FMI

Bachelorarbeit

Android-Offline-Routenplaner mit Contraction Hierarchies

Tobias Hübl

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr. Stefan Funke

Betreuer/in: M.Sc. Inf. Felix Weitbrecht

Beginn am: 12. November 2021

Beendet am: 12. Mai 2022

Abstract

Contraction Hierarchies sind ein Ansatz zur Optimierung von Pfadsuch-Algorithmen, bei welcher eine Graphstruktur um zusätzliche Elemente erweitert wird. Inhalt dieser Arbeit ist die Implementierung einer Android-App zur Berechnung kürzester Pfade auf Teilgraphen unter Verwendung ebensolcher Contraction Hierarchies. Dabei wird unter anderem der Prozess der Extraktion eines Teilgraphen von einem um Contraction Hierarchies erweiterten Graphen auf potentielle Probleme untersucht. Die dabei entdeckten Probleme der Pfadkorrektheit und der Gewährleistung der Absenz ungültiger Kanten werden auf ihre Ursache analysiert und mögliche Lösungsansätze formuliert. Dem folgt eine Dokumentation der Umsetzung der Lösungsansätze in der implementieren Software. Ebenfalls Teil der Arbeit ist eine Untersuchung der Effizienz von einem für Contraction Hierarchies modifizierten Dijkstra-Algorithmus gegenüber einem unmodifizierten Dijkstra. Anhand der Messungen ergab sich eine signifikante Optimierung der Laufzeit des Algorithmus, wenn Contraction Hierarchies zum Einsatz kommen.

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Motivation . . . 10

2 Theoretische und Technische Grundlagen **11**
2.1 Graph . . . 11
2.2 Dijkstra-Algorithmus . . . 12
2.3 Contraction Hierarchies . . . 13
2.4 OpenStreetMap . . . 16
2.5 Osmdroid . . . 16

3 Zielstellung **19**
3.1 Karten-Server . . . 19
3.2 Android-App . . . 20

4 Forschungsfragen **23**
4.1 Ungültige Shortcut-Kanten im Subgraphen . . . 23
4.2 Pfadkorrektheit . . . 25

5 Lösungsansätze zur Subgraphextraktion **29**
5.1 Intuitiv . . . 29
5.2 Erweiterung für CHs: Absenz ungültiger Kanten . . . 30
5.3 Erweiterung für CHs: Keine ungültigen Kanten und Pfadkorrektheit . . . 31

6 Implementierung **33**
6.1 Karten-Server . . . 33
6.2 Android App *OffRoAD* . . . 35

7 Performanceanalyse **47**
7.1 Erzeugung der Offline-Daten . . . 47
7.2 Optimierung durch Contraction Hierarchies . . . 48

8 Zusammenfassung und Ausblick **53**
8.1 Ausblick . . . 53

Literaturverzeichnis **57**

Abkürzungsverzeichnis

CH Contraction Hierarchies. 9

FMI Institut für Formale Methoden der Informatik. 19

OSM OpenStreetMap. 16

1 Einleitung

Die grundlegende Frage und Bedarf nach Mobilität ist sehr einfach und doch gleichzeitig sehr kompliziert. So haben wir heutzutage eine große Menge an Möglichkeiten, von einem Ort zum anderen zu gelangen, ob per Zug, mit dem Auto, zu Fuß oder auf eine von vielen weiteren Arten. Dadurch ist der grundlegende Bedarf bereits gedeckt, was zu einer neuen Frage führt: die Effizienz einzelner Mobilitätsmöglichkeiten. So ist es fast immer möglich, mit einem Auto ein Ziel über mehrere Routen zu erreichen, ganz getreu dem Sprichwort "Alle Wege führen nach Rom". Dabei ist aber meist eine Route die Schnellste, eine andere dafür kürzer, und wieder eine andere am Spritsparendsten.
Die Wahl der Route ist letztlich dem Einzelnen und dessen Präferenzen und Bedürfnissen überlassen. Damit aber überhaupt erst eine Wahl gelassen werden kann, muss zuvor ein Programm eine oder gar mehrere Routen berechnen. Dafür bedarf es Algorithmen, die diese Aufgabe nicht nur richtig, sondern vor allem möglichst effizient erfüllen. Die Entwicklung und Optimierung von ebensolchen Algorithmen zur Berechnung von Fahrstrecken ist daher ein wichtiges Thema, denn Mobilität ist in unserer globalisierten Welt omnipräsent, egal ob im Privaten, geschäftlich, oder eben für Speditions- und Transportunternehmen.

Ein Ansatz, diese Herausforderungen zu lösen, wäre die Entwicklung einer Software, welche einem Nutzer die Möglichkeit gibt, effizient Routen zwischen zwei Punkten zu berechnen. Die Wahl der Metrik, wie Distanz oder Verbrauch, ist dabei abhängig vom Kontext oder sogar durch den Nutzer festzulegen. Um die Kartendaten aktuell zu halten ohne dabei häufige Updates durch den Nutzer zu forcieren, sollte die Software eine Verbindung zu einem Server haben, welcher immer die neuesten Datensätze zur Verfügung stellt. Dabei ist zu bedenken, dass es in der Natur einer solchen Software liegt, dass sie oft unterwegs verwendet wird und nicht nur von stationären Geräten. Dementsprechend sollte auch keine konstante Internetverbindung nötig sein, da es mitunter ein Problem darstellen kann, diese Anforderung überall und jederzeit zu erfüllen. Die Frage der Mobilität der Software selbst ist also auch zu lösen.

Diese Arbeit beschäftigt sich mit Entwicklung einer ebensolchen Applikation für Android-Geräte, welche die Möglichkeit zur Berechnung der kürzesten Route zwischen zwei beliebigen Orten bietet. Die zugrundeliegenden Kartendaten beschränken sich dabei auf das Straßennetz Deutschlands. Diese sind von einem Server herunterladbar und daher bei Bedarf auch ohne Internetverbindung zugänglich. Aus Gründen der Speichereffizienz ist dem Nutzer dabei die Wahl gelassen, lediglich einen individuell selektierten Bereich der Karte herunterzuladen und abzuspeichern. Für die Routenberechnung können zwei beliebige Punkte auf der Karte ausgewählt und dann die kürzeste Route zwischen diesen Punkten berechnet werden. Der Algorithmus für die Berechnung verwendet dabei als Grundlage den Dijkstra-Algorithmus. Ein Fokus der Arbeit ist dabei die Verwendung sogenannter Contraction Hierarchies (CH). Diese werden auf einem Graphen gebildet und erweitern diesen um zusätzliche Daten und Informationen, welche dann zur Laufzeit von einem speziell dafür modifizierten Dijkstra-Algorithmus aufgegriffen werden. Dadurch wird eine signifikante

Optimierung der Berechnungsdauer erzielt.

In Kapitel 2 werden für das Verständnis benötigte Grundkenntnisse besprochen. Darauf folgt eine tiefergehende Ausarbeitung der Ziele dieser Arbeit in Kapitel 3, unterstützt von Kapitel 4, in welchem auf die resultierenden Forschungsfragen eingegangen wird, die aus der Zielstellung hervorgehen. Mögliche Lösungsansätze für diese werden in Kapitel 5 diskutiert und bewertet, bevor deren Implementierung in Kapitel 6 dokumentiert wird. Kapitel 7 befasst sich mit der Untersuchung der Laufzeiten einzelner Prozesse. Abschließend wird der Inhalt und das Ergebnis dieser Arbeit in Kapitel 8 resümiert, sowie ein kurzer Überblick über die Zukunft dieses Projektes und von CHs allgemein gegeben.

1.1 Motivation

Der Dijkstra-Algorithmus ist ein Greedy-Algorithmus, sprich er wählt zu Beginn jeder Iteration immer die bestmögliche Ausgangssituation. Ansonsten arbeitet er kontextunabhängig, für ihn ist bis auf die Distanz jeder Knoten und jede Kante grundsätzlich gleich. Er zieht also nicht eine Kante der anderen bei der Traversierung vor, weil sie auf Lange Sicht besser sein könnte. Wendet man ihn auf einen beliebigen Graph an, um den kürzesten Pfad zwischen zwei Knoten zu berechnen, wird der Algorithmus diesen in alle Richtungen suchen, nicht nur in die angenäherte Richtung des Zielknotens. In einem dichten Graphen lässt sich die Suche daher graphisch in etwa kreisförmig, mit zunehmendem Radius des Suchkreises, beschreiben. Das mag für kleine Graphen- und Testumgebungen ausreichend sein, für große Graphen, wie nationale oder gar internationale Straßennetze, ist dieser naive Ansatz aber nicht mehr ausreichend. Entsprechend ist die Optimierung der Berechnungsdauer eine relevante Thematik. Diese Arbeit verfolgt den Lösungsansatz der Verwendung sogenannter CHs, bei welchem ein existierender Graph erweitert und ein dafür modifizierter Dijkstra angewendet wird. Dabei wird neben dem technischen Vorgehen auch auf Probleme mit CHs in der Praxis, zum Beispiel bei der Betrachtung von Subgraphen, eingegangen, sowie Lösungsansätze vorgebracht und umgesetzt.

2 Theoretische und Technische Grundlagen

Dieser Abschnitt behandelt für das Verständnis der Arbeit notwendige Konzepte und Strukturen. Ebenfalls wird die Form der Notation deklariert, welche im darauffolgenden Teil der Arbeit verwendet wird.

2.1 Graph

Um eine Pfadberechnung auf einer Karte durchführen zu können, benötigt es zugrundeliegende Daten und Metriken, welche eine solche Evaluation überhaupt erst ermöglichen. Als Datengrundlage verwendet dieses Projekt einen Graph, welcher im Folgenden $\mathcal{G}$ genannt wird.

Die simpelste Form eines Graph ist ein ungewichteter, ungerichteter Graph. Bei diesem handelt es sich um eine Struktur, welche eine Menge an Elementen (Knoten) sowie zwischen diesen bestehende Verbindungen (Kanten) repräsentiert.
Graphen setzen sich dabei immer aus einer Knotenmenge V und einer Kantenmenge E zusammen, also $\mathcal{G} = (V, E)$. Die einzelnen Knoten $v \in V$ besitzen eine Menge an Attributen, welche verschiedene Eigenschaften dieser definieren. So besitzt ein Knoten des in diesem Projekt verwendeten Graphen eine eindeutige Knoten-ID, eine Längengrad und einen Breitengrad. Eine Kante $e \in E$ setzt sich dagegen aus 2 Knoten $v_1, v_2 \in V$ zusammen, wobei im Rahmen dieser Arbeit $v_1 \neq v_2$ gilt. Ist der zugrundeliegende Graph ungerichtet, bedeutet das, dass eine bilaterale Verbindung zwischen Knoten v_1 und Knoten v_2 existiert. In diesem Projekt steht eine solche Verbindung für einen Abschnitt zwischen zwei Punkten innerhalb des deutschen Straßennetzes.

Der verwendete Graph $\mathcal{G}$ erweitert diese Struktur, indem den Kanten ein Attribut hinzugefügt wird - ein Kantengewicht. Im Rahmen dieser Arbeit handelt es sich dabei um eine Festlegung der Kosten, welche für das Traversieren der Kante, also einem Abschnitt des Straßennetzes, veranschlagt werden. Die Kosten bilden sich hierbei aus der Länge des Abschnitts und stellen somit eine Repräsentation der benötigten Zeit der Traversierung dar. Durch diese Erweiterung gilt $\mathcal{G}$ als kantengewichteter Graph.

Zusätzlich gilt für den in dieser Arbeit verwendeten Graphen $\mathcal{G}$, dass er gerichtet ist. Das beschreibt die Eigenschaft, dass eine Kante $e \in E$ nun keine bilaterale, sondern eine unilaterale Verbindung bezeichnet. Für eine Kante $e = (v_1, v_2)$ bedeutet das, dass es zwar eine Verbindung von Knoten v_1 nach v_2 gibt, aber nicht implizit auch eine Verbindung von v_2 nach v_1 - außer eine Kante $e' = (v_2, v_1)$ ist ebenfalls Teil der Kantenmenge. Übertragen auf das Straßennetz kann man sich solche Kanten als Einbahnstraßen vorstellen, welche ebenfalls nur in eine Richtung verwendet werden können. Für eine zweispurige Straße, mit einer Spur in jede Richtung, existiert daher meist eine Kante für jede Richtung, wodurch indirekt eine bilaterale Verbindung existiert, jedoch mit der

Möglichkeit, dass beide Kanten ein unterschiedliches Kantengewicht aufweisen.
Durch Zusammenführen beider Eigenschaften entsteht ein kantengewichteter, gerichteter Graph (Abbildung 2.1), welcher im Folgenden verwendet wird.

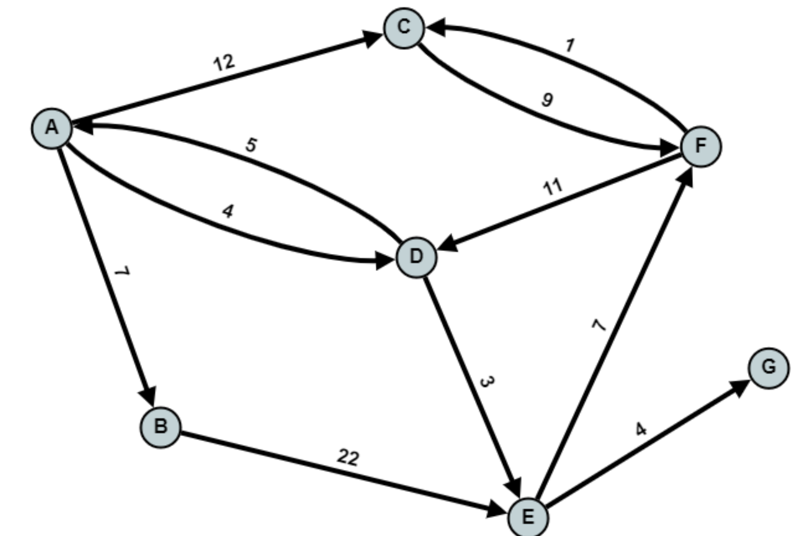


Abbildung 2.1: Beispiel für einen kantengewichteten, gerichteten Graph

2.2 Dijkstra-Algorithmus

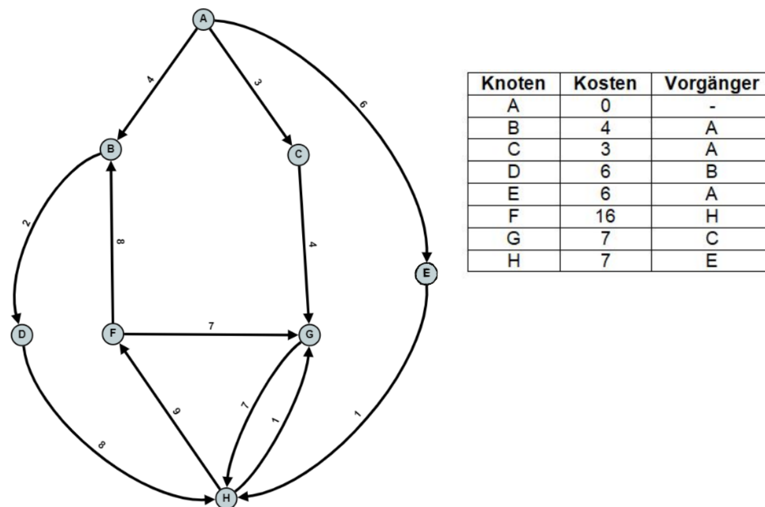
Der Algorithmus zur Pfadberechnung in diesem Projekt greift auf den Dijkstra-Algorithmus zurück. Dieser beschreibt ein Vorgehen zur Berechnung der kürzesten Pfade von einem Knoten eines Graphen, dem Startknoten, zu allen anderen Knoten. Dabei wird nach Terminierung des Algorithmus nicht nur die minimale Distanz zu jedem Knoten, sondern auch die Pfade zu den Knoten, die diese Minimalkosten ergeben, ausgegeben. Dies wird effizient erreicht, indem für jeden Knoten nicht der gesamte Pfad, sondern lediglich der Knotenvorgänger eines solchen Pfades gespeichert wird. Um damit einen Pfad zu rekonstruieren, wird, von einem Zielknoten ausgehend, der Graph immer zum aktuellen Knotenvorgänger durchwandert, bis der Startknoten erreicht ist. Der dabei traversierte Pfad ist der invertierte kürzeste Pfad vom Start- zum Zielknoten.
Auf Eingabe eines Graphen $\mathcal{G} = (V, E)$ und eines Startknoten $v_{start} \in V$ initialisiert der Algorithmus die Distanz von v_{start} zu allen anderen Knoten $v \in V$ mit ∞, wobei die Distanz zum Startknoten selbst 0 beträgt. Des Weiteren wird eine der Distanz nach aufsteigend sortierte Knotenmenge V' initialisiert, welche zu Beginn nur den Startknoten v_{start} enthält, sowie eine Knotenmenge V^*, welche alle abgearbeiteten Knoten enthält.
Dann wird iterativ wie folgt gearbeitet, solange $|V'| > 0$:

- Entnehme vorderstes Element v aus V'.
- für alle von v ausgehenden Kanten $e = (v, v')$ und deren jeweiliges Kantengewicht g mit $v' \notin V^*$:

1. Falls $Distanz(v) + g < Distanz(v')$: Setze $Distanz(v') = Distanz(v) + g$ und $Vorgänger(v') = v$.
2. Falls $v' \notin V^*$: Füge v' der Menge V' hinzu.

- Füge v der Menge V^* hinzu.

Wenn dieser Algorithmus terminiert, wurde die Distanz zu allen von v_{start} erreichbaren Knoten berechnet. Soll lediglich die Distanz zu einem bestimmten Knoten v_{ziel} berechnet werden, kann der Algorithmus auch vorzeitig terminiert werden, sobald $v_{ziel} \in V^*$ oder $Distanz(v_{ziel}) < Distanz(v)$ für einen beliebigen Knoten v zu Beginn einer Iteration.



Knoten	Kosten	Vorgänger
A	0	-
B	4	A
C	3	A
D	6	B
E	6	A
F	16	H
G	7	C
H	7	E

Abbildung 2.2: Beispiel eines Ergebnis des Dijkstra-Algorithmus

Dieser Algorithmus kann zusätzlich um Bidirektionalität erweitert werden. Dabei wird unter Angabe eines Startknoten s und eines Zielknoten t ein Dijkstra-Algorithmus sowohl an s als auch an t gestartet. Hierbei ist zu beachten, dass der von t startende Dijkstra Kanten invertiert betrachtet, also nur dann von einem Knoten u zu einem Knoten v traversiert, falls die Kante $(v, u) \in E$. Diese Erweiterung hat den Vorteil, dass terminiert werden kann, sobald für einen Knoten v von beiden Seiten ein Wert gesetzt ist, es somit also einen Treffpunkt beider Algorithmen gibt, und anhand der noch übrigen Knoten und Werte kein kürzerer Pfad existieren kann. Dann werden die beiden Teilpfade von s nach v sowie von v nach t konkateniert, deren Kosten aufsummiert, und ein Gesamtpfad ausgegeben,

2.3 Contraction Hierarchies

Gegeben ein Graph $\mathcal{G} = (V, E)$. Unter einer CH[GSSV12][GSSD08] versteht man einen Graphen $\mathcal{G}' = (V', E')$, bei welchem die Kantenmenge E um zusätzliche Kanten, sogenannte Shortcut-Kanten, sowie jeder Knoten $v \in V$ zusätzlich um ein sogenanntes Level als Attribut erweitert ist. Dafür ist jedoch eine globale Ordnung aller Knoten, dargestellt durch die Level, nötig. Diese kann entweder randomisiert sein, was aber zu ineffizienten Ergebnissen führen

kann, oder nach einer individuellen, kontextabhängigen Logik geschehen. Auf der sortierten Knotenmenge wird dann iterativ dem Level nach aufsteigend wie folgt gearbeitet. Jeder Knoten wird dabei anhand seiner ein- und ausgehenden Kanten *kontraktiert*. Angenommen, es wird momentan Knoten v betrachtet, zudem existieren Kanten $e_1 = (a, b)$ und $e_2 = (b, c)$. Dann kann der Knoten b kontraktiert werden, indem aus den beiden Kanten e_1, e_2 eine Shortcut-Kante $e_{1,2} = (a, c)$ gebildet wird, deren Kantengewicht der Summe der Kantengewichte von e_1 und e_2 entspricht, siehe Abbildung 2.3. In diesem Fall spricht man davon, dass e_1 und e_2 die *Kinder* beziehungsweise *Kindkanten* der *Elternkante* $e_{1,2}$ sind. Existiert jedoch ein alternativer Pfad von a nach c mit gleichen Kosten, so wird die Shortcut-Kante nicht gebildet, da sie nicht wird, um die Menge kürzester Pfade aufrechtzuerhalten. Sollte bereits eine solche Kante von a nach

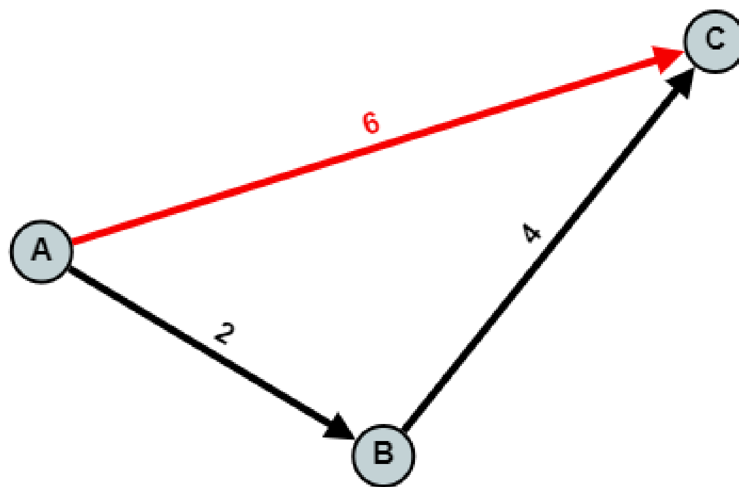


Abbildung 2.3: Neu erstellte Shortcut-Kante von Knoten a zu c mit Kantengewicht 6

c existieren, so wird lediglich deren Kantengewicht aktualisiert, sollte das der Shortcut-Kante kleiner sein. Die auf diese Weise erzeugten Kanten können auf gleiche Weise zur Bildung weiterer Shortcut-Kanten verwendet werden. Der gesamte Algorithmus kann, leicht vereinfacht, wie folgt definiert werden: Nach Terminierung des Algorithmus besitzt der resultierende Graph eine

Algorithmus 2.1 Erstellen einer CH

```
for all v ∈ V in order of Level do
    for all (u, v) ∈ E with Level(u) > Level(v) do
        for all (v, w) ∈ E with Level(w) > Level(v) do
            if < u, v, w > is shortest path from u to v then
                E = E ∪ (u, w) with weight((u, w)) = weight((u, v)) + weight((v, w))
            end if
        end for
    end for
end for
```

CH, auf welcher dann ein modifizierter Dijkstra-Algorithmus zur Pfadsuche angewandt werden kann.

Hierfür wird zuerst der Dijkstra-Algorithmus bidirektional angewandt, wobei dessen zwei Instanzen im Folgenden Vorwärts-Dijkstra (startend am Startknoten) und Rückwärts-Dijkstra (startend am Zielknoten) genannt werden. Des Weiteren gilt von nun an für den Vorwärts-Dijkstra, dass eine Kante $(u, v) \in E$ nur in Betrachtung gezogen wird, falls $Level(u) < Level(v)$. Für den Rückwärts-Dijkstra gilt dies ebenfalls, hierbei ist aber zu beachten, dass Kanten invertiert durchlaufen werden (siehe Abschnitt 2.2 auf Seite 13). Es wird zudem nur dann eine Kante (v, u) von u nach v traversiert, falls $Level(v) > Level(u)$. Für den *Treffpunktknoten* des bidirektionalen Dijkstra gilt in diesem Fall, dass er das höchste Level aller Knoten innerhalb eines gefundenen Pfads hat.
Aufgrund der Shortcut-Kanten terminiert dieser Algorithmus schneller, lässt aber das Problem zurück, dass ein Teil der verwendeten Kanten durch die CH konstruiert wurden und in der "Realität"nicht existieren. Daher ist der nächste Schritt, die Shortcut-Kanten wieder zu expandieren. Dabei wird jede Shortcut-Kante im berechneten kürzesten Pfad durch die zwei Kanten ersetzt, aus welchen sie ursprünglich gebildet wurde. Da Shortcut-Kanten wiederum das Produkt aus Shortcut-Kanten sein können, wird dieser Prozess rekursiv angewandt, bis nur noch Kanten aus der originalen Kantenmenge übrig sind. Das Resultat der Expansion ist dann der tatsächliche kürzeste Pfad.

Um zu zeigen, dass die Einführung neuer Regelungen bezügliche der Traversierung keinen Einfluss auf die Berechnung des kürzesten Pfades hat, gilt es die Korrektheit des Algorithmus zu zeigen. Der gezeigte Beweis folgt dabei dem Muster von Stefan Funke[Fun19].
Im Folgenden sei geltend, dass ein kürzester Pfad immer einzigartig sei, der kürzeste Pfad zwischen zwei beliebigen Knoten also eindeutig ist. Angenommen, es existiert ein Startknoten a und ein Zielknoten i, sowie ein kürzester Pfad $\phi = < a, b, c, d, e, f, g, h, i >$ von a nach i aus mehreren (schwarzen) Kanten (siehe Abbildung 2.4 auf der nächsten Seite). Dabei spiegelt die Höhe der Knoten in der Abbildung ihr relatives Level, und somit auch den Zeitpunkt der Kontraktion, wider. Die Kontraktionsreihenfolge wäre in diesem Fall also $b \longrightarrow c \longrightarrow h \longrightarrow a \longrightarrow g \longrightarrow i \longrightarrow e \longrightarrow d \longrightarrow f$. Der in Algorithmus 2.1 auf der vorherigen Seite beschriebene, modifizierte Dijkstra würde diesen Pfad jedoch nicht finden, da er von Knoten a nach Knoten f nur aufsteigende Kanten, beziehungsweise von f nach i nur absteigende Kanten in Betracht zieht. Diese Problematik wird durch die Shortcut-Kante gelöst, hier gezeigt anhand dem Pfadabschnitt $< d, e, f >$. Aufgrund der Einzigartigkeit des kürzesten Pfades ist der kürzeste Pfad von d nach f immer über e, daher wird bei der Kontraktion von e die Kante (d, f) hinzugefügt. Damit ist das Problem in diesem Abschnitt behoben. Allgemein gilt, dass für zwei Kanten (u, v), (v, w) mit $Level(u) > Level(v) < Level(w)$ gilt, dass wenn der kürzeste Pfad von u nach w über v geht, eine Shortcut-Kante (u, w) existiert. Nun kann auch mit einer Shortcut-Kante diese Problematik weiterhin existieren (Pfadabschnitt $< a, b, c, d >$), aber dieses kann mit selbiger Argumentation gelöst werden. Auch für den Pfadabschnitt $< f, g, h, i >$ lässt sich die Argumentation anwenden, wodurch letztlich mithilfe der Shortcut-Kanten ein durch den modifizierten Dijkstra findbarer Pfad existiert.

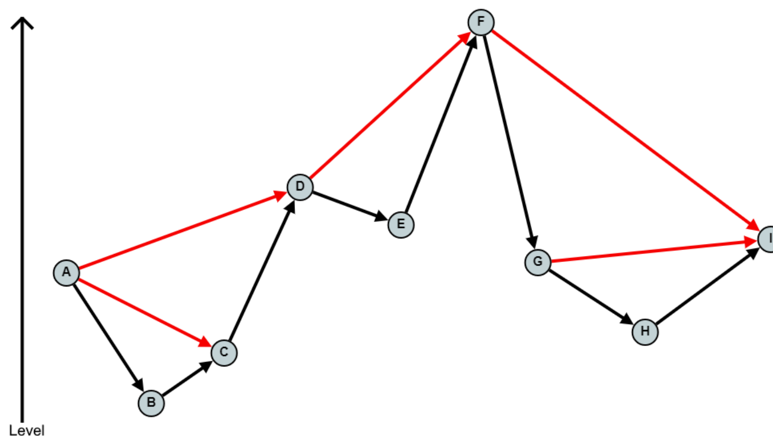


Abbildung 2.4: Korrektheit des Pfades

2.4 OpenStreetMap

OpenStreetMap (OSM)[Ope17] ist ein Open-Source Community-Projekt, in welchem von Nutzern weltweit bereitgestellte Daten gesammelt und veröffentlicht werden. Dabei handelt es sich beinahe ausschließlich um geographische Daten, wie beispielsweise GPS-Aufzeichnungen, welche verarbeitet werden, um Karten mit Informationen zu Infrakstrukturnetzen wie Straßen- und Bahnstrecken, aber auch Höhenmeter zu erzeugen. Gleichzeitig stellt OSM auch Tiles (Kacheln) einer Karte zur Abrufung bereit, mit welchen sich Nutzer bestimmte Bereiche der Welt anschauen können (Abbildung Abbildung 2.5 auf der nächsten Seite). Damit gleicht es in seiner Präsentation und Funktionalität anderen Kartendiensten wie `maps.google.com` oder `bing.com/maps`, die verwendeten Tiles haben dabei lediglich ein leicht anderes Design.

2.5 Osmdroid

Osmdroid[Gra] ist ein eine Android Bibliothek, die Funktionalitäten basierend auf OSM-Daten zur Verfügung stellt und Alternativen zu einigen Basisklassen in Android, zum Beispiel die MapView-Klasse, bereitstellen. Standardmäßig erlaubt Osmdroid unter anderem die Darstellung von Karten durch Tiles. Dazu muss ein Server oder Dateiarchiv, welcher oder welches die darzustellenden Tiles beinhaltet, angegeben werden. Die Bibliothek liest die einzelnen Tiles dann selbstständig aus und setzt diese zusammen, sodass eine vollständig dargestellte Karte gebildet wird.
Bei Verwendung eigener Tiles ist es dabei wichtig, die vorgegebene Ordnerstruktur einzuhalten, ansonsten ist eine eigenhändige Implementierung des Einlesealgorithmus nötig. Nativ unterstützt Osmdroid das ZXY-Format, welches als Standard von OSM[Ope17] übernommen wurde. Dabei werden lokal die einzelnen Tiles zuerst anhand der Zoom-Stufe (Z), dann anhand des Längengrads (X) und letztlich anhand des Breitengrades (Y) sortiert, den sie darstellen. Benötigt man ein bestimmtes Tile, um einen Bereich der Karte darzustellen, so lässt sich der Pfad des Tiles auf dem Server anhand der vorliegenden Werte (Zoom, Längengrad, Breitengrad) berechnen.

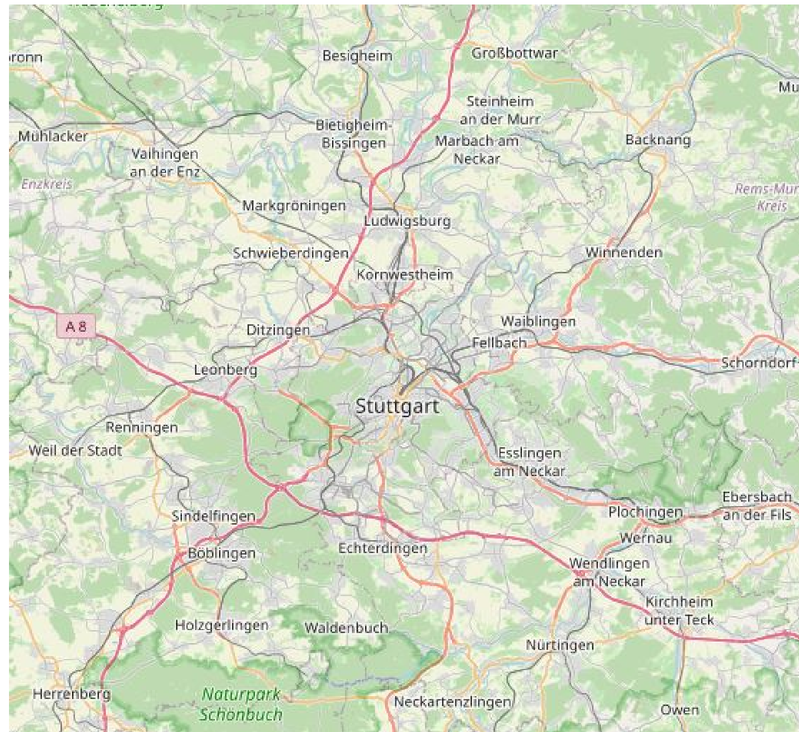


Abbildung 2.5: Beispiel der Karte auf der Webseite von OSM

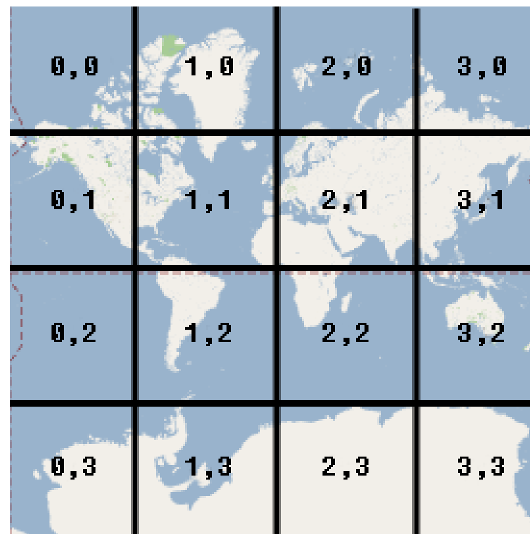


Abbildung 2.6: Beispiel für ZXY-Format (`https://developers.google.com/maps/documentation/ios-sdk/tiles`). Dargestellt ist die Karte für Zoom-Stufe 2, bestehend aus 16 Tiles mit eindeutig identifizierenden X,Y-Werten.

Eine genauere Beschreibung der Umrechnung sowie die Formeln selbst finden sich unter `https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames`.

3 Zielstellung

Die Arbeit lässt sich inhaltlich in zwei Kapitel teilen. Der erste Teil bestand darin, einen Server in Java aufzusetzen, welcher auf Anfragen zu Kartendaten reagiert und den Nutzer dann die angeforderten Daten herunterladen lässt. Diese werden für den zweite Teil benötigt, bei welchem es sich um die Implementation einer Android-App handelt. In dieser soll es möglich sein, Kartendaten vom Server herunterzuladen, um auf diesen dann mithilfe des Dijkstra-Algorithmus (Abschnitt 2.2 auf Seite 12) den kürzesten Pfad zwischen zwei Punkten berechnen zu können.

3.1 Karten-Server

Als Grundlage zur Verfügungsstellung der Kartendaten, speziell unterwegs, soll ein Server in der Programmiersprache Java implementiert werden. Dieser soll mithilfe der parallel zu implementierenden App über eine URL angesteuert werden können.

Der Server soll im Hintergrund einen Graphen des Straßennetzes von Deutschland im Speicher halten, welcher die Datengrundlage für nachfolgende Funktionalitäten darstellt. Beim Start des Servers soll dieser einmalig aus einer Datei eingelesen werden, welche alle benötigten Daten des Graphen beinhaltet und durch das Institut für Formale Methoden der Informatik (FMI) bereitgestellt wird. Ein erneutes Einlesen des Graphen während der Laufzeit ist, nicht zuletzt aus Gründen der Performance, zu vermeiden. Die Datei beschreibt dabei einen kantengewichteten, gerichteten Graph (siehe Abschnitt 2.1 auf Seite 11), in welcher zeilenweise die einzelnen Knoten und Kanten beschrieben werden.
Die ersten beiden Zeilen beinhalten die Anzahl an Knoten beziehungsweise Kanten, aus denen der Graph besteht, gefolgt von allen Knoten und danach allen Kanten.
Im Format der Datei setzt sich dabei ein Knoten aus einer (einzigartigen) Knoten-ID, einer OSM-ID, dem Breitengrad, dem Längengrad und seinem Höhenlevel zusammen. Für diese Arbeit ist jedoch nur die Knoten-ID, sowie Längen- und Breitengrad relevant.
Eine Kante hat immer eine Startknoten-ID und eine Zielknoten-ID, welche die oben beschriebene Knoten-ID referenzieren, eine Gewichtung, einen Typ, sowie eine *Maximalgeschwindigkeit*. Die beiden letzteren Eigenschaften sind für diese Arbeit ebenfalls nicht weiter relevant. Anzumerken ist, dass Kanten im Gegenteil zu Knoten keine eigene Identifikationsnummer haben.
Beim Einlesen des Graph können unwichtige Informationen übersprungen werden, müssen also nicht unnötig im Speicher bereitgehalten werden.

Nach erfolgreichem Start des Servers soll dieser auf Anfragen reagieren, welche durch Aufrufen einer durch den Server verwalteten URL gestellt werden. Bei Anfrage des Servers soll durch Verwendung von URL-Parametern ein rechteckiger Bereich innerhalb der deutschen Landesgrenzen spezifiziert werden. Dies kann unter anderem durch die Übermittlung von zwei

GPS-Koordinaten geschehen, welche dann zwei gegenüberliegende Eckpunkte eines Rechtecks festlegen. In der Folge einer solchen Anfrage ist es dann die Aufgabe des Servers, einen Subgraph des eingelesenen Graphen zu erstellen. Dieser muss alle Knoten innerhalb des Bereichs beinhalten, sowie alle Kanten, welche zwischen den extrahierten Knoten verlaufen. Hinausgehende Kanten, welche von einem Knoten von innerhalb des Bereichs ausgehen, aber auf einen Knoten außerhalb des Bereichs zeigen, sind zu verwerfen. Gleiches gilt für in den Bereich hineinführende Kanten.

Ist der Subgraph vollständig erstellt, ist es ebenfalls Aufgabe des Servers, diesen zurück an die Android-Applikation zu senden. Dies kann in Form einer Datei oder als Text in der HTML-Antwort sein. Der extrahierte Subgraph kann dann verworfen werden, er muss also nicht weiter im Speicher behalten oder als Datei gespeichert werden. Auf nachfolgende Anfragen wird also ein komplett neuer, unabhängiger Subgraph erstellt.

3.2 Android-App

Parallel zum Server soll eine Android-Applikation entwickelt werden, welche als die Bedienoberfläche des Nutzers zur Anfrage des Servers und für Operationen auf den Kartendaten verwendet werden soll.
Die Anwendung lässt sich dabei in zwei Abschnitte unterteilen - einem Online- und einem Offline-Part. Dafür soll es eine Art Menü geben, in welchem ein Nutzer zwischen der Online- und der Offline-Ansicht wechseln kann.

In der Online-Hälfte der Anwendung wird dem Benutzer eine Karte präsentiert. Diese setzt sich aus den Kartenkacheln des Tile-Servers des FMI zusammen, welche über eine Internetverbindung abgerufen werden. Grundlegend soll sich die Karte verwenden lassen wie von vergleichbaren Karten-Anwendungen, zum Beispiel Google Maps (https://maps.google.com/). Ansonsten ist die Karte leer, es existieren also keine Landkartenmarkierungen und es existiert auch kein Datensatz wie ein Graph auf Seiten des Nutzers.
Innerhalb dieser Karte soll es dann dem Nutzer möglich sein, ein rechteckiges Gebiet auszuwählen - zum Beispiel durch Aufspannen eines Kastens mit zwei Fingern. Dieser dient als Grundlage für die Extraktion eines Kartenabschnitts. So soll es nach Auswahl eines Bereichs die Möglichkeit zum Starten eines Downloads geben, welcher dann den selektierten Bereich anhand von GPS-Koordinaten ausliest und eine Anfrage an den Karten-Server schickt. Als Antwort auf diese Anfrage soll der Subgraph erhalten werden, welcher durch den Server extrahiert wurde (siehe Abschnitt 3.1). Dies kann als Datei sein oder als Text, welcher direkt verarbeitet oder zuerst als Datei abgespeichert wird. Mithilfe des Textes beziehungsweise dem ausgelesenen Dateiinhalt soll dann der Subgraph rekonstruiert werden, damit er im App-Speicher hinterlegt ist, und für Operationen zur Verfügung steht.
Gleichzeitig zum Download des Subgraphen sollen ebenfalls alle Tiles der Karte heruntergeladen werden, welche benötigt werden, um den Kartenabschnitt innerhalb des Bereichs darzustellen. Das umfasst auch alle Tiles, welche nur teilweise innerhalb des Bereichs liegen. Hierfür sollen diese direkt vom Server des FMI heruntergeladen und auf dem Mobilgerät gespeichert werden. Der heruntergeladene Datensatz soll dabei die Zoomstufen 0 bis 17 unterstützen.
Sind beim Start eines Downloads bereits heruntergeladene Dateien lokal vorhanden, soll der redundante Anteil dieser gelöscht werden. Dies umfasst den alten Subgraphen, sowie alle Tiles,

welche nicht mehr innerhalb des Bereichs liegen.

Die Offline-Hälfte der Applikation nutzt die zuvor im Online-Modus heruntergeladenen Daten. Wie auch im Online-Part wird eine Karte dargestellt, jedoch nur anhand der lokalen Tiles, sprich ohne aktive Internetverbindung. Auf dieser Karte soll es nun möglich sein, zwei Wegpunkte, einen Start- und einen Zielpunkt, festzulegen. Daraufhin soll ein Nutzer die Möglichkeit haben, den kürzesten Weg zwischen diesen beiden Punkten mithilfe des Dijkstra-Algorithmus (siehe Abschnitt 2.2 auf Seite 12) zu berechnen. Als Datengrundlage für die Ausführung des Dijkstra wird hierfür der zuvor im Online-Modus ebenfalls heruntergeladene Subgraph verwendet werden. Dabei dienen die Kosten in Form der Kantengewichte als Metrik für die Distanz zweier Knoten innerhalb des Graphen. Der Algorithmus soll terminieren, sobald ein kürzester Pfad vom Start- zum Zielknoten bekannt ist, und es keinen kürzeren Pfad mehr geben kann. Nach Beendigung des Algorithmus soll der eben berechnete Pfad noch auf der dargestellten Karte angezeigt werden.

Da der normale Dijkstra-Algorithmus auf einem großen Graphen wie dem Straßennetz Deutschlands nicht mehr effizient genug ist, soll zusätzlich zu den oben genannten Grundanforderungen eine Optimierung des Dijkstra durch Verwendung von CHs (siehe Abschnitt 2.3 auf Seite 13) implementiert werden.

Dazu soll der Karten-Server auf eine Anfrage mit einem Subgraphen antworten, auf dem eine CH existiert. Die Berechnung der CH selbst ist dabei nicht Teil der Arbeit, eine Datei mit einem bereits um eine CH erweiterten Graphen wird bereitgestellt. Ebenfalls bereitgestellt wird ein Programm[And] zu Berechnung einer CH auf einer beliebigen Graphdatei. Dies unterliegt der Annahme, dass die Datei ein unterstütztes Format für den Graphen, wie das der originalen Graphdatei des FMI, verwendet.
Die Antwort des Server soll dann wie zuvor von der Android-Applikation empfangen und verarbeitet werden, sodass der CH-Graph im Arbeitsspeicher der App vorliegt.

Der im Offline-Modus verwendete Dijkstra-Algorithmus soll ebenfalls an den nun vorliegenden Graphen angepasst werden, sodass dieser sich die Existenz der CH zunutze macht. Ein Vergleich der Performance in Form eines Laufzeitvergleichs des modifizierten Dijkstra-Algorithmus mit dem ursprünglichen Dijkstra ohne Verwendung von CH soll dabei die Effektivität dieser Optimierung testen.

4 Forschungsfragen

Durch die Möglichkeit, einen Subgraphen des bereitgestellten Graphen des Straßennetzes zu extrahieren, stellen sich neue Herausforderungen. Dieses Kapitel soll diese aufzeigen und erklären, warum sie ein Problem darstellen.

4.1 Ungültige Shortcut-Kanten im Subgraphen

Bei der Extraktion eines Subgraphen aus einem normalen Graphen ist die Frage, welche Knoten und Kanten zu exportieren sind, oft intuitiv beantwortbar. In dieser Arbeit wird der Bereich des zu extrahierenden Subgraphen durch eine rechteckige Box definiert. Die logische Schlussfolgerung ist, alle Knoten innerhalb des Bereichs in den Subgraph aufzunehmen, sowie alle Kanten, welche zwischen den auf diese Weise exportieren Knoten verlaufen und daher ebenfalls innerhalb der Box liegen. Dieser Ansatz funktioniert, da die Kanten nicht voneinander, sondern nur von ihrem Start- und Zielknoten abhängig sind. Anders gesagt hat die Entfernung einer beliebigen Kante keinen Einfluss auf den Rest des Graphen - maximal kann es dadurch passieren, dass es keinen Pfad mehr von einem Knoten zu einem anderen gibt, was aber an dieser Stelle nicht weiter relevant ist. Entsprechend können bedenkenlos alle Kanten entfernt werden, welche nicht komplett innerhalb der Box liegen. Falls zuvor ein Pfad aus Kanten innerhalb des Bereichs existierte, so existiert dieser auch nach der Entfernung all solcher Kanten (siehe Abbildung 4.1 auf der nächsten Seite).

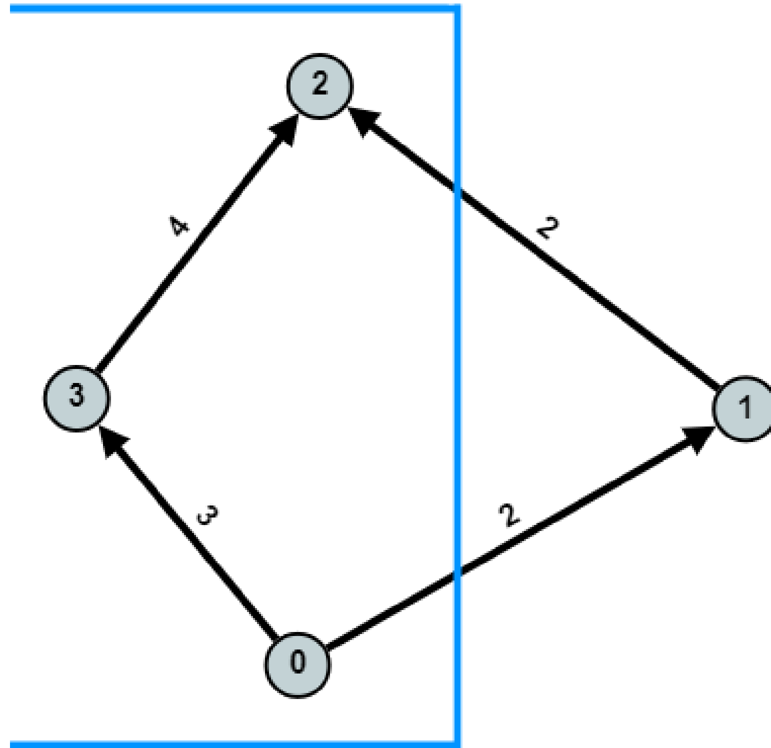


Abbildung 4.1: Beispiel für Kanten am Rand des Bereichs (blau), welche bei Extraktion entfernt werden. Im Subgraph wird nur noch der Pfad $< 0, 3, 2 >$ mit Kosten 7 existieren. Das Entfernen der beiden anderen Kanten verursacht also kein Problem.

Diese Aussage ist nicht mehr gültig, sobald die Verwendung von CH hinzukommt, denn durch das Hinzufügen der Shortcut-Kanten, kann es bei Verwendung der gleichen Methode zu Schwierigkeiten kommen. So ist es durchaus möglich, dass eine Shortcut-Kante komplett innerhalb des Bereichs liegt, die Kanten, aus welchen sich diese zusammensetzt, jedoch nicht. Das hätte zur Folge, dass die Shortcut-Kante selbst im Subgraph enthalten ist, ihre Kindkanten aber entfernt wurden (siehe Abbildung 4.2 auf der nächsten Seite). Wird nun auf dem Subgraph ein Dijkstra-Algorithmus ausgeführt und dessen Ergebnis beinhaltet eine solche Kante, kann diese nicht korrekt entpackt werden. Entsprechend kann der Dijkstra auch keinen Pfad anzeigen, der im eigentlichen Straßennetz verläuft.

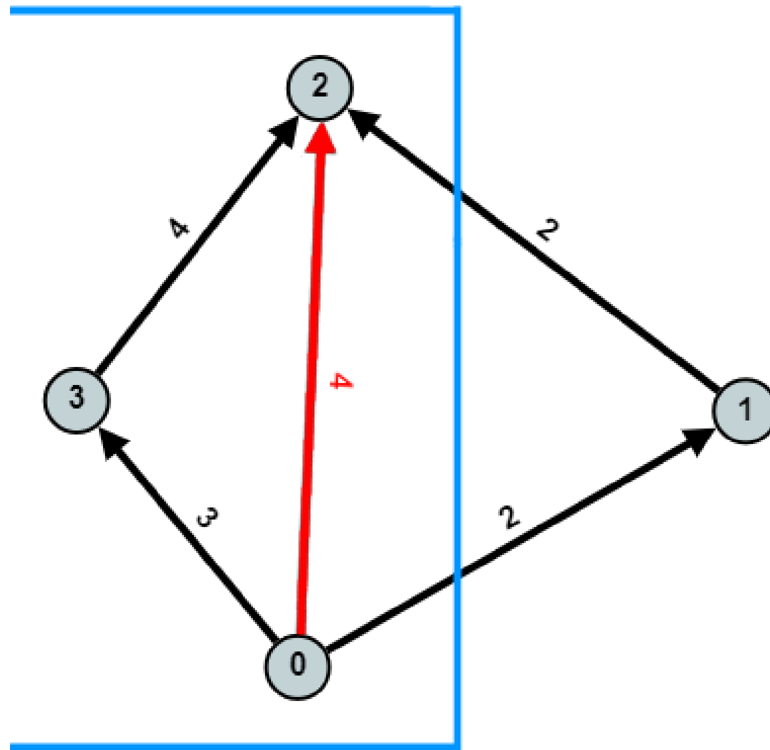


Abbildung 4.2: Die Kanten $(0, 1)$ und $(1, 2)$ am Rand des Bereichs (blau), würden bei der Extraktion entfernt werden. Die Shortcut-Kante $(0, 2)$ aber bleibt bestehen, weil sie innerhalb des Bereichs liegt. Da sie aber durch die beiden entfernen Kanten entsteht, welche im Subgraphen nicht enthalten sind, kann sie auch nicht wieder entpackt werden.

Die Frage, wie die Extraktion von Kanten geschehen kann, ohne dabei ungültige Shortcut-Kanten innerhalb des Subgraphen zu erhalten, ist daher ein wichtiger Aspekt dieser Arbeit.

4.2 Pfadkorrektheit

Eine weitere Herausforderung beim Extrahieren eines Subgraphen aus einem Graphen mit existierender CH ist die fehlende Garantie für die Korrektheit eines berechneten Pfades. Dabei handelt es sich um die Frage, ob ein berechneter kürzester Pfad tatsächlich die optimale Route zwischen zwei Punkten bildet, oder ob es nicht doch eine bessere Alternative gibt.

Angenommen, die Problematik der Existenz von Shortcut-Kanten ohne Kindkanten im Subgraph wird dadurch gelöst, dass alle nicht vollständig expandierbaren Shortcut-Kanten entfernt werden. Dadurch würde das ursprüngliche Problem zwar gelöst werden, aber auch direkt ein neues entstehen.
Der Korrektheitsbeweis für CHs (Abschnitt 2.3 auf Seite 15) baut darauf auf, dass in einem kürzesten Pfad für einen Knoten b, dessen Level kleiner als das seines Vorgängers ist, eine Shortcut-Kante von seinem Vorgängerknoten a zu seinem Nachfolgerknoten d existiert (siehe Abbildung 4.4 auf Seite 27).

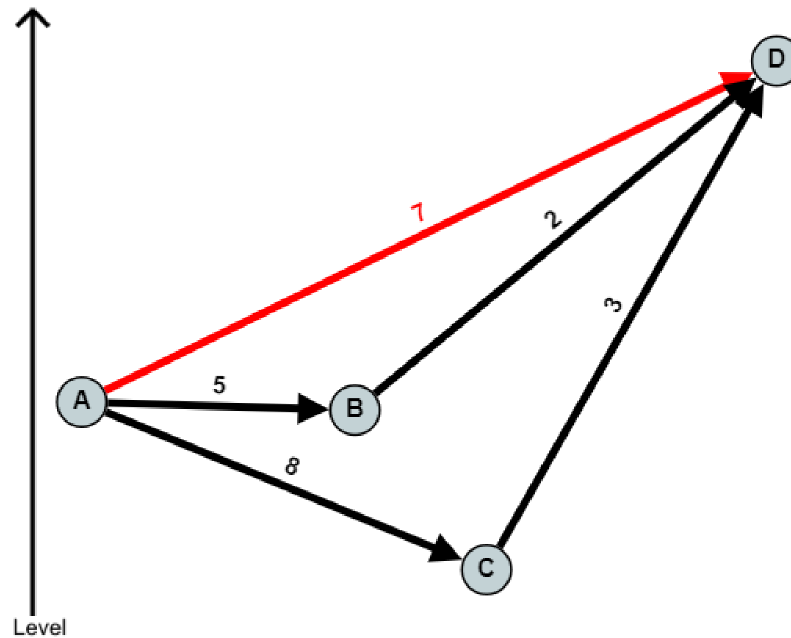


Abbildung 4.3: Der kürzeste Pfad von a nach d über b würde durch die Existenz der (roten) Shortcut-Kante (a, d) durch den modifizierten Dijkstra gefunden werden.

Liegt der Knoten außerhalb des Bereichs, wird er selbst und damit auch die Shortcut-Kante (a, d) entfernt. Dadurch existiert der bisher kürzeste Pfad von a nach d über b nicht mehr, jedoch existiert potentiell ein weiterer Pfad über einen Knoten c, der jetzt den kürzesten Pfad von a nach d bildet. Gilt für c jedoch auch, dass $Level(c) < Level(a)$, so würde der modifizierte Dijkstra diesen Pfad nicht finden, da eine entsprechende Shortcut-Kante nicht existiert, ein unmodifizierter Dijkstra jedoch schon (Abbildung 4.2 auf der vorherigen Seite). Entsprechend wäre das Ergebnis des CH-Dijkstra nicht optimal, da der normale Dijkstra einen kürzeren Pfad findet.

Entsprechend handelt es sich hierbei um eine Thematik, welche bei Nichtbeachtung die Korrektheit berechneter Pfade gefährdet. Lösungsansätze hierfür zu präsentieren und umzusetzen ist daher ein Fokus dieser Arbeit.

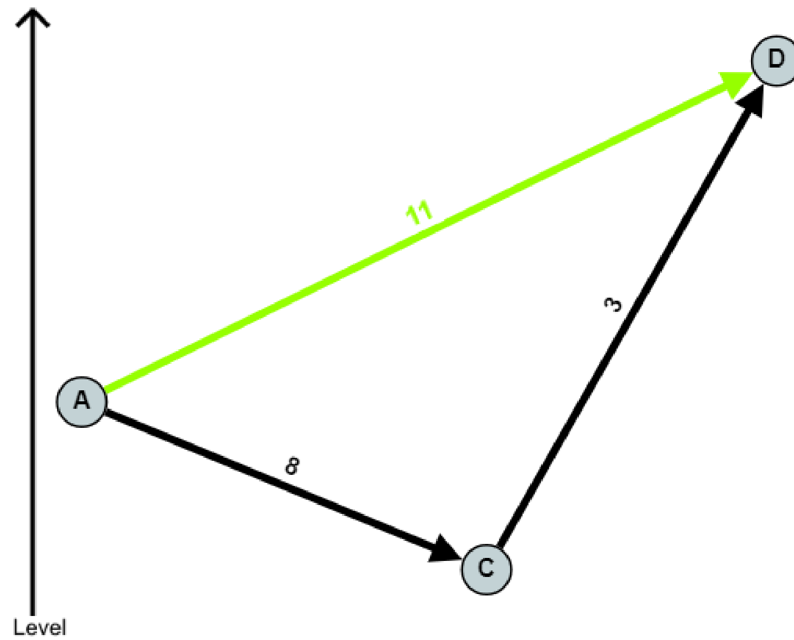


Abbildung 4.4: Nach dem Entfernen von b verläuft der kürzeste Pfad von a nach d über c. Dieser wird aber nicht durch den modifizierten Dijkstra gefunden, da die dafür benötigte Shortcut-Kante (grün) nicht existiert. Ein unmodifizierter Dijkstra-Algorithmus dagegen kann den Pfad $< a, c, d >$ traversieren.

5 Lösungsansätze zur Subgraphextraktion

Für den Prozess der Subgraphextraktion durch Deklaration einer Bounding Box (Begrenzungsrahmen) innerhalb eines größeren Graphen ergeben sich mehrere Ansätze diese umzusetzen. Dabei sind diese speziell darin zu unterscheiden, inwiefern sie die in Kapitel 4 auf Seite 23 genannten Herausforderungen aufgreifen und versuchen zu lösen. Dieses Kapitel bietet einen Überblick über einige dieser Lösungsansätze, erklärt deren Ideen und zeigt ihre individuellen Vor- und Nachteile auf.

5.1 Intuitiv

Die grundlegendste und simpelste Implementierung eines Algorithmus zur Extraktion eines Subgraphen ist, wie bereits in Abschnitt 4.1 auf Seite 23 erwähnt, das sinnbildliche Abschneiden aller Knoten und Kanten, welche außerhalb des festgelegten Areals liegen.
Dabei wird iterativ jeder Knoten einzeln betrachtet und auf seine Position in Relation zur Bounding Box getestet. Liegt er außerhalb des Bereichs, wird er nicht in den Subgraph übertragen beziehungsweise aus dem Graphen entfernt (abhängig davon, ob der Subgraph parallel zu seinem Original neu erstellt wird oder der ursprüngliche subtraktiv um Knoten reduziert wird).
Nachdem alle Knoten überprüft wurden, werden die Kanten betrachtet. Dabei werden all die Kanten in den Subgraph übertragen, für welche beide Knoten innerhalb des Subgraphen existieren. Bei subtraktivem Vorgehen können Kanten auch direkt zusammen mit einem Knoten entfernt

Algorithmus 5.1 Extraktion eines Subgraphen

```
// Given a BoundingBox and Graph G = (V, E)
Create Subgraph G′ = (V′, E′), V′ = ∅, E′ = ∅
for all v ∈ V do
    if v is inside BoundingBox then
        V′ = V′ ∪ v
    end if
end for
for all e ∈ E, e = (u, v) do
    if u ∈ V′ ∧ v ∈ V′ then
        E′ = E′ ∪ e
    end if
end for
```

werden, unter der Bedingung, dass Knoten Referenzen zu ihren ein- beziehungsweise ausgehenden Kanten besitzen. Dieser Ansatz wurde im Rahmen dieser Arbeit jedoch nicht weiter verfolgt.

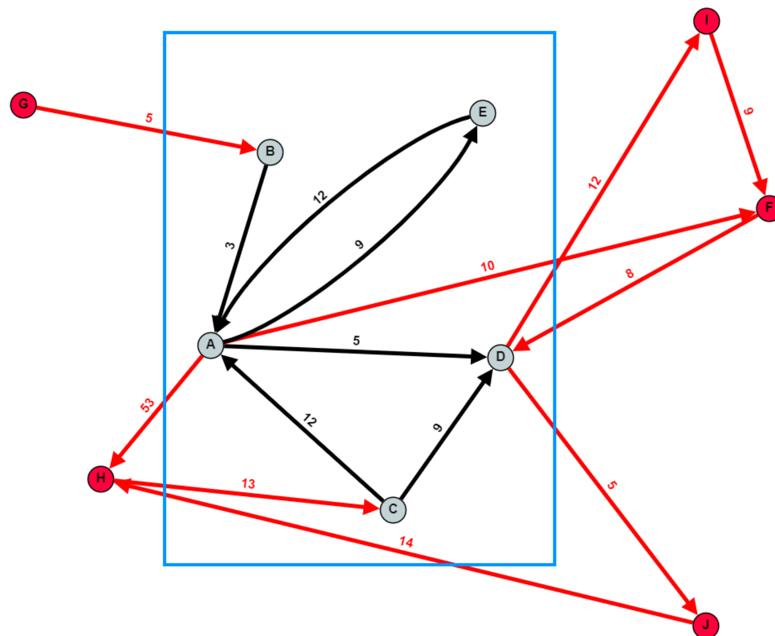


Abbildung 5.1: Der durch die Bounding Box (blau) definierte Bereich würde bei intuitiver Extraktion eines Subgraphen die rot markierten Knoten und Kanten entfernen. Die schwarzen Elemente würden den dabei neu erstellten Subgraphen bilden.

Der resultierende Subgraph erfüllt bei "normalen" Graphen ohne CH auch weiterhin alle wichtigen Eigenschaften wie die Absenz ungültiger Kanten oder eben Pfadkorrektheit. Wie aber bereits festgestellt (Kapitel 4 auf Seite 23) reicht dieser Ansatz nicht aus, um einen Subgraphen aus einem Graphen mit existierender CH zu erstellen. Jedoch ist diese Methode durch ihre simple und effiziente Implementierung ein guter Ausgangspunkt, um weitere Algorithmen darauf aufzubauen. Daher ist dieser Ansatz auch Ausgangspunkt für den Algorithmus, der in dieser Arbeit verwendet wird.

5.2 Erweiterung für CHs: Absenz ungültiger Kanten

Ein grundsätzliches Problem unter Hinzunahme von CHs ist die Gewährleistung, dass alle Shortcut-Kanten vollständig expandiert werden können. Um das zu erreichen, muss für jede Shortcut-Kante in einem Subgraph auch jede ihrer (rekursiven) Kindkanten Teil des Subgraphen sein. Damit das eingehalten wird, gibt es zwei mögliche Ansätze, einen additiven und einen subtraktiven.

Der additive Ansatz fügt dabei für jede Shortcut-Kante innerhalb der Bounding Box rekursiv alle deren Kindkanten ebenfalls dem Subgraphen hinzu. Dadurch wäre gewährleistet, dass jede Shortcut-Kante expandiert werden kann. Dieser Ansatz hätte zudem die Folge, dass der Subgraph nicht nur die Eigenschaft erfüllt, dass er keine ungültigen Shortcut-Kanten enthält, sondern dass zudem auch für alle Resultate des Dijkstra-Algorithmus Pfadkorrektheit

gewährleistet ist. Implizit kann dabei jedoch der Fall auftreten, dass einige der Kindkanten partiell außerhalb des Bereichs verlaufen, und damit potentiell auch ein Teil des berechneten Pfads des Dijkstra-Algorithmus außerhalb des Bereichs verläuft. Bei dieser Methode wäre es also wichtig, solche Kanten zur Wiedererkennung zu markieren. Sollte dann der Dijkstra-Algorithmus eine auf diese Weise markierte Kante traversieren, kann dieser entsprechend darauf reagieren und zum Beispiel dem Nutzer eine Rückmeldung geben, Alternativrouten berechnen (dann jedoch ohne Gewährleistung von Pfadkorrektheit) oder den betroffenen Abschnitt visuell hervorheben.

Der subtraktive Ansatz hingegen fügt nicht weitere Kanten hinzu, sondern entfernt problematische Shortcut-Kanten. Dafür kann, nachdem der Subgraph wie in Algorithmus 5.1 auf Seite 29 beschrieben extrahiert wurde, erneut über alle Kanten iteriert werden, wobei jede Shortcut-Kante entfernt wird, deren rekursive Kindkanten nicht vollständig im Subgraph auffindbar sind. Das Ergebnis ist ein Subgraph, der nur noch "gültige" Shortcut-Kanten aufweist. Zudem gilt für jeden gefundenen Pfad, der durch die Ausführung des Dijkstra-Algorithmus berechnet wird, dass dieser komplett innerhalb der Bounding Box verläuft. Es muss also nicht wie im additiven Ansatz auf potentielle Ausreißer geachtet werden. Im Gegensatz dazu hat diese durch das Entfernen zusätzlicher Kanten den Nachteil, dass der resultierende Subgraph grundsätzlich keine Pfadkorrektheit gewährleistet.

Die Wahl zwischen additivem und subtraktivem Ansatz ist kontextabhängig zu treffen. Falls die Beschränkung der Bounding Box nur approximiert ist, so ist die additive Methode sinnvoller, da die Ausreißer meist im Umfeld des ausgewählten Bereichs vorzufinden sind und in diesem Fall keine signifikante Abweichung existiert. Ist die Bounding Box dagegen strikt, ist der subtraktive Ansatz besser, mit dem Nachteil, dass eine berechnete Route nicht die tatsächlich kürzeste ist oder gar nicht erst gefunden wird.

5.3 Erweiterung für CHs: Keine ungültigen Kanten und Pfadkorrektheit

Bisher wurde der in Abschnitt 5.1 auf Seite 29 beschriebene Algorithmus insofern an CHs angepasst, dass er gewährleistet, dass der Graph keine ungültigen Kanten aufweist. Dabei kam es zum Konflikt mit der Pfadkorrektheit, denn das simultane Erfüllen beider Eigenschaften stellte eine Herausforderung dar. Insofern stellt sich die Problematik dahin um, dass beide Anforderungen simultan abzudecken nicht ganz trivial ist. Eine Möglichkeit jedoch, beide Eigenschaften zu ermöglichen, bietet sich darin, keine vorausberechnete CH zu verwenden, sondern diese erst auf dem Subgraph selbst zu bilden.

Beim Start des Karten-Servers wird eine Graphdatei ohne CH eingelesen. Empfängt dieser eine Anfrage durch die Android-Applikation, wird zuerst mithilfe des in Algorithmus 5.1 auf Seite 29 präsentierten Verfahrens ein Subgraph extrahiert. Auf dem dadurch erstellten Subgraph wird dann das in Abschnitt 3.2 auf Seite 21 erwähnte Programm zur Konstruktion einer CH[And] ausgeführt. Der resultierende Subgraph wird dann als Antwort auf die Anfrage an die Applikation zurückgesendet und entsprechend verarbeitet.
Dieser Ansatz hat den großen Vorteil, dass das Endergebnis sowohl nur gültige Kanten als auch Pfadkorrektheit aufweist. Da die CH erst auf dem Subgraphen selbst gebildet wird, befinden sich alle betrachteten Knoten innerhalb der Bounding Box, und damit implizit auch alle Shortcut-Kanten,

welche zwischen diesen gebildet werden. Und da auch im Weiteren keine (Shortcut-)Kanten entfernt werden, ist auch die Pfadkorrektheit gewährleistet. Der mitunter größte Nachteil dieses Ansatzes ist die deutliche höhere Rechendauer im Vergleich zu den Methoden, welche nur einen Subgraph extrahieren. Hierbei nimmt die Erstellung der CH den größten Anteil ein.
Insgesamt zeigt sich dieser Ansatz am Vielversprechendsten, nicht zuletzt da er als einziger sowohl korrekte als auch optimale Ergebnisse liefert. Zumal ist die höhere Berechnungsdauer auch deshalb nicht schwerwiegend, da dieser Prozess einmal während des Downloads ausgeführt werden muss und dessen Dauer, wie später in Abbildung 7.2 auf Seite 49 festgestellt, ohnehin durch das Herunterladen der Tiles relativiert wird.

6 Implementierung

Dieses Kapitel behandelt die Umsetzung der in Kapitel 3 auf Seite 19 spezifizierten Anforderungen unter Hinzunahme der Ergebnisse aus Kapitel 5 auf Seite 29. Die beschriebene Implementierung reflektiert den Zustand zum Zeitpunkt der Beendigung der Arbeit. Sowohl der Karten-Server als auch die Android-Applikation wurden auf Basis von Java Version 11.0.13 implementiert. Zudem benötigt die App mindestens API-Level 27 von Android, was Androids Version 8.1 *Oreo* entspricht. Offiziell unterstützt die App bis zu API-Level 31 und damit Android Version 12 *Snow Cone*, womit sie den aktuellen Mindestanforderungen einer Versionsunterstützung bis mindestens November 2023 genügt [Bel].

6.1 Karten-Server

Die finale Version der Arbeit verwendet als Methode für die Extraktion des Subgraphen den Ansatz, dass die CH erst auf dem Subgraphen berechnet wird (Abschnitt 5.3 auf Seite 31). Entsprechend wird als Datengrundlage eine Graphdatei verwendet, welche keine vorausberechnete CH besitzt. Bei der Ausführung liest der Karten-Server die Graphdatei des deutschen Straßennetzes ein, indem zeilenweise deren Inhalt verarbeitet wird. Zur Bereithaltung der Daten im Arbeitsspeicher wird ein Graph-Objekt angelegt, welches alle relevanten Informationen aus der Datei speichert. Für die Knoten und Kanten wurde sich gegen die Verwendung klassenbasierter Objekte entschieden, da diese zu einer deutlich längeren Zugriffszeit beim Auslesen der Daten führen. Stattdessen werden die Knoten und Kanten im Graphen als mehrere eindimensionale Arrays gespeichert, wobei je ein Array pro Attribut der Knoten/Kanten verwendet wird. Somit existieren für den ganzen Graphen letztlich zwei Arrays für die Knoten (Längen- und Breitengrad), sowie drei Arrays für die Kanten (Startknoten, Zielknoten, Kantengewicht).

Sobald der Graph vollständig eingelesen ist, steht der Server für Anfragen bereit. Eine Anfrage muss dabei an das Root-Verzeichnis des lokalen Netzwerks gerichtet sein und zwei GPS-Koordinaten für die Bounding Box als URL-Parameter beinhalten. Beispiel für eine valide URL wäre `localhost:8080/?coordinate1=49.00,8.2&coordinate2=52,12`. Diese würde einen Subgraphen für das rechteckige Gebiet, welches durch die Koordinaten (49.00, 8.20) und (52.00, 12.00) definiert ist, anfragen.

Erhält der Server eine Anfrage, so beginnt er mit der Extraktion des Subgraphen anhand des Algorithmus 5.1 auf Seite 29. Dafür wird auf dem Graphen jeder Knoten iterativ geprüft, ob er sich anhand seines Längen- und Breitengrades innerhalb der spezifizierten Bounding Box befindet, und "entfernt", falls dies nicht der Fall ist. In der Umsetzung geschieht dies durch einen weiteren Array, im Folgenden $RemovedNodes$, welcher alle seine Einträge, deren Indizes mit den IDs "entfernter" Knoten übereinstimmen, markiert. Die Arrays mit den tatsächlichen

Attributen der Knoten bleiben dabei unberührt, damit der ursprüngliche Graph nicht modifiziert und dementsprechend am Ende des Prozesses auch nicht neu eingelesen werden muss. Nachdem alle Knoten betrachtet wurden, werden die Knoten-IDs neu zugeordnet. Dies geschieht aus Konsistenzgründen, da im ursprünglichen Format auch bereits Knoten aufsteigend sortierte, lückenlose IDs vorweisen. Dafür wird über $RemovedNodes$ iteriert, wobei der Wert jedes nicht markierten Eintrags mit einem aufsteigenden Zähler gesetzt wird, sodass letztlich jeder Eintrag entweder eine einzigartige, lückenlos hochzählende ID darstellt oder markiert ist (siehe Abbildung 6.1).

12	13	-1	-1	14	15	-1	16	17

Abbildung 6.1: Beispiel für den Array $RemovedNodes$, der Knoten markiert und neu indiziert. Entfernte Knoten sind durch eine -1 gekennzeichnet, alle anderen Einträge sind aufsteigend, lückenlos indiziert.

Damit sind die Knoten abgearbeitet und die Kanten werden betrachtet. Wie der Algorithmus 5.1 auf Seite 29 beschreibt, wird nun jede Kante entfernt, deren Start- oder Zielknoten nicht Teil des Subgraphen ist. Da auch hier das Problem besteht, dass die ursprünglichen Arrays nicht modifiziert werden sollen, wird erneut ein weiterer Array, $RemovedEdges$, erstellt. Dann wird iterativ für jede Kante die ID ihres Start- und Zielknotens ausgelesen und der Eintrag des selben Indizes in $RemovedNodes$ betrachtet. Ist der Eintrag in $RemovedNodes$ für einen der beiden Knoten -1, so wurde dieser im vorherigen Schritt entfernt und damit wird auch die Kante "entfernt". Dafür wird, bei Betrachtung der n-ten Kante, der n-te Eintrag von $RemovedEdges$ gesetzt. Auf diese Weise wird eine Indizierung der Kanten simuliert, da diese, im Gegensatz zu Knoten, ursprüngliche keine eigene ID besitzen.

Nachdem sowohl Knoten als auch Kanten fertig bearbeitet sind, wird der resultierende Subgraph zwischengespeichert, indem dieser, das originale Format erhaltend, in eine temporäre Datei geschrieben wird. Der Prozess wandelt dazu den originalen Graph im Speicher wieder in Textform um. Dieser überspringt jedoch alle in $RemovedNodes$ markierten Knoten sowie in $RemovedEdges$ markierten Kanten, und ersetzt alle Knoten-IDs in den Start- und Zielknoten der übernommenen Kanten durch die neu gesetzten Indizes in $RemovedNodes$. Das Resultat ist eine Datei mit dem extrahierten, neu indizierten und korrekt formatierten Subgraphen.
Um abschließend eine CH auf diesem zu konstruieren, wird die Datei als Eingabe für das bereitgestellte Programm, den *CH-Constructor* [And], verwendet. Dieses berechnet selbstständig die CH, wobei die nähere Funktionsweise nicht Teil dieser Arbeit ist. Da der Sourcecode des Programms nicht verändert wurde, kann dieses in Folge eines Updates oder bei Bedarf durch eine neue/andere Version ausgetauscht werden. Als Ausgabe des *CH-Constructor* entsteht eine weitere Graphdatei, welche den Subgraphen mit gebildeter CH beinhaltet. Der Inhalt der erstellten Datei wird letztlich vom Server ausgelesen und an die Android-Applikation gesendet, womit die Kommunikation mit dem Karten-Server abgeschlossen ist. Die beiden serverseitig erstellten Dateien werden am Ende dieses Prozesses wieder gelöscht, um den Speicherplatz wieder freizugeben.

6.2 Android App *OffRoAD*

Die Android-Applikation *OffRoAD* (Offline Routeplaner for Android Devices) besteht wie geplant aus zwei Hauptmodulen, einem für Online- und einem für Offlinedienste.

Beim Start der App wird während des Startbildschirm zuerst geprüft, ob bereits heruntergeladene Daten existieren und eventuell neu eingelesen werden, Genaueres dazu später in diesem Abschnitt (Abschnitt 6.2.1 auf Seite 38).

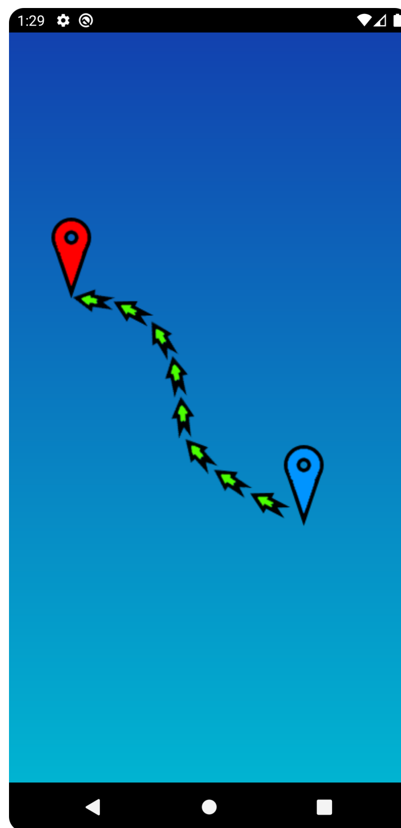


Abbildung 6.2: Startbildschirm der App. Im Hintergrund werden bei Bedarf Daten neu eingelesen.

Im darauffolgenden Hauptmenü (Abbildung 6.3 auf der nächsten Seite) existieren zwei Tasten, mit welchen der Online- und der Offlinemodus gestartet werden können. Zudem lässt sich über eine weitere Taste in der oberen, rechten Ecke das Einstellungsmenü aufrufen.

Die internen Einstellungsmöglichkeiten beschränken sich dabei auf die wichtigsten Punkte. So ist es mitunter möglich, die IP-Adresse und den Standard-Port des Karten-Servers festzulegen. Da hierdurch bereits alle notwendigen Einstellungsmöglichkeiten direkt in der App getroffen werden können, ist kein editieren des Sourcecodes der App nötig, um diese zu verwenden. Eine letzte mögliche Einstellungsoption ist das Festlegen einer Maximalgröße für die Downloads im Onlinemodus durch die Angabe einer Anzahl an Karten-Tiles. Wird im Onlinemodus ein Gebiet selektiert, welches eine größere Menge an Tiles umfasst, wird ein Nutzer vor dem entsprechend

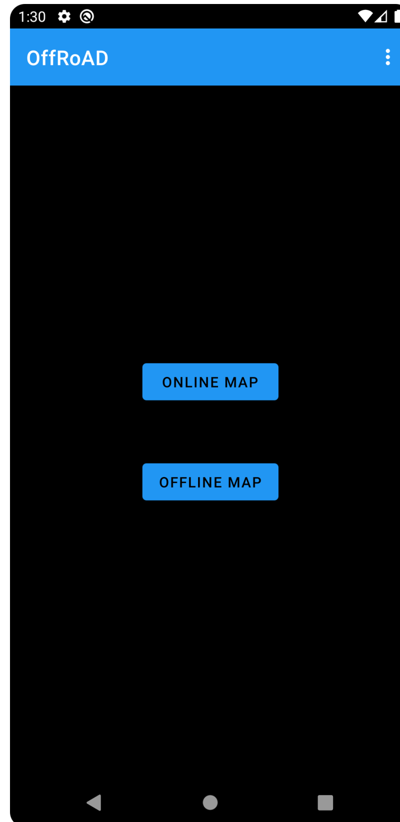


Abbildung 6.3: Über das Hauptmenü lässt sich auf den Online- und Offlinemodus, sowie die Einstellungen zugreifen.

großen Download gewarnt. Grund für diese Entscheidung ist, dass es gerade für unerfahrene Nutzer sehr schwer vorzustellen ist, aus wie vielen Tiles sich ein ausgewählter Bereich zusammensetzt. Es handelt sich dabei also um eine Art Absicherung für den Nutzer. Da auch die Vorstellung davon, wie viel Speicherplatz eine gewisse Menge an Tiles braucht, schwer fällt, wird ein geschätzter Speicherbedarf angegeben, der bei Ausschöpfung der festgelegten Grenze benötigt wird. Dieser Wert ist dabei pessimistisch abgeschätzt, sodass er in der Realität nie erreicht wird, trotzdem aber als grobe Orientierung verwendet werden kann. Des Weiteren sei gesagt, dass dieser Wert eine Maximalgrenze von 200.000 hat, welche im Code gesetzt ist und durch den Nutzer nicht überschritten werden kann. Das liegt daran, dass der Graph mit zunehmender Größe auch immer speicherintensiver wird, bis die App nicht mehr genug Zwischenspeicher besitzt um den gesamten Graphen zu halten. Bei Bedarf ist der Wert im Sourcecode leicht zu ändern, sollte es in Zukunft nötig sein, diesen zu erhöhen. Als Vergleich: Baden-Württemberg lässt sich mit etwa 1.000.000 Tiles abdecken.

6.2.1 Online-Modus

Der Online-Modus stellt eine Karte von Deutschland dar, vergleichbar mit verwandten Kartendiensten wie Google Maps (`maps.google.com`). Hierfür verwendet die App das Open-Source Projekt *Osmdroid*[Gra], bei welchem es sich um eine Bibliothek zur Darstellung von Karten

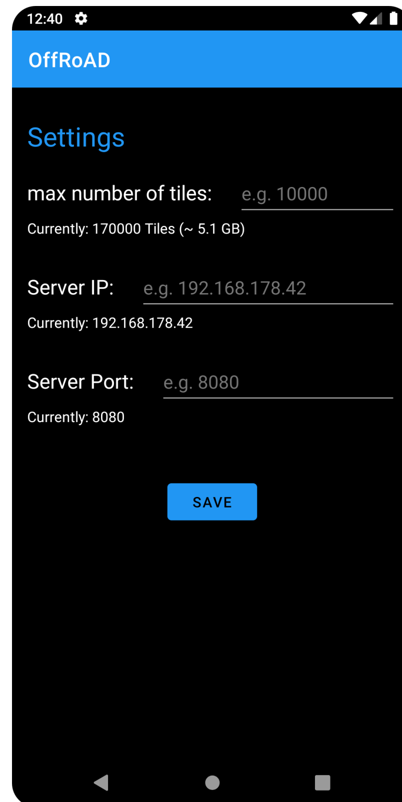


Abbildung 6.4: Im Einstellungsmenü lassen sich neben IP-Adresse und Port des Karten-Servers auch die maximale Größe eines Downloads festlegen.

mithilfe von Tiles auf Basis von OSM handelt. So stellt *Osmdroid* unter Angabe einer Quelle der Tiles die anzuzeigende Karte anhand des momentanen Sichtfensters selbst zusammen, speichert bereits heruntergeladene Tiles in einem internen Cache, und erlaubt die Implementierung eigener Nutzerinteraktionen mit der Karte. Standardmäßig implementiert ist dabei das Bewegen des Sichtfensters, sowie die Möglichkeit zu Zoomen.

Für diese Arbeit wird als Quelle der Tiles der Tile-Server des FMI (`https://tiles.fmi.uni-stuttgart.de/`) verwendet. Dessen Ordnerstruktur unterstützt bereits das ZXY-Format, somit ist bereits Kompatibilität mit Osmdroid (siehe Abschnitt 2.5 auf Seite 16) gewährleistet.

Um dem Nutzer die Möglichkeit zu geben, einen Teil der Karte herunterzuladen, wurden die standardmäßige Interaktion des Zoomens überschrieben, sodass stattdessen ein rechteckiger Bereich aufgezogen wird (Abbildung 6.6 auf Seite 39). Um weiterhin die Möglichkeit zu zoomen zu ermöglichen, existieren zwei entsprechende Tasten mit dieser Funktionalität an der unteren Seite des Bildschirms. Zudem lässt sich diese Funktion bei Bedarf auch mithilfe eines Schalters in der oberen linken Ecke abschalten, wodurch das direkte Zoomen mithilfe der Finger wieder möglich ist.

Hat der Nutzer einen Bereich ausgewählt, so kann dieser den Downloadprozess des Kartenabschnitts durch betätigen der Download-Taste starten. Vor dem Start dessen wird ausgelesen, aus wie vielen Tiles sich der selektierte Bereich zusammensetzt, und ob diese den in den

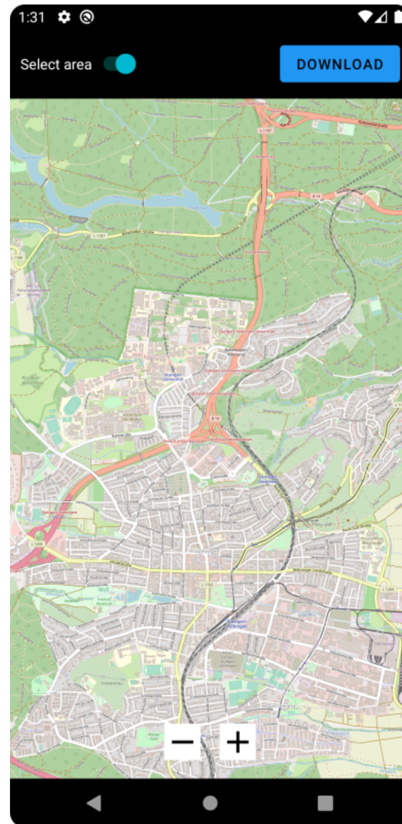


Abbildung 6.5: Der Online-Modus der App. In diesem kann ein Bereich ausgewählt und anschließend heruntergeladen werden.

Einstellungen festgelegten Wert überschreitet. Falls er überschritten wird, zeigt die App dem Nutzer einen entsprechenden Dialog an, welcher die berechnete Anzahl Tiles mit dem festgelegten Wert vergleicht und den Vorgang abbricht. Ist dies nicht der Fall, so startet der Download. Der Prozess besteht dabei aus zwei Teilen, dem Herunterladen der Tiles und dem Berechnen des Subgraphen des Abschnitts.
Für Ersteren liest der Prozess zuerst die definierenden GPS-Koordinaten des Bereichs aus und berechnet anhand dessen den Abschnitt an Tiles, die er vom Server herunterladen muss. Daraufhin werden alle benötigten Tiles heruntergeladen und, das ZXY-Format erhaltend, auf dem Android-Gerät gespeichert. Für alle Speicherzugriffe wird der interne Speicher der App verwendet, um in Einklang mit den geänderteren Richtlinien bezüglich Speicherzugriff durch Android 11 zu stehen. Um in späteren Verfahren einfachen Zugriff auf die Dateien zu haben, werden diese unarchiviert (beispielsweise ZIP-Archiv) in Ordnerstruktur angelegt. *Osmdroid* unterstützt jedoch nativ lediglich einige wenige Dateiformate (unter anderem SQLite, ZIP und Mapsforge). Daher musste zuerst eine eigene Implementierung eines *ArchiveFileProvider* von *Osmdroid* umgesetzt werden, welcher das Auslesen loser, unarchivierter Dateien ermöglicht.
Parallel dazu stellt die App eine Anfrage an den Karten-Server zu Extraktion eines Subgraphen, indem sie die ermittelten GPS-Koordinaten des Bereichs übermittelt. Der serverseitige Prozess läuft wie in Abschnitt 6.1 auf Seite 33 beschrieben ab. Am Ende erhält die App den Subgraphen als Antwort in Textform, speichert diesen als Datei ab und liest diese wiederum nach einem ähnlichen Prinzip ein, wie der Karten-Server die originale Graphdatei. Im Gegensatz zum Graphen auf

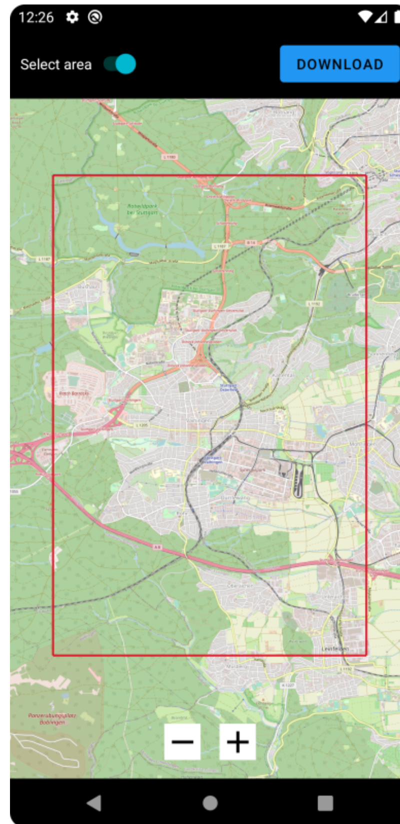


Abbildung 6.6: Ein durch den Nutzer ausgewählter Bereich wird in rot dargestellt.

dem Karten-Server merkt sich der lokale Subgraph jedoch nicht nur alle ausgehenden Kanten eines Knoten, sondern zudem auch alle eingehenden Kanten. Diese Information ist wichtig für die Ausführung des bidirektionalen Dijkstra (Abschnitt 2.2 auf Seite 13) im Offline-Bereich, welcher die Kanten rückwärts läuft und somit den Graphen von einem Knoten über dessen eingehende Kanten zu einem anderen Knoten traversiert.
Wird die App geschlossen und neu gestartet, müsste die Graphdatei jedes Mal erneut ausgelesen werden, was für größere Graphen/Dateien nicht sehr performant ist. Daher wird das auf diese Weise erstellte Graph-Objekt zusätzlich als Binärdatei gespeichert, welche stattdessen bei einem erneuten Start während des Startbildschirms eingelesen wird. Auf diese Weise dauert der Einleseprozess nur einen Bruchteil der Zeit, wodurch längere Ladezeiten beim Start der App verhindert werden. Dadurch liegen zwar zwei Dateien vor, da die Textdatei nicht gelöscht wird, dafür kann nach einem Update der App, in welcher der Graph intern verändert und die Binärdatei damit ungültig wird, dieser anhand der Textdatei trotzdem rekonstruiert werden. Ein erneutes Herunterladen eines Bereichs ist dann nicht nötig.
Zur Optimierung der Performance sind zudem beide Hälften des Downloads parallelisiert, für den Download der Tiles sogar der jedes einzelnen Tiles.

Nach Terminierung des Downloadprozesses liegen also alle Tiles vor, die benötigt werden, um den Bereich darzustellen, sowie ein Subgraph, welcher das Straßennetz innerhalb des Bereichs abdeckt.

Um dem Nutzer während des Downloads über den Fortschritt zu informieren, zeigt die App einen Dialog mit je einem Ladebalken für den Download des Subgraphen und den der Tiles an (Abbildung 6.7). Nach Abschluss verschwindet der Dialog und die GPS-Koordinaten des

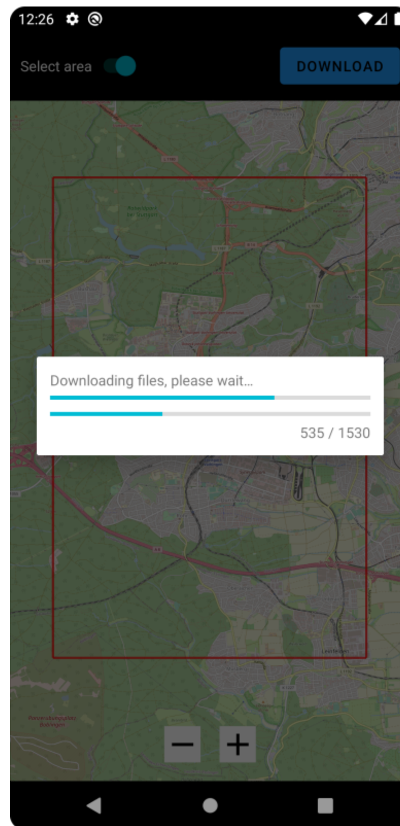


Abbildung 6.7: Der Fortschrittsdialog zeigt den Stand des Downloads des Subgraphen und der Tiles. Für Letztere ist auch eine numerische Übersicht gegeben.

heruntergeladenen Bereichs werden appintern gespeichert. Zudem wird anhand dieser das eben heruntergeladene Gebiet dem Nutzer auf der Karte markiert (Abbildung 6.8 auf der nächsten Seite). Durch das Abspeichern der Koordinaten kann diese Markierung auch in zukünftigen Sitzungen geladen werden, um dem Nutzer einen Überblick über das aktuell heruntergeladene Gebiet zu geben.

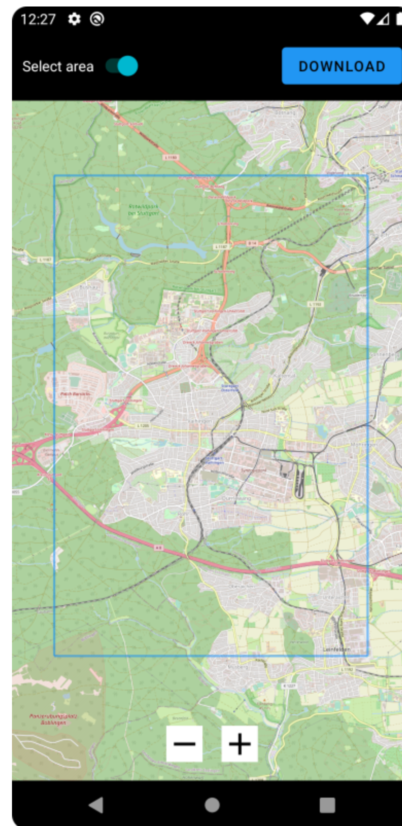


Abbildung 6.8: Der heruntergeladene Bereich ist durch ein blaues Rechteck gekennzeichnet.

Startet der Nutzer einen Download, während bereits ein (anderer) Bereich heruntergeladen ist, so wird vor dem Download zuerst geprüft, ob bereits heruntergeladene Tiles wiederverwendet werden können. Entsprechend werden alle Tiles, welche sich außerhalb des neuen Bereichs befinden, gelöscht, während der Downloadprozess das Herunterladen aller Tiles überspringt, welche sich bereits im lokalen Speicher des Geräts befinden. Der Subgraph hingegen wird jedes Mal neu konstruiert, da dieser nicht zuletzt aufgrund der in Kapitel 4 auf Seite 23 diskutierten Herausforderungen nicht einfach aus dem existierenden Subgraphen extrahiert werden kann.

Sobald der Nutzer einen Bereich im Online-Modus heruntergeladen hat, kann er die Funktionen des Offline-Modus nutzen.

6.2.2 Offline-Modus

Bevor im Online-Modus Kartendaten heruntergeladen werden, findet der Nutzer im Offline-Bereich eine leere Karte vor sich, mit der keine weitere Interaktion möglich ist. Sobald Daten offline zur Verfügung stehen, kann der Nutzer den Offline-Modus nutzen. In diesem greift *Osmdroid* auf die lokalen Tiles zu und stellt aus diesen eine Karte zusammen. Dabei wird eine eventuell bestehende Internetverbindung ignoriert, sodass keine weiteren Tiles während der Sitzung geladen werden. Das verhindert einen durch den Nutzer ungewollten Gebrauch der Verbindung, die je nach Aufenthaltsort unter anderem Kosten verursachen könnte. Eventuell im Cache von *Osmdroid*

befindliche Tiles werden aber trotzdem angewandt, da sie ohnehin bereits lokal vorhanden sind.

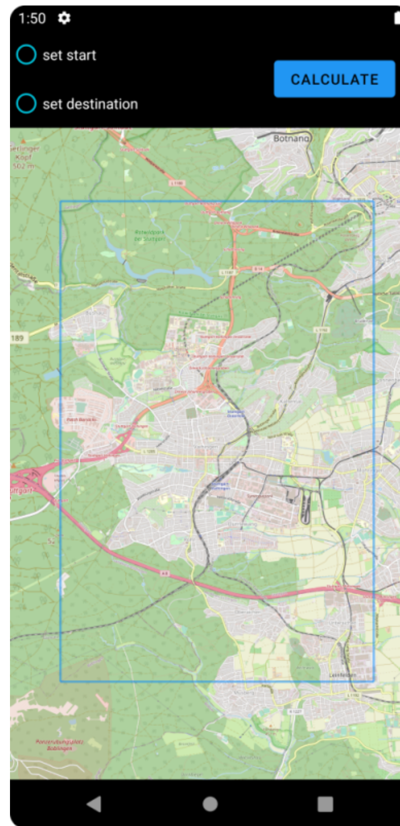


Abbildung 6.9: Die Standardansicht der Offline-Karte.

Als Orientierung dafür, welcher Bereich vom Subgraph abgedeckt wird und damit für Pfadberechnungen in Frage kommt, wird die Kennzeichnung anhand eines blauen Rechtecks aus dem Online-Modus übernommen.
Auf der Offline-Karte kann ein Nutzer zwei Punkte auf der Karte auswählen, indem er eine der beiden Möglichkeiten in der oberen linken Ecke auswählt. Ist entweder die Möglichkeit zum Setzen eines Start- oder Zielpunktes aktiv, lässt sich per Drücken einer Stelle auf der Karte der entsprechende Punkt setzen. Ein auf diese Weise gesetzter Punkt befindet sich dabei immer auf einem Knoten des Graphen. Dafür wird bei der Interaktion des Nutzer mit dem Display die GPS-Koordinate der Stelle des Fingers auf der Karte ausgelesen und mit den Knoten des Subgraphen verglichen. Hierbei wird auf dem Subgraphen der naheliegendste Knoten anhand der euklidischen Distanz bestimmt, und der Start- beziehungsweise Zielpunkt an den Koordinaten des Punktes gesetzt. Wäre dies nicht der Fall, müsste diese Berechnung spätestens bei der Ausführungs des Dijkstra-Algorithmus ohnehin durchgeführt werden, da die Pfadberechnung nur auf Knotenpunkten des Graphen, welche nur einen Anteil der gesamten Karten prägen, ausgeführt werden kann. In der Realität funktioniert dieser Ansatz aber, da die Auflösung des Graphen hoch genug ist, wodurch eine nahezu detailgetreue Pfaddsuche gewährleistet ist. Dementsprechend ist ein direktes Feedback an den Nutzer durch Rasterisierung des Selektionspunktes eine sinnvolle Umsetzung. Ein weiterer, nicht verwendeter Ansatz, wäre die Berechnung des nächsten Knoten bei der Ausführung des Dijkstra-Algorithmus und Einzeichnen einer Linie vom selektieren Start-/Zielpunkt zu seinem tatsächlichen Äquivalent. Dadurch wird dem Nutzer vermittelt, dass der berechnete Pfad lediglich

bis zu einem gewissen Punkt gilt, und der letzte Abschnitt des Pfads aufgrund fehlender Daten nicht durchgeführt werden kann.

Sind beide Punkte gesetzt, lässt sich die kürzeste Route vom Start- zum Zielknoten anhand der Kantengewichte berechnen. Dafür wird ein bidirektionaler Dijkstra-Algorithmus (Abschnitt 2.2 auf Seite 13) verwendet, welcher zudem für die Verwendung der CH auf dem Subgraphen modifiziert wurde.
Dieser initialisiert zu Beginn je einen Array für die Distanz aller Knoten vom Start- und vom Zielknoten und setzt die Werte auf unendlich (hier in Form des Maximalwerts eines Elements der Klasse *Double*). Des Weiteren existieren zwei Arrays, welche sich für je eine Richtung die IDs der Knotenvorgänger jedes Knoten merken, sowie zwei weitere Arrays, welche sich die Menge der bereits abgearbeiteten Knoten vermerken. Zuletzt werden zwei Priority Queues für Knoten erstellt, eine Vorwärts- und eine Rückwärts-Queue, welche zu Beginn den Start- beziehungsweise Zielknoten enthalten. Diese enthalten zu jedem Zeitpunkt eine Menge an Knoten, welche stets anhand deren Distanz zum Startknoten für die Vorwärts-Queue beziehungsweise zum Zielknoten für die Rückwärts-Queue sortiert sind.
Der Algorithmus arbeitet nun iterativ, wobei abwechselnd eine Iteration des Vorwärts- und eine des Rückwärts-Dijkstra durchgeführt wird, nach der in (Abschnitt 2.2 auf Seite 13) präsentierten allgemeinen Arbeitsweise eines bidirektionalen Dijkstra. Das heißt, es wird (für einen Schritt des Vorwärts-Dijkstra) das erste Element der Vorwärts-Queue entnommen und geprüft, ob dieser bereits in einer vorherigen Iteration fertig bearbeitet wurde. Gilt dieser nicht als abgearbeitet, werden für den Knoten alle ausgehenden Kanten betrachtet, deren Zielknoten ein höheres Level als er selbst aufweisen. Ist für eine daraufhin betrachtete Kante zu einem Zielknoten die neue Distanz geringer als die bisher gespeicherte, so wird die Distanz des Nachbarknoten aktualisiert und dieser in die Queue aufgenommen. Die Shortcut-Kanten der CH werden dabei durch den Algorithmus wie gewöhnliche Kanten interpretiert und entsprechend für die Berechnung des kürzesten Pfades gehandhabt. Für aktualisierte Distanzen wird der Knotenvorgänger auf den entnommenen Knoten gesetzt. Zudem wird dabei betrachtet, ob die Distanz zu diesem Knoten bereits durch den Rückwärts-Dijkstra gesetzt wurde, in welchem Fall ein Treffpunktknoten und damit ein potentieller kürzester Pfad gefunden ist. Der entsprechende Knoten sowie die Summe der beiden Distanzen werden vermerkt. Am Ende der Iteration wird der Knoten als fertig bearbeitet markiert und der Algorithmus wird auf Abbruchkonditionen geprüft. Diese sind erfüllt, sobald ein eventuell existierender Gesamtpfad geringere Kosten aufweist, als sowohl das erste Element der aktuellen Vorwärts-Queue als auch der Rückwärts-Queue, in welchem Fall es sich um den kürzesten Pfad handelt. Eine andere Möglichkeit ist, dass beide Queues leer sind und noch kein Pfad gefunden wurde, wobei in diesem Fall kein Pfad zwischen beiden selektierten Punkten existiert.
Ist keine Abbruchbedingung erfüllt, so folgt eine Iteration der anderen Algorithmus-Instanz (in diesem Fall des Rückwärts-Dijkstra). Dabei ist das Vorgehen gleich zum Vorwärts-Dijkstra, mit dem Unterschied, dass die eingehenden Kanten des aus der Queue entnommenen Knoten betrachtet werden. Eine Übersicht über die beiden Algorithmen finden sich in Algorithmus 6.1 auf der nächsten Seite für den Vorwärts-Dijkstra und Algorithmus 6.2 auf der nächsten Seite für den Rückwärts-Dijkstra.

Algorithmus 6.1 Vorwärts-Dijkstra

$Distance_{start} \longleftarrow$ Array of distances from start node
node u $\longleftarrow$ get first element of Queue-Forward
for all outgoing edges $e = (u, v)$ of u **do**
  if $Level(u) < Level(v) \wedge v$ not settled **then**
    if $Distance_{start}(u) + Weight(e) < Distance_{start}(v)$ **then**
      $Distance_{start}(v) = Distance_{start}(u) + Weight(e)$
      $EdgePredecessor(v) = u$
      if $Distance_{destination}(v)$ is set **then**
        check if $Path(start, v) + Path(v, destination)$ is shortest path
      end if
    end if
    Queue-Forward $\longleftarrow v$
  end if
end for
set node u as settled

Algorithmus 6.2 Rückwärts-Dijkstra

$Distance_{start} \longleftarrow$ Array of distances from destination node
node v $\longleftarrow$ get first element of Queue-Backwards
for all incoming edges $e = (u, v)$ of v **do**
  if $Level(v) < Level(u) \wedge u$ not settled **then**
    if $Distance_{destination}(v) + Weight(e) < Distance_{destination}(u)$ **then**
      $Distance_{destination}(u) = Distance_{destination}(v) + Weight(e)$
      $EdgePredecessor(u) = v$
      if $Distance_{start}(u)$ is set **then**
        check if $Path(start, u) + Path(u, destination)$ is shortest path
      end if
    end if
    Queue-Backwards $\longleftarrow u$
  end if
end for
set node v as settled

Sobald eine Abbruchkondition erfüllt ist, terminiert der gesamte Algorithmus und gibt den berechneten kürzesten Pfad aus, sollte dieser existieren. Wurde kein solcher Pfad gefunden, so liegt das entweder daran, dass der heruntergeladene Subgraph zu klein ist, und ein potentiell existierender Pfad nicht vollständig extrahiert wurde, oder dass selbst auf dem originalen Graphen kein Pfad existiert, in welchem Fall nur eine andere Wahl von Start- oder Zielpunkt das Problem lösen kann. Existiert ein Pfad, so enthält dieser meist Shortcut-Kanten, welche zunächst entpackt werden müssen. Dazu wird für jede Kante im Pfad geprüft, ob sie Kindkanten besitzt, und bei Bedarf entsprechend im Pfad durch ihre Kinder ersetzt. Da es sich bei den Kindkanten ebenfalls um Shortcut-Kanten handeln kann, wird jedes Kind einer entpackten Kante rekursiv auf Kinder getestet und wenn nötig entpackt.

Ist der Pfad vollständig entpackt, wird dieser auf der Karte angezeigt, indem die beinhalteten Kanten eingefärbt werden (Abbildung 6.10). Der Nutzer kann dann anhand der Route den effizientesten Weg vom Start- zum Zielknoten navigieren. Damit während der, wenn auch kurzen,

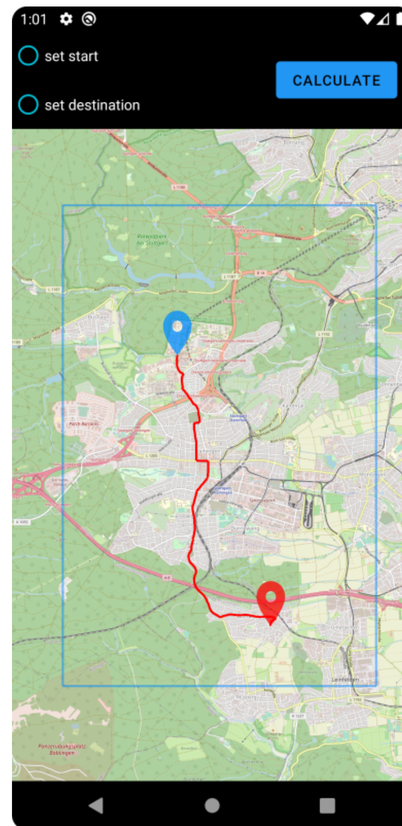


Abbildung 6.10: Das Ergebnis einer erfolgreichen Pfadsuche mit dem bidirektionalen Dijkstra-Algorithmus. Die Kanten des gefundenen kürzesten Pfads sind auf der Karte rot hervorgehoben.

Rechenzeit des Dijkstra-Algorithmus die App nicht einfriert, wurde der Algorithmus unter Nutzung eines separaten Threads parallelisiert.

7 Performanceanalyse

Das Folgende Kapitel beschäftigt sich mit der allgemeinen Performance der Applikation, in Bezug auf die Erzeugung von Offline-Daten durch den Download der Tiles und des Subgraphen, vor allem aber hinsichtlich der Laufzeit des Dijkstra-Algorithmus. Dabei soll speziell betrachtet werden, ob unter Verwendung von CH tatsächliche eine Optimierung der Laufzeit erreicht wird.

Durchgeführt wurden die Messungen auf einem Huawei P20 Lite mit Android API-Level 28. Die Geschwindigkeit der Internetverbindung lag bei im Durchschnitt bei etwa 45MBps.

7.1 Erzeugung der Offline-Daten

Da der Download zweigeteilt ist, ist auch die Frage der Dauer aus zwei Perspektiven zu betrachten. So wird sowohl die Dauer für den Download des Subgraphen, als auch für den Download aller Tiles gemessen, und anhand der Messungen Rückschluss darauf gezogen, was der primäre Faktor für die Gesamtlaufzeit des Prozesses ist. Dabei wird auch untersucht, wie sich diese Zeiten über größere werdende Gebiete entwickeln, da für einen Bereich aus wenigen Tiles ein anderes Ergebnis zu erwarten ist, als für ein großes, aus vielen Tiles bestehendes Gebiet. Um möglichst genau Ergebnisse zu liefern, wird zudem für jede Messung der interne Speicher der App zurückgesetzt, sodass nicht bereits heruntergeladene Tiles existieren, welche gelöscht oder wiederverwendet werden. Für jede Messung wurden zwei Durchläufe durchgeführt und der Mittelwert der zwei vorliegenden Zeiten genommen. Das Ergebnis findet sich in Tabelle 7.1 und Abbildung 7.2 auf Seite 49.

Anzahl Knoten/Kanten	Anzahl Tiles	Subgraph-Download [in sek.]	Tile-Download [in sek.]
1.200 / 4.000	100	3.315	2.27
10.000 / 35.000	1.000	5.45	3.99
90.000 / 320.000	10.000	36.94	35.43
340.000 / 665.000	50.000	161.47	173.50
550.000 / 1.100.000	100.000	257.93	303.45
840.000 / 3.100.000	200.000	431.95	622.42

Tabelle 7.1: Tabelle mit den Ergebnissen der Messungen. Größenangaben der Tiles und des Graphen sind dabei zur Übersichtlichkeit gerundet

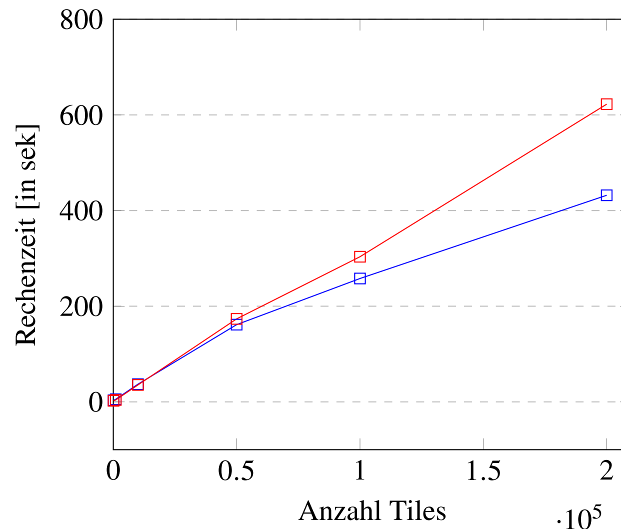


Abbildung 7.1: Plot der Messungen aus Tabelle 7.1 auf der vorherigen Seite. Die blaue Linie stellt dabei die Werte des Downloads des Subgraphen dar, die rote Linie die des Downloads der Tiles.

Anhand der Messungen zeigt sich, dass für sehr kleine Gebiete der Graph und der Download der Tiles eine vergleichbare Dauer aufweisen, mit steigender Tileanzahl aber eine zunehmend größer werdende Differenz bilden. So steigt die Dauer des Tile-Downloads annähernd linear, da der gleiche Prozess lediglich öfter ausgeführt wird. Das Berechnen der CH, welche beim Download des Subgraphen der primäre Faktor für die Rechenzeit ist, wird dagegen zunehmend effizienter mit größer werdendem Graphen, und steigt weniger als linear.
Es lässt sich also festhalten, dass für das Herunterladen eines beliebig großen Bereichs der Karte der Download der Tiles die Gesamtdauer des Prozesses bestimmt.

7.2 Optimierung durch Contraction Hierarchies

Um zu testen, ob die Berechnung einer CH und deren Verwendung für die Ausführung des Dijkstra-Algorithmus einen signifikanten Effekt hat, werden Messungen der Laufzeit vorgenommen. Dabei wird ein Vergleich zwischen dem implementierten, modifizierten Dijkstra für CHs mit einem normalen Dijkstra unternommen. Beide Algorithmen werden auf steigende Distanzen ausgeführt und deren Laufzeiten festgehalten. Die Ergebnisse finden sich in Tabelle 7.2 auf der nächsten Seite. Pro Kategorie wurden die Laufzeit von zehn Routen mit jeweils zehn Ausführungen gemessen, in Summe also 100 Messungen pro Kategorie. Die festgehaltenen Werte stellen also den Mittelwert der übrigen 8 Zeiten dar. Der verwendete Graph für die Rechnungen besteht aus etwa 800.000 Knoten und 2,8 Millionen Kanten. Aus Gründen der Nachvollziehbarkeit finden sich einige beispielhafte Abbildungen der berechneten Routen am Ende dieses Kapitels. (Abschnitt 7.2 auf Seite 50).

Distanz (Kosten)	Rechenzeit CH-Dijkstra [in ms.]	Rechenzeit Dijkstra [in ms.]
10.000	1.28	1.9
100.000	3.47	144.54
500.000	11.13	1556.69
1.000.000	13.79	3466.10

Tabelle 7.2: Tabelle mit den Ergebnissen der Messungen

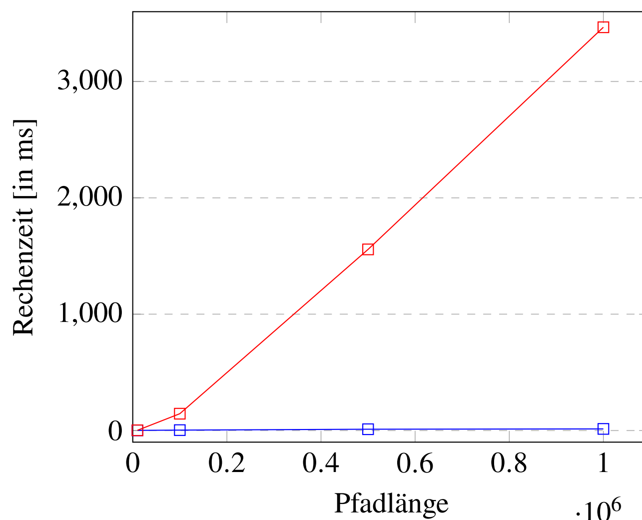


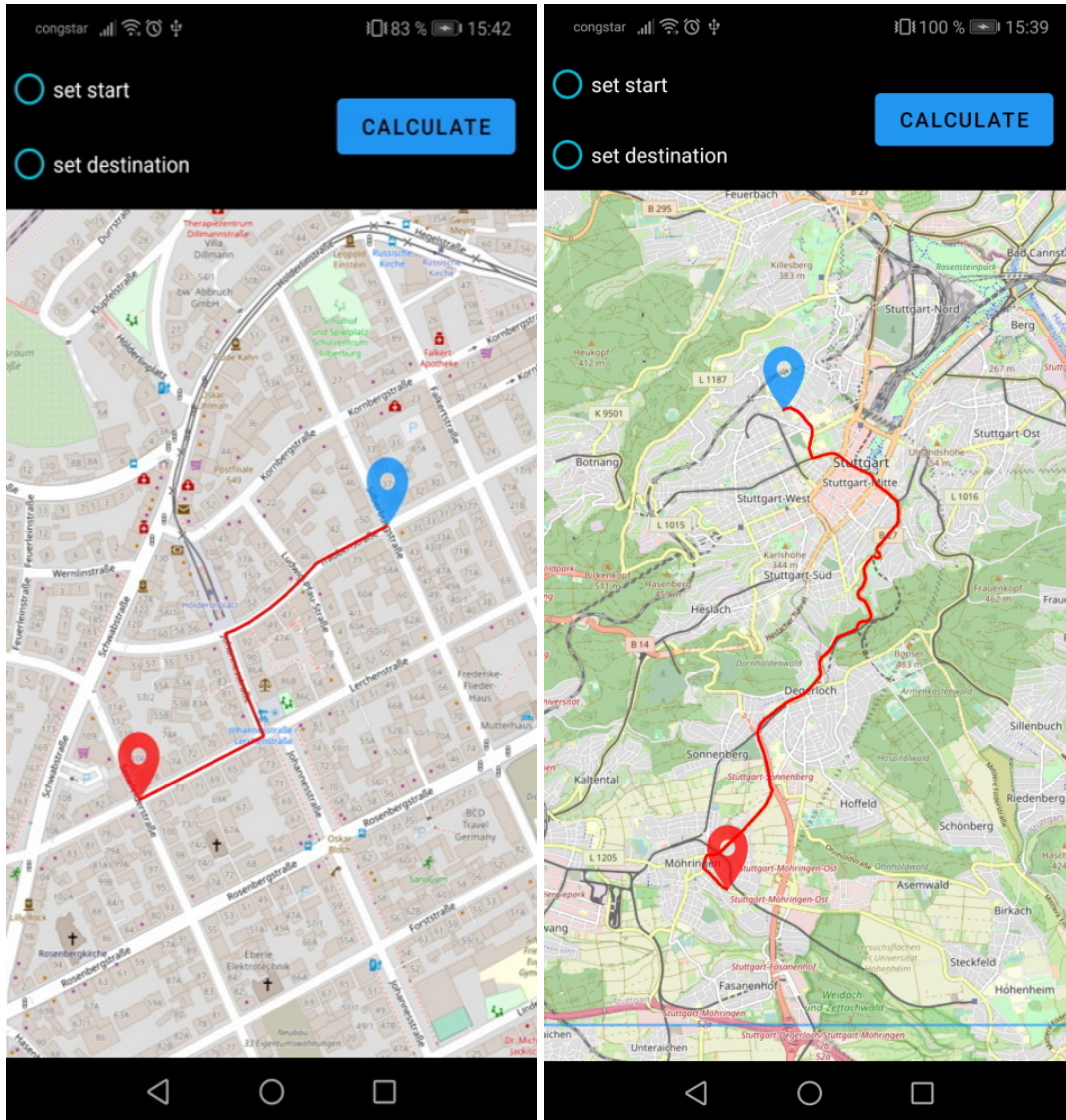
Abbildung 7.2: Plot der Messungen aus Tabelle 7.2. Die blaue Linie stellt dabei die Laufzeit des für CHs angepassten Dijkstra dar, die roten Linie die des normalen Dijkstra.

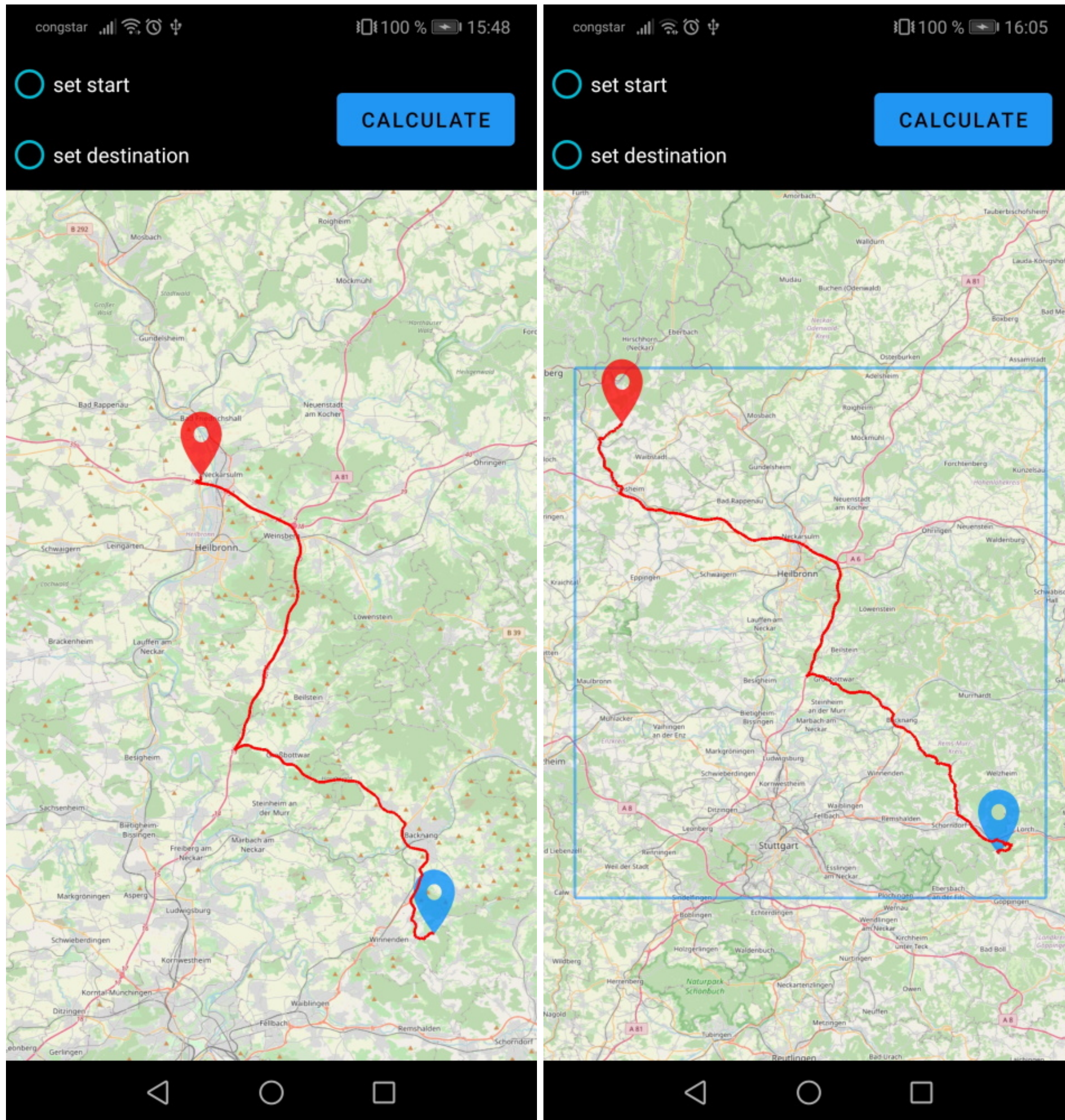
Anhand der Werte lässt sich eine signifikante Differenz der Laufzeiten der beiden Algorithmen feststellen. So weist der normale Dijkstra bei der Pfadberechnung auf sehr kurze Distanz noch äquivalente Rechenzeit auf. Aufgrund der Arbeitsweise des Dijkstra ist dies aber nur natürlich, da die Existenz einer CH auf solch kleine Distanzen kaum bis gar nicht genutzt werden kann. Mit zunehmender Distanz des Start- und Zielpunktes, und dem damit zunehmenden Rechenaufwand, steigen zwar beide Rechenzeiten an, jedoch lässt sich eine deutliche Differenz im Umfang der Zunahme feststellen. So benötigt der CH-optimierte Dijkstra für fast alle berechneten Pfade lediglich einen Bruchteil der Rechenzeit des normalen Dijkstra.
Es wurde also tatsächlich eine signifikante Optimierung der Laufzeit durch Verwendung von CHs erreicht.

An dieser Stelle sei aber angemerkt, dass die Werte mit Vorsicht zu betrachten sind. Für zwei Strecken mit gleichen Kosten können signifikant voneinander abweichende Rechenzeiten auftreten, abhängig davon, wie der der Graph und die CH konstruiert sind. Trotz allem ist eine Aussage über die Tendenz der Laufzeit bezüglich der beiden Algorithmen anhand der Ergebnisse eindeutig möglich.

Bilder der berechneten Pfade

Im Folgenden finden sich einige berechnete Pfade für die Messungen aus Tabelle 7.2 auf der vorherigen Seite, wobei jeweils ein Beispielpfad pro Kategorie aufgezeigt wird. Die Bilder sind in aufsteigender Reihenfolge sortiert, sodass das Erste den Pfad mit Kosten 10.000 zeigt und das Letzte den mit Kosten 1.000.000.




congstar
100 %
15:48
set start
CALCULATE
set destination
Heilbronn
congstar
100 %
16:05
set start
CALCULATE
set destination
Heilbronn
Stuttgart

8 Zusammenfassung und Ausblick

Diese Arbeit hat sich mit Herausforderungen und Auswirkungen der Nutzung von CHs auf Graphen beschäftigt. Dabei wurde gezeigt, dass diese einen signifikanten Effekt auf die Laufzeit eines Pfadsuch-Algorithmus wie dem Dijkstra besitzen.
Ein weiteres Thema war die effektive und korrekte Extraktion von Teilgraphen aus einem Graphen mit existierender CH. Hierfür wurden verschiedene Ansätze diskutiert und diese auf potentielle Probleme untersucht. So war die Findung von Lösungsansätzen zur Gewährleistung von Pfadkorrektheit und dem Entfernen von nicht expandierbaren Shortcut-Kanten der CH ein wichtiger Teil dieser Arbeit. Die aufgezeigte und letztlich sogar implementierte Lösung einer Berechnung der CH erst auf dem Subgraphen ist dabei nicht nur ein in dieser Arbeit funktionaler Ansatz, sondern lässt sich auch auf andere Arbeiten im Themengebiet von Graphen und CHs anwenden.

Insgesamt sind CHs also eine kostengünstige Optimierung für die vielseitig existente Thematik der Pfadsuche. Ein überschaubarer und zudem einmaliger Rechenaufwand sowie die in der heutigen Zeit eher triviale Problematik von erhöhtem Speicherbedarf sind dabei ein geringer Preis für eine signifikant bessere Laufzeit.

8.1 Ausblick

Im Bezug auf das in dieser Arbeit implementierte Projekt bestehen noch einige Punkte, die optimiert werden können, um die Laufzeit einzelner Prozesse weiter zu verbessern. So kann der Graph bereits beim Einlesen durch den Server neu angeordnet werden, um in späteren Schritten schnellere Entscheidungen bei Fallunterscheidungen zu treffen. Das Problem unzureichenden Zwischenspeichers für große Graphen ist ebenfalls ein interessantes Forschungsthema. Auch der implementierte Dijkstra-Algorithmus kann weiter optimiert werden, sowohl bezüglich der Speichereffizienz als auch der Laufzeit. Speziell das doppelte Vorbehalten von Arrays zum speichern von Informationen ist ein Punkt, welcher optimiert werden könnte.

Außerhalb dieser Arbeit bleiben auch CHs weiterhin relevantes und vor allem vielversprechendes Forschungsthema, da sie großes Potential, auch für weitere Optimierungen, zeigen.

Anmerkungen

Die in dieser Arbeit enthaltenen Bilder von Graphen wurden mithilfe von [Uni] erstellt.

Literaturverzeichnis

[And] Andre Nusser. *CH-Constructor*. (nur innerhalb des Universitäts-Netzwerks erreichbar). URL: theogit.fmi.uni-stuttgart.de/nusserae/chconstructor (besucht am 06. 04. 2022) (zitiert auf S. 21, 31, 34).

[Bel] E. Belinski. *Android API Levels*. URL: https://apilevels.com/#fn:2 (besucht am 24. 04. 2022) (zitiert auf S. 33).

[Fun19] Funke, Stefan. *Lecture notes for Computational Geometry*. (nur innerhalb des Universitäts-Netzwerks erreichbar). 2019. URL: https://fmi.uni-stuttgart.de/files/alg/teaching/s19/compgeo/scribe_notes_19.pdf (besucht am 14. 04. 2022) (zitiert auf S. 15).

[Gra] Gramlich, Nicolas and others. *Osmdroid*. URL: github.com/osmdroid/osmdroid (besucht am 05. 04. 2022) (zitiert auf S. 16, 36).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. „Contraction hierarchies: Faster and simpler hierarchical routing in road networks". In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, S. 319–333. (Besucht am 11. 04. 2022) (zitiert auf S. 13).

[GSSV12] R. Geisberger, P. Sanders, D. Schultes, C. Vetter. „Exact routing in large road networks using contraction hierarchies". In: *Transportation Science* 46.3 (2012), S. 388–404. (Besucht am 11. 04. 2022) (zitiert auf S. 13).

[Ope17] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2017. (Besucht am 14. 04. 2022) (zitiert auf S. 16).

[Uni] Unick Soft. *Graphonline*. URL: https://graphonline.ru/de/ (besucht am 13. 04. 2022) (zitiert auf S. 55).

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