Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Personalisierbarer Android-Offline-Routenplaner

Niven Ratnamaheson

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Felix Weitbrecht, M.Sc.
Beginn am:	16. November 2021
Beendet am:	16. Mai 2022

Kurzfassung

Routenplaner sind heutzutage selbstverständlich geworden. Das Gebiet der klassischen Routenplanung bietet jedoch keine Antwort auf individuelle Anforderungen eines Nutzers. Diese Anforderungen eröffnen ein anderes Gebiet der Routenplanung: Das sogenannte *personalized route planning*. Das Hauptproblem des *personalized route planning* liegt bei der dynamischen Gewichtung der Kanten in dem zu untersuchenden Graphen. Diese Gewichtung verhindert die Nutzung etablierter Algorithmen der klassischen Routenplanung, die einen Vorbereitungsschritt benötigen.

In dieser Bachelorarbeit wird die Konzeption und Entwicklung eines personalisierbaren Android-Offline-Routenplaners vorgestellt und dokumentiert. Dabei wird ein nachvollziehbarer Ansatz unter Berücksichtigung des Prinzips des *personalized route planning* und einer Offlinefunktion konzipiert und verfolgt. Die Entwicklung umfasst die Implementierung eines *Servers* mit Java 17 und einer Android-Applikation mit Kotlin. Die für die Implementierung benötigten Geoinformationen werden einerseits aus den SRTM-Datensätzen und andererseits über den *OsmGraphCreator* des FMI der Universität Stuttgart aus OpenStreetMap Datensätzen extrahiert.

Das Ergebnis der Arbeit ist eine Android-Applikation, die das Smartphone zu einem unabhängigen Routenplaner macht. Der Nutzer der Applikation kann einen Kartenausschnitt des deutschen Straßennetzwerks herunterladen und eine durch drei Metriken (Zeit, Distanz und positive Höhendifferenz) personalisierbare Route berechnen lassen.

In einer abschließenden Evaluation wird auf Hindernisse, die im Laufe der Arbeit auftraten, eingegangen, und die gesamte Implementierung auf Performance und Skalierbarkeit überprüft.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Android-App Android-Applikation. 9

API Application Programming Interface. 19

CPU central processing unit. 49

HGT Height. 24

HTTP Hypertext Transfer Protocol. 26

ID Identification. 16

OSM OpenStreetMap. 19

Radar radio detection and ranging. 19

SRTM Shuttle Radar Topography Mission. 19

URI Uniform Resource Identifier. 29

URL Uniform Resource Locator. 31

1 Einleitung

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint, soweit es für die Aussage erforderlich ist.

Routenplaner sind heutzutage selbstverständlich geworden. Mit einem internetfähigen Smartphone ist es ohne Aufwand möglich, einen meist schon vorinstallierten Routenplaner zu nutzen, um innerhalb einer kurzen Zeitspanne eine Route zwischen zwei Standorten zu erhalten. Diese Route ist in der Regel darauf optimiert, in Bezug auf den kürzesten Zeitaufwand ein Ergebnis zu liefern. Dabei werden verschiedene Faktoren miteinbezogen, wie zum Beispiel die Live-Daten des Verkehrsgeschehens. Diese Faktoren werden dabei vom Anbieter festgelegt und können, falls überhaupt, nur minimal vom Nutzer geändert werden. Zur Verdeutlichung können bei der Routenplanung mit Google Maps lediglich verschiedene Straßentypen gemieden, verschiedene Transportmittel gewählt und verschiedene Optionen für die Geschwindigkeit bei der Nutzung eines Kraftfahrzeugs selektiert werden. So werden jedem Nutzer in der Regel die gleichen oder zumindest sehr ähnliche Routen vorgeschlagen. Es wird also nicht berücksichtigt, dass unterschiedliche Personen eine individuelle Vorstellung der besten Route zwischen zwei Standorten haben können. So kann es einer Person wichtig sein, eine Strecke so schnell wie möglich zu bewältigen. Einer anderen könnte es zum Beispiel wichtig sein, größere Städte zu vermeiden, Sprit zu sparen oder Sehenswürdigkeiten während der Fahrt zu besuchen. Die Anforderungen an eine Route sollten somit so individuell anzupassen sein, wie Personen individuell sind. Herkömmliche Routenplaner kommen bei diesen Anforderungen folglich an ihre Grenze.

Die Forderung nach einer personalisierbaren Route deklariert ein neues Gebiet im Bereich der Routenplanung. Das sogenannte *personalized route planning* [FS15] bringt im Vergleich zur klassischen Routenplanung neue Probleme mit sich. Die klassische Routenplanung basiert auf Algorithmen, die auf einem gerichteten, gewichteten Graphen (siehe Definition 2.1.1) mit statischer Gewichtung der Kanten arbeiten. In den letzten Jahrzehnten wurden verschiedene Algorithmen entwickelt, die den Basisalgorithmus von Dijkstra (siehe Abschnitt 2.1) beschleunigen. Diese Algorithmen basieren in der Regel auf zwei Schritten. Der erste Schritt ist ein Vorverarbeitungsschritt. In diesem werden die gegebenen Informationen des Graphen mit spezifischen Zusatzinformationen angereichert. Diese Zusatzinformationen beschleunigen dann die Kürzeste-Wege-Suche im zweiten Schritt.

Wenn dieser zweiteilige Ansatz nun auf das *personalized route planning* übertragen wird, ergibt sich ein Problem. Das Prinzip des *personalized route planning* fordert, dass ein Nutzer seine Anforderungen an die Route durchgehend und vollkommen individuell verändern kann. Somit kann die Routenberechnung zwar auf einem gerichteten, gewichteten Graphen (siehe Definition 2.1.1) stattfinden, sobald der Nutzer jedoch seine Anforderungen ändert, ändert sich auch die Gewichtung der Kanten. Die Gewichtungen haben folglich dynamische Werte. Dieser Fakt verhindert die Übertragung eines in der klassischen Routenplanung verwendeten Vorverarbeitungsschritts auf

das *personalized route planning*. Folglich musste für die Optimierung der Berechnung einer personalisierbaren Route wieder vom *Dijkstra-Algorithmus* (siehe Abschnitt 2.1) ausgegangen werden.

Mittlerweile gibt es Ansätze, die den *Dijkstra-Algorithmus* (siehe Abschnitt 2.1) beschleunigen. Dieser Beschleunigungsfaktor ist jedoch noch lange nicht auf dem Niveau der Algorithmen für die klassische Routenplanung. Es ist außerdem anzuzweifeln, dass ein vergleichbarer Beschleunigungsfaktor möglich ist. [FS15]

Ein anderer Aspekt, der immer noch ein Problem darstellt, sind Qualität, Abdeckung und Kosten des Mobilfunknetzes. Ein Nutzer kann sich nicht darauf verlassen, dass er in jeder Situation und an jedem Ort mit seinem Smartphone eine Internetverbindung aufbauen kann. Der Anspruch an einen Routenplaner sollte somit sein, dass er auch ohne Internetverbindung Routen berechnen kann. Dass wiederum impliziert die Nutzung des Smartphones zur Berechnung der Route. Da Smartphones heutzutage in der Regel sehr leistungsstark sind, scheint das im Rahmen der Möglichkeit.

Das Ziel dieser Arbeit ist die Konzeption und Entwicklung eines personalisierbaren Offline-Routenplaners in Form einer *Android-Applikation (Android-App)*. Dieser soll eine Route nach dem Prinzip des *personalized route planning* berechnen, wobei die Metriken Zeit, Distanz und positive Höhendifferenz beachtet werden. Diese Routenberechnung soll dabei auf dem Smartphone, also offline, stattfinden. Die spezifische Aufgabenstellung findet sich in Abschnitt 1.1.

In Kapitel 2 werden zunächst die grundlegenden Begrifflichkeiten und Definitionen für die Implementierung der *Android-App* dargelegt. Hierbei wird auf den *Dijkstra-Algorithmus*, das Prinzip des *personalized route planning*, die Datenstruktur zur Graphdarstellung, die Prozedur der *Dreiecksinterpolation* und auf die Herkunft der Rohdaten eingegangen. Schließlich werden essenzielle Komponenten der Android-Programmierung erläutert.

In Kapitel 3 findet sich der Hauptteil der Arbeit, der sich mit der Umsetzung des Projekts befasst. Hier werden grundlegenden Komponenten des Projekts genannt und der Ansatz erläutert. Es folgt die Dokumentation des Ansatzes, indem zuerst die genutzten Daten, dann der *Server* und schließlich die *Android-App* beschrieben werden.

Kapitel 4 beinhaltet die Evaluation des Projekts, wobei auch auf Hindernisse, die im Laufe der Arbeit auftraten, eingegangen wird. Darüber hinaus wird in diesem Kapitel die Performance und Skalierbarkeit der Implementation diskutiert.

Den Abschluss der Arbeit bildet das Kapitel 5, das eine Zusammenfassung und einen Ausblick beinhaltet. Hierbei werden die Ergebnisse dieser Bachelorarbeit zusammengefasst und ein Ausblick auf weitere Entwicklungsmöglichkeiten gegeben.

1.1 Aufgabenstellung

Aufgabenstellung war die Konzeption und Entwicklung eines Offline-Routenplaners für Android mittels Kotlin. Die Implementierung sollte die folgenden Features und Komponenten umfassen:

Einen herunterladbaren Kartenausschnitt Es soll ein Kartenausschnitt auswählbar sein. Die App soll über eine Datenverbindung den Subgraphen und die Map-Tiles des Ausschnitts herunterladen.

Offline-Routing Der Nutzer soll die Möglichkeit haben, eine Route zwischen einem selbst ausgewählten Start- und Zielpunkt offline berechnen zu lassen. Dies soll über die heruntergeladenen Daten vollzogen werden.

Server Es soll ein *Server* implementiert werden, der den Graphen des Straßennetzwerks Deutschlands bereithält und bei einer Anfrage einen bestimmten Subgraphen des gesamten Graphs zurückgibt.

Personalisierbarkeit Der Nutzer soll die Möglichkeit haben, die gewünschte Route zu personalisieren. Dabei sollen drei Metriken (Zeit, Distanz und die positive Höhendifferenz) beachtet werden. Die Priorisierung der einzelnen Metriken soll mithilfe eines Dreiecks in der UI stattfinden. Insbesondere muss die positive Höhendifferenz im Vorfeld aus einer Höhenkarte extrahiert werden.

Fortbewegungsart Die Routenberechnung soll neben der Personalisierbarkeit auch mit verschiedenen Transportmitteln (zu Fuß, mit dem Fahrrad, mit dem Auto) berechnet werden.

2 Präliminarien

Im folgenden Kapitel werden alle grundlegenden Begriffe und Definitionen eingeführt.

2.1 Dijkstra-Algorithmus

Der *Dijkstra-Algorithmus* gehört zu der Klasse der Greedy-Algorithmen und ist ein bekanntes Lösungsverfahren für die Kürzeste-Wege-Suche zwischen einem Start- und Zielknoten.

Der Algorithmus wird auf einem gewichteten Graphen ausgeführt. In dieser Arbeit wird ausschließlich der Fall für einen gerichteten und gewichteten Graphen betrachtet, was allerdings irrelevant für die Ausführung des Algorithmus ist.

Somit ist zu klären, was ein gerichteter, gewichteter Graph ist.

Definition 2.1.1
Sei $G = (V, E, \delta)$, wobei V als die Menge der Knoten und E als die Menge der Kanten definiert wird. E ist eine Teilmenge von $V \times V$ und δ eine Abbildung $E \rightarrow \mathbb{R}$, die die Gewichtung der einzelnen Kanten darstellt. Dann heißt G gerichteter, gewichteter Graph ([[KN09], S. 167-168]).

2.1.1 Algorithmus nach Dijkstra

E. W. Dijkstra definierte 1959 seine Lösung zur Kürzeste-Wege-Suche. Es folgt eine abgewandelte Definition des *Dijkstra-Algorithmus* auf Basis der Notiz Dijkstras ([Dij+59], S. 270):

Gegeben: Ein gewichteter Graph $G = V, E, \delta$. Ein Startknoten S und ein Zielknoten Z.

Gesucht: Der kürzeste Weg zwischen S und Z.

Für den Algorithmus werden jeweils 3 Teilmengen für die Knoten $v \in V$ und die Kanten $e \in E$ wie folgt definiert:

Teilmenge A: Beinhaltet die Knoten v, für die der kürzeste Weg zu S bestimmt wurde.

Teilmenge B: Beinhaltet die Knoten v, die einen Kandidaten für die A darstellen.

Teilmenge C: Beinhaltet die restlichen Knoten v.

Teilmenge I: Beinhaltet die Kanten e, die auf dem kürzesten Weg von S nach Z liegen.

Teilmenge II: Beinhaltet die Kanten e, die einen Kandidaten für I darstellen.

Teilmenge III: Beinhaltet die restlichen Kanten e.

Anfangs befinden sich alle Knoten v in C und alle Kanten e in III. Nun wird der Startknoten S nach A geschoben.

Schritt 1:

Es werden alle Kanten e, die von dem Knoten v_{weg} ausgehen und die zuletzt nach A geschoben wurde berücksichtigt. Alle Kanten e, die den besagten Knoten v_{weg} mit einem Knoten $v_{kandidat}$ verbinden, der in der Menge B oder C liegt, werden weiter betrachtet. Bei diesen Kanten wird geprüft, ob die Nutzung der jeweiligen Kante $e_{kandidat}$, unter Berücksichtigung ihrer Gewichtung δ, einen kürzeren Weg darstellt, als ein möglicherweise schon berechneter Weg zum Knoten $v_{kandidat}$ über eine andere Kante e in II. Falls das nicht zutrifft, wird der geprüfte Weg über die Kante $e_{kandidat}$ verworfen und die restlichen oben genannten Kanten werden überprüft. Falls das jedoch zutrifft, wurde ein kürzerer Weg zwischen dem Startknoten S und dem Knoten $v_{kandidat}$ über den Knoten v_{weg} und die Kante $e_{kandidat}$ gefunden. Es folgt die Verschiebung von $e_{kandidat}$ nach II. Dabei werden jegliche vorhandenen Einträge zu $e_{kandidat}$ in II überschrieben. Außerdem wird der Knoten $v_{kandidat}$ nach B verschoben, falls er in C lag.

Schritt 2:

Da der Weg zwischen dem Startknoten S und jeglichem Knoten $v_{kandidat}$ in B über die Kanten e in I und der zu ihm gehörenden Kante $e_{kandidat}$ in II klar definiert ist, wird jedem Knoten $v_{kandidat}$ in B eine Distanz zum Startknoten S zugeschrieben. Nun wird derjenige Knoten $v_{kandidat}$ von B nach A verschoben, dessen Distanz zum Startknoten S am geringsten ist. Die zu ihm gehörende Kante $e_{kandidat}$ wird von II nach I verschoben. Schließlich wird die Prozedur bei Schritt 1 fortgesetzt.

Dies wiederholt sich solange, bis der Zielknoten Z nach A geschoben wird. Falls dies passiert, wurde eine Lösung gefunden.

2.1.2 Beispiel zum Algorithmus

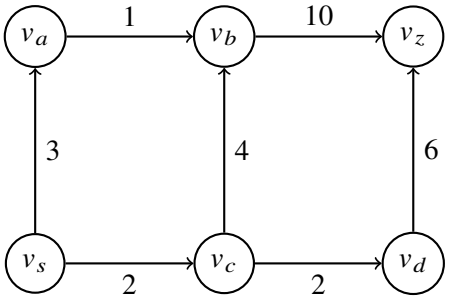
Als Beispiel wird der folgende Graph betrachtet mit Startknoten v_s und Zielknoten v_z.

Dabei werden die oben definierten Teilmengen nach jedem Durchlauf analysiert.

Initialisierung:

$$A = \{\}, B = \{\}, C = \{v_a, v_b, v_c, v_d, v_s, v_z\}$$

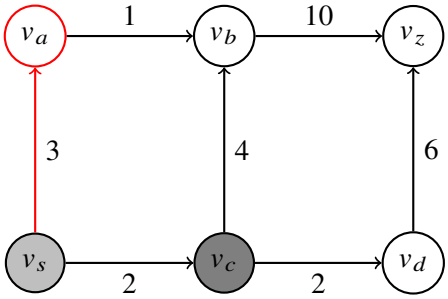
$$I = \{\}, II = \{\}, III = \{(v_s, v_a), (v_a, v_b), (v_b, v_z), (v_s, v_c), (v_c, v_b), (v_c, v_d), (v_d, v_z)\}$$



Nach Durchlauf 1:

$$A = \{v_s, v_c\}, B = \{v_a\}, C = \{v_b, v_d, v_z\}$$

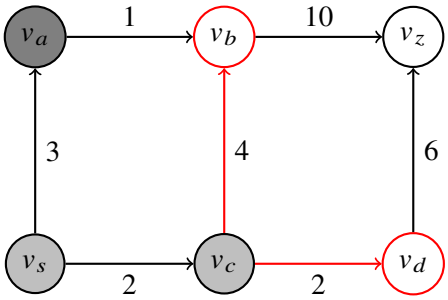
$$I = \{(v_s, v_c)\}, II = \{(v_s, v_a)\}, III = \{(v_a, v_b), (v_b, v_z), (v_c, v_b), (v_c, v_d), (v_d, v_z)\}$$



Nach Durchlauf 2:

$$A = \{v_s, v_c, v_a\}, B = \{v_b, v_d\}, C = \{v_z\}$$

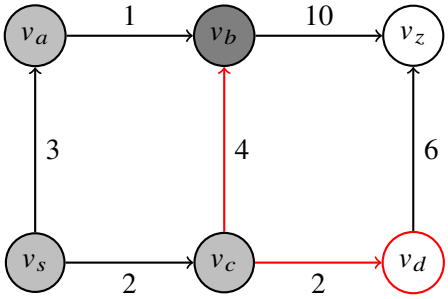
$$I = \{(v_s, v_c), (v_s, v_a)\}, II = \{(v_c, v_b), (v_c, v_d)\}, III = \{(v_a, v_b), (v_b, v_z), (v_d, v_z)\}$$



Nach Durchlauf 3:

$$A = \{v_s, v_c, v_a, v_b, v_d\}, B = \{v_z\}, C = \{\}$$

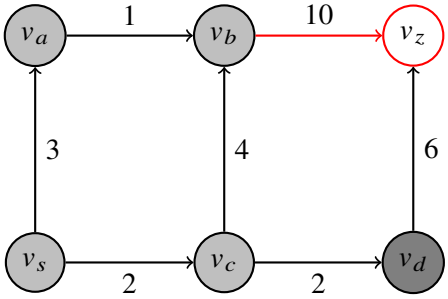
$$I = \{(v_s, v_c), (v_s, v_a), (v_a, v_b)\}, II = \{(v_c, v_b), (v_c, v_d)\}, III = \{(v_b, v_z), (v_d, v_z)\}$$



Nach Durchlauf 4:

$$A = \{v_s, v_c, v_a, v_b, v_d\}, B = \{v_z\}, C = \{\}$$

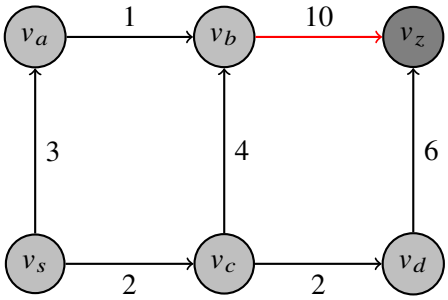
$$I = \{(v_s, v_c), (v_s, v_a), (v_a, v_b), (v_c, v_d)\}, II = \{(v_b, v_z)\}, III = \{(v_d, v_z)\}$$



Nach Durchlauf 5:

$$A = \{v_s, v_c, v_a, v_b, v_d, v_z\}, B = \{\}, C = \{\}$$

$$I = \{(v_s, v_c), (v_s, v_a), (v_a, v_b), (v_c, v_d), (v_d, v_z)\}, II = \{(v_b, v_z)\}, III = \{\}$$



Der Algorithmus ist beendet.

Bei der Betrachtung der Menge I wird der Weg über die Kanten $(v_s, v_c), (v_c, v_d), (v_d, v_z)$ und somit über die Knoten v_s, v_c, v_a, v_d, v_z gefunden.

Die Distanz des minimalen Weges von v_s nach v_z ist somit $2 + 2 + 6 = 10$.

2.1.3 Umsetzung des Algorithmus

Der oben beschriebene *Algorithmus nach Dijkstra* ist durch folgenden Pseudocode realisierbar:

```
1:  procedure DIJKSTRA(Graph G(V, E, δ), Startknoten, Zielknoten)
2:      minHeap: PriorityQueue
3:      vorgänger: Array mit n Einträgen
4:      distanz: Array mit n Einträgen
5:      for all v in V do
6:          vorgänger[v] = 0
7:          distanz[v] = ∞
8:      end for
9:      minHeap.add(Knoten(Startknoten,0))
10:     distanz[Startknoten] = 0
11:     while minHeap nicht leer do
12:         aktuellerKnoten = erster Eintrag von minHeap
```

```
13:         if aktuellerKnoten != Zielknoten then
14:             for all Nachbarn v von aktuellerKnoten do
15:                 distanzKandidat = distanz[aktuellerKnoten] + δ((aktuellerKnoten, v))
16:                 if distanzKandidat < distanz[v] then
17:                     distanz[v] = distanzKandidat
18:                     vorgänger[v] = aktuellerKnoten
19:                     minHeap.add(Knoten(v,distanzKandidat))      // Füge v in minHeap hinzu
20:                 end if
21:             end for
22:         else                                                    // Beendet den Algorithmus
23:             ergebnis = distanz[aktuellerKnoten]
24:             leere minHeap
25:         end if
26:         aktuellerKnoten ist abgeschlossen
27:         minHeap.remove(aktuellerKnoten)             // Lösche aktuellerKnoten aus minHeap
28:     end while
29: end procedure
```

Falls Interesse an dem Verlauf des kürzesten Weges vorhanden ist, wird am Ende des Algorithmus vom Zielknoten aus über die Elemente in *vörgänger* (siehe Zeile 2 Abschnitt 2.1.3) iteriert. Dieses Vorgehen wird unter Berücksichtigung der *Dijkstra*-Prozedur in Abschnitt 2.1.3, durch den folgenden Pseudocode verdeutlicht:

```
1: function findeWeg(vorgänger: Array, Startknoten, Zielknoten)
2:     weg: list
3:     aktuellerKnoten = Zielknoten
4:     weg.add(aktuellerKnoten)
5:     while aktuellerKnoten != Startknoten do
6:         aktuellerKnoten = vorgänger[aktuellerKnoten]
7:         weg.add(aktuellerKnoten)
8:     end while
9:     return weg
10: end function
```

2.2 Personalized Route Planning

Der Inhalt des Abschnitts bezieht sich auf die Arbeit von Funke und Storandt [FS15].

Das Problem des *personalized route planning* wird wie folgt definiert:

Gegeben ist ein Graph $G(V, E)$, der ein Straßennetzwerk repräsentiert. In der Menge V befinden sich die Knoten und in der Menge E die Kanten.

Für jede Kante $e \in E$ existiert jeweils ein n-dimensionaler nicht negativer Vektor $c(e) \in \mathbb{R}^n$, der die Kosten der Kante beinhaltet. Die Kosten der einzelnen Einträge $c_1, c_2, \cdots, c_n$ beziehen sich dabei auf verschiedene Metriken. Zum Beispiel kann sich c_1 auf die Distanz, c_2 auf die Höhendifferenz, und c_3 auf die Zeit beziehen. Eine Anfrage besteht aus einem Startknoten s, einem Zielknoten z und nicht negative Gewichtungen der Metriken $\alpha_1, \alpha_2, \cdots, \alpha_n$, wobei $s, z \in V$. Dabei werden die

α_i durch Faktoren definiert, wie zum Beispiel Tankpreis, Charakteristiken des Fahrzeugs oder einer manuellen Priorisierung der Metrik i durch den Nutzer. Das Ziel ist die Berechnung der Route r zwischen s und z, wobei r die Summe $\sum_{e \in r} \alpha^T c(e)$ minimiert.

Die Problematik dahinter sind die wechselnden Werte α_i. Diese verhindern die entwickelten Techniken der klassischen Routenplanung, die eine klassische Lösung über den Dijsktra-Algorithmus signifikant verbessern.

Für das *personalized route planning* gibt es auch schnellere Alternativen als den klassischen *Dijkstra-Algorithmus*. Aufgrund des hohen Aufwands der Alternativen wird in dieser Bachelorarbeit trotzdem der klassische Dijkstra genutzt.

2.3 Adjazenzarray

Der Inhalt dieses Abschnitt basiert auf dem Buch *Algorithms and Data Structures* von Mehlhorn und Sanders ([MS08] S. 169-171).

Um einen statischen Graphen $G(V, E)$ abzuspeichern, bietet sich ein *Adjazenzarray* an, da diese Datenstruktur einen schnellen Zugriff auf die ausgehenden Kanten eines spezifischen Knotens fördert.

Das Grundprinzip ist die Abspeicherung aller ausgehenden Kanten der Knoten in separate Subarrays, die mindestens den Zielknoten der einzelnen Kanten als Eintrag haben. Im statischen Fall ist es dann möglich die Subarrays zusammen in ein Kantenarray abzuspeichern. Dieser hat somit in seinen Einträgen repetitive Abschnitte von oben definierten Subarrays.

Zusätzlich zu diesem Kantenarray wird ein Hilfsarray erstellt, welcher mit $|V| + 1$ Einträgen initialisiert wird. Die Indizes des Hilfsarrays werden jeweils als Identification (ID) verwendet, wobei die ID mit einem spezifischen Knoten assoziiert werden muss. Die Einträge selbst entsprechen dabei der Startposition des zu dem Knoten gehörenden Subarrays. Der zusätzliche Eintrag am Ende des Hilfsarrays wird unter der Annahme, dass der erste Eintrag eines Arrays den Index 0 hat, mit dem Dummy-Wert $|E|$ initialisiert.

Um alle ausgehenden Kanten $e_{ausgehend}$ eines Knotens $v \in V$ durch das Kanten- und Hilfsarray zu bestimmen, wird zuerst die Anzahl der ausgehende Kanten von v berechnet. Die Form der Berechnung begründet die Notwendigkeit des Dummy-Werts.

$$|e_{ausgehend}| = Hilfsarray[v_{id} + 1] - Hilfsarray[v_{id}]$$

Die Zuordnung der ausgehenden Kanten $e_{ausgehend}$ erfolgt nun, indem ab der Startposition des zu v gehörenden Subarrays $|e_{ausgehend}|$ Einträge des Kantenarrays ausgelesen werden.

Als Beispiel dient der folgende Graph G:

Das Kantenarray:

B	C	D	A	C	D	F	E	F	E

Das Hilfsarray:

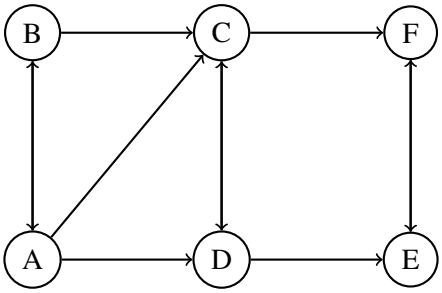


Abbildung 2.1: Graph G

0	3	5	7	8	9	10	11

Die Anzahl der von Knoten A ausgehenden Kanten ist $3 - 0 = 3$.

Einsetzen in das Kantenarray:

$$Kantenarray[0] = B$$
$$Kantenarray[0 + 1] = C$$
$$Kantenarray[0 + 2] = D$$

Somit gehen folgende Kanten vom Knoten A aus:

$$(A, B), (A, C), (A, D)$$

Äquivalent dazu verhält es sich mit der Bestimmung der ausgehenden Kanten für die restlichen Knoten.

2.4 Dreiecksinterpolation

Eine *Dreiecksinterpolation* ist über die *baryzentrischen Koordinaten* eines Punktes möglich.

Die folgende Definition und das dazugehörige Lemma der *baryzentrischen Koordinaten* oder auch *Dreieckskoordinaten*, stammt von Koecher und Krieg ([[KK07]]):

„Definition. Es sei K ein beliebiger Körper und a, b, c ein Dreieck in K^2.

Lemma. *Zu jedem $x \in K^2$ gibt es eindeutig bestimmte $\alpha, \beta, \gamma \in K$ mit*

(1)

$$x = \alpha a + \beta b + \gamma c$$

und

(2)

$$\alpha + \beta + \gamma = 1$$

Genauer gilt hier

(3)

$$\alpha = \frac{[x,b,c]}{[a,b,c]}, \beta = \frac{[a,x,c]}{[a,b,c]}, \gamma = \frac{[a,b,x]}{[a,b,c]}$$

[...]. Das Tripel (α, β, γ) in (1) mit der Nebenbedingung (2) nennt man die (auf das Dreieck a,b,c bezogenen) *Dreieckskoordinaten* oder *baryzentrischen Koordinaten* des Punktes $x \in K^2$." ([KK07] S.75)

Zu (3) muss noch die Definition der eckigen Klammern erörtert werden. Diese wurde wiederum aus [KK07] abgeleitet.

Es sei weiterhin K ein beliebiger Körper und es seien die Punkte $x, y, z \in K^2$ gegeben.

wobei:

$$x := \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, y := \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}, z := \begin{pmatrix} z_1 \\ z_2 \end{pmatrix},$$

Es sei

$$[x,y] := det(x,y) = x_1 y_2 - x_2 y_1$$

Weiterhin sei

$$[x,y,z] := [x,y] + [y,z] + [z,x]$$

Die oben definierten *baryzentrischen Koordinaten* bilden die Basis für die *Dreiecksinterpolation*. Die Interpolation findet zum Beispiel Nutzen in folgendem Szenario:

K sei ein beliebiger Körper. Gegeben sei ein Dreieck, aufgespannt durch die Punkte $a, b, c \in K^2$. Die einzelnen Punkte haben jeweils einen Wert $v_a, v_b, v_c \in K$. Weiterhin sei ein Punkt $x \in K^2$ gegeben, der sich innerhalb des Dreiecks befindet.

Gesucht wird nun ein sinnvoller Wert v_x in Bezug auf v_a, v_b, v_c in Betrachtung der Distanz zwischen x und den Eckpunkten des Dreiecks a, b, c.

Um dieses Problem zu lösen, werden die *baryzentrischen Koordinaten* α für a, β für b und γ für c, wie in (3) definiert, berechnet.

Nun ist das Ergebnis des Werts v_x von x wie folgt zu erhalten:

$$v_x = \alpha v_a + \beta v_b + \gamma v_c$$

2.5 OpenStreetMap

OpenStreetMap (OSM) ist ein im Jahr 2004 gegründetes Projekt, das es sich zur Aufgabe gemacht hat, eine freie Weltkarte zu kreieren. Es werden weltweit Geoinformationen gesammelt, die kostenfrei für jeden erhältlich sind. Die Daten werden durch die eigene Community erfasst und verwaltet. Dabei ist es jedem, der dies möchte, möglich zu partizipieren. [Ope22a]

2.5.1 Relevanz von OpenStreetMap

Geoinformationen sind in der Regel nicht frei erhältlich. Es gibt Anbieter wie Google, die eine Nutzung ihrer Karten kostenlos zur Verfügung stellen. Die kostenlose Nutzung wird allerdings an Bedingungen geknüpft, zum Beispiel an einen eingeschränkter Zugriff, der nur über die Webseite beziehungsweise die *Application Programming Interface (API)* des Anbieters erfolgen kann. Hinzu kommen Bedenken im Bezug auf den Datenschutz. Ein weiterer Nachteil bei der Nutzung der Dienste eines solchen Anbieters ist der Mangel an Rohdaten von Geoinformationen. Die Nutzung von Google-Diensten in diesem Kontext verhindert somit eine freie manuelle Konfiguration. Es ist einem Nutzer dieser Dienste zum Beispiel weder möglich, eine eigene Landkartendarstellung auszuwählen noch einen eigenen Routing-Algorithmus zu programmieren.

Um den oben genannten Mangel der Rohdaten und die damit einhergehenden Probleme zu lösen, ist die Nutzung von *OSM* empfehlenswert. Über die *OSM*-Datensätze ist es kostenlos möglich Rohdaten von Geoinformationen herunterzuladen und zu nutzen. [Ope22a]

2.6 Shuttle Radar Topography Mission

Die Shuttle Radar Topography Mission (SRTM) begann am 11.02.2000. Zehn Tage lang wurden dabei Daten der Erdoberfläche durch radio detection and ranging (Radar) gemessen. [JPL22a]

Dabei wurde sich auf den Bereich zwischen dem nördlichen Breitengrad 60 und dem südlichen Breitengrad 54 eingeschränkt. Das entspricht ungefähr 80% der Erdoberfläche. [JPL22b]

Die Daten wurden mittels Interferometrie errechnet. Bei der Technik der Interferometrie werden, oberflächlich betrachtet, zwei Bilder der gleichen Region von zwei verschiedenen Punkten aus aufgenommen. Durch die leichten Unterschiede der Bilder werden die Höhen der Oberfläche bestimmt. [JPL22c]

Bei der SRTM wurden zwei Antennen genutzt. Eine der Antennen war direkt am Spaceshuttle angebracht. Die zweite Antenne war in 60 Meter Entfernung an einem mit dem Spaceshuttle verbundenen Masten befestigt. Jede Antenne nutzte sein eigenes Radarsignal, wodurch zwei verschiedene Datensätzte erzeugt wurden. Diese Datensätze wurden schließlich für die Interferometrie genutzt. [JPL22c]

Die *SRTM*-Daten sind mittlerweile frei zugänglich und mit der Auflösung von drei Bogensekunden (ungefähr 90 Meter zwischen den Messungen) oder einer Bogensekunde (ungefähr 30 Meter zwischen den Messungen) erhältlich. Es ist zu beachten, dass die ursprünglichen *SRTM*-Daten leere Messproben enthalten. Die *NASA Version 3 SRTM Global* Daten, die seit August 2015 öffentlich

verfügbar sind, enthalten keine leeren Messproben mehr. Diese sogenannten *voids* wurden mit Erhebungsdaten aus den Datensätzen von *ASTER GDEM2* und *USGS GMTED 2010* gefüllt oder mit vorhandenen Erhebungsdaten interpoliert. [LPD22a]

2.7 Android

Der folgende Abschnitt gibt eine kurze Einführung in essenzielle Komponenten der Android-Entwicklung, die in der *Android-App* genutzt werden.

Activity

Eine *Activity* ist eine Komponente einer *Android-App*, die dem Nutzer durch die Erstellung eines *Views*, eine Bildschirm-Seite (im Folgenden *Screen* genannt) repräsentiert. Dabei werden die einzelnen GUI-Elemente, wie zum Beispiel Buttons, durchgehend von der *Activity* verwaltet. [And22c]

Activities haben dabei ihre eigenen Lebenszyklen mit verschiedenen *callbacks*, die es einem Programmierer ermöglichen, spezifische Funktionen zu tätigen. Dies ist nötig, falls zum Beispiel der *callback onCreate()* aufgerufen wird, in dem die *Activity* erstmals erstellt wird. [And22b]

Fragment

Ein *Fragment* ist ein wiederverwendbarer Baustein mit seinem eigenen Layout, das durch das *Fragment* selbst erstellt und verwaltet wird. Dabei erhält auch das *Fragment* die graphische Darstellung über die Erstellung eines *View*-Objekts. Zudem besitzt ein *Fragment* seinen eigenen Lebenszyklus, der wie bei einer *Activity* seine eigenen *callbacks* besitzt. Die besondere Beziehung zwischen *Fragment* und *Activity* besteht darin, dass ein *Fragment* nur mit einer *Activity* als Host genutzt werden kann. Es können sogar mehrere *Fragments* in einer *Activity* gehostet und somit angezeigt werden. [And22a]

2.7.1 Osmdroid

Osmdroid ist ein open-source Framework für Android zur Verarbeitung von *OSM* Geoinformationen. [osm22g]

Im Folgenden werden ausgesuchte Komponenten dieses Frameworks vorgestellt.

MapView

Die Kernkomponente *osmdroids* ist die *MapView*-Klasse. Diese benötigt einen *TileProvider* und eine *TileSource*, um schließlich eine Landkarte darzustellen. Mit dieser Landkarte kann ein Nutzer über verschiedene Gestiken interagieren und sich so zum Beispiel auf der Karte bewegen. Zur Manipulation der Landkarte können einem *MapView*-Objekt *Overlays* hinzugefügt werden. [osm22a]

TileProvider

Der *TileProvider* definiert wie die Kacheln (im folgenden Tiles genannt) der Landkarte geladen werden. Tiles können im Allgemeinen aus Assets, offline Archiven (ZIP, SQLite), dem *Tilecache* oder von einem *Tileserver* geladen werden. [osm22b]

TileSource Die *TileSource* definiert die Art der Tiles, die dargestellt werden. [osm22b]

Overlays

Es gibt mehrere Objekte, die als *Overlay* genutzt werden können. Die folgende Auflistung beschränkt sich auf drei Objekte. [osm22c]

Marker Ein Objekt der *Marker*-Klasse ist als Markierung zu verstehen, die einen definierten Icon auf der Landkarte abbilden kann. [osm22d]

Polyline Ein *Polyline*-Objekt ist als Linie zu verstehen, die verschiedene Punkte miteinander verbindet. Mit diesem Objekt lassen sich somit aus Linien bestehende Zeichnungen auf der Landkarte darstellen. [osm22e]

MapEventsOverlay Ein *MapEventsOverlay*-Objekt wird durch ein *MapEventsReceiver*-Objekt instanziiert und kann dem *MapView*-Objekt als *Overlay* hinzugefügt werden. Dabei deklariert das *MapEventsReceiver*-Objekt Funktionen, die bei verschiedenen Interaktionen mit der Landkarte ausgeführt werden sollen. So erkennt das *MapEventsOverlay*-Objekt zum Beispiel einen *longPress*, also ein längeres Drücken auf die Landkarte. [osm22f]

3 Ansatz

Das Projekt lässt sich in verschiedene Komponenten aufteilen, die in Abbildung 3.1 dargestellt sind und miteinander interagieren. Dabei sollen die Komponenten *Server* und *Android-App* die in Abschnitt 1.1 definierten Aufgaben lösen. Hinzu kommt ein *Tileserver*, der bei einer Anfrage der *Android-App* spezifische Tiles für die Visualisierung einer Landkarte zurückgeben kann. Da das FMI bereits einen *Tileserver* zur Verfügung stellt, ist die Implementierung dieser Komponente nicht notwendig.

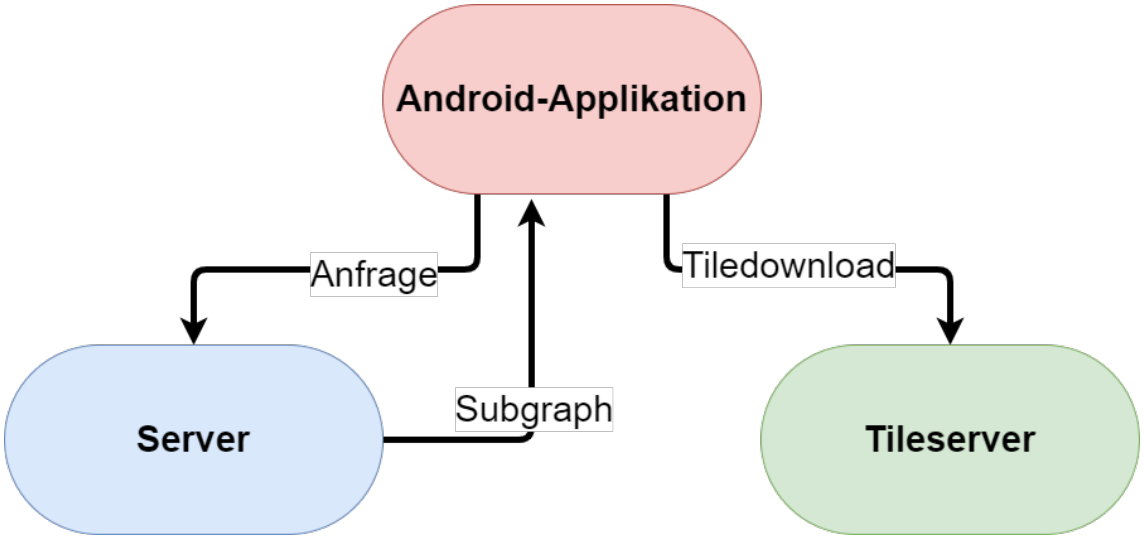


Abbildung 3.1: Komponenten

Die *Android-App* hängt vom *Server* ab, da sie einen Subgraphen des *Servers* für das geforderte Offline-Routing nach dem Prinzip des *personalized route planning* (siehe Abschnitt 2.2) benötigt. Angefangen wurde folglich mit der Implementierung des *Servers*. Für diesen sind die Datensätze des deutschlandweiten Graphen essenziell. Diese Datensätze zu sammeln war somit der erste Schritt.

3.1 Daten

Die für das Projekt benötigten Datensätze beschränken sich auf Daten aus den *OSM*-Datensätzen und den Höhendaten in Deutschland.

3.1.1 Vorhandene Graphdaten

Die notwendigen Informationen für die Erstellung eines Graphen sind vorgegeben. Ursprünglich stammen diese Daten aus den *OSM*-Datensätzen. Als Werkzeug zur Extrahierung wird dabei der *OsmGraphCreator* (siehe [Alg22a]) genutzt. Die Daten beinhalten jegliche Knoten und Kanten in Deutschland, die entweder zu Fuß, mit dem Fahrrad oder mit dem Auto zu erreichen sind. Diese Graphdaten sind in einer Textdatei mit der .fmi-Endung in strukturierter Form gespeichert.

3.1.2 Format der Graphdaten

Das .fmi-Format besteht aus drei verschiedenen Formaten. Spezifisch gibt es ein Format für die Metadaten, eines für die Kanten und eines für die Knoten.

Metadaten

Format:

```
ID: <unsigned integer>
Timestamp: <UNIX Zeitstempel>
Type: <type of the graph>
Revision: <unsigned integer>
Anzahl der Knoten: <unsigned integer>
Anzahl der Kanten: <unsigned integer>
```

Im Datensatz der Metadaten ist für das Projekt ausschließlich die Anzahl der Knoten und Kanten relevant.

Knoten

Jeder Knoten wird durch eine Zeile beschrieben. Das Format dieser Zeile sieht wie folgt aus:

```
Knoten_ID    OSM_ID      Breitengrad   Laengengrad
<uint32_t>   <int64_t>   <double>      <double>
```

Im Datensatz der Knoten sind die Knoten-ID sowie der Breiten- und Längengrad relevant.

Die Knoten-ID ist nötig, um den Start- und Zielknoten der Kanten zu bestimmen.

Die Kombination der Breiten- und Längengrade liefert die Koordinaten der Knoten und ist somit die Schnittstelle zwischen Graph und Karte.

Kanten

Jede Kante wird durch eine Zeile beschrieben. Das Format dieser Zeile sieht wie folgt aus:

```
Start_Knoten_ID   Ziel_Knoten_ID   Zeit in cs   Kantentyp   max_Geschwindigkeit in km/h
<uint32_t>        <uint32_t>       <int32_t>    <int32_t>   <int32_t>
```

Im Datensatz der Kanten werden alle Attribute außer der maximalen Geschwindigkeit benötigt.

Über die Startknoten- und Zielknoten-ID kann man einen gerichteten Graphen erstellen.

Die Zeit in Zentisekunden dient als Metrik bei der Routenberechnung.

Das Attribut des Kantentyps gibt die Art der Straße wieder, auf der die Kante liegt. Für eine Differenzierung zwischen den verschiedenen Transportmitteln, ist der Kantentyp nötig. Die Relation zwischen Kantentyp und Transportmittel wird aus der Dokumentation des *OsmGraphCreator* (siehe [Alg22b]) abgeleitet und in Tabelle 3.1 dargestellt:

Transportmittel	**Kantentyp-ID**
Auto	1, 2
Zu Fuß	18, 19, 21, 22
Zu Fuß und Fahrrad	17, 20
Alle	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Tabelle 3.1: Transportmittel

3.1.3 Höhendaten

Um die Höhe als Metrik für die Routenplanung zu nutzen, wird die Höhe bei jedem Knoten des Deutschlandgraphen benötigt. Hierfür werden die Höhendaten mit der Auflösung von 1″ (Bogensekunde) der *SRTM* (siehe Abschnitt 2.6) im HGT-Format genutzt.

Height-Format

Jede Height (HGT)-Datei beinhaltet die Höhendaten eines Tiles, wobei das Tile einen Breitengrad hoch und einen Längengrad breit ist. Die Position der Tiles wird im Namen der Datei festgehalten, indem dieser den Breiten- und Längengrad der unteren linken Ecke des Tiles beinhaltet. Die 1″ (Bogensekunde) HGT-Datei selbst besteht aus 3601 Zeilen und 3601 Spalten, also insgesamt aus 12 967 201 Werten. Bei den Werten handelt es sich um 2-Byte signed Integer Werte, also variieren die Werte maximal zwischen -32 767 und 32 767. Sie wurden mit einem Abstand von einer Bogensekunde aufgenommen. Somit entspricht die Distanz zwischen benachbarten Werten ungefähr 30 Meter. Da 1° = 3600″, ist klarzustellen, dass die Werte, die direkt auf den Graden liegen, in mehreren HGT-Dateien zu finden sind. Es ist außerdem zu beachten, dass die erste Zeile der Datei die nördlichsten Werte des Tiles darstellt. Die letzte Zeile beinhaltet folglich die

südlichsten Werte, also den im Namen der Datei definierten Breitengrad. Äquivalent dazu verhält es sich mit den westlichsten Werten, die sich am Anfang jeder Zeile befinden und im Namen der Datei definiert sind. Folglich befinden sich die östlichsten Werte am Ende jeder Zeile. [LPD22b]

Ein Beispiel: Die Datei *N47E005.hgt* enthält die Höhendaten, die sich in dem aufgespannten Bereich zwischen dem Breitengrad N47 und N48 und zwischen dem Längengrad E5 und E6 befinden.

3.2 Server

Mit den erhaltenen Daten (siehe Abschnitt 3.1) kann nun der *Server* implementiert werden. Die Implementierung des *Servers* wurde mittels Java 17 und Maven 3.8.5 vollzogen.

3.2.1 FMI-Daten

Das Straßennetzwerk von Deutschland für die drei Fortbewegungsarten (zu Fuß, mit dem Fahrrad, mit dem Auto) enthält 70 914 829 Knoten und 150 103 104 Kanten. Somit ist die dazugehörige .fmi-Datei 8,5 GB groß. Um diese Menge an Daten optimiert im Arbeitsspeicher zu halten, kommt eine objektorientierte Implementierung über Knoten- und Kantenobjekte nicht in Frage. Die Knoten und Kanten werden stattdessen in Arrays abgespeichert. Als Datenstruktur für den Graphen bietet sich eine Darstellung als Array in Form eines *Adjazenzarray* (siehe Abschnitt 2.3) an. Diese Datenstruktur hat den Vorteil, dass die ausgehenden Kanten von einem Knoten schnell zurückgegeben werden können. Es sind folgende Arrays nötig, um einen Graphen mit den für die Routenberechnung essenziellen Daten bereitzuhalten:

```
breitengradDerKnoten: doubleArray[Anzahl der Knoten]
längengradDerKnoten: doubleArray[Anzahl der Knoten]
transportmittelIdDerKnoten: byteArray[Anzahl der Knoten]
hilfsarray: intArray[Anzahl der Knoten + 1]
zielknotenDerKante: intArray[Anzahl der Kanten]
zeitDerKante: intArray[Anzahl der Kanten]
distanzDerKante: shortArray[Anzahl der Kanten]
transportmittelIdDerKanten: byteArray[Anzahl der Kanten]
```

Lediglich bei den Arrays für die Transportmittel-IDs der Knoten und Kanten sowie bei dem Array für die Distanz einer Kante ist die weitere Vorgangsbetrachtung interessant. Die Daten der restlichen Arrays werden aus der .fmi-Datei eingelesen (siehe Abschnitt 3.1.2).

Die Transportmittel-ID der Kanten wird durch die Relation zu den Kantentypen bestimmt. Sie wird dabei wie folgt definiert:

```
Auto = 1
Zu Fuß = 3
Zu Fuß und Fahrrad = 23
Alle = 123
```

Die Transportmittel-ID der Knoten wird benötigt, um den nächsten für das Transportmittel erreichbaren Routenstart- und Routenzielknoten zu finden, nachdem ein Nutzer die Punkte auf der Karte durch eine Berührung ausgewählt hat. Um die Transportmittel-ID zu bestimmen, wird sie sukzessive während der Erstellung des *Adjazenzarrays* aktualisiert.

Mit der Annahme, dass die *Zeit in cs* vom *OsmGraphCreator* mit der *max_Geschwindigkeit in km/h* aus Abschnitt 3.1.2 und der Distanz der Kante berechnet wurde, kann nun die Distanz der Kante ausgerechnet werden. Um die Distanz in Metern auszurechnen, kann dann für jede Kante wie folgt vorgegangen werden:

$$distanz = \frac{maxgeschwindigkeit}{360} \cdot zeit$$

3.2.2 Höhendaten

Um die Höhendaten der *SRTM* (siehe Abschnitt 2.6) nutzen zu können, ist die Registrierung auf einer Plattform der NASA (siehe [NAS22]) erforderlich. Nach der Registrierung lassen sich gewünschte Daten der *SRTM* über ein Interface herunterladen. Diese Daten sind im HGT-Format gegeben.

Um die Höhendaten aus den HGT-Dateien zu extrahieren, werden für jede HGT-Datei die Koordinaten der einzelnen Knoten aus der .fmi-Datei mit den Koordinaten der jeweiligen HGT-Datei verglichen. Somit werden alle Knoten selektiert, die sich innerhalb der HGT-Datei befinden. Für jeden einzelnen dieser Knoten wird nun die dazugehörige Höhe bestimmt. So werden für diese Arbeit sukzessive alle Höhen aller Knoten bestimmt. Dafür sind 70 HGT-Dateien nötig.

Die Bestimmung der Höhen erfolgt, indem zuerst die Dezimalstellen der Knotenkoordinaten berechnet werden. Diese Dezimalstellen werden dann mit der Anzahl der Proben aus der HGT-Datei multipliziert, um die Position des Knotens in der Datei zu bestimmen. Nun werden die drei Proben, die dieser Position am nächsten sind, bestimmt und als Ecken eines Dreiecks betrachtet. Innerhalb dieses Dreiecks positioniert sich folglich auch der Knoten. Als nächstes werden die *baryzentrischen Koordinaten* der Position des Knotens berechnet. Die erhaltenen Massen m_1, m_2, m_3 werden mit den Höhenwerten der jeweils dazugehörigen Probe multipliziert und ergeben somit einen interpolierten Höhenwert für unseren Knoten. Es wird also die im Abschnitt 3.3.3 beschriebene Prozedur der *Dreiecksinterpolation* auf dem oben genannten Dreieck ausgeführt.

3.2.3 HTTP Server

Der *Server* beinhaltet eine simple Klasse. Diese Klasse initialisiert einen *Hypertext Transfer Protocol (HTTP)*-Server. Um den Subgraphen einer bestimmten Region zu erhalten, ist es möglich eine Anfrage an den *HTTP*-Server zu stellen. Diese Anfrage muss die Koordinaten der Eckpunkte der gewünschten Region als Parameter beinhalten. Falls eine valide Anfrage gestellt wird, initialisiert die Klasse die Berechnung des Subgraphen und gibt schließlich das Ergebnis zurück.

3.2.4 Subgraph

Die Herausforderung bei der Berechnung des Subgraphen ist die Optimierung der weiteren Verarbeitung in der *Android-App*. Somit war zu beachten, dass alle möglichen Verarbeitungsschritte der Daten für die spätere Routenberechnung auf dem *Server* berechnet werden. Dies resultiert in den folgenden beiden Aufgaben:

- Neue Indizes für Knoten und Kanten, da für die Erstellung eines *Adjazenzarray* Indizes von $0, 1, \cdots, n-1$ sinnvoll sind. Dabei ist zu beachten, dass n der Anzahl der Knoten im Subgraphen entspricht.
- Anpassung des .fmi-Formats, um irrelevante Datensätze für die weitere Verarbeitung zu verhindern.

Die neuen Indizes werden während der Bestimmung des Subgraphen in einem Array gespeichert, sodass die Neuindizierung die alten Indizes, die für eine neue Anfrage nötig sind, nicht beeinflussen.

Das angepasste .fmi-Format ist weiterhin in Metadaten, Knoten und Kanten aufgeteilt, wobei lediglich die für diese Arbeit relevanten Daten genutzt wurden.

Metadaten

Die ersten beiden Zeilen beinhalten die Metadaten.

```
Anzahl der Knoten: <unsigned integer>
Anzahl der Kanten: <unsigned integer>
```

Knoten

Nach den Metadaten folgen die Daten für die Knoten, die weiterhin jeweils aus einer Zeile bestehen.

```
Knoten_ID     Breitengrad     Längengrad     Höhe     Transportmittel_ID
  <int>        <double>        <double>     <int>         <byte>
```

Kanten

Zuletzt werden die Daten für die Kanten abgebildet, die auch weiterhin jeweils aus einer Zeile bestehen.

```
Start_Knoten_ID  Ziel_Knoten_ID  Zeit_cs  Transportmitteltyp_ID  Länge   Höhendifferenz
    <int>            <int>        <int>         <byte>          <int>       <byte>
```

Vorgehen Subgraphen

Die Berechnung eines Subgraphen, wird nur vom *HTTP*-Server (siehe Abschnitt 3.2.3) initiiert. Somit wird die Region, die der Subgraph abdecken muss, mitgeliefert.

Als erstes werden alle Knoten des Deutschlandgraphen, die sich in der gewünschten Region befinden, zwischengespeichert und neu indiziert. Nun wird das *Adjazenzarray* genutzt, um die Anzahl der Kanten im Subgraphen zu zählen. Danach wird die Summe der Knoten und Kanten berechnet, um zu ermitteln, ob der angefragte Subgraph den Arbeitsspeicher des Smartphones überbelasten würde. Falls die Summe eine statisch festgelegte Schranke überschreitet, wird die Anfrage abgebrochen.

Ansonsten wird wie folgt vorgegangen: Die Anzahl der Knoten, die Anzahl der Kanten und die Knoten des Subgraphen werden über den *HTTP*-Server zurückgegeben. Währenddessen werden die neuen Indizes zwischengespeichert und wiederum das *Adjazenzarray* genutzt, um die Kanten des Subgraphen zu bestimmen und dabei die Indizes der Start- und Zielknoten jeder Kante anzupassen. Schließlich werden die Kanten über den *HTTP*-Server zurückgegeben.

3.2.5 Allgemeines Vorgehen

Nachdem die wichtigsten Komponenten des *Servers* angeschnitten wurden, wird nun der Ablauf ab Start des *Servers* beschrieben.

Vorverarbeitung Sobald der *Server* gestartet wird, beginnt die Vorverarbeitung der Daten. Anfangs werden alle nötigen Arrays zur Speicherung des Deutschlandgraphen initialisiert. Als nächstes werden die Längen- und Breitengrade der Knoten im dafür vorgesehenen Array gespeichert. Diese werden dann genutzt, um die Höhendaten (siehe Abschnitt 3.2.2) aller Knoten auszulesen und in einem Array zu speichern. Nun wird über die Anzahl der Kanten des Deutschlandgraphen iteriert und währenddessen der *Adjazenzarray* in Form von mehreren Arrays (siehe Abschnitt 3.2.1), sowie der Hilfsarray erstellt. In jeder Iteration wird die Distanz und die Höhendifferenz der Kante berechnet und abgespeichert. Des Weiteren wird währenddessen die Transportmittel-ID zugeordnet, wie in Abschnitt 3.2.1 beschrieben.

In Betrieb Nach der Vorverarbeitung startet der *Server* den *HTTP*-Server. Nun können Anfragen gestellt werden. In diesem Fall wird, wie in Abschnitt 3.2.4 beschrieben, vorgegangen.

Es kann immer nur eine Anfrage verarbeitet werden. Falls also ein großer Subgraph angefragt wird, wird der *Server* neue Anfragen erst im Anschluss bearbeiten.

3.3 Android-Applikation

Für den folgenden Abschnitt wird das Vorwissen aus Abschnitt 2.7 benötigt. Nachdem der Server implementiert wurde, ist der nächste Schritt die Implementierung der *Android-App*. Nach anderen Ansätzen wurde sich für eine *Single-Activity* Architektur entschieden. Die *Android-App* besteht also aus einer *Activity* und mehreren *Fragments*. Hinzu kommt noch ein *ViewModel* für die Logik und Variablen, auf die mehrere *UI-Komponenten* zugreifen müssen. Des Weiteren gibt es mehrere Komponenten für die Verwaltung und Verarbeitung des Graphen.

Die folgende Abbildung illustriert eine Zusammenfassung der Architektur:

3.3.1 Backend

Im Folgenden werden die in Abbildung 3.2 lila eingefärbten Klassen des *Backends* der *Android-App* genauer betrachtet.

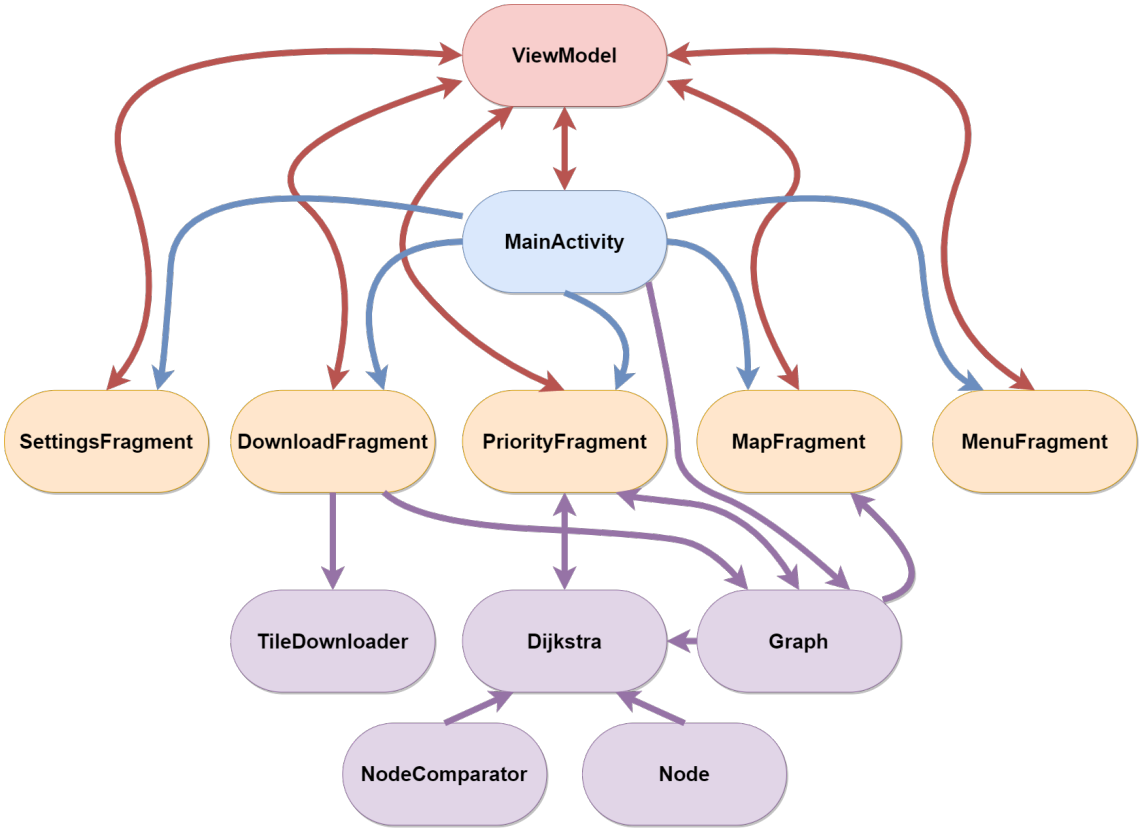


Abbildung 3.2: Architektur

TileDownloader

Die *TileDownloader*-Klasse ist für den Download der Tiles eines Subgraphen vom *Tileserver* des FMI zuständig. Innerhalb der Klasse sind drei Funktionen definiert.

getXYTiles Die Funktion *getXYTiles* benötigt die Koordinaten der vier Himmelsrichtungen des Subgraphen. Mit diesen Parametern werden die Anzahl und die Uniform Resource Identifiers (URIs) aller nötigen Tiles berechnet. Dabei werden die URIs in einem Array abgespeichert. Die Bestimmung der URIs ist dabei vom *Tileserver* abhängig. Da sich dieser auf die Konventionen für eine *Slippy Map* bezieht, haben die URIs der Tiles das folgende Format:

$$Tileserver\_URL/Zoomlevel/x/y.png.$$

Die x- und y-Werte der URI eines Tiles werden in *getXYTiles*, durch die *Web Mercator projection* berechnet. [Ope22b]

manualDownloader Die *manualDownloader*-Funktion lädt nun die benötigten Tiles des Subgraphen über Coroutinen herunter. Dabei ist klarzustellen, dass die Tiles eines angefragten Bereichs nicht für alle Zoomlevel heruntergeladen werden. Es werden nur die Zoomlevel beachtet, die sich zwischen dem Zoomlevel der Landkarte während der Anfrage ($zoomLvl$) und einem maximalen Wert

$maxZoom$ befinden. Falls $zoomLvl \geq 16$, dann folgt $maxZoom = 18$, sonst gilt $maxZoom = 16$. Beim Download wird außerdem auf die Ordnerstruktur der Daten Wert gelegt wird. Die Struktur basiert auf dem in *getXYTiles* eingeführten Format der URIs. Die Ordnerstruktur des *Tileservers* entspricht somit der Struktur auf dem Smartphone und wird im Folgenden beschrieben:

Für jedes beachtete Zoomlevel wird ein Ordner mit dem Wert als Name angelegt. In den jeweiligen Ordnern wird nun für alle unterschiedlichen x Koordinaten der Tiles, die dem Zoomlevel zuzuordnen sind, ein Unterordner mit dem Wert der x Koordinate angelegt. Anschließend werden alle Tiles, in Betrachtung ihres Zoomlevels und ihrer x Koordinate, in den passenden Unterordner gespeichert. Dabei ist zu beachten, dass die Benennung der Tiles dem Wert der y Koordinate entspricht. Somit ist zum Beispiel das Tile, das dem Zoomlevel 17 zugeordnet wird, das die x-Koordinate 68 858 und die y-Koordinate 45 138 besitzt, im Unterordner 68 858 des Ordners 17 in Form der Datei 45138.png zu finden.

Die Nutzung von *osmdroid* fordert außerdem, dass die oben beschriebene Ordnerstruktur sich in einem nach der *Tilesource* benannten Ordner befindet.

Nachdem alle Tiles heruntergeladen wurden, wird nun die *zip*-Funktion aufgerufen werden.

zip Die Funktion *zip* speichert die heruntergeladenen Tiles nun in ein ZIP-Archiv. Dabei wird die in *manualDownloader* angelegte Ordnerstruktur beachtet. Aufgrund der Struktur kann *osmdroid* das resultierende Archiv als *TileProvider* nutzen.

Es ist anzumerken, dass die heruntergeladenen Tiles und das dazugehörige Archiv bei der Anfrage nach einem neuen Subgraphen überschrieben werden. Es ist somit immer nur möglich einen Subgraphen offline zu nutzen.

Graph

Die *Graph*-Klasse spiegelt den Subgraphen wider, der für die Routenplanung benötigt wird. Die Klasse besitzt drei Funktionen.

graphDataToRam Die Funktion *graphDataToRAM* lädt den Subgraphen aus der heruntergeladenen .fmi-Datei in den Arbeitsspeicher. Dabei wurde die gleiche Datenstruktur, wie beim *Server* gewählt. Diese besteht somit aus einzelnen Arrays, die Geoinformationen der Knoten und Kanten des Subgraphen beinhalten. Des Weiteren wird ein *Adjazenzarray* (siehe Abschnitt 2.3) zusammen mit seinem *Hilfsarray* benötigt. Eine Besonderheit der *graphDataToRAM*-Funktion ist die Anpassung der Metriken. Die Werte der Zeit sind in Zentisekunden (siehe Abschnitt 3.1.2) angegeben. Diese sind im Vergleich zu den Werten für die Distanz (siehe Abschnitt 3.2.1) und der positiven Höhendifferenz (siehe Abschnitt 3.2.2), welche beide in Meter angegeben sind, viel zu hoch. Da die Personalisierung der Route visuell über ein Dreieck (siehe Abschnitt 3.3.3) stattfindet, ist das ein Problem. Falls zum Beispiel eine ausgeglichene Priorisierung gewählt wird, hätte die Zeitmetrik ohne eine Angleichung deutlich mehr Einfluss auf die Route als die anderen beiden Metriken. Deswegen wird für die Distanz und die positive Höhendifferenz jeweils ein *Gewichtungsfaktor* ausgerechnet, der die Angleichung bewirkt. Dieser wird berechnet, indem zuerst der Mittelwert

aller Werte jeder Metrik berechnet wird. Im Anschluss wird die Differenz zwischen dem jeweiligen Mittelwert und dem Mittelwert der Zeit berechnet. Der *Gewichtungsfaktor* der jeweiligen Metrik ist dann das Ergebnis der Differenz multipliziert mit dem jeweiligen Mittelwert.

getGraph Die Funktion *getGraph* benötigt die Koordinaten der vier Himmelsrichtungen und die Uniform Resource Locator (URL) des *Tileservers*. Mit diesen Parametern wird eine Anfrage an den *Server* (siehe Abschnitt 3.2) gestellt. Das Resultat, in Form der Daten eines Subgraphs, wird schließlich in einer .fmi-Datei abgespeichert.

Es ist auch hier anzumerken, dass der somit heruntergeladene Subgraph durch die nächste Anfrage überschrieben wird.

findNearest Die Funktion *findNearest* benötigt den Längen- und Breitengrad eines Punktes. Berechnet und zurückgegeben wird dann der Knoten aus dem Subgraphen, der sich am nächsten zu diesem Punkt befindet und mindestens eine Kante besitzt, die mit dem ausgewählten Transportmittel befahren werden kann.

Dijkstra

Die *Dijkstra*-Klasse implementiert lediglich die Funktion *dijkstra* für den in Abschnitt 2.1 beschriebenen *Dijkstra-Algorithmus*, der auf dem Subgraphen ausgeführt wird. Die Umsetzung des Algorithmus wurde durch die beiden Hilfsklassen *Node* und *NodeComparator* optimiert. Eine Besonderheit, die sich durch das *personalized route planning* (siehe Abschnitt 2.2) ergibt, ist die Berechnung der Gewichtung δ einer Kante. Um δ für eine Kante zu berechnen wird der Wert jeder Metrik mit dem Priorisierungsfaktor der jeweiligen Metrik und, im Falle der Distanz und der positiven Höhendifferenz, mit dem *Gewichtungsfaktor* der Metrik (siehe Abschnitt 3.3.1) multipliziert. Die Gewichtung δ der Kante entspricht nun der Summe dieser drei Produkte. Die zweite Besonderheit ist die Nichtberücksichtigung der Knoten und Kanten, welche nicht durch das ausgewählte Transportmittel erreicht werden können.

3.3.2 ViewModel

Im *ViewModel* sind mehrere Variablen und Funktionen definiert, die von den *UI-Komponenten* geändert und abgerufen werden können. Somit ist das *ViewModel* mit allen *UI-Komponenten* verbunden und bildet eine Art *Controller*. Jegliche Interaktion zwischen zwei *UI-Komponenten* wird nur über das *ViewModel* vollzogen. Die Interaktion funktioniert dabei über das Beobachtermuster. Die verschiedenen *UI-Komponenten* beobachten also den Wert einer Variablen des *ViewModels* und reagieren dann auf spezifische Werte der Variablen. Des Weiteren gibt es auch Variablen im *ViewModel*, die in einer *UI-Komponente* aktualisiert, und in einer anderen *UI-Komponente* zur Initiierung einer Methode im *Backend* genutzt werden.

3.3.3 UI-Komponenten

MainActivity

Die *MainActivity* ist die einzige *Acitivtiy* der *Android-App* und bleibt durchgehend aktiv. Diese *Activity* ist die grundlegende *UI-Komponente* der *Android-App*, da sie durchgehend der Host für ausgewählte *Fragments* ist. Im Folgenden werden die weiteren Kernaufgaben der *Activity* beschrieben.

Initiierung des Subgraphen: Beim Start der *Android-App* wird hier die in Abschnitt 3.3.1 beschriebene Funktion *graphDataToRAM* aufgerufen.

Statusvariablen: Die *MainActivity* lädt beim Start die Statusvariablen der *Android-App*. Dabei handelt es sich um Variablen, die folgende Werte speichern:

```
offlineNorth: Speichert die nördliche Koordinate der zuletzt heruntergeladenen Landkarte
offlineSouth: Speichert die südliche Koordinate der zuletzt heruntergeladenen Landkarte
offlineWest: Speichert die westliche Koordinate der zuletzt heruntergeladenen Landkarte
offlineEast: Speichert die östliche Koordinate der zuletzt heruntergeladenen Landkarte
offlineLat: Speichert den Breitengrad des Zentrums der zuletzt heruntergeladenen Landkarte
offlineLon: Speichert den Längengrad des Zentrums der zuletzt heruntergeladenen Landkarte
offlineZoom: Zoomlevel der zuletzt heruntergeladenen Landkarte
dataUse: War der Nutzer zuletzt offline oder online?
url: url des Servers für die Graphdaten
```

Durch die ersten vier gelisteten Statusvariablen wird die heruntergeladene Region gesichert. Die folgenden drei gelisteten Statusvariablen sichern wiederum den Standort und die korrekte Darstellung der heruntergeladenen Landkarte. Ohne eine Sicherung dieser zuletzt genannten drei Variablen wäre das Auffinden der heruntergeladenen Landkarte mit Aufwand des Nutzers verbunden. Durch die Speicherung der *dataUse*- und *url*-Variable wird die *User Experience* verbessert.

Berechtigungen: Die *MainActivity* ist dafür verantwortlich eine Anfrage nach den nötigen Berechtigungen der *Android-App* zu stellen. Diese Anfrage erfolgt bei einer Zusage einmalig nach dem ersten Start der *Android-App*.

Navigation: In der *MainActivity* wurde die Logik der UI-Navigation implementiert. Es werden drei verschiedene Container definiert, die bestimmte Abschnitte des Displays einnehmen, die für die Darstellung der UI benutzt werden. Die sich ändernden *Screens* werden über einen *Observer* erreicht, der die Variable *uiId* im *ViewModel* beobachtet. Falls sich der Wert von *uiId* ändert, wird der zu diesem Wert gehörende *Screen* durch die Erstellung eines *Views* angezeigt. Spezifischer werden verschiedene *Views* über die *Fragments* erstellt und dann über einzelne *transactions* in die gewünschten Container geladen. Es folgt somit ein *View*, der dem gewünschten *Screen* entspricht.

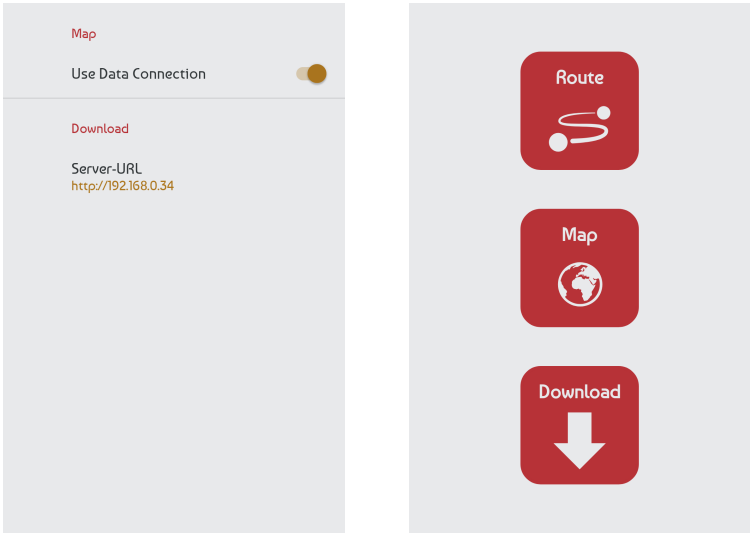
SettingsFragment

Das *SettingsFragment* liest und schreibt Werte in zwei Variablen des *ViewModels* (siehe Abschnitt 3.3.2). Genauer betrachtet ändert und liest das *Fragment* den Wert, der die IP-Adresse des *Servers* (siehe Abschnitt 3.2) darstellt. Weiterhin ist die Variable, die den Offline- beziehungsweise Onlinemodus bestimmt, der wiederum Einfluss auf das *MapFragment* (siehe Abschnitt 3.3.3) und *DownloadFragment* (siehe Abschnitt 3.3.3) hat, abhängig vom *SettingsFragment*.

Der *View* (siehe Abbildung 3.3a) des *SettingsFragments* wird nur im *SettingsScreen* dargestellt.

MenuFragment

Das *MenuFragment* erstellt und verwaltet das in Abbildung 3.3b dargestellte Layout. Dieses *Fragment* hat die Aufgaben eines einfachen Menüs. Somit besitzt es nur rudimentäre Funktionen. Es interagiert mit dem *ViewModel* (siehe Abschnitt 3.3.2). Genauer überschreibt das *MenuFragment* den Wert der *uiId* des *ViewModels*, falls einer der drei Buttons gedrückt wird. Somit wird die Änderung der UI in der *MainActivity* (siehe Abschnitt 3.3.3) initiiert. Des Weiteren wird eine Variable der *Graph*-Klasse gelesen. Dieser Wert gibt an, ob ein heruntergeladener Subgraph in den Arbeitsspeicher geladen ist. Ist dies nicht der Fall, wird die Navigation angepasst.



(a) SettingsFragment **(b)** MenuFragment

Abbildung 3.3: SettingsFragment und MenuFragment

MapFragment

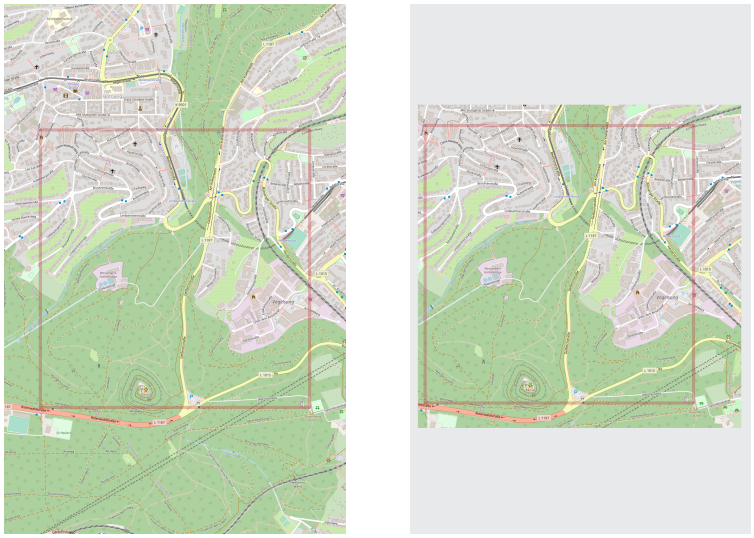
Das *MapFragment* ist für die Erstellung und Verwaltung eines *MapView*-Objekts verantwortlich. Somit verläuft jegliche Interaktion mit der Landkarte über das *MapFragment*. Generell wird nur ein *MapView*-Objekt initialisiert, das während der gesamten Laufzeit der *Android-App* genutzt wird. Falls die *Android-App* durch das *SettingsFragment* (siehe Abschnitt 3.3.3) einen Wechsel zwischen

Online- und Offlinemodus vollzieht, muss eine neue Instanz des *MapView* erstellt werden. Dadurch wird der *Tileprovider* gewechselt, was wiederum die Nutzung der Tiles eines *Tileservers* verhindert und somit die Nutzung der Tiles aus dem heruntergeladenen Tilearchiv und des *Tilecache* erzwingt. Lediglich die *Tilesource* bleibt mit *Mapnik* immer gleich.

Da verschiedene *Fragments* eine Änderung der Landkarte initiieren sollen, ist das *MapFragment* darauf angewiesen, mehrere Variablen aus dem *ViewModel* (siehe Abschnitt 3.3.2) zu beobachten und spezifisch auf Änderungen dieser zu reagieren. Die meisten Funktionen des *MapFragments* werden auf diese Weise aufgerufen. Die einzige Funktion, die unabhängig von anderen *Fragments* aufgerufen wird, ist im *MapEventsReceiver*-Objekt des *MapFragments* deklariert. Diese Funktion ruft die *findNearest*-Funktion der *Graph*-Klasse (siehe Abschnitt 3.3.1) mit den Koordinaten des Interaktionspunkts auf und schreibt das Ergebnis als Start- oder Zielknoten in die jeweils dafür vorgesehene Variable in der *Graph*-Klasse (siehe Abschnitt 3.3.1). Weiter wird dabei der jeweilige Punkt als Start- oder Zielknoten in die Landkarte eingezeichnet. Dabei ist zu beachten, dass dieses *Overlay* nur im *MapScreen* aktiv genutzt wird.

Der durch das *Mapfragment* erstellte *MapView* kann sich auf Grund verschiedener *Overlays* verändern. Die Abbildung 3.4 zeigt den *View* des *MapFragment* im Online- und Offlinemodus. Das rote Rechteck zeigt dabei die heruntergeladene Region an.

Das *MapFragment* tritt in mehreren *Screens* in verschiedener Größe und mit unterschiedlicher Funktionalität auf



(a) Online **(b)** Offline

Abbildung 3.4: MapFragment

PriorityFragment

Das *PriorityFragment* ermöglicht die Definition eines Start- und Zielpunkts, die Änderung des Transportmittels sowie die Personalisierbarkeit der Route. Des Weiteren initiiert das *Fragment* die Berechnung der ausgewählten Route, indem der *Dijkstra-Algorithmus* aus der in Abschnitt 3.3.1 beschriebenen Klasse aufgerufen wird. Die Änderung des Start- oder Zielpunkts sowie des

Transportmittels wird dabei durch jeweils einen Button umgesetzt. Die Personalisierbarkeit wird über ein Dreieck realisiert, in dem der Nutzer seine Priorisierung der verschiedenen Metriken visuell festlegen kann.

Das *PriorityFragment* erstellt und verwaltet den in Abbildung 3.5 dargestellten *View*. Dieses *Fragment* ist eng gekoppelt mit dem *MapFragment* (siehe Abschnitt 3.3.3), da es Werte bestimmter Variablen des *ViewModels* (siehe Abschnitt 3.3.2) ändert, um die Manipulation der Landkarte zu initiieren. In diesem Stil initiieren die beiden Buttons links von dem Dreieck eine Änderung der Landkarte.

Des Weiteren hat das *PriorityFragment* eine enge Bindung zur Klasse *Dijkstra* (siehe Abschnitt 3.3.1). Über die drei Buttons rechts vom Dreieck konfiguriert das *Fragment* das Transportmittel, das für den *Dijkstra-Algorithmus* genutzt wird. Eine weitere Konfiguration bezieht sich auf die Priorisierung der Route, wobei hierfür das Dreieck implementiert wurde.

Dreieck Für das visualisierte Dreieck definiert das *PriorityFragment* eine Metrik in jeweils einer Ecke. Diese Metriken sind Zeit, Distanz und positive Höhendifferenz. Jeder Metrik wird auch ein RGB-Farbwert zugeteilt. Genauer sind das Rot für die positiven Höhenmeter, Blau für die Zeit und Grün für die Distanz. Wenn nun ein Punkt im Dreieck ausgewählt wird, berechnet das *Fragment* die in Abschnitt 2.4 definierten *baryzentrischen Koordinaten* dieses Punktes in Bezug auf das Dreieck. Das Ergebnis wird als Gewichtung der einzelnen Metriken genutzt. Visuell wird das Dreieck in Abhängigkeit der Gewichtung eingefärbt. Dabei werden die zugeteilten Farbwerte der Metriken genutzt, die außerdem bei der Wahl eines Punktes in den jeweiligen Ecken zu sehen sind. Ferner erfolgt die Routenberechnung, über die *Dijkstra*-Klasse (siehe Abschnitt 3.3.1), mit den deklarierten Gewichtungen zum Zeitpunkt der Initiierung.

Nachdem somit das Transportmittel und die Gewichtung der Metriken konfiguriert wurde, kann das *PriorityFragment* die *dijkstra*-Funktion der *Dijkstra*-Klasse aufrufen, falls der Start- und Zielpunkt der Route durch das *MapFragment* (siehe Abschnitt 3.3.3) definiert ist.

Das *Fragment* wird nur im *RouteScreen* genutzt.

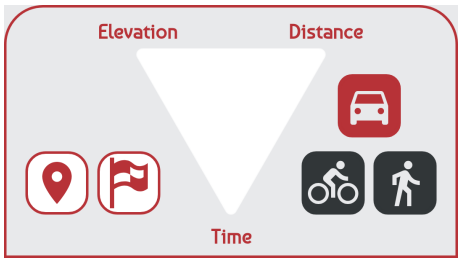


Abbildung 3.5: PriorityFragment

DownloadFragment

Das *DownloadFragment* ist das einzige *Fragment*, das sein Layout während der Ausführung der *Android-App* ändern kann. Der *View* des *Fragments* ist visuell als *Overlay* zu verstehen und besteht lediglich aus zwei Buttons, von denen immer nur einer sichtbar ist. Das Layout wird in Abbildung 3.6 dargestellt.

Dabei interagiert das *DownloadFragment* einerseits mit dem *MapFragment* (siehe Abschnitt 3.3.3), indem es die Zeichnung eines Bereichs, der heruntergeladen werden soll oder heruntergeladen ist, über eine beobachtete Variable initiiert. Andererseits schreibt das *MapFragment* (siehe Abschnitt 3.3.3) die Werte der Koordinaten des Bereichs in Variablen des *ViewModels* (siehe Abschnitt 3.3.2). Diese Werte der Variablen nutzt das *DownloadFragment* als Parameter, um die *getGraph*-Funktion in der *Graph*-Klasse (siehe Abschnitt 3.3.1), sowie alle Funktionen der *TileDownloader*-Klasse (siehe Abschnitt 3.3.1) nacheinander aufzurufen. Falls neue Tiles und der dazugehörige Subgraph heruntergeladen wurden, ruft das *DownloadFragment* im Anschluss die *graphDataToRAM*-Funktion der *Graph*-Klasse (siehe Abschnitt 3.3.1) auf. Somit interagiert das *DownloadFragment* auch mit der *Graph*- (siehe Abschnitt 3.3.1) und *TileDownloader*-Klasse (siehe Abschnitt 3.3.1). Die Initiierung eines Downloads ist dem *DownloadFragment* außerdem nur möglich, falls sich die *Android-App* im Onlinemodus befindet. Da dafür das *SettingsFragment* (siehe Abschnitt 3.3.3) benötigt wird, muss auch indirekt mit diesem interagiert werden.

Das *DownloadFragment* wird nur im *Download-Screen* genutzt.

(a) Layout 1 **(b)** Layout 2

Abbildung 3.6: DownloadFragment

3.3.4 Nutzerinteraktion mit der Android-Applikation

Im Anschluss an den Einblick in die einzelnen Komponenten werden nun die UI und die damit verbundenen Funktionalitäten bei der Interaktion eines Nutzers beschrieben. Dabei ist zu beachten, dass das Design der *Android-App* im Moment nur für den Porträt-Modus optimiert ist.

In Abbildung 3.7 ist die Navigation zwischen den *Screens* dargestellt. Dabei ist zu erwähnen, dass der *MenuScreen* rot eingefärbt ist, da er der Startpunkt der *Android-App* ist. Des Weiteren ist der *SettingsScreen* von allen *Screens* über die *App Bar* erreichbar. Jedoch ist die Navigation im *SettingsScreen* auf die Zurück-Taste beschränkt. Somit kann der Nutzer von hier aus immer nur zu dem vorherigen *Screen* zurückkehren.

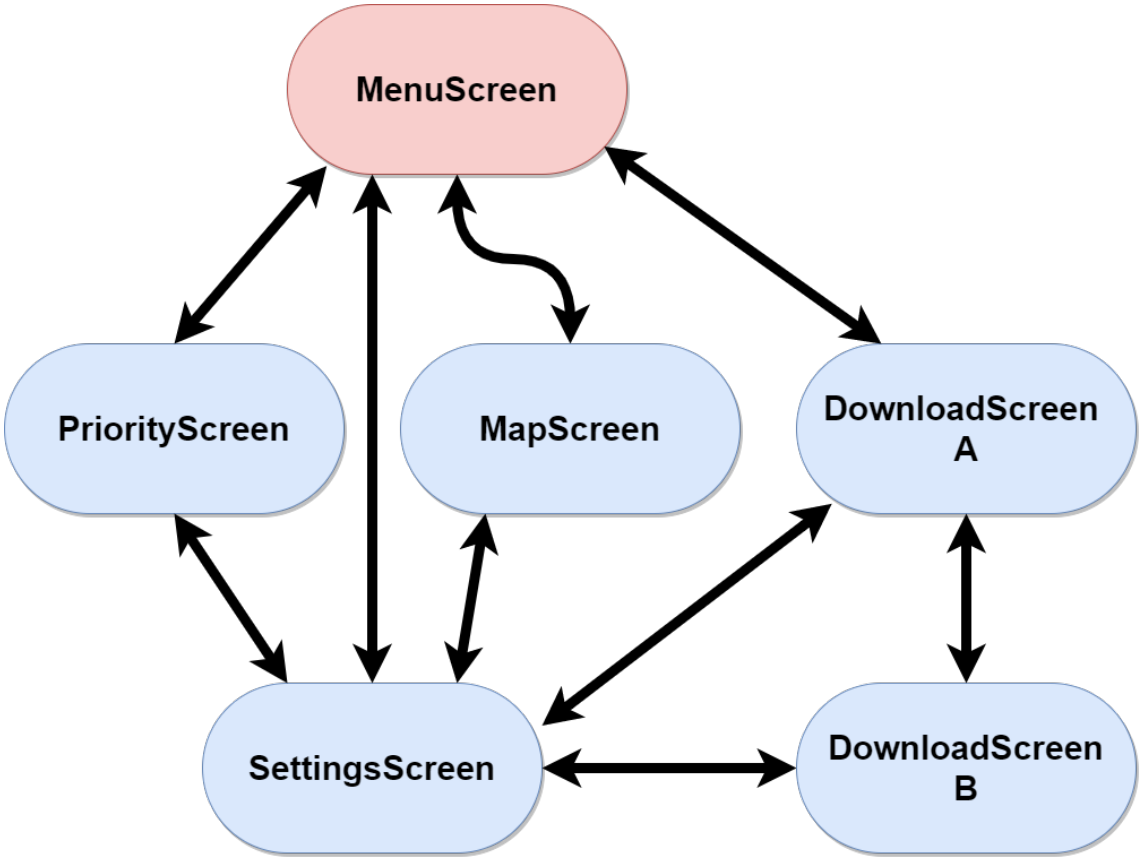


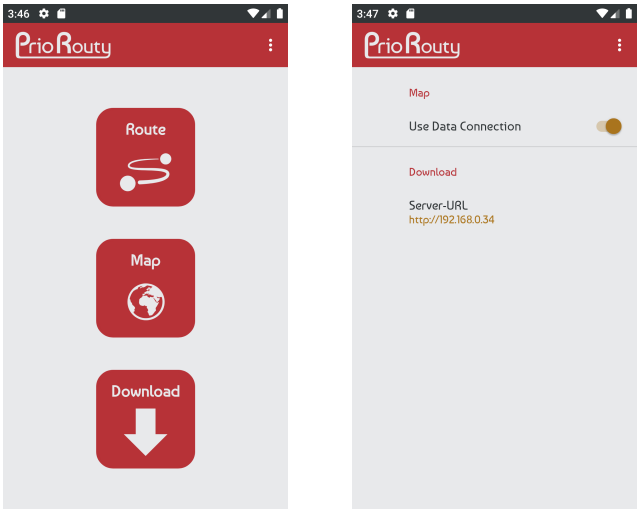
Abbildung 3.7: UI-Navigation

MenuScreen

Der *MenuScreen* (siehe Abbildung 3.8a) ist der Startpunkt der *Android-App* und besteht aus der in Abschnitt 3.3.3 beschriebenen *UI-Komponente MenuFragment* und der *App Bar*. Somit dient der *Screen* dem Nutzer als Menü, das es ihm ermöglicht, zwischen den verschiedenen *Screens* der *Android-App* zu navigieren. Diese Navigation wird über die drei verschiedenen *Buttons* des *MenuFragments* ermöglicht, wobei trivial ist, welcher Button zu welchem *Screen* führt. Der *SettingsScreen* wird über die *App Bar* erreicht.

SettingsScreen

Der *SettingsScreen* (siehe Abbildung 3.8b) besteht aus dem *SettingsFragment* (siehe Abschnitt 3.3.3) und der *App Bar*. Hier kann der Nutzer den Schalter neben *Use Data Connection* aktivieren und deaktivieren. Je nachdem, welchen Status der Schalter hat, wird es dem *MapView* des *Mapfragments* (siehe Abschnitt 3.3.3) ermöglicht, die Tiles der visualisierten Landkarte über den *Tileserver* herunterzuladen oder stattdessen nur das lokale Tilearchiv und den *Tilecache* zu nutzen. Die zweite Option, die der Nutzer konfigurieren kann, ist die *Server-URL* des in Abschnitt 3.2 beschriebenen *Servers*. Die Eingabe des Nutzers muss somit der IP-Adresse des *Servers* entsprechen, ansonsten ist es unmöglich einen Subgraphen herunterzuladen.



(a) MenuScreen **(b)** SettingsScreen

Abbildung 3.8: MenuScreen und SettingsScreen

MapScreen

Der *MapScreen* (siehe Abbildung 3.9) besteht aus dem *MapFragment* (siehe Abschnitt 3.3.3) und der *App Bar*. Dieser *MapScreen* soll dem Nutzer einen ersten Einstieg in die *Android-App* geben. Hier kann der Nutzer das Layout der Landkarte kennenlernen und sich beliebig auf der Landkarte bewegen. Den Ausschnitt der Landkarte, den der Nutzer beim Verlassen des *Screens* ausgewählt hat, wird beibehalten. Somit befindet sich der Nutzer nach dem Wechsel in einen anderen *Screen* an der im vorherigen *Screen* ausgewählten Position. Dies geschieht nur unter der Annahme, dass dieser neue *Screen* das *MapFragment* (siehe Abschnitt 3.3.3) beinhaltet.

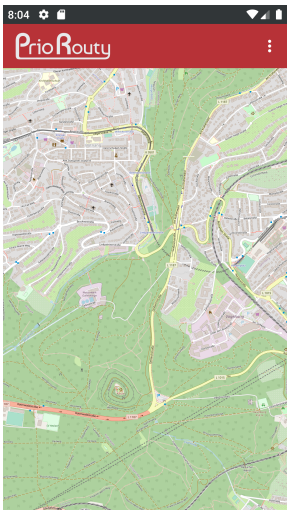


Abbildung 3.9: MapScreen

RouteScreen

Der *RouteScreen* (siehe Abbildung 3.10) besteht aus dem *MapFragment* (siehe Abschnitt 3.3.3), dem *PriorityFragment* (siehe Abschnitt 3.3.3) und der *App Bar*. Mit der Routenberechnung enthält dieser *Screen* das essenzielle *Feature* der *Android-App*.

Die Routenberechnung erfolgt in mehreren Schritten, die in beliebiger Reihenfolge durchgeführt werden können.

Start- und Zielpunkt Sobald sich der Nutzer im *RouteScreen* befindet, kann er, indem er auf die Landkarte drückt, einen Startpunkt auswählen. Dadurch wird dieser durch einen *Marker* auf der Karte visualisiert. Ein zweiter *longPress* auf die Landkarte wählt den Zielpunkt aus, der ebenfalls visualisiert wird. Die zwei Buttons links neben dem Dreieck, mit jeweils der gleichen Darstellung wie die einzelnen *Marker*, kann der Nutzer drücken, um den jeweils ausgewählten Punkt zu löschen und ihn nun neu zu setzen.

Auswahl des Transportmittels Die sich rechts vom Dreieck befindlichen drei Buttons stehen für die drei verschiedenen Transportmittel, die bei der Routenberechnung ausgewählt werden können. Der Nutzer kann durch eine Berührung eines dieser Buttons auswählen, welches Transportmittel genutzt werden soll. Der zu diesem Zeitpunkt ausgewählte Button ist rot eingefärbt. Das per Standard ausgewählte Transportmittel ist das Auto.

Priorisierung Eine Priorisierung für das *personalized route planning* (siehe Abschnitt 2.2) kann der Nutzer über das Dreieck (siehe Abschnitt 3.3.3) definieren. Sobald der Nutzer das Dreieck berührt wird es in der zur Priorisierung stimmigen Farbe eingefärbt und die berührte Stelle als weißer Punkt visualisiert. Der Nutzer hat nun die Möglichkeit den weißen Punkt nach Belieben zu verschieben. Sobald er seinen Touchscreen loslässt wird die Route mit der neuen Priorisierung berechnet und schließlich als *Polyline* zwischen Start- und Zielpunkt dargestellt. Da die Route allerdings nur berechnet und visualisiert wird, falls der Start- und Zielpunkt konfiguriert ist, macht es für den Nutzer am meisten Sinn, die Priorisierung als letztes einzustellen.

In Abbildung 3.11 wird der *RouteScreen* nach der Berechnung demonstrativer Routen dargestellt. Dabei wurden alle Routen mit dem gleichen Start- und Zielpunkt und mit der Transportmitteleinstellung Fußgänger berechnet. Die Route 1 priorisiert dabei hauptsächlich die Metrik der positiven Höhendifferenz. Die Route 2 wiederum beachtet die Metrik der Zeit und der Distanz in ungefähr gleichem Maß, berücksichtigt die positive Höhendifferenz jedoch kaum. Die letzte Route, Route 3, priorisiert hauptsächlich die positive Höhendifferenz, beachtet jedoch auch die Zeit, wodurch sich diese Route ebenfalls von den anderen unterscheidet.

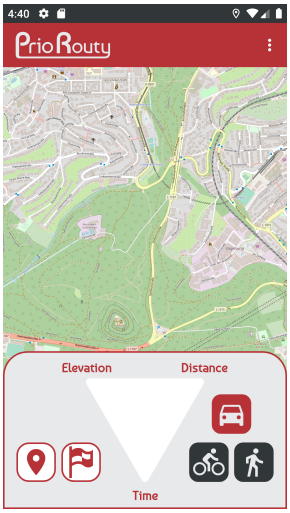
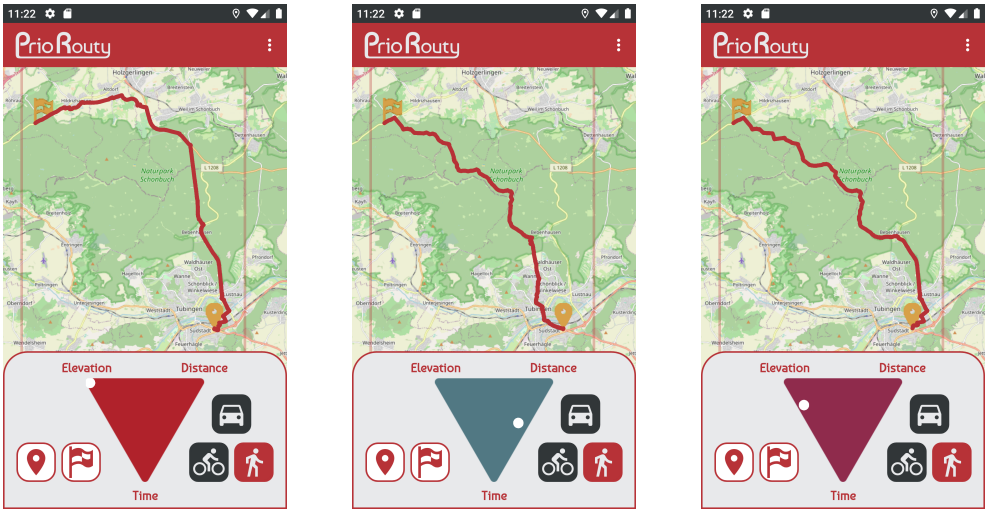


Abbildung 3.10: RouteScreen



(a) Route 1 **(b)** Route 2 **(c)** Route 3

Abbildung 3.11: Demonstrative Routen

DownloadScreen

Der *DownloadScreen* (siehe Abbildung 3.12) verwendet das *MapFragment* (siehe Abbildung 3.4), das *DownloadFragment* (siehe Abbildung 3.6) und die *App Bar*. Ein Nutzer startet im Layout 1 (siehe Abbildung 3.12) und hat hier die Möglichkeit mit der Landkarte zu interagieren oder durch eine Berührung des *Buttons* das Layout 2 (siehe Abbildung 3.12) zu betreten. Im Layout 2 kann der Nutzer das eingezeichnete Rechteck durch einen *longPress* auf einen der Buttons mit anschließendem Ziehen, nach Belieben ändern. Dabei verschiebt er das Rechteck mit dem linken Button oder vergrößert beziehungsweise verkleinert es mit dem rechten Button. Der Bereich innerhalb des Rechtecks zeigt dabei den Bereich an, der heruntergeladen werden soll. Nachdem der Nutzer den gewünschten Bereich ausgewählt hat, kann er durch eine Berührung des Buttons das

Downloadverfahren starten. Im Folgenden wird dem Nutzer zunächst ein Dialog angezeigt, während des Downloads des Subgraphen. Danach wird ein Dialog gestartet, der durchgehend anzeigt wie viele Tiles schon heruntergeladen sind. Infolgedessen wird der Archivierungsprozess durch einen Dialog dargestellt. Und schließlich wird dem Nutzer durch einen Dialog mitgeteilt, dass der Graph in den Arbeitsspeicher geschrieben wird. Diese Dialoge werden in Abbildung 3.12 dargestellt. Nach dem Downloadverfahren wird auf der Landkarte ein rotes Rechteck angezeigt, das dem zuvor Ausgewählten entspricht. Dieses rote Rechteck ist in Abbildung 3.4 zu sehen.

(a) Layout 1 **(b)** Layout 2 **(c)** Subgraphdownload

(d) Tiledownload **(e)** Archivierungsprozess **(f)** Einlesen des Subgraphs

Abbildung 3.12: DownloadScreen

4 Evaluation

In diesem Kapitel werden Statistiken zur Performance und Skalierbarkeit der Implementierung diskutiert. Anschließend werden verschiedene Hindernisse beleuchtet, die während des Projektes auftraten.

4.1 Performance und Skalierbarkeit

In diesem Abschnitt wird die Performance und die Skalierbarkeit der Implementierung betrachtet. Einerseits sind die verschiedenen Prozesse sehr individuell. Andererseits ist die Performance stark abhängig von den Testgeräten und vor allem auch der Internetverbindung. Deswegen wurde auf Referenzwerte außerhalb der eigenen Implementierung verzichtet. Selbst die Performance des *Dijkstra-Algorithmus* ist durch die Personalisierbarkeit der Route schwer mit Richtwerten zu vergleichen.

4.1.1 Spezifikation der Testgeräte

PC

- Betriebssystem: Windows 10 Education
- Prozessor: Intel Core i7-6700K mit vier Kernen und 4.00 GHZ pro Kern
- Arbeitsspeicher: 32 GB

Smartphone Die Smartphone Tests wurden mit einem HUAWEI P1090 getätigt.

- Betriebssystem: Android 9
- Prozessor: Hisilicon Kirin 960 mit acht Kernen und 2.4 GHZ pro Kern
- Arbeitsspeicher: 4 GB

4.1.2 Vorverarbeitung

Die Vorverarbeitung wird hierfür in verschiedene Schritte eingeteilt. Als Testgerät wurde der PC gewählt. Auf ihm wurde der *Server* (siehe Abschnitt 3.2) fünfzig Mal gestartet und im Anschluss der Durchschnitt jedes Schrittes ausgerechnet. Somit folgten die durchschnittlichen Werte, die in Tabelle 4.1 dargestellt werden. Die gesamte Vorverarbeitung dauerte im Schnitt 110 906 Millisekunden.

Schritt	∅ Zeit in Millisekunden
Initialisierung der Arrays	1761 ms
Einlesen der Knoten	29567 ms
Einlesen der Höhenmeter	24315 ms
Erstellung des *Adjazenzarrays* und einlesen der Kanten	55263 ms

Tabelle 4.1: Vorverarbeitung

4.1.3 Subgraph

Die ab der Anfrage an den *Server* (siehe Abschnitt 3.2) bis zur Speicherung des Subgraphen auf dem Smartphone benötigte Zeit, wurde mit fünfzig verschiedenen Subgraphen getestet. Dabei handelt es sich um Subgraphen von randomisierter Größe. Der PC wurde hierbei als *Server* genutzt.

Es ergab sich das in Abbildung 4.1 dargestellte Schaubild. Dabei wird auf der x-Achse die Zeit in ms und auf der y-Achse die Summe der Knoten und Kanten (im Folgenden KK_{sum}) des Subgraphen abgebildet. Die Summe ist auf der Achse in Millionen angegeben. Die einzelnen Punkte entsprechen den einzelnen Durchgängen. Die Gerade entspricht dem durchschnittlichen Wert der Knoten und Kanten, die pro Millisekunde heruntergeladen wurden.

Durchschnittlich wurden 770 Knoten und Kanten innerhalb einer Millisekunde verarbeitet. Es ist zu erkennen, das für eine kleine KK_{sum} die Werte unter der Geraden liegen, wobei bei einer größeren KK_{sum} eine signifikante Menge über der Geraden liegen. Somit ist die Verarbeitung von Knoten und Kanten pro Millisekunde für eine kleine KK_{sum} schlechter als für große KK_{sum}. Genauer verarbeitet ein größerer Subgraph mehr Knoten und Kanten in der Millisekunde, als ein kleinerer Graph. Das ist durch konstante Kosten zu erklären, wie zum Beispiel die Zeit der Anfrage oder die Erstellung der .fmi-Datei auf dem Smartphone. Somit ergibt sich ein positives Ergebnis in Bezug auf die Skalierbarkeit. Da jedoch der Arbeitsspeicher von herkömmlichen Smartphones begrenzt ist (siehe Abschnitt 4.3), folgt eine angepasste Grenze der Skalierbarkeit.

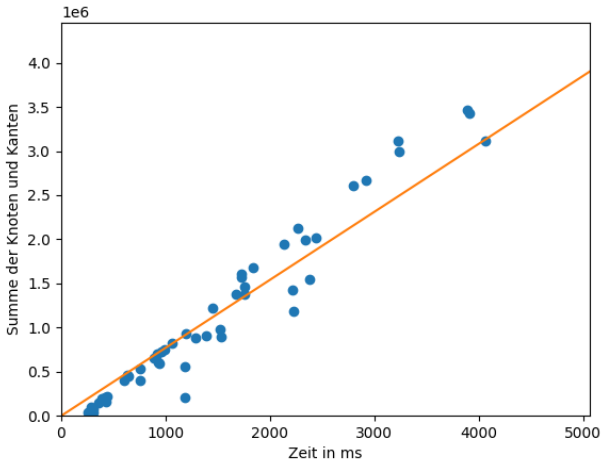


Abbildung 4.1: Download und Speicherung des Subgraphen

4.1.4 Download der Tiles und Archivierung

Der Download und die Archivierung der Tiles wurde fünfzig Mal mit randomisierter Größe des Ausschnitts durchgeführt. Das Zoomlevel der heruntergeladenen Tiles betrug 3 bis 16.

Download

Bei dem Download der Tiles wurden die in Abbildung 4.2 dargestellten Werte ausgegeben. Hierbei wird auf der x-Achse die Zeit für den Prozess in Millisekunden angegeben und auf der y-Achse die Anzahl der Tiles. Genauer geben die Punkte die einzelnen Werte der Durchgänge wider. Die Gerade entspricht dem durchschnittlichen Wert heruntergeladener Tiles pro Millisekunde.

Im Durchschnitt benötigt der Download 2,28 Millisekunden pro Tile. Dabei ist zu erkennen, dass die sich die Zeit pro Tile linear verhält.

Das liegt daran, dass die konstanten Kosten pro Tiles stattfinden. Die Skalierbarkeit ist somit in Bezug auf die Zeit pro Tile konstant.

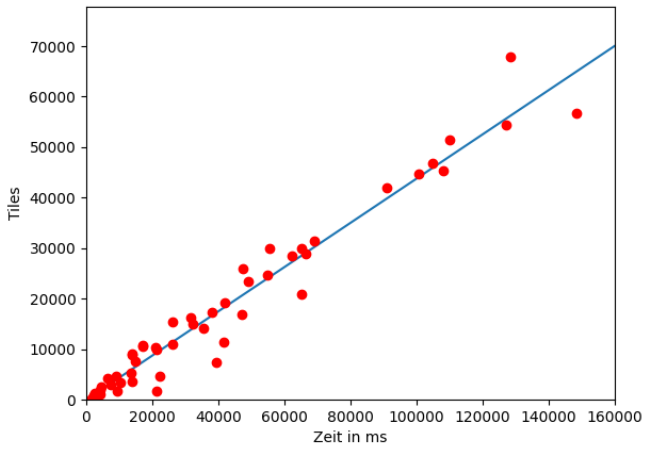


Abbildung 4.2: Download der Tiles

Archivierung

Die Darstellung der Archivierung der Tiles (siehe Abbildung 4.3) ist an die des Downloads angepasst. Somit befindet sich hier auf der x-Achse die Zeit der Archivierung in Millisekunden und auf der y-Achse die Anzahl der Tiles. Die Punkte entsprechen den einzelnen Werten pro Durchgang und die Gerade bezieht sich auf den durchschnittlichen Wert der archivierten Tiles pro Millisekunde.

Im Durchschnitt dauert die Archivierung 1,78 Millisekunde pro Tile. Allerdings ist zu erkennen, dass mehr Zeit für größere Subgraphen benötigt wird als der Durchschnitt erwarten lässt.

Diese Erkenntnis ist wie folgt zu erklären: Je größer der Subgraph, desto höher die Wahrscheinlichkeit, dass eine geringe Anzahl an Tiles in einzelnen Ordnern platziert ist, die archiviert werden müssen. Somit wird die Ordnerstruktur (siehe Abschnitt 3.3.1) unter Umständen signifikant größer. Die Skalierbarkeit der Archivierung ist somit von diesem Zufallsfaktor abhängig und wird sich negativ auf die Tiles pro Millisekunde auswirken.

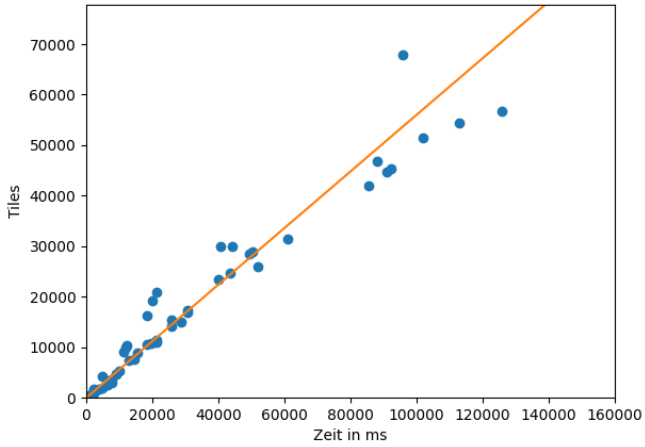


Abbildung 4.3: Archivierung der Tiles

4.1.5 Vergleich

In der Abbildung 4.4 werden die durchschnittliche Zeit pro Tile für den Download und die Archivierung verglichen. Es ist zu erkennen, dass die Zeit für die Archivierung in der Regel kürzer ist als für den Download. Jedoch werden sich die beiden Zeiten bei großen Subgraphen angleichen, da die Archivierung der Tiles pro Millisekunde nicht konstant bleibt.

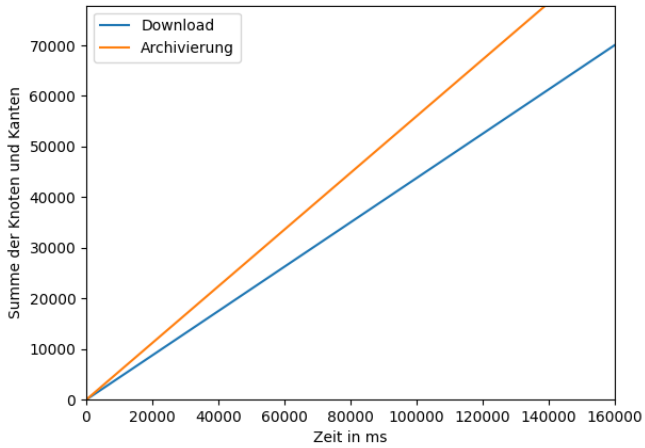


Abbildung 4.4: Download versus Archivierung

4.1.6 Einlesen des Graphen auf dem Smartphone

Für den Prozess, der den Subgraphen in den Arbeitsspeicher des Smartphones liest, wurden vier repräsentative Ausschnitte ausgewählt und der Graph dann zehn Mal eingelesen. Daraufhin wurde der Durchschnitt dieser Werte berechnet und in Tabelle 4.2 dargestellt. Wie die Tabelle 4.2 veranschaulicht, ist das Einlesen des Subgraphen für kleine sowie für große Ausschnitte in Bezug auf Knoten und Kanten pro Millisekunde nicht signifikant unterschiedlich. Somit ist die Skalierbarkeit lediglich durch fehlenden Arbeitsspeicher (siehe Abschnitt 4.3) gehemmt.

Anzahl der Knoten	Anzahl der Kanten	∅ Zeit in ms	Knoten/Kanten pro ms
237664	506793	2377 ms	313
430775	923904	4067 ms	332
2460437	5251543	23631 ms	326
6201172	13176763	61112 ms	317

Tabelle 4.2: Einlesen des Subgraphen

4.1.7 Dijkstra-Algorithmus

Der *Dijkstra-Algorithmus* wurde für drei repräsentative Ausschnitte getestet. Dabei wurde er zehn Mal pro Ausschnitt und Transportmittel ausgeführt, wobei die Routen bewusst so groß wie möglich gewählt wurden. Die Ergebnisse wurden in Tabelle 4.3 festgehalten. Dabei wurde für die Zeit in Millisekunden der Durchschnitt der Durchläufe gewählt.

Zu erkennen ist, dass der Algorithmus am schnellsten für das Auto abläuft. Langsamer ist er für das Fahrrad und am langsamsten für den Fußgänger. Das liegt daran, dass für das Fahrrad im gleichen Graphen mehr potentielle Routen geprüft werden müssen. Für den Fußgänger gibt es wiederum mehr potentielle Routen als für den Radfahrer.

Anzahl der Knoten	Anzahl der Kanten	Transportmittel	∅ Zeit in Millisekunden
237664	506793	Auto	179 ms
		Fahrrad	214 ms
		Zu Fuß	361 ms
430775	923904	Auto	347 ms
		Fahrrad	415 ms
		Zu Fuß	651 ms
2460437	5251543	Auto	2265 ms
		Fahrrad	2864 ms
		Zu Fuß	3719 ms

Tabelle 4.3: Dijkstra-Statistik

4.2 SRTM-Daten

In diesem Abschnitt wird sich tiefer mit den Daten der *SRTM* beschäftigt.

4.2.1 Genauigkeit der Höhendaten

Im Folgenden wird sich kurz mit der Genauigkeit der einzelnen Höhenwerte der Messproben beschäftigt. Die *SRTM*-Daten wurden durch die Technik der Interferometrie berechnet (siehe Abschnitt 2.6). Hierbei kommen statische und zeitabhängige Fehler vor. [RMB06]

Eine weitere Problematik bei der Messung der Daten ergibt sich dadurch, dass 5, 6-Zentimeter-Wellen für das Radarsignal genutzt wurden. Diese Wellenlänge durchdringt Vegetation nicht konsistent. Somit wurde zum Beispiel leicht bewachsene Böschung durchdrungen, jedoch stärker bewachsene nicht. Folglich gibt es im Datensatz Höhenmessungen, die sich auf Böschungen, wie zum Beispiel Baumkronen beziehen. [JPL22d]

Infolgedessen variiert die Genauigkeit der Daten von Region zu Region, da verschiedene geografische Gegebenheiten vorhanden sind. Auch die optimierten *NASA Version 3 SRTM Global* Daten sind in Bezug auf ihre Genauigkeit nicht optimal. Im Artikel [Elk18] von Elkhrachy werden die Daten zum Beispiel in einer kleinen Region von $900 km^2$ mit den Höhendaten einer topographischen Karte verglichen. Spezifisch für diese Region waren Höhen von 1080 bis 2252 Metern, sowie allgemein große Höhendifferenzen. Dabei betrug das Ergebnis für die *NASA Version 3 SRTM Global* Daten mit einer Auflösung von einer Bogensekunde, die vertikale Genauigkeit von ±6, 87 Metern.

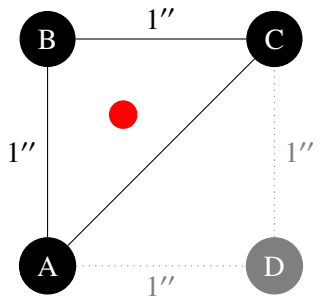
4.2.2 Auflösung der Höhendaten

Durch die *SRTM*-Daten und deren Auflösung gab es Hindernisse bei der Implementierung. Die maximale Auflösung der Daten von $1''$ entspricht ungefähr einer Distanz von 30 Metern zwischen den Messproben (siehe Abschnitt 3.1.3) und ist somit nicht genau genug für spezifische Koordinaten und deren Höhendaten. Die Tatsache, dass nicht alle Knoten im Subgraphen genau einer Messprobe zuzuordnen sind, kann in bestimmten Fällen zu extremen Differenzen zwischen der zugeordneten Höhe und der realen Höhe führen. In Gegenden mit einer großen Höhendifferenz zwischen zwei benachbarten Messproben, also in der Regel in bergigen Regionen, ist es so teilweise nicht möglich sinnvolle Höhendaten auszulesen.

So kann es in bestimmten Regionen passieren, dass ein Knoten zwischen drei Messproben liegt, wobei die Werte der Messproben, Differenzen von 500 Metern und mehr zueinander besitzen. Folglich wird in der Implementierung in bestimmten Fällen, trotz der Durchführung der *Dreiecksinterpolation* (siehe Abschnitt 2.4), ein Wert mit signifikanter Differenz zu naheliegenden Werten der Höhe berechnet. Diese Differenz entspricht somit nicht ansatzweise der realen Höhendifferenz.

Das folgende Schaubild zeigt einen Knoten, der Rot gefärbt ist. Zu sehen sind außerdem vier Messpunkte, wobei die schwarz eingefärbten A, B und C die drei nächsten Messpunkte zum Knoten sind. Diese drei schwarzen Messpunkte spannen ein Dreieck auf, um die *Dreiecksinterpolation* (siehe Abschnitt 2.4) zu symbolisieren. Dabei haben die Messpunkte die folgenden Höhenwerte: $A = 500$, $B = 1200$, $C = 1100$. In der Realität kann der Messpunkt A zum Beispiel in einem See liegen, der von Bergen umgeben ist. Die Messpunkte B und C wurden auf dem Berg gemessen und

der gesuchte Knoten befindet sich auf einem Wanderweg, der sich auf einem Plateau zwischen See und Berg befindet. Das Plateau liegt mit 600 Metern vergleichsweise nur leicht über dem See. Jedoch wird dem gesuchten Knoten wegen der zwischen Plateau und den Messpunkten B und C liegenden Bergwand durch die *Dreiecksinterpolation* (siehe Abschnitt 2.4) ein Wert von 933 Metern zugewiesen.



Um die daraus resultierenden falschen Werte der Kanten zwischen den besagten Knoten in der Implementierung anzupassen, wurde die Höhendifferenzen aller Kanten analysiert. In Tabelle 4.4 wird die Anzahl der Kanten E angegeben, die eine Höhendifferenz h_e besitzen, die größer ist als eine untere Schranke d.

$\|E\| : h_e \geq d$	d
687592	10 m
79230	20 m
15849	30 m
4274	40 m
1521	50 m
634	60 m
311	70 m
186	80 m
107	90 m
74	100 m
6	200 m
1	300 m
0	320 m

Tabelle 4.4: Höhendifferenz

Da die durchschnittliche Distanz aller Kanten 32 Meter beträgt, wurden die Werte aller Kanten für die Höhendifferenz auf 30 Meter gesetzt, falls dieser größer als 30 Meter betrug. Dieses Vorgehen ist keine Lösung des Problems, jedoch gibt es den verfälschten Kanten einen realistischeren Wert für die Höhendifferenz, wodurch die Routenplanung in der Region dieser Kanten zumindest die Möglichkeit hat, die beste Route zu finden.

4.3 Graphlimitierung

Der auf 4 GB limitierte Arbeitsspeicher des benutzten Testgeräts führt zu einem weiteren Hindernis. Die *Android-App* speichert die Daten des heruntergeladenen Subgraphen während der Ausführung in den Arbeitsspeicher (siehe Abschnitt 3.3.1). Dabei ist zu beachten, dass die ursprüngliche .fmi-Datei des Straßennetzwerks Deutschlands mit den drei Transportmitteln 8,5 GB groß ist (siehe Abschnitt 3.2.1). Diese Menge an Daten im Arbeitsspeicher des Testgeräts zu halten, ist unmöglich. Um die Menge an Daten des Subgraphen zu verkleinern, werden irrelevante Daten aussortiert (siehe Abschnitt 3.2.1). Außerdem wird die Typisierung der einzelnen Arrays an die Daten angepasst (siehe Abschnitt 3.2.1). Die für die Darstellung des gesamten Straßennetzwerks benötigte Speicherkapazität kann durch die Berechnung der Speicherkapazität der einzelnen Arrays (siehe Abschnitt 3.2.1) ermittelt werden. Die folgenden Werte werden lediglich durch die Multiplikation der Anzahl der Elemente des Arrays und der Größe des Typs berechnet. [Kot22]

```
breitengradDerKnoten: 567,3 MB
längengradDerKnoten: 567,3 MB
transportmittelIdDerKnoten: 70,9 MB
hilfsarray: 283,7 MB
zielknotenDerKante: 600,4 MB
zeitDerKante: 600,4 MB
distanzDerKante: 300,2 MB
transportmittelIdDerKanten: 150,1 MB
```

Somit benötigt alleine die Darstellung des Deutschlandgraph mindestens 3,14 GB des Arbeitsspeichers. Diesen Graphen als Subgraphen für die *Android-App* zu nutzen, ist folglich mit dem Testgerät nicht möglich.

Eine Lösung für dieses Problem, das sich spezifisch auf den Arbeitsspeicher des genutzten Smartphones bezieht, konnte aus Zeitgründen nicht implementiert werden. Jedoch wird im *Server* (siehe Abschnitt 3.2) eine statische Schranke eingebaut, die wiederum mit der Summe der Knoten und Kanten des angefragten Subgraphen verglichen wird (siehe Abschnitt 3.2.4) und bei Überschreitung der Schranke die Anfrage abbricht. Um diese Schranke für das Testgerät zu finden, werden manuell Subgraphen getestet, bis eine Größe ermittelt wird, mit der die *Android-App* ohne Komplikationen läuft. Diese ermittelte Größe wird aus zwei Gründen bewusst kleiner als Schranke gewählt. Zum einen soll so gut wie möglich verhindert werden, dass die Android Garbage Collection aktiv wird. Das schont die central processing unit (CPU) und mindert die Wahrscheinlichkeit auf ungewolltes Verhalten der *Android-App* . Der zweite Grund für eine kleinere Schranke ist, dass zwei Kartenausschnitte die gleiche Summe der Knoten und Kanten besitzen können und gleichzeitig verschieden viele Tiles zur Darstellung benötigen.

Mit diesem Vorgehen wurde die Schranke von 19 Millionen Knoten und Kanten ausgewählt. Die Speicherkapazität des Subgraphen liegt dabei ungefähr bei 275 MB. Die zum Subgraph gehörende .fmi-Datei hat dabei eine Größe von ungefähr 595 MB und muss in *graphDataToRAM* (siehe Abschnitt 3.3.1) geöffnet werden. Somit wird bis zu 870 MB des Arbeitsspeichers bei der Einlesung des Graphen benötigt.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Bachelorarbeit wurden zuerst verschiedene Begriffe und Definitionen erörtert. Zunächst wurde mit dem *Dijkstra-Algorithmus* eine Lösung für die Kürzeste-Wege-Suche definiert. Die Definition wurde durch ein ausführliches Beispiel und eine Pseudocode-Darstellung des Algorithmus erläutert. Infolgedessen wurde das Problem des *personalized route planning* nach Funke und Storandt [FS15] definiert, das durch die dynamischen Werte der Kantengewichte entsteht. Diese sich wechselnden Kantengewichte verhindern eine Vorverarbeitung, auf die moderne Algorithmen der Routenplanung angewiesen sind. Als nächstes wurde die Erstellung eines *Adjazenzarrays* nach Mehlhorn und Sanders [MS08] beschrieben und durch ein Beispiel verdeutlicht. Dabei handelt es sich um eine Datenstruktur, durch die ein statischer Graph in Arrays abgespeichert werden kann und trotzdem ein schneller Zugriff auf die ausgehenden Kanten eines spezifischen Knotens gewährleistet wird. Im Anschluss daran wurde die Prozedur der *Dreiecksinterpolation* definiert, die auf der Definition der *baryzentrischen Koordinaten* eines Punktes nach Koecher und Krieg [KK07] basiert. Hierbei wurde die Prozedur nach der Definition durch ein Beispiel verdeutlicht. Anschließend wurden Hintergrundinformationen in Bezug auf die Herkunft der genutzten Daten beleuchtet, indem die *SRTM* und das *OSM*-Projekt beschrieben wurden. Im letzten Teil des Kapitel 2 wurden die für die *Android-App* essenziellen *Android*- und *osmdroid*-Objekte beschrieben.

Im Kapitel 3 folgte die Konzipierung und Dokumentation der Entwicklung eines personalisierbaren Android-Offline-Routenplaners. Dabei wurden zuerst die drei Komponenten des Routenplaners bestehend aus dem *Server*, der *Android-Applikation* und dem *Tileserver* und ihre Beziehung zueinander erläutert. Da der *Tileserver* bereits durch das FMI gegeben war und die *Android-Applikation* vom *Server* und dem *Tileserver* abhängig ist, wurde die Implementierung des *Servers* bevorzugt. Für die Implementierung des *Servers* waren Geoinformationen notwendig, die somit zuerst beschrieben wurden. Dabei handelte es sich um *SRTM*- und *OSM*-Datensätze, deren Formate näher betrachtet wurden. Anschließend wurde der *Server* genauer beschrieben. Dabei handelt es sich um einen mit Java 17 implementierten *HTTP*-Server, der den deutschlandweiten Graphen aus den *OSM*-Datensätzen und die dazugehörigen Höhendaten aus den *SRTM*-Datensätzen in einem Vorverarbeitungsschritt im Arbeitsspeicher abspeichert. Des Weiteren gibt der *Server* bei einer Anfrage einen spezifischen Subgraphen des deutschlandweiten Graphen zurück. Als nächstes wurde die *Android-Applikation* thematisiert. Dabei wurden die *Single-Activity*-Architektur und die dazugehörigen Komponenten ausführlich beschrieben. Darüber hinaus wurden das Design der *Android-Applikation* und die dazugehörige Nutzerinteraktion durch Screenshots veranschaulicht.

Nach der ausführlichen Dokumentation des Vorgehens wurden in Kapitel 4 zuerst Statistiken zur Performance und Skalierbarkeit der Implementierung diskutiert. Zunächst wurde die Vorverarbeitung des *Servers* gemessen (siehe Abschnitt 4.1.2). Dabei wurde der Vorverarbeitungsvorgang im Schnitt nach 110 906 Millisekunden beendet. Allerdings hat das Ergebnis aufgrund fehlender Vergleichswerte keine Aussagekraft über die Performance der Komponente. Als nächstes wurde die ab der Anfrage an den *Server* bis zur Speicherung des Subgraphen auf dem Smartphone benötigte Zeit gemessen (siehe Abschnitt 4.1.3). Die Messungen ergaben dabei, dass durchschnittlich 770 Knoten

und Kanten in einer Millisekunde verarbeitet werden. Des Weiteren kann durch die Messungen angenommen werden, dass die Skalierbarkeit nur durch den Arbeitsspeicher des Smartphones begrenzt wird. Darauf folgend wurden der Download der Tiles und die Archivierung dieser gemessen (siehe Abschnitt 4.1.4). Dabei beläuft sich die durchschnittliche Zeit des Downloads eines Tiles auf 2,28 Millisekunden, während die Archivierung pro Tile 1,78 Millisekunden benötigt. Die Skalierbarkeit ist in Bezug auf die Zeit pro Tile beim Download gegeben, jedoch nicht bei der Archivierung. Anschließend wurde die Zeit beim Einlesen des Subgraphs auf dem Smartphone gemessen, wobei die durchschnittliche Zeit von vier verschieden großen Subgraphen beachtet wurde (siehe Abschnitt 4.1.6). Die durchschnittliche Anzahl der Knoten und Kanten die pro Millisekunde eingelesen wurden, variierte dabei von 313 bis 332 Millisekunden. Da der größte Graph mit 317 Millisekunden gut abschnitt, impliziert das wiederum die Skalierbarkeit dieser Komponente. Die letzten Messwerte wurden für den *Dijkstra-Algorithmus* der *Android-App* gemessen (siehe Abschnitt 4.1.7). Dabei ist zu erkennen, dass der Algorithmus für die unterschiedlichen Transportmittel im Durchschnitt unterschiedlich lange braucht. Das ist mit der Menge der potentiellen Kanten der Transportmittel zu erklären. Nach den Statistiken wurden zwei Hindernisse während des Projektes beleuchtet. Das erste Hindernis bezieht sich auf Auflösung und Genauigkeit der Höhendaten des *SRTM*-Datensatzes (siehe Abschnitt 4.2). Es ergab sich, dass die Höhendaten an bestimmten Stellen für die Routenplanung unbenutzbar sind. Dieses Problem wurde durch die Manipulation der spezifischen Höhendaten insofern gelöst, als dass in den betroffenen Gebieten zumindest die Möglichkeit auf eine sinnvolle Routenplanung besteht. Abschließend wurde das zweite Hindernis der Graphlimitierung betrachtet (siehe Abschnitt 4.3). Das Problem hierbei war der auf 4 GB limitierte Arbeitsspeicher des genutzten Testgeräts. Diese Menge an Speicher reicht nicht ansatzweise aus, um den gesamten Graphen des deutschen Straßennetzwerks zu halten. Um die *Android-App* funktionstüchtig zu implementieren war es somit notwendig, eine spezifisch auf das Testgerät abgestimmte obere Schranke für die Größe des Subgraphen zu deklarieren.

Ausblick

Die Aufgabenstellung wurde erfüllt und das Ergebnis in Form des personalisierbaren Android-Offline-Routenplaners ist funktionstüchtig, jedoch noch lange nicht ausgereift. Das Design der *Android-App* ist im Moment auf den Porträt-Modus limitiert, was eine erste Optimierung darstellen könnte. Außerdem können die in Kapitel 4 beschriebenen Hindernisse nur als provisorisch gelöst angesehen werden. Somit ist eine Lösung dieser Probleme erstrebenswert. Dies würde es außerdem ermöglichen, den Routenplaner mit neuen Funktionen zu bestücken, wie zum Beispiel die Einbindung anderer Metriken oder die Nutzung eines schnelleren Algorithmus für die Routenberechnung.

Literaturverzeichnis

[Alg22a] F. .-. A. Algorithmik. 12. Mai 2022. URL: https://github.com/fmi-alg/OsmGraphCreator (zitiert auf S. 23).

[Alg22b] F. .-. A. Algorithmik. 12. Mai 2022. URL: https://github.com/fmi-alg/OsmGraphCreator/blob/master/data/configs/all.cfg (zitiert auf S. 24).

[And22a] Android. *Fragments*. 9. Mai 2022. URL: https://developer.android.com/guide/fragments (zitiert auf S. 20).

[And22b] Android. *The Activity Lifecycle*. 9. Mai 2022. URL: https://developer.android.com/guide/components/activities/activity-lifecycle (zitiert auf S. 20).

[And22c] W. Android. *Activity — Android Wiki*. 9. Mai 2022. URL: https://www.droidwiki.org/w/index.php?title=Activity&oldid=10657 (zitiert auf S. 20).

[Dij+59] E. W. Dijkstra et al. „A note on two problems in connexion with graphs“. In: *Numerische mathematik* 1.1 (1959), S. 269–271 (zitiert auf S. 11).

[Elk18] I. Elkhrachy. „Vertical accuracy assessment for SRTM and ASTER Digital Elevation Models: A case study of Najran city, Saudi Arabia“. In: *Ain Shams Engineering Journal* 9.4 (2018), S. 1807–1817. ISSN: 2090-4479. DOI: https://doi.org/10.1016/j.asej.2017.01.007. URL: https://www.sciencedirect.com/science/article/pii/S2090447917300084 (zitiert auf S. 47).

[FS15] S. Funke, S. Storandt. „Personalized Route Planning in Road Networks“. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '15. Seattle, Washington: Association for Computing Machinery, 2015. ISBN: 9781450339674. DOI: 10.1145/2820783.2820830. URL: https://doi.org/10.1145/2820783.2820830 (zitiert auf S. 8, 9, 15, 50).

[JPL22a] J. P. L. JPL. 12. Mai 2022. URL: https://www2.jpl.nasa.gov/srtm/mission.htm (zitiert auf S. 19).

[JPL22b] J. P. L. JPL. 12. Mai 2022. URL: https://www2.jpl.nasa.gov/srtm/coverage.html (zitiert auf S. 19).

[JPL22c] J. P. L. JPL. 12. Mai 2022. URL: https://www2.jpl.nasa.gov/srtm/instr.htm (zitiert auf S. 19).

[JPL22d] J. P. L. JPL. 12. Mai 2022. URL: https://www2.jpl.nasa.gov/srtm/faq.html (zitiert auf S. 47).

[KK07] M. Koecher, A. Krieg. *Ebene Geometrie*. Springer-Verlag, 2007 (zitiert auf S. 17, 18, 50).

[KN09] S. Krumke, H. Noltemeier. *Graphentheoretische Konzepte und Algorithmen*. Springer, 2009. ISBN: 978-3-8348-0629-1. DOI: 10.1007/978-3-8348-9592-9 (zitiert auf S. 11).

[Kot22] F. Kotlin. *Basic Types*. 14. Mai 2022. URL: https://kotlinlang.org/docs/basic-types.html#numbers (zitiert auf S. 49).

[LPD22a] LPDAAC. *NASA Shuttle Radar Topography Mission (SRTM) Global 1 arc second Data Released Over the Middle East*. 1. Mai 2022. URL: https://lpdaac.usgs.gov/news/nasa-shuttle-radar-topography-mission-srtm-global-1-arc-second-data-released-over-the-middle-east/ (zitiert auf S. 20).

[LPD22b] LPDAAC. *The Shuttle Radar Topography Mission (SRTM) CollectionUser Guide*. 1. Mai 2022. URL: https://lpdaac.usgs.gov/documents/179/SRTM_User_Guide_V3.pdf (zitiert auf S. 25).

[MS08] K. Mehlhorn, P. Sanders. „Algorithms and Data Structures: The Basic Toolbox“. In: *Algorithms and Data Structures: The Basic Toolbox* (Jan. 2008). DOI: 10.1007/978-3-540-77978-0 (zitiert auf S. 16, 50).

[NAS22] NASA. 12. Mai 2022. URL: https://urs.earthdata.nasa.gov/ (zitiert auf S. 26).

[Ope22a] F. OpenStreetMap. 1. Mai 2022. URL: https://www.openstreetmap.de/ (zitiert auf S. 19).

[Ope22b] F. OpenStreetMap. 12. Mai 2022. URL: https://wiki.openstreetmap.org/wiki/Slippy_map_tilenames (zitiert auf S. 29).

[osm22a] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/javadocs/osmdroid-android/debug/index.html?org/osmdroid/views/MapView.html (zitiert auf S. 20).

[osm22b] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/Map-Sources.html (zitiert auf S. 21).

[osm22c] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/javadocAll/org/osmdroid/views/overlay/Overlay.html (zitiert auf S. 21).

[osm22d] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/javadocs/osmdroid-android/debug/index.html?org/osmdroid/views/overlay/Marker.html (zitiert auf S. 21).

[osm22e] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/javadocAll/org/osmdroid/views/overlay/Polyline.html (zitiert auf S. 21).

[osm22f] osmdroid. 4. Mai 2022. URL: https://osmdroid.github.io/osmdroid/javadocAll/org/osmdroid/views/overlay/MapEventsOverlay.html (zitiert auf S. 21).

[osm22g] osmdroid. *osmdroid*. 4. Mai 2022. URL: https://github.com/osmdroid/osmdroid#readme (zitiert auf S. 20).

[RMB06] E. Rodriguez, C. S. Morris, J. E. Belz. „A global assessment of the SRTM performance“. In: *Photogrammetric Engineering & Remote Sensing* 72.3 (2006), S. 249–260 (zitiert auf S. 47).

Alle URLs wurden zuletzt am 15. 05. 2022 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