 Experimente und Evaluation

Die im Weiteren beschriebenen Experimente wurden mit einer C++ Implementierung auf einem Skylake Single-Core-Prozessor von Intel mit 3.2 GHz und 32 GB RAM durchgeführt.

4.1 Testgraphen

Die Datenstruktur wurde an den Graphen von Mecklenburg-Vorpommern, Baden-Württemberg, sowie dem des Regierungsbezirks Stuttgart getestet. Diese Graphen wurden aus Daten von OpenStreetMap extrahiert und um ihre Contraction Hierarchy erweitert. Sie werden im Weiteren mit *MV*, *BW* und *ST* abgekürzt.

Tabelle 4.1 zeigt relevante Informationen zu den bereits um ihre Contraction Hierarchies erweiterten Graphen, sowie den Speicherverbrauch und Konstruktionszeit der Datenstruktur.

Die Datenstruktur speichert für jede Kante und jeden Knoten den zugehörigen Rahmen, bestehend aus 4 Floats oder 16 Bytes. Zusätzlich muss für jeden Knoten noch in einem Bool gespeichert werden, ob er für die gegebene Pfadmenge relevant ist. Es werden also 16 Bytes für jede Kante und 17 Bytes für jeden Knoten benötigt.

4.2 Erstellung von Testpfaden

Um die Datenstruktur zu testen wurden für jeden Graphen mehrere Mengen an Pfaden erstellt. Da es sich bei den meisten Trajektorien auf OpenStreetMap um nicht sehr lange Wege handelt, wurde ein Gitter benutzt, um die Testpfade räumlich zu beschränken. Dieses Gitter teilt den Rahmen des Graphen in 100x100 gleich große Zellen ein und speichert für jeden Knoten die Zelle in der er liegt. Der Rahmen des Graphen ist identisch mit dem Knotenrahmen des höchstleveligen Knotens,

	MV	**ST**	**BW**
$\lvert V\rvert$	618,125	2,561,811	8,297,953
$\lvert E\rvert$	1,254,664	5,481,707	17,603,309
$\lvert E'\rvert$	958,159	5,738,394	18,238,928
Höchstes Knotenlevel	89	253	327
Speicherverbrauch	44,837MB	217,844MB	697,794MB
Konstruktionszeit	1.0s	4.1s	18.0s

Tabelle 4.1: Testgraphen

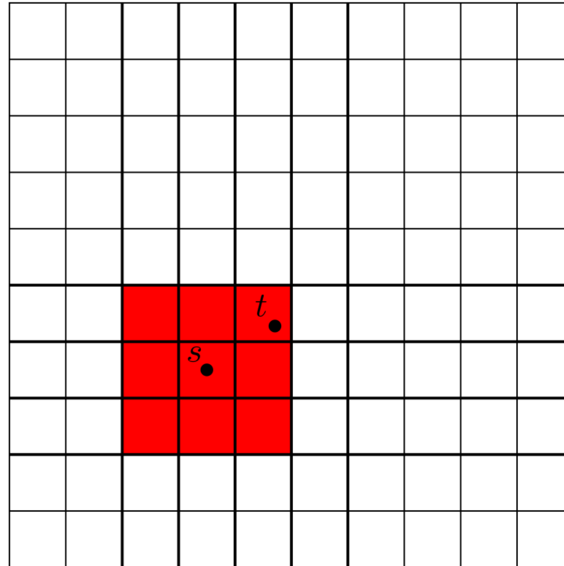

Abbildung 4.1: Beispielhaftes 10x10 Gitter mit 3x3 Maske

	7x7	13x13	19x19	25x25
Größe unkomprimiert	445	650	820	979
Größe komprimiert	12	13	15	15
Zeit für invertierten Index	3,529ms	3,759ms	3,818ms	3,894ms

Tabelle 4.2: Pfade für Mecklenburg-Vorpommern

er ist also das minimale Rechteck, das den Graphen umfasst. Die Zellen des Gitters ermöglichen es, zufällige kürzeste Pfade zu erstellen, die eine wählbare Länge nicht überschreiten. Dazu wird ein zufälliger Knoten s und ein zufälliger Knoten t aus Zellen in der Umgebung von s gewählt. Der Pfad ist dann der kürzeste Pfad von s nach t.

Abbildung 4.1 illustriert ein Beispielgitter, das den Rahmen des Graphen in 10x10 Zellen einteilt. Für einen zufälligen Punkt s wird aus einer 3x3 Maske, mit der Zelle von s als Mittelpunkt, ein anderer zufälliger Punkt t bestimmt.

Für jeden Graphen wurden jeweils vier Pfadmengen erstellt. Diese Mengen bestehen alle aus 1,000,000 kürzesten Pfaden, unterscheiden sich aber in der Größe der Maske, innerhalb der die kürzesten Pfade gesucht wurden. Gewählt wurden Masken der Größe 7x7, 13x13, 19x19 und 25x25.

Nachdem die Pfade erstellt wurden, wurden sie, wie in Kapitel 3.2 beschrieben, komprimiert und für die komprimierte Darstellung anschließend der invertierte Index erstellt. Die Tabellen 4.2, 4.3 und 4.4 zeigen für die verschiedenen Pfadmenge die durchschnittliche Kantenzahl pro Pfad sowie die für die Konstruktion des invertierten Indexes benötigte Zeit. Wie sich sehen lässt, erhöht sich die Kantenanzahl der Pfade in der komprimierten Darstellung mit zunehmender Pfadlänge kaum.

	7x7	13x13	19x19	25x25
Größe unkomprimiert	1,238	1,515	1,722	1,907
Größe komprimiert	13	14	14	15
Zeit für invertierten Index	5,203ms	5,701ms	5,491ms	6,231ms

Tabelle 4.3: Pfade für Stuttgart

	7x7	13x13	19x19	25x25
Größe unkomprimiert	2,502	3,019	3,406	3,718
Größe komprimiert	14	15	16	16
Zeit für invertierten Index	7,290ms	7,555ms	7,741ms	7,832ms

Tabelle 4.4: Pfade für Baden-Württemberg

4.3 Laufzeit von Anfragen

Die erstellten Testpfade wurden verwendet, um die Laufzeit für Anfragen mit der vorgestellten Datenstruktur mit der Laufzeit eines naiveren Ansatzes zu vergleichen. Bei diesem Ansatz werden die Kanten der unkomprimierten Pfade solange untersucht, bis der Rahmen einer Kante das Anfragerechteck R schneidet und der Pfad somit durch R verläuft. Schneidet keiner der Kantenrahmen R, so verläuft der Pfad nicht durch das Anfragerechteck.

Zur Laufzeitanalyse wurde mit unterschiedlich großen Anfragerechtecken experimentiert. Dazu wurden Rechtecke mit einer Höhe von y, $\frac{y}{2}$, ..., $\frac{y}{16}$, $\frac{y}{32}$ und einer Breite von x, $\frac{x}{2}$, ..., $\frac{x}{16}$, $\frac{x}{32}$ zufällig innerhalb des Graphenrahmens platziert, wobei y die Höhe und x die Breite des Graphenrahmens beschreibt. Für jede Pfadmenge und Rechteckgröße wurden jeweils 100 zufällige Anfragen berechnet. Die Tabellen 4.5, 4.6 und 4.7 zeigen die jeweiligen durchschnittlichen Laufzeiten für eine Anfrage, während in den Tabellen 4.8, 4.9 und 4.10 die durchschnittliche Anzahl von gefundenen Pfaden festgehalten sind.

Die vorgeschlagene Datenstruktur war, abgesehen von dem Fall, in dem das Anfragerechteck den gesamten Graphen überdeckt, für alle Anfragerechteckgrößen schneller und benötigte teilweise weniger als ein zwanzigstel der Zeit, die der naive Algorithmus benötigte. Generell skaliert die Anfragezeit deutlich besser mit zunehmender Pfadlänge und abnehmender Größe des Anfragerechtecks. Dies liegt zum einen daran, dass bei kleineren Anfragerechtecken weniger Shortcutrahmen das Anfragerechteck schneiden. Schneidet der Rahmen eines Shortcuts das Anfragerechteck nicht, so muss auch keine der Kanten, die von dem Shortcut repräsentiert werden, weiter untersucht werden. Zum anderen wächst die Anzahl der Kanten in der komprimierten Darstellung eines Pfades mit zunehmender Pfadlänge kaum. Auch mit steigender Pfadanzahl dürfte die Laufzeit besser skalieren, da Kanten, die in mehreren Pfaden vorkommen, dennoch nur einmal untersucht werden.

		1	1/2	1/4	1/8	1/16	1/32
7x7		467ms	676ms	745ms	774ms	792ms	809ms
	Naiv	325ms	1,970ms	4,597ms	7,444ms	10,293ms	13,155ms
13x13		468ms	678ms	744ms	773ms	792ms	809ms
	Naiv	411ms	2,652ms	6,336ms	10,315ms	14,292ms	18,234ms
19x19		462ms	661ms	728ms	755ms	774ms	790ms
	Naiv	479ms	3,197ms	7,551ms	12,315ms	17,101ms	21,767ms
25x25		464ms	668ms	738ms	767ms	786ms	802ms
	Naiv	548ms	3,496ms	8,519ms	14,056ms	19,678ms	25,051ms

Tabelle 4.5: Laufzeiten für MV

		1	1/2	1/4	1/8	1/16	1/32
7x7		2,174ms	3,040ms	3,383ms	3,575ms	3,721ms	3,854ms
	Naiv	635ms	8,881ms	20,851ms	33,936ms	47,069ms	60,297ms
13x13		2,265ms	3,208ms	3,559ms	3,754ms	3,902ms	4,036ms
	Naiv	757ms	11,992ms	29,596ms	49,093ms	68,730ms	88,117ms
19x19		2,235ms	3,168ms	3,540ms	3,745ms	3,898ms	4,031ms
	Naiv	838ms	13,455ms	33,340ms	54,861ms	76,595ms	97,635ms
25x25		2,353ms	3,363ms	3,766ms	3,979ms	4,133ms	4,265ms
	Naiv	913ms	14,024ms	35,428ms	59,451ms	84,011ms	105,491ms

Tabelle 4.6: Laufzeiten für ST

		1	1/2	1/4	1/8	1/16	1/32
7x7		6,804ms	9,810ms	10,880ms	11,481ms	11,919ms	12,296ms
	Naiv	1,186ms	23,483ms	64,685ms	111,779ms	159,876ms	209,138ms
13x13		6,816ms	9,941ms	11,053ms	11,645ms	12,073ms	12,427ms
	Naiv	1,363ms	27,252ms	76,879ms	132,562ms	181,738ms	222,949ms
19x19		6,669ms	9,677ms	10,724ms	11,285ms	11,675ms	12,024ms
	Naiv	1,492ms	31,301ms	88,824ms	152,561ms	200,661ms	247,774ms
25x25		6,036ms	8,797ms	9,768ms	10,295ms	10,710ms	11,077ms
	Naiv	1,558ms	31,155ms	87,344ms	151,769ms	215,212ms	281,437ms

Tabelle 4.7: Laufzeiten für BW

	1	1/2	1/4	1/8	1/16	1/32
7x7	1,000,000	454,198	133,445	40,429	12,140	6,045
13x13	1,000,000	493,452	145,276	48,517	17,722	7,564
19x19	1,000,000	512,301	172,068	48,168	23,406	11,535
25x25	1,000,000	549,897	200,859	66,066	24,807	11,747

Tabelle 4.8: Ergebnismengen für MV

	1	1/2	1/4	1/8	1/16	1/32
7x7	1,000,000	416,791	139,025	51,901	21,989	12,253
13x13	1,000,000	458,878	151,573	57,995	24,736	12,194
19x19	1,000,000	480,688	176,614	73,446	36,334	15,292
25x25	1,000,000	517,739	199,972	83,379	37,811	13,399

Tabelle 4.9: Ergebnismengen für ST

	1	1/2	1/4	1/8	1/16	1/32
7x7	1,000,000	497,794	159,360	63,329	27,151	13,314
13x13	1,000,000	544,043	189,086	74,708	35,803	14,404
19x19	1,000,000	556,951	187,668	72,651	33,239	13,953
25x25	1,000,000	581,978	209,092	82,739	46,454	19,054

Tabelle 4.10: Ergebnismengen für BW

5 Zusammenfassung und Ausblick

In dieser Bachelorarbeit wurde ein Ansatz vorgestellt, mit dem GPS-Trajektorien, die bereits auf Pfade eines Graphen abgebildet wurden, verwaltet werden können. Der Ansatz besteht im Wesentlichen aus einem Verfahren zur komprimierten Darstellung der entsprechenden Pfade sowie einer Datenstruktur, die es ermöglicht, auf Pfade innerhalb einer beliebigen Rechteckregion effizient zugreifen zu können. Sowohl das Verfahren als auch die Datenstruktur basieren auf Contraction Hierarchies. Die Zugriffszeiten der Datenstruktur wurden experimentell mit einem naiven Algorithmus verglichen. Dazu wurden mehrere Testmengen von zufälligen, kürzesten Pfaden erstellt und Anfragen mit zufälligen Rechtecken gestellt. Hierbei wurde sowohl in der Maximallänge der einzelnen Pfade als auch in der Größe der Anfragerechtecke variiert. Es hat sich gezeigt, dass der vorgestellte Ansatz fast immer deutlich besser abschneidet. Nur bei Anfragerechtecken, die so groß wie der Graph selber waren, war die Laufzeit des naiven Verfahrens besser. Solche Anfragen sind aus praktischer Sicht aber wenig interessant.

Ausblick

Insbesondere bei Algorithmus 3.4 zur Untersuchung von Shortcuts, die potenziell das Anfragerechteck schneiden, bestehen Möglichkeiten zur Verbesserung. Eine Heuristik die entscheidet welche der Kanten, die von dem Shortcut repräsentiert werden, zuerst untersucht werden sollte, wäre vorteilhaft, um die Laufzeit zu senken. Weiterhin besteht die Möglichkeit die bisherige Datenstruktur noch zu erweitern, sodass Metadaten der Bewegungstrajektorien aggregiert werden können. Dazu können beispielsweise die Art der Straßen, Tempolimits oder die Anzahl der Fahrspuren gehören.

Literaturverzeichnis

[Dij59] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische mathematik* 1.1 (1959), S. 269–271 (zitiert auf S. 16).

[FSS17] S. Funke, N. Schnelle, S. Storandt. „URAN: A Unified Data Structure for Rendering and Navigation“. In: *International Symposium on Web and Wireless Geographical Information Systems*. Springer. 2017, S. 66–82 (zitiert auf S. 13).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. „Contraction hierarchies: Faster and simpler hierarchical routing in road networks“. In: *Experimental Algorithms* (2008), S. 319–333 (zitiert auf S. 17).

[OSMD17] OpenStreetMap Project. *OpenStreetMap Data Stats*. 2017. URL: https://www.openstreetmap.org/stats/data_stats.html (zitiert auf S. 13).

[OSMP17] OpenStreetMap Project. *OpenStreetMap Project*. 2017. URL: https://www.openstreetmap.org (zitiert auf S. 13).

[QON07] M. A. Quddus, W. Y. Ochieng, R. B. Noland. „Current map-matching algorithms for transport applications: State-of-the art and future research directions“. In: *Transportation research part c: Emerging technologies* 15.5 (2007), S. 312–328 (zitiert auf S. 13).

[SSZZ14] R. Song, W. Sun, B. Zheng, Y. Zheng. „PRESS: A novel framework of trajectory compression in road networks“. In: *Proceedings of the VLDB Endowment* 7.9 (2014), S. 661–672 (zitiert auf S. 13).

Alle URLs wurden zuletzt am 12. 11. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift