Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Trajektorienaufzeichnung mit Map-Matching auf Android-Geräten

Lukas Trautwein

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Prof. Dr. Stefan Funke
Beginn am:	1. Februar 2022
Beendet am:	11. Juli 2022

Kurzfassung

Map-Matching ist der Prozess, aus einer Sequenz von möglicherweise ungenauen Positionsmessungen den am wahrscheinlichsten traversierten Pfad auf einem Wegenetz zu rekonstruieren. Gegenstand dieser Arbeit ist die Entwicklung einer Android-App, welche durch Nutzung von GPS-Positionsmessungen die Aufzeichnung von Bewegungsmustern auf einem Wegenetz erlaubt. Hierzu wird ein performanter Map-Matching-Algorithmus implementiert, welcher die Ausführung der nötigen Kalkulationen in Echtzeit auf dem Mobilgerät erlaubt. Im Zuge der Implementierung werden Probleme und Performance-Einschränkungen des grundlegenden Map-Matching-Algorithmus bei einer Verwendung in der echten Welt aufgezeigt und entsprechende Lösungen und Optimierungen für diese erarbeitet und implementiert. Es wird gezeigt, dass eine performante und energiesparende Ausführung von Map-Matching-Algorithmen auf Mobilgeräten möglich ist, auch wenn diese im Hintergrund ausgeführt werden. Die begrenzten Hardware-Ressourcen, die auf Mobilgeräten zur Verfügung stehen, setzen jedoch gewisse Grenzen bei den Datenmengen, die durch den Algorithmus verarbeitet werden können.

Inhaltsverzeichnis

1 Einleitung

Smartphones sind im letzten Jahrzehnt zu allgegenwärtigen Gegenständen geworden, die von den meisten Personen fast immer mitgeführt werden. Mit ihrer leistungsstarken Hardware, Kommunikationsschnittstellen und Sensoren ermöglichen sie die Aufzeichnung und Verarbeitung von vielfältigen Daten.
Da Smartphones von Natur aus mobile Geräte sind, spielt die Position des Gerätes hierbei oft eine zentrale Rolle. Smartphones bieten aus diesem Grund mehrere Möglichkeiten, um ihre aktuelle Position mit unterschiedlicher Präzision zu bestimmen.

Die Sammlung von Rohdaten ohne eine weitere Zuordnung oder Verarbeitung dieser bietet jedoch in den meisten Fällen keinen großen Nutzen. Gerade bei Positionsdaten ist eine weitere Verarbeitung sinnvoll, da die Rohdaten oft wenig Aussagekraft besitzen. So ist eine GPS-Messung ein diskreter Datenpunkt, jedoch bewegt sich das Gerät auch in der Zeit zwischen den Messungen. Ferner ist eine Sequenz von GPS-Messungen nur eine Sequenz von Koordinaten, die Punkte auf der Erde beschreiben. Die reale Welt ist jedoch keine einfache Kugel. Die Bewegung eines Menschen auf Punkte auf der Oberfläche der Kugel zu reduzieren, lässt die Komplexität der Wegfindung in der realen Welt außer Acht.
Menschen bewegen sich im Regelfall mithilfe von Wegen, seien diese nun Bahnstrecken oder Straßen. Von diskreten Positionsmessungen auf den Pfad zurückschließen zu können, den ein Nutzer in einem Wegenetz zurückgelegt hat, kann für die weitere Interpretation von gesammelten Daten ein integraler Schritt sein. Dieses Verfahren der Zuordnung von Positionsmessungen ist als *Map-Matching* bekannt.

Gerade bei Prozessen, die auf Mobilgeräten über längere Zeit ablaufen, wie es bei der Aufzeichnung von Bewegungsmustern der Fall ist, spielt jedoch der Energieverbrauch eine große Rolle. Insbesondere GPS ist weithin dafür bekannt, Batterien schnell zu entleeren. Algorithmen, die mit einer niedrigeren Frequenz von GPS-Messungen weiterhin eine akzeptable Genauigkeit der Erfassung von Bewegungsmustern erlauben, sind deshalb von besonderem Interesse.
Aber auch leistungsintensive Algorithmen, die den Prozessor des Gerätes belasten, können einen negativen Einfluss auf die Akkulaufzeit haben und sollten vermieden werden.

Ein System, welches einen performanten und energiesparenden Algorithmus nutzt, um die Erfassung von Bewegungsmustern mit akzeptabler Genauigkeit bei gleichzeitig vergleichsweise weit auseinanderliegenden GPS-Messungen zu ermöglichen, kann somit sehr nützlich sein.

1.1 Zielsetzung der Arbeit

Ziel dieser Arbeit ist die Entwicklung einer vollumfänglichen Applikation für das Mobilgerätebetriebssystem *Android*, welche die kontinuierliche Aufzeichnung der Position des Gerätes per GPS ermöglicht und aus den gesammelten Positionsdaten einen Bewegungspfad auf einem Wegenetz ermittelt. Die Applikation soll außerdem grundlegende Funktionalitäten zum Auslesen und Anzeigen der gespeicherten Daten durch den Nutzer besitzen.

Es soll untersucht werden, inwiefern die Aufzeichnung von Positionen und der Prozess des Map-Matching energiesparend auf einem Mobilgerät ausgeführt werden kann. Relevante Faktoren sind hierbei die benötigte Prozessorleistung und die damit in Verbindung stehende Akkulaufzeit, sowie der Speicherplatz- und Arbeitsspeicherverbrauch zur Ausführung des Algorithmus.

1.2 Verwandte Arbeiten

Für Mobilgeräte existieren eine Vielzahl an GPS-Tracker-Apps, welche die kontinuierliche Aufzeichnung von GPS-Positionen erlauben. Auch viele Fitness-Tracker-Geräte bieten heutzutage eine Aufzeichnung der zurückgelegten Strecke per GPS an. Jedoch liegt das Hauptaugenmerk bei diesen Apps und Geräten weniger bei einer Zuordnung des GPS-Pfades zu einem Pfad in einem Wegenetz und ein Map-Matcher ist in der Regel nicht vorhanden.

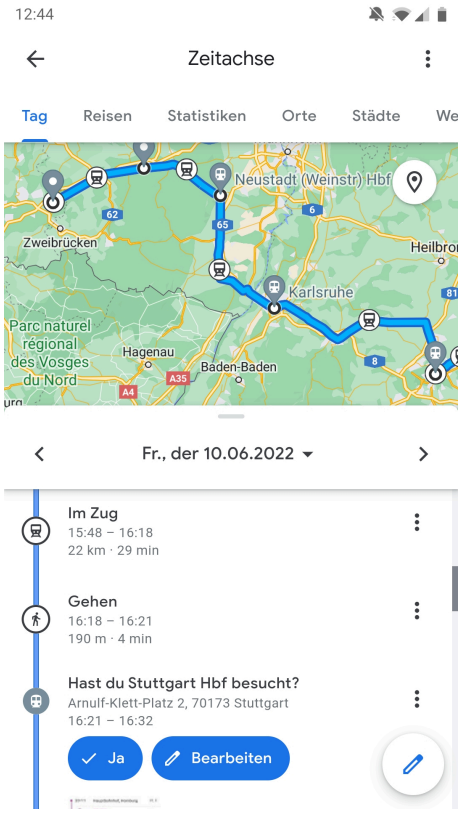
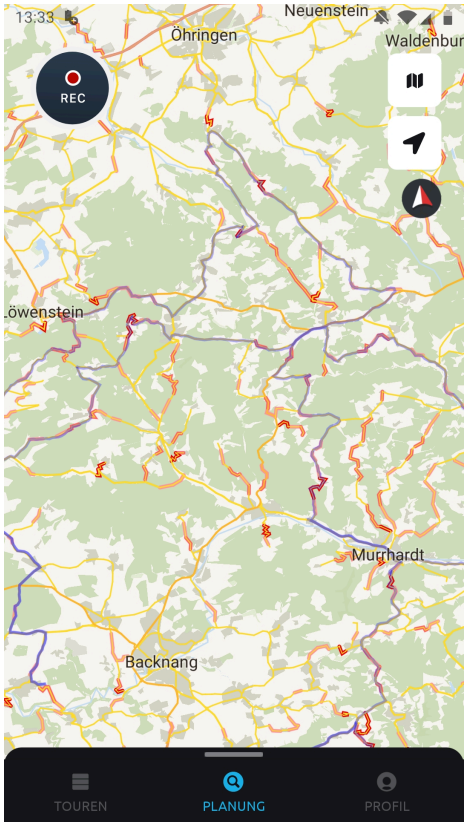


Abbildung 1.1: Bildschirmausschnitt der Google Zeitachse mit der Aufzeichnung einer Bahnreise.

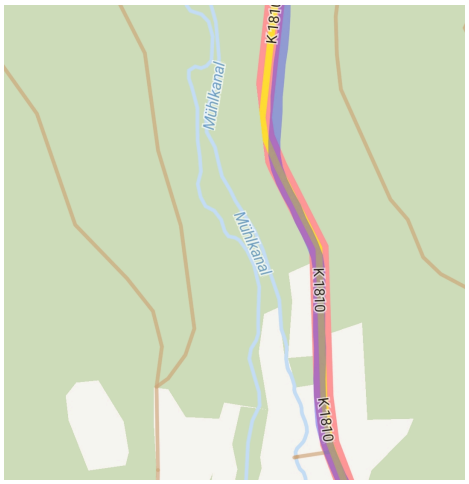
Google Maps[6] bietet auf Mobilgeräten die Funktion *Zeitachse* an. Diese erlaubt dem Nutzer das Einsehen der durch die Standortdienste des Gerätes kontinuierlich im Hintergrund aufgezeichneten Positionsdaten.

Während es sich um eine closed-source Applikation handelt und die genauen Algorithmen somit nicht einsehbar sind, deuten einige Funktionalitäten der *Zeitachse* darauf hin, dass zumindest in Teilen eine Art Map-Matcher oder Activity Detector verwendet wird, welcher jedoch höchstwahrscheinlich nicht auf dem Gerät, sondern auf Google-Servern ausgeführt wird.

So kann die App zwischen verschiedenen Verkehrsmitteln unterscheiden, mit denen eine Strecke zurückgelegt wurde. Zum Beispiel wird zwischen einer Autofahrt, einer Fortbewegung zu Fuß oder per Bahn automatisch unterschieden.
Besonders interessant ist hierbei die Tatsache, dass bei Bahnreisen die exakten Züge und Umstiege ermittelt werden, die für die Reise genutzt wurden.
Trotz alledem werden in der Kartenansicht der *Zeitachse* weiterhin nur GPS-Positionen angezeigt, die durch Linien verbunden werden. Da die Standortdienste des Geräts im Hintergrund nur sehr selten GPS-Positionen aufzeichnen, ergeben sich so Linien, die dem Verlauf von Straßen oder Bahnstrecken nicht sehr akkurat folgen.



(a) Übersicht der Nutzereroberfläche von *calimoto* mit den aufgezeichneten Fahrten in Blau.



(b) Nahaufnahme einer aufgezeichneten Fahrt. Die Abweichung vom tatsächlichen Verlauf der Straße ist deutlich zu erkennen.

Abbildung 1.2: Bildschirmaufnahmen der App *calimoto*.

calimoto[1] ist eine App zur Routenplanung und Aufzeichnung von Vergnügungsfahrten mit dem Motorrad oder Auto. Die zurückgelegte Strecke wird zwar aufgezeichnet, es ist jedoch deutlich zu erkennen, dass die Genauigkeit der Aufzeichnung nicht durch Map-Matching verbessert wird.

Bei *Strava* handelt es sich um eine App mit angeschlossenem Sozialen Netzwerk, mit welcher Sportaktivitäten wie Laufen, Radfahren und andere aufgezeichnet werden können. Auch diese App könnte von Map-Matching profitieren, verwendet es aber nicht.

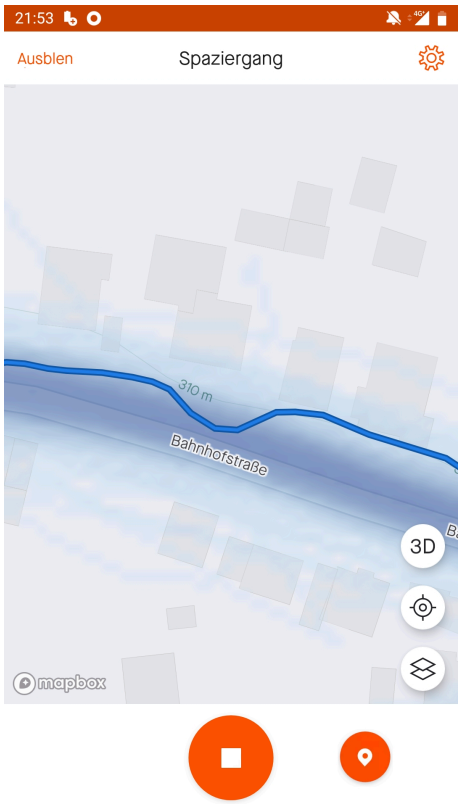


Abbildung 1.3: Aufzeichnung einer Aktivität in *Strava*. Die Abweichung des Tracks (blaue Linie) vom Verlauf der Straße ist deutlich erkennbar.

1.3 Aufbau

Zu Beginn der Arbeit werden in Kapitel 2 die theoretischen Grundlagen und Konzepte betrachtet, die für die Implementierung der gewünschten Funktionalitäten von integraler Bedeutung sind. Die in der Arbeit verwendeten Algorithmen werden ausführlich erläutert.

In Kapitel 3 wird auf die Anforderungen und Funktionen eingegangen, die das Softwareprodukt nach Abschluss der Arbeit aufweisen soll.

Um die geforderten Ziele, insbesondere im Hinblick auf die Performance der Applikation, zu erreichen, sind einige Anpassungen und Optimierungen der zuvor vorgestellten grundlegenden Algorithmen nötig. In Kapitel 4 werden diese Problemstellen genauer beleuchtet und Lösungen für diese aufgezeigt.

Die konkrete Umsetzung und Implementierung der geforderten Funktionen wird in Kapitel 5 präsentiert.

Eine Performanceanalyse und Evaluation von Speicherplatz- und Arbeitsspeicherverbrauch findet in Kapitel 6 statt.

Zum Schluss werden in Kapitel 7 die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf zukünftige in diesem Themengebiet anzusiedelnde Arbeiten gegeben.

2 Grundlagen

2.1 Das OpenStreetMap-Projekt

OpenStreetMap ist ein Open-Data-Projekt, welches sich das Ziel gesetzt hat, eine weltumspannende Datenbank aus Kartendaten zu erstellen, die durch ehrenamtliche Nutzer gesammelt werden.[8]

Eine Verwendung der Geodaten stellt hierbei der ebenfalls durch das Projekt betriebene Online-Kartendienst, der gleichermaßen den Namen *OpenStreetMap* verwendet, dar.

Die Geodaten des Projekts unterliegen hierbei der *Open Data Commons Open Database Lizenz* (ODbL).[7]

2.2 Graphen

Um auf einem Wegenetz Berechnungen durchführen zu können, ist es nötig, dieses Konstrukt der realen Welt auf eine Struktur der Mathematik abzubilden. Hierfür eignet sich die Struktur des Graphen aus der namensgebenden Graphentheorie.

Grundlegend besteht ein Graph $G = (V, E)$ aus einer Menge an Knoten V (engl. nodes/vertices) und einer Menge an Kanten E (engl. edges). Jede Kante $e \in E$ ist hierbei eine zweielementige Teilmenge von V: $e = \{v_1, v_2\}$ mit $v_1, v_2 \in V$. Somit verbindet e v_1 und v_2 ohne Festlegung einer Richtung. Ein solcher Graph wird auch als ungerichtet bezeichnet.

In einem gerichteten Graphen ist jedes $e = (v_1, v_2)$ hingegen ein 2-Tupel. Somit ist klar festgelegt, welcher Knoten der Quell- und welcher der Zielknoten der Kante ist.

Ein gewichteter Graph beinhaltet zusätzlich eine Gewichtungsfunktion $f : E \to \mathbb{R}$. Jeder Kante wird somit ein Gewicht zugeordnet, was beispielsweise Distanzen oder andere Kosten für den Weg zwischen den beiden verbundenen Knoten repräsentieren kann.

In unserem Fall sind Knoten Punkte auf den Straßen, mit denen mindestens ein Straßensegment verbunden ist. Somit trägt jeder Knoten unseres Graphen noch die zusätzliche Information der geographischen Breite und Länge des Punktes in der echten Welt.

Kanten sind hierbei Straßensegmente zwischen den Punkten auf den Straßen. Es wird für die korrekte Repräsentation eines Straßennetzes ein gerichteter Graph benötigt, um Einbahnstraßen adäquat abbilden zu können. Für eine in beide Richtungen befahrbare Straße werden somit zwei Kanten benötigt.

Das Kantengewicht repräsentiert in unserem Fall den geschätzten Zeitaufwand, der benötigt wird, um die Kante beziehungsweise das Straßensegment zu traversieren. Es setzt sich aus der geographischen Länge der Kante und einer für diesen Straßentyp (Wohngebiet, Landstraße, Autobahn etc.) angesetzten Reisegeschwindigkeit zusammen.

2.3 Kürzester-Pfad-Algorithmus nach Dijkstra

Ein integraler Bestandteil des Map-Matching-Algorithmus, auf den die Applikation zurückgreift, ist das Finden von kürzesten Wegen zwischen Knoten im Graphen. Zur Berechnung dieser kann Dijkstras Algorithmus[3] genutzt werden.

Dijkstras Algorithmus ermittelt auf der Eingabe eines gewichteten (gerichteten) Graphens $G = (V, E, f)$ und eines Startknotens $v_{start} \in V$ den kürzesten Pfad von v_{start} zu einem, mehreren oder allen Knoten in G. Für jeden betrachteten Knoten v werden hierbei $d(v)$, die kürzeste Distanz des Knotens v zum Startknoten v_{start}, und $p(v)$, der Vorgängerknoten auf dem kürzesten Weg von v_{start} zu v, bestimmt. Der Distanzwert und Vorgängerknoten eines Knotens werden während der Ausführung des Algorithmus kontinuierlich aktualisiert. Erst wenn ein Knoten Teil der Menge abgearbeiteter Knoten U wird, sind diese Werte für den Knoten final.

1. Initialisiere Distanzwerte für alle Knoten: $\forall v \in V \setminus \{v_{start}\} : d(v) = \infty$ und $d(v_{start}) = 0$
2. Initialisiere Menge abgearbeiteter Knoten $U = \varnothing$ und Menge aktuell betrachteter Knoten $W = \{v_{start}\}$
3. Wiederhole solange $|W| > 0$:
   (a) Entnehme aus W ein Element v, für welches gilt $\forall v' \in W : d(v) \leq d(v')$ (v ist das Element mit kürzester Distanz in V)
   (b) Füge v in U ein: $U = U \cup \{v\}$
   (c) Für jede von v ausgehende Kante $e_i = (v, v_i) \in E$ mit Kantenlänge $l = f(e_i)$:
      (i) Falls $v_i \in W$: Wenn $d(v)+l < d(v_i)$ gilt, dann setze $d(v_i) = d(v)+l$ und $p(v_i) = v$
      (ii) Falls $v_i \notin W \wedge v_i \notin U$: Setze $W = W \cup v_i$ und $d(v_i) = d(v) + l$ und $p(v_i) = v$

Wenn der Algorithmus terminiert, wurden kürzeste Pfade und deren Länge zu allen von v_{start} erreichbaren Knoten berechnet. Wenn nur der Pfad zu einem oder mehreren bestimmten Knoten V_{target} berechnet werden soll, so kann der Algorithmus verfrüht terminiert werden, sobald $V_{target} \subseteq U$ gilt.

Der für die Performance des Algorithmus ausschlaggebende Teil der Implementierung ist hierbei Schritt 3(a). Bei einer naive Implementierung der Datenstruktur müsste die ganze Liste durchsucht werden, um den Knoten mit kürzester Distanz zu finden. Aus diesem Grund ist die Verwendung einer Vorrangwarteschleifen-Datenstruktur sinnvoll, da diese für derartige Retrieval-Operationen optimiert sind.

2.3.1 Modifizierter Dijkstra-Algorithmus

Für die Verwendung im Map-Matching-Algorithmus wird ein leicht modifizierter Dijkstra-Algorithmus benötigt, welcher zwei zusätzliche Funktionen bietet.

Einerseits erlaubt er es, einen kürzesten Pfad von mehreren möglichen Startknoten V_{start} zu bestimmen. Ein kürzester Pfad zu einem Knoten hat den Knoten aus V_{start} als ersten Knoten, von dem aus der kürzeste Pfad erreichbar ist.

Zweitens können im modifizierten Dijkstra-Algorithmus die Startknoten bereits eine initiale Distanz aufweisen. Die Länge aller von einem Startknoten ausgehenden Pfade wird um diese initiale Distanz erhöht. Hierzu wird für jeden Startknoten $v \in V_{start}$ im Gegensatz zum regulären Algorithmus in Schritt 1 eine Distanz $d(v) \geq 0$ zugewiesen.

Konzeptionell kann man sich diese Modifikation so vorstellen, dass ein "virtueller" Startknoten v_v in den Graphen eingefügt wird, von dem folgende Kanten ausgehen: $\forall v \in V_{start} : e = (v_v, v)$ mit $f(e) = d(v)$.

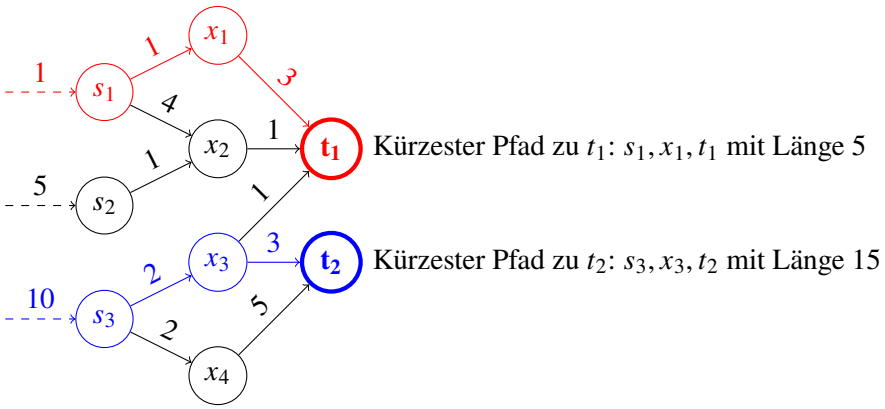
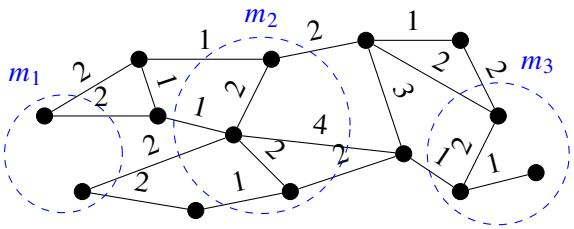
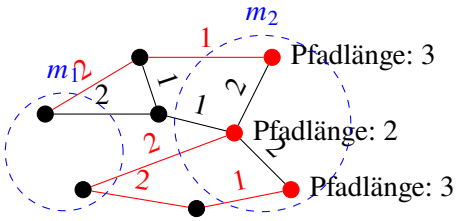


Abbildung 2.1: Beispielhafter Graph mit Startknoten $s_1, ..., s_3$ mit initialen Distanzwerten (gestrichelte Kanten) und Zielknoten t_1, t_2. Die jeweils kürzesten Pfade zu den Zielknoten sind farbig hervorgehoben.

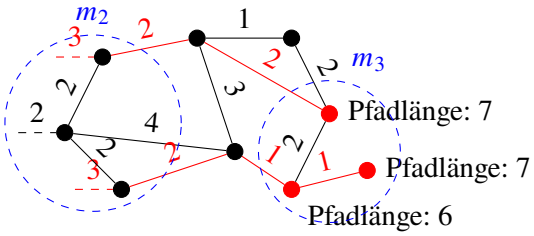
2.4 Map-Matching



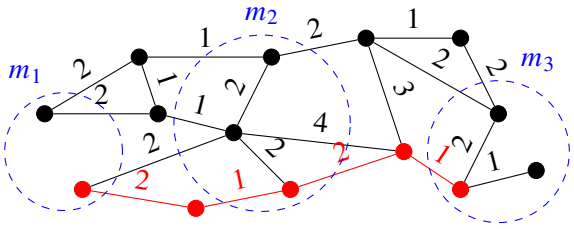
(a) Der Eingabegraph mit den drei Positionsmessungen.



(b) Erster Schritt: Kürzester-Pfad-Berechnung zu allen Knoten in m_2 mit Knoten von m_1 als Startknoten.



(c) Zweiter Schritt: Kürzester-Pfad-Berechnung zu allen Knoten in m_3 mit Knoten von m_2 als Startknoten. Initiale Distanzwerte aus dem letzten Schritt sind als gestrichelte Kanten an den Startknoten dargestellt.



(d) Letzter Schritt: Von den drei Knoten in m_3 wird derjenige ausgewählt, dessen Pfad die kürzeste Länge hat.

Abbildung 2.2: Map-Matching-Prozess am Beispiel eines einfachen Graphen und drei Positionsmessungen. Zur Vereinfachung werden in diesem Beispiel ungerichtete Kanten verwendet.

Map Matching ist der Prozess, aus einer Folge von Positionsmessungen $M = m_1, ..., m_s$, wobei jede Messung m_i einen Kreis mit einem Mittelpunkt c_i und einem Fehlerradius r_i darstellt, einen traversierten Pfad auf einem Wegenetz, repräsentiert als Graph $G = (V, E, f)$, zu rekonstruieren.

Ein möglicher performanter Algorithmus hierfür wird von Eisner et al. präsentiert.[4] Dieser rekonstruiert einen plausiblen Pfad basierend auf der Beobachtung, dass Nutzer sich im Regelfall für den schnellsten Pfad zwischen zwei Punkten entscheiden. Folglich kann man das Problem in mehrere einfach zu lösende Wegfindungsprobleme aufspalten, die separat für alle jeweils aufeinanderfolgenden Messungen m_i und m_{i+1} betrachtet werden können.

Der Algorithmus läuft folgendermaßen ab:

1. Für jede Messung $m_i \in M$:
   (a) Bestimme alle Knoten von G, die innerhalb des Kreises von m_i liegen. Diese kommen als mögliche Zwischenknoten auf dem Pfad infrage. Sie werden fortan als *Kandidatenknoten von* m_i beziehungsweise K_{m_i} bezeichnet.

(b) Berechne die kürzesten Wege und deren Länge zu allen Kandidatenknoten K_{m_i} mit den Kandidatenknoten $K_{m_{i-1}}$ des vorigen Schritts als Startknoten. Hierfür kann der in 2.3.1 beschriebene modifizierte Dijkstra-Algorithmus verwendet werden. Initialisiere die Distanzwerte für die Startknoten mit den Distanzwerten, die für sie im vorigen Schritt errechnet wurden beziehungsweise mit 0, falls es die erste Positionsmessung ist.

2. Wähle letztlich aus den Kandidatenknoten der letzten Positionsmessung den Knoten aus, dessen berechneter Pfad der kürzeste ist. Der Pfad, der zu diesem Knoten führt, wird vom Map-Matcher als wahrscheinlichster Pfad ausgewählt. Der konkrete Verlauf des Pfades muss hierbei aus den einzelnen Teilpfaden, die aus den Dijkstra-Berechnungen resultieren, zusammengesetzt werden.

3 Anforderungen an die Software

Die Software soll auf Mobilgeräten, welche das Betriebssystem Android in der Version 8 oder höher nutzen, lauffähig sein. Sie soll es dem Nutzer erlauben, GPS-Aufzeichnungen zu tätigen. Anhand der einzelnen Messpunkte einer GPS-Aufzeichnung soll ein wahrscheinlicher Bewegungspfad entlang eines Wegenetzes errechnet werden.

Alle Berechnungen sollen auf dem Mobilgerät ausgeführt werden, sodass die Software auch ohne eine aktive Internetverbindung verwendbar ist. Weiterhin soll eine Funktion bereitgestellt werden, um Kartendaten für die Offline-Verwendung herunterzuladen. Weiterhin soll die Möglichkeit gegeben werden, dass die für das Map-Matching zu verwendenden Graphdaten durch den Nutzer ausgewählt werden können, sodass das Map-Matching auf einem beliebigen Wegenetz durchgeführt werden kann.

Die errechneten Bewegungspfade sollen für den Nutzer sinnvoll aufbereitet dargestellt werden.

3.1 Aufzeichnen von Bewegungen in einem Wegenetz

Die Software soll die Aufzeichnung der Position mittels des in Mobilgeräten integrierten GPS-Empfängers erlauben. Die kontinuierliche Aufzeichnung muss auch bei geschlossener Software und im Ruhezustand des Mobilgeräts sichergestellt sein.

Um die GPS-Messpunkte auf einen Pfad im Wegenetz zu übertragen, soll der in Kapitel 2.4 beschriebene Map-Matching-Algorithmus auf dem Gerät ausgeführt werden. Das mittels der bisherigen Positionsdaten errechnete Ergebnis soll dem Nutzer noch während der Aufzeichnung unmittelbar in einer Kartendarstellung präsentiert werden.

3.2 Speichern und Verwalten von Aufzeichnungen

Die getätigten Aufzeichnungen und berechneten Pfade sollen auf dem Gerät gespeichert werden. Es soll eine Benutzeroberfläche existieren, mit welcher der Benutzer die Aufzeichnungen im Nachhinein betrachten und verwalten kann.

So sollen Basisinformationen über die Aufzeichnung, wie Startzeitpunkt, Länge der Strecke, Anzahl an GPS-Messungen und Anzahl an besuchten Straßensegmenten bereitgestellt werden. Zudem soll der Streckenverlauf der Aufzeichnung auf einer Karte visualisiert werden.

Weiterhin muss es dem Nutzer ermöglicht werden, Aufzeichnungen zu verwalten. Hierzu zählt die Möglichkeit, einer Aufnahme einen Titel zuzuordnen, die Funktionalität, getätigte Aufzeichnungen zu löschen und die Fähigkeit, Aufzeichnungen als Datei zu exportieren und später oder auf einem anderen Gerät wieder zu importieren.

3.3 Verarbeitung gesammelter Daten

Die durch die Aufzeichnungen gesammelten Daten sollen für den Nutzer aufbereitet werden. Durch diese Funktionalitäten soll es dem Nutzer ermöglicht werden, seine Bewegungsmuster besser analysieren zu können.

So soll die Applikation eine Kartenansicht enthalten, in welcher die Pfade von allen getätigten Aufzeichnungen angezeigt werden. Bei einem Heranzoomen auf einen kleineren Kartenausschnitt sollen die einzelnen besuchten Straßensegmente eingefärbt werden, wobei die Farbe des Segments repräsentieren soll, wie oft das entsprechende Straßensegment in allen Aufzeichnungen besucht wurde.

Weiterhin soll eine Listenansicht von Straßensegmenten bereitgestellt werden, welche nach Anzahl von Besuchen durch den Nutzer sortiert ist. Durch einen Klick auf den entsprechenden Listeneintrag soll die Kartenansicht, zentriert auf das entsprechende Straßensegment, geöffnet werden.

4 Herausforderungen und algorithmische Lösungen

4.1 Beschleunigung geographischer Abfragen

Bevor der Map-Matching-Algorithmus ausgeführt werden kann, müssen alle Knoten im Graphen des Straßennetzes gefunden werden, die als Kandidaten für einen GPS-Fix infrage kommen. Ein Knoten ist ein Kandidat, wenn er im Fehlerradius um den Punkt der GPS-Messung liegt. Ohne jegliche Optimierungen muss hierzu über jeden Knoten des Graphen iteriert werden, um zu prüfen, ob die Koordinaten des Knotens innerhalb des gewünschten geographischen Bereichs liegen. Insbesondere bei großen Graphen hat diese Operation eine nicht zu vernachlässigende Laufzeit.

Es existieren verschiedene Datenstrukturen, die solche räumlichen Abfragen optimieren können. Eine simple Datenstruktur, die jedoch für dieses Projekt vollkommen ausreichend ist, ist ein Gitter mit festen Zellgrößen. Hierbei wird das gesamte geographische Gebiet, das der Graph abdeckt, in Gitterzellen mit einer zuvor festgelegten Größe unterteilt. Für jede Gitterzelle wird eine Liste angelegt, welche alle Knoten enthält, deren geographische Position in der entsprechenden Gitterzelle liegen. Um eine geographische Abfrage durchzuführen, müssen dann nur noch die Knoten geprüft werden, welche in Gitterzellen liegen, die das geographische Gebiet der Abfrage überlappen. Hierzu kann einfach über die Knotenlisten der überlappenden Gitterzellen iteriert werden.

4.2 Frühzeitiger Abbruch der Dijkstra-Berechnung

Ein weiteres Problem, welches die Laufzeit des Map-Matching-Algorithmus deutlich erhöhen kann, tritt auf, wenn einer oder mehrere der Kandidatenknoten mit dem restlichen Straßennetz nicht oder nur über einen sehr großen Umweg verbunden sind. Dies tritt zum Beispiel häufig bei Wirtschaftswegen oder bei Straßenbrücken, die über andere Straßen führen, auf.

Im Map-Matching-Algorithmus wird der Dijkstra-Algorithmus verwendet, um den kürzesten Pfad zwischen den Kandidatenknoten des letzten GPS-Fix und den Kandidatenknoten des aktuellen GPS-Fix zu errechnen. Dieser Algorithmus terminiert, wenn ein kürzester Pfad zu allen Zielknoten gefunden wurde. Im oben beschriebenen Fall existiert jedoch kein Pfad zu einem oder mehreren der Zielknoten. Somit wird der Dijkstra-Algorithmus erst dann terminieren, wenn über alle erreichbaren Knoten des Graphen iteriert wurde, weil nur so sichergestellt werden kann, dass ein Zielknoten nicht erreichbar ist. Dies kann bei größeren Graphen viel Zeit in Anspruch nehmen.

Um eine weitere Abbruchbedingung zu schaffen, kann die Information über die verstrichene Zeit zwischen den GPS-Fixes genutzt werden, die in unserem Anwendungsfall zur Verfügung steht. In der Zeitspanne zwischen den Fixes kann sich das Gerät realistischerweise nur höchstens eine

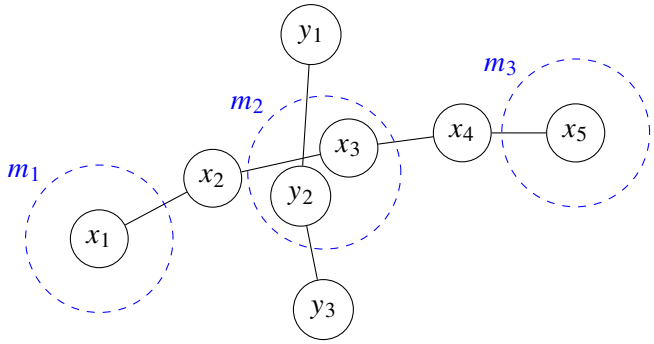


Abbildung 4.1: Beispielhafter Ausschnitt eines Wegenetzes, wie er auftreten kann, wenn ein Weg einen anderen Weg höhenfrei überquert. Die beiden Stränge $x_1, ..., x_5$ und $y_1, ..., y_3$ sind möglicherweise nur über einen großen Umweg oder gar nicht verbunden. In Rot die GPS-Messungen $m_1, ..., m_3$ eines Gerätes mit ihren Fehlerkreisen. Das Gerät bewegt sich entlang der Achse $x_1, ..., x_5$, jedoch liegt y_2 trotzdem in m_2. Bei einer naiven Implementierung des Map-Matching-Algorithmus kann dies zu einer drastisch erhöhten Laufzeit führen, da dieser den gesamten Graphen auf eine mögliche Verbindung von x_1 zu y_2 untersuchen muss.

bestimmte Distanz bewegen. Somit kann die Dijkstra-Berechnung abgebrochen werden, sobald die Distanz des aktuell betrachteten Knoten vom Startknoten diesen Schwellenwert überschreitet, da durch die iterative Vorgehensweise des Dijkstra-Algorithmus, in der immer der Knoten von den verbleibenden als nächstes betrachtet wird, der die geringste Distanz zum Startknoten aufweist. Somit werden alle folgenden Knoten den Schwellwert ebenfalls überschreiten.

4.3 Kanten mit Länge größer als der GPS-Fehlerradius

Da die in diesem Projekt verwendete Variante des Map-Matching-Algorithmus nur überprüft, welche Knoten im Fehlerradius der Positionsmessung liegen, anstatt auch zu prüfen, ob Kanten, deren Knoten außerhalb liegen, den Fehlerkreis schneiden, kann es zu Problemen kommen, wenn eine Kante länger ist, als der Durchmesser des Fehlerkreises der Positionsmessung. Wenn in diesem Fall die beiden Endknoten der Kante außerhalb des Fehlerkreises liegen, die Kante jedoch den Kreis durchquert, so werden die Knoten dieser Kante nicht im Algorithmus beachtet, obwohl diese valide Kandidaten darstellen.

Eine einfache Lösung für dieses Problem stellt der Ansatz dar, Kanten des Graphen mit einer sehr großen Länge in einem Vorverarbeitungsschritt aufzuteilen, sodass die Länge der neue entstandenen Segmente unter einen gewissen Schwellenwert fällt. Wenn nun für ein minimaler Suchradius um GPS-Positionen festgelegt wird, in dem auch dann gesucht wird, wenn die durch das GPS bestimmte Ungenauigkeit eigentlich geringer als dieser minimale Radius ausfällt, dann kann sichergestellt werden, dass immer mindestens ein Knoten der Straße innerhalb des Suchradius liegt.

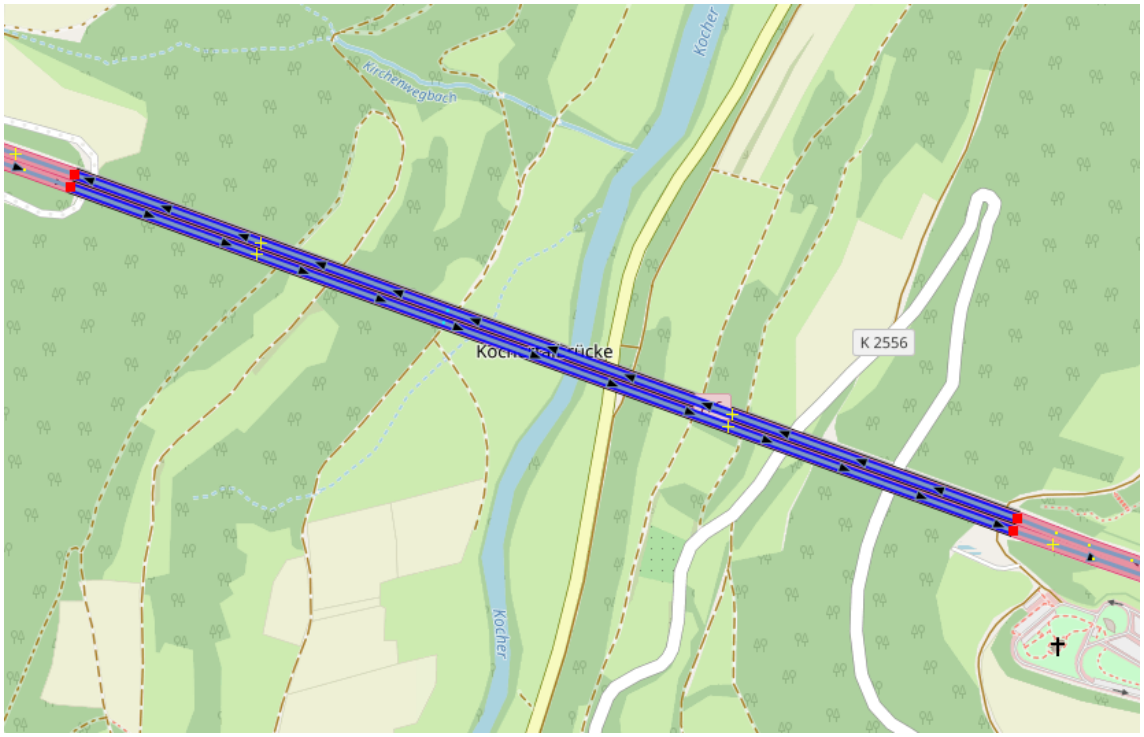


Abbildung 4.2: Die Kochertalbrücke auf der Bundesautobahn 6 (Deutschland, Koordinaten 49,1795225°N 9,7780160°O) ist über einen Kilometer lang, jedoch nur durch 2 Knoten an den Enden (im Bild rot markiert) in den OpenStreetMap-Daten kartiert. Darstellung der OpenStreetMap-Daten in der Software JOSM.

4.4 Level-Of-Detail-System für die Kartendarstellung von Aufzeichnungen

Pfade von Aufzeichnungen bestehen aus vielen Knoten. Insbesondere wenn in der Kartendarstellung mehrere lange Pfade dargestellt werden müssen, kann dies zu Leistungseinbrüchen führen, da das Gerät nur eine begrenzte Anzahl an Linien zeitgleich darstellen kann. Im Allgemeinen sind vor allem auf niedrigen Zoomstufen mehr Pfade gleichzeitig sichtbar, jedoch ist auf niedrigen Zoomstufen gleichzeitig ein voller Detailgrad der Pfade ohnehin nicht nötig.
Weiterhin macht es Sinn, für den Pfad jeder Aufzeichnung ein Hüllrechteck zu berechnen und abzuspeichern. Dieses kann genutzt werden, um Aufzeichnungen, die komplett außerhalb des aktuellen Fensterausschnitts liegen, komplett zu überspringen.

Aus diesem Grund ist ein Level-Of-Detail-System nötig, welches den Detailgrad der Pfade entsprechend der aktuellen Zoomstufe dynamisch anpasst.

Um das Laden von Knoten für einen beliebigen Detailgrad zu ermöglichen, können die Knoten hierbei in einer bestimmten Reihenfolge abgespeichert werden.

Für einen Pfad mit Knoten $v_1, ..., v_n$ können diese in der Reihenfolge $v_1, v_n, v_{n/2}, v_{n/4}, v_{3n/4}, ...$ abgespeichert werden, also sodass mit jedem Schritt diejenigen Knoten der Liste hinzugefügt werden, die sich in der Mitte der Pfade zwischen den bereits in der Liste enthaltenen Knoten befinden.

Knoten:	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9
Speicher-reihenfolge:	v_1	v_9	v_5	v_3	v_7	v_2	v_4	v_6	v_8

Abbildung 4.3: Knoten einer Aufzeichnung und die Reihenfolge, in der sie gespeichert werden, um das Laden eines geringer aufgelösten Pfades zu ermöglichen.

Um nun die Knoten für einen Detailgrad $l \in \mathbb{N}$ zu laden, müssen lediglich die ersten m Knoten aus der Liste geladen werden. Die Anzahl der zu ladenden Knoten für einen Detailgrad l kann mittels der geometrischen Reihe bestimmt werden:

$$m = \sum_{i=0}^{l} 2^i + 2 = \frac{1 - 2^{l+1}}{1 - 2} + 2 = 2^{l+1} + 1 \tag{4.1}$$

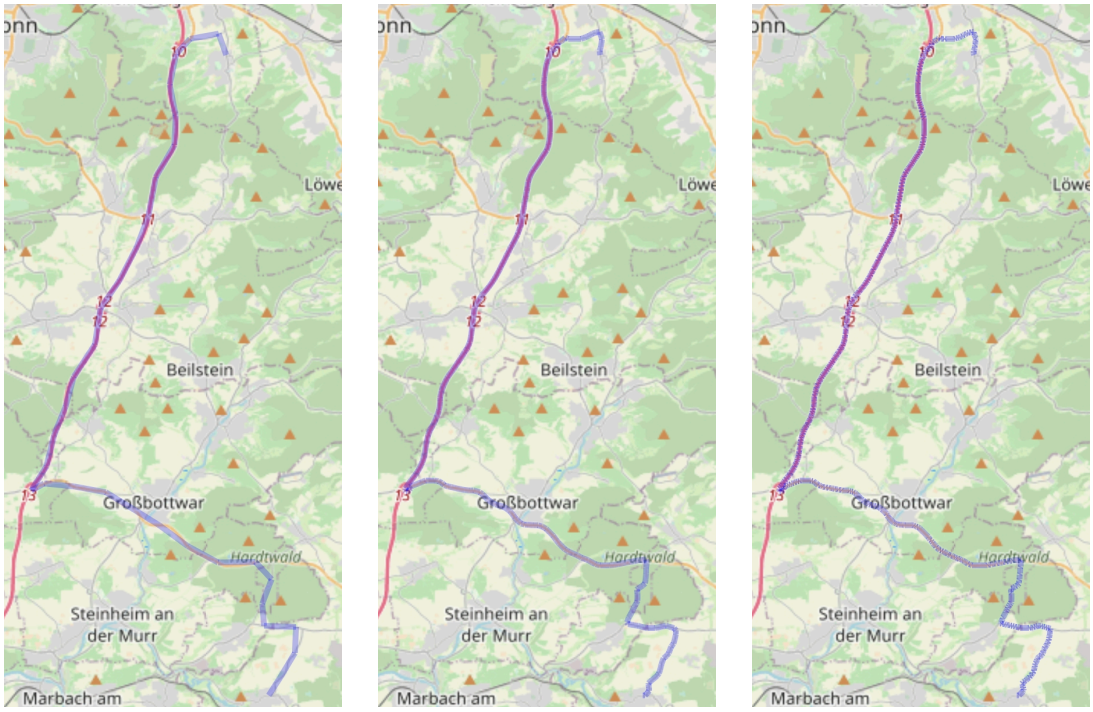
Um die korrekte Anzahl an Knoten zu erhalten, wird auf das Resultat der geometrischen Reihe 2 addiert, um Start- und Endknoten zu beachten.

In der realen Applikation sind die wählbaren Detailgrade l jedoch nicht sehr nützlich, da diese nur angeben, mit wie vielen Knoten ein Pfad repräsentiert werden soll. So generiert $l = 1$ einen Pfad bestehend aus 3 Knoten, $l = 2$ einen Pfad aus 7 Knoten und so weiter.
Verschiedene Aufzeichnungen haben jedoch in der Regel unterschiedliche Längen. Es macht keinen Sinn den Pfad einer Aufzeichnung, welche aus insgesamt 1000 Knoten besteht, mit der gleichen Anzahl an Knoten darzustellen wie der einer Aufzeichnung, welche aus nur insgesamt 200 Knoten besteht.

Aus diesem Grund wird ein normiertes Level-Of-Detail l_n eingeführt, welches die Anzahl an darzustellenden Knoten in Relation zur Gesamtzahl an Knoten der Aufzeichnung angibt. Somit gilt für das normierte Level-Of-Detail $0 < l_n <= 1$. Für $l_n = 1$ würden alle Knoten für die Darstellung genutzt werden, während für $l_n = 0,5$ der gezeichnete Pfad nur aus der Hälfte der Knoten (also jedem zweiten Knoten) bestehen würde.

l kann aus l_n berechnet werden, indem das kleinste $l \in \mathbb{N}$ gewählt wird, für welches $\frac{2^{l+1}-1}{n-2} \geq l_n$ gilt.

l_n wird in Abhängigkeit von der aktuellen Zoomstufe z der Kartenansicht gewählt: $l_n = \frac{2^z}{2^{14}}$. Dieser Wert wurde empirisch bestimmt, um einen guten Kompromiss zwischen Detailtreue und reduziertem Darstellungsaufwand zu finden. Mit dieser Formel würde der Detailgrad bei Zoomstufe 14 volles Detail erreichen, jedoch schaltet die Kartenansicht ohnehin bereits bei Zoomstufe 12 auf den Detailmodus um, welcher ohnehin volle Details darstellt.



(a) $l_n = 0.02$, deutlicher Detailverlust erkennbar.

(b) $l_n = 0.125$, der Standardwert für diese Zoomstufe.

(c) $l_n = 1$, Kanten sind zu kurz für saubere Darstellung.

Abbildung 4.4: Darstellung einer Aufzeichnung auf Zoomstufe 11 mit 3 verschiedenen Detailgraden.

5 Umsetzung

Die Applikation wird in der Programmiersprache Java für das Mobilgerätebetriebssystem *Android* entwickelt.

5.1 Verwendete Bibliotheken

5.1.1 Osmdroid

Osmdroid[9] ist eine Open-Source Bibliothek, die ein Element zur Verwendung in der Benutzerschnittstelle von Android-Applikationen bereitstellt. Mithilfe dieses Elements kann eine Kartenansicht dargestellt werden, mit welcher der Nutzer durch Verschieben des Fensterausschnitts und mittels Zoomaktionen interagieren kann. Es können außerdem weitere Elemente wie Punkte, Linien, Polygone u.a. in der Karte dargestellt werden.

Die Komponentenbibliothek ist unter der Apache Lizenz 2.0 lizenziert.

5.2 Benutzeroberfläche

5.2.1 Übersichtsseite

Über die Übersichtsseite können alle Funktionen der Applikation erreicht werden, wie das Aufzeichnen neuer oder das Betrachten gespeicherter Aufzeichnungen. Die Einstellungen der Applikation finden sich im Optionsmenü in der rechten oberen Ecke. Die Seite verwendet ein Layout, welches in drei Tabs unterteilt ist. Das Optionsmenü erlaubt außerdem den Export aller Aufzeichnungen, den Import von zuvor exportierten *.trk*-Dateien und zu Testzwecken den Import von CSV-Dateien, welche GPS-Daten enthalten. Jede Zeile der CSV-Datei entspricht hierbei einer GPS-Messung. Die Zeilen müssen folgende Struktur aufweisen:

```
latitude;longitude;accuracy_meters;gps_timestamp_ms
```

Liste der Aufzeichnungen

In dieser Listenansicht werden alle auf dem Gerät gespeicherten Aufzeichnungen angezeigt. Für jede Aufzeichnung werden deren Titel, Startzeit der Aufzeichnung, Länge des durch den Map-Matcher bestimmten Pfades der Aufzeichnung und Dateigröße.

Ein Klick auf das entsprechende Listenelement führt den Nutzer zur Detailansicht dieser Aufzeichnung.

Über die Knöpfe rechts kann für jede Aufzeichnung ein Menü geöffnet werden, welches weitere Optionen beinhaltet. So kann die entsprechende Aufzeichnung gelöscht oder auch in eine externe Datei export werden. Der Export erlaubt zwei verschiedene Dateiformate.

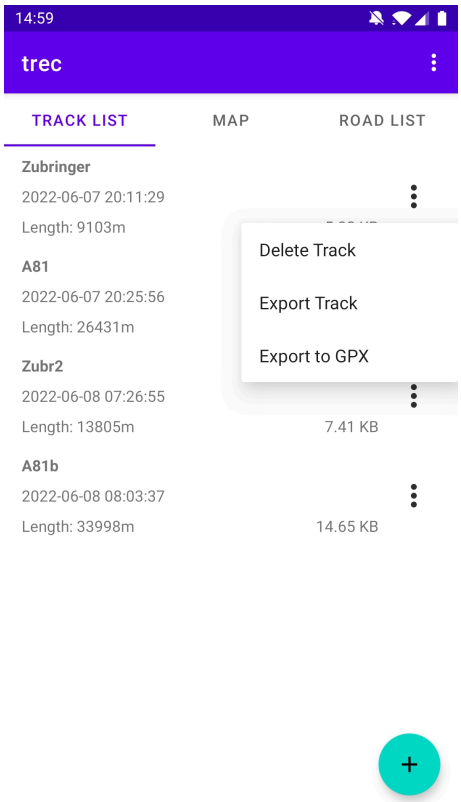


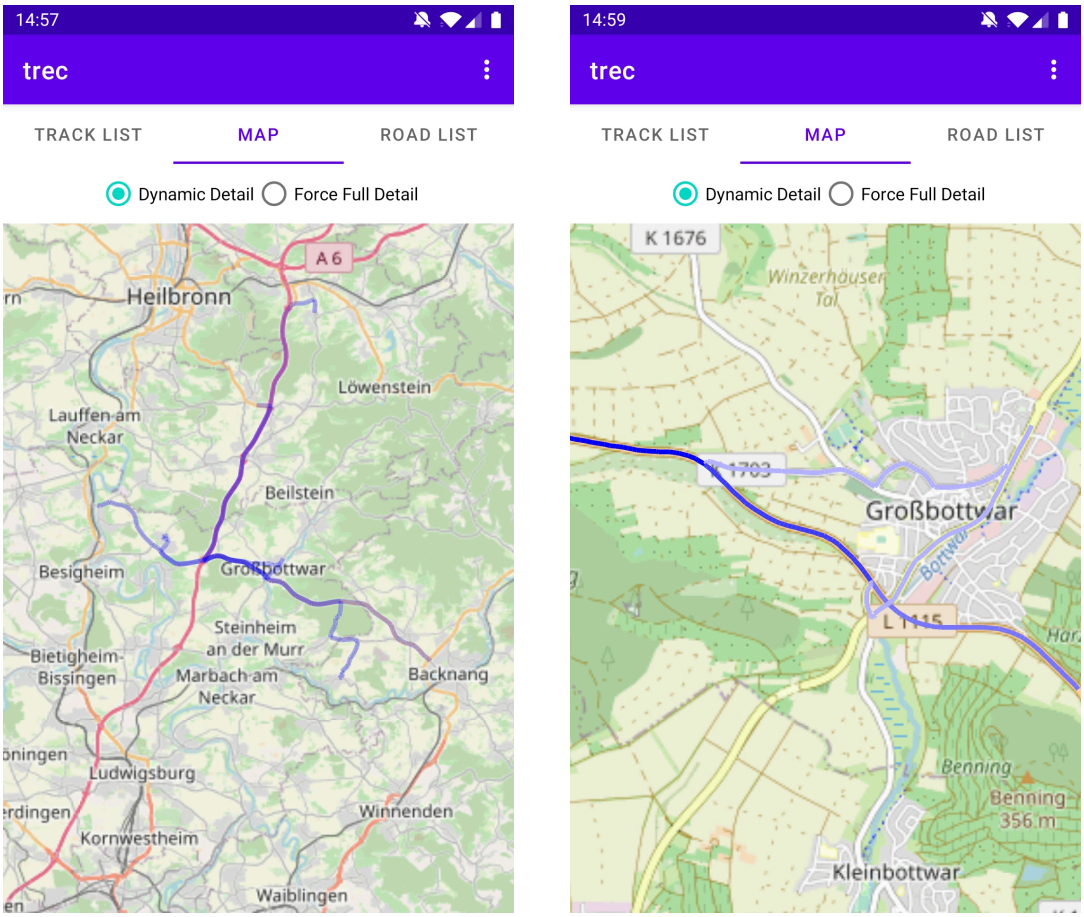
Abbildung 5.1: Listenansicht mit vier auf dem Gerät gespeicherten Aufzeichnungen. Das Optionsmenü der ersten Aufzeichnung ist geöffnet.

Entweder kann die Aufzeichnung direkt im durch die Applikation genutzten *.trk*-Format exportiert werden, welches alle Daten der Aufzeichnung beinhaltet. So kann die Aufzeichnung zu einem späteren Zeitpunkt oder auf einem anderen Gerät ohne Verlust von Daten wieder in die Applikation importiert werden.

Es kann auch ein Export als Datei im *GPX*-Format durchgeführt werden. Hierbei werden die GPS-Messungen als Wegpunkte und der durch den Map-Matcher bestimmte Pfad als Track gespeichert.

Durch das Betätigen des “+”-Knopfes unten rechts kann eine neue Aufzeichnung begonnen werden.

Kartenansicht der Aufzeichnungen



(a) Darstellung von Aufzeichnungen als einzelne Linien im Level-Of-Detail-Modus auf einer niedrigeren Zoomstufe.

(b) Kartenausschnitt im Detail-Modus. Zu erkennen ist die Einfärbung der Straßensegmente basierend auf der Anzahl an Besuchen.

Abbildung 5.2: Kartendarstellung gespeicherter Aufzeichnungen in den zwei Modi.

In dieser Ansicht werden alle auf dem Gerät gespeicherten Aufzeichnungen gemeinsam auf einer Karte angezeigt. Es existieren hierbei zwei verschiedene Modi:

Auf höheren Zoomstufen oder falls der “Force Full Detail”-Modus aktiviert wurde, wird jedes besuchte Straßensegment einzeln gezeichnet. Hierbei wird jedes Segment basierend auf der Anzahl an Besuchen eingefärbt. Die Anzahl an Besuchen für jedes Segment im Straßennetz wird von der Applikation im Voraus berechnet und aktualisiert, falls Aufzeichnungen gespeichert oder gelöscht werden. Um eine akzeptable Performance zu erreichen, sollten jedoch nicht alle besuchten Segmente des Straßennetzes gezeichnet werden, sondern nur die, die im aktuellen Kartenausschnitt sichtbar sind. Um dies zu erreichen, kann auf die in Kapitel 4.1 beschriebene Methode zurückgegriffen

werden, um performant alle Knotenpunkte des Straßennetzes im aktuellen Kartenausschnitt zu erhalten. Dann kann für jeden Knoten über die anliegenden Kanten iteriert werden und diese bei Bedarf, also wenn sie mindestens einmal besucht wurden, gezeichnet werden.

Auf niedrigeren Zoomstufen (unter 12) werden die Pfade der Aufzeichnungen durch den in Kapitel 4.4 beschriebenen Level-Of-Detail-Mechanismus dargestellt. Dieser reduziert dynamisch den Detailgrad der Pfade basierend auf der aktuellen Zoomstufe der Karte. Eine exakte Einfärbung der Straßensegmente basierend auf der Anzahl an Besuchen ist hier nicht möglich, ein ähnlicher Effekt kann aber erzeugt werden, wenn die Linien der einzelnen Aufzeichnungen semitransparent gezeichnet werden. Bei einer Überlappung wirkt die Farbe dann kräftiger. Auch im Level-Of-Detail-Modus findet eine Performance-Optimierung statt. Hier geschieht diese jedoch auf Aufzeichnungs-Ebene statt auf Segment-Ebene, indem das für jede Aufzeichnung gespeicherte Hüllrechteck genutzt wird, um nur jene Aufzeichnungen darzustellen, deren Hüllrechteck den Kartenausschnitt überlappen.

Liste von besuchten Straßensegmente

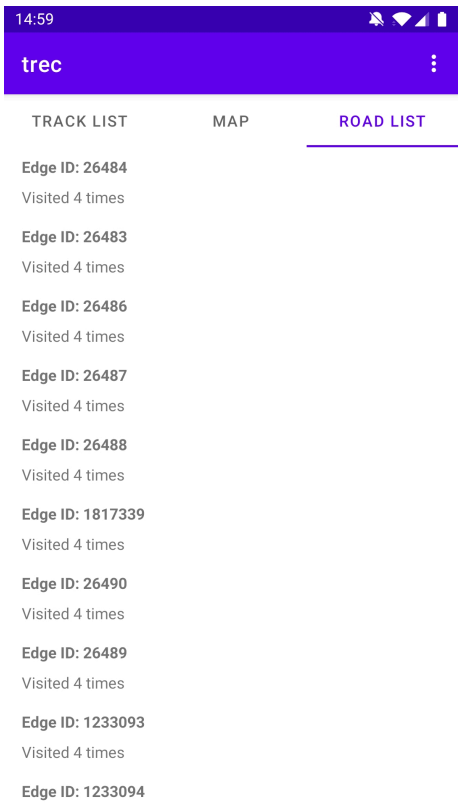


Abbildung 5.3: Die Liste der besuchten Straßensegmente. Sichtbar sind einige Segmente, welche viermal besucht wurden.

Die Liste enthält alle besuchten Straßensegmente. Sie ist in absteigender Reihenfolge nach Anzahl der Besuche sortiert. Durch einen Klick auf ein Listenelement wechselt die Applikation in die Kartenansicht und zeigt das entsprechende Kartensegment.

5.2.2 Einstellungen

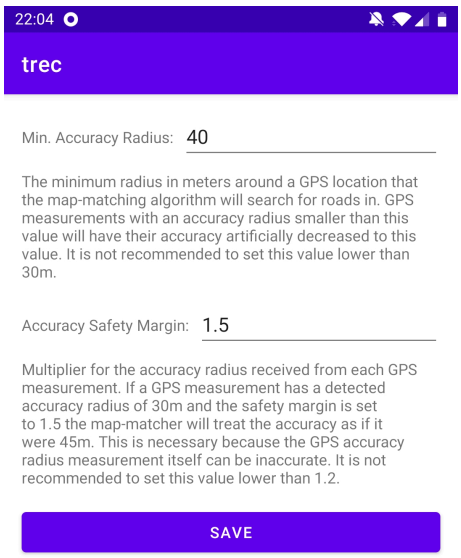


Abbildung 5.4: Die Einstellungsseite der App.

Die Einstellungen der App können über das Optionsmenü auf der Übersichtsseite aufgerufen werden.

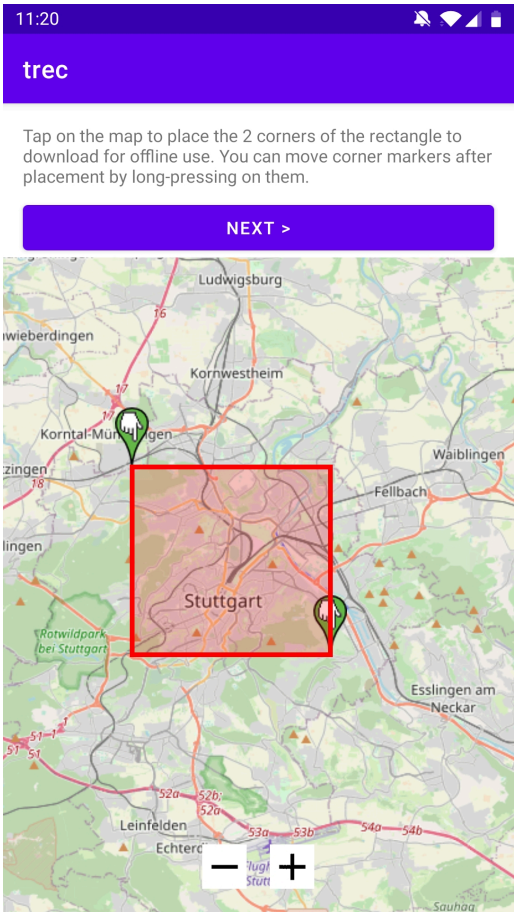
In den Einstellungen können der minimale Suchradius und der Sicherheitsfaktor für den Map-Matching-Algorithmus bestimmt werden. Ein ungenauer GPS-Empfang kann es unter Umständen nötig machen, diese Werte zu vergrößern.

Eine genaue Erklärung, wie die Werte durch den Algorithmus verwendet werden, findet sich im Kapitel 5.5.2.

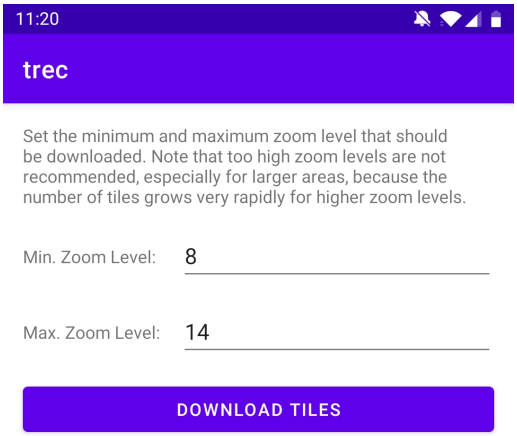
5.2.3 Kartendownload für Offlinenutzung

Mittels des Optionenmenüs auf der Übersichtsseite kann ein Fenster geöffnet werden, welches es dem Nutzer erlaubt, Kartenausschnitte herunterzuladen, sodass diese auch ohne aktive Internetverbindung verfügbar sind.

Der Nutzer kann hierbei in einer Kartenansicht die zwei Eckpunkte des Rechtecks platzieren, welches heruntergeladen werden soll.



(a) Platzierung eines Rechtecks auf der Karte. Die zwei Eckpunkte können per Tippen platziert und durch gedrückt halten verschoben werden.



(b) Wahl der zu herunterladenden Zoomstufen. Der Download kann mit einem Klick auf "Downlod Tiles" gestartet werden.

Abbildung 5.5: Benutzeroberfläche zum Herunterladen von Kartendaten.

Nachdem ein Rechteck selektiert wurde, kann der Nutzer zur nächsten Seite fortfahren. Auf dieser kann ausgewählt werden, welche Zoomstufen heruntergeladen werden soll, bevor der Download gestartet wird. Diese Möglichkeit der Regulierung ist wichtig, da die Datenmenge bei höheren Zoomstufen rapide wächst. Somit erlaubt die Downloadfunktion dem Nutzer die Abwägung zwischen einem kleinen Gebiet mit einer hohen Auflösung oder einem größeren Gebiet mit einer niedrigeren Auflösung.

Für den eigentlichen Download des selektierten Gebiets wird der integrierte CacheManager von *Osmdroid* verwendet. Dieser erlaubt es, ausgewählte Bereiche in den vereinten Cache aller Kartendarstellungen in der Applikation zu laden.

5.2.4 Verwaltung von Graphdaten

Um dem Nutzer zu erlauben, Datensätze für beliebige Wegenetze zu verwenden, kann über das Optionsmenü auf der Übersichtsseite die Verwaltungsoberfläche für Graphdaten aufgerufen werden.

In dieser können neue Graphen aus *.bfmi*-Dateien installiert werden. Die Daten der Datei werden hierbei in das Datenverzeichnis der App kopiert, die originale Datei kann daraufhin gelöscht werden.

In der Übersicht der Graphdatenverwaltung werden alle installierten Graphdatensätze angezeigt. Hierzu zählt auch der mit der Applikation mitgelieferte Datensatz. Der Nutzer kann durch eine Berührung den aktiven Datensatz wechseln, welcher durch die Applikation genutzt werden soll. Für die Pfadberechnungen neuer Aufzeichnungen wird immer der aktive Datensatz genutzt. Weiterhin kann die Detailansicht für eine gespeicherte Aufzeichnung nur aufgerufen werden, wenn diese Aufzeichnung auf dem aktuell aktiven Datensatz ausgeführt wurde.

Graphen können deinstalliert werden, indem der entsprechende Menüpunkt im Optionsmenü rechts angewählt wird.

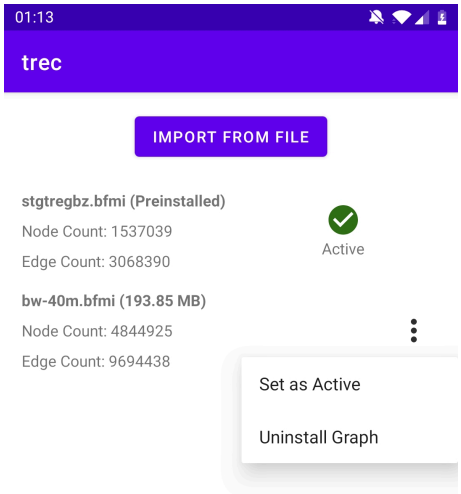


Abbildung 5.6: Die Verwaltungsoberfläche für Graphdaten. Sichtbar sind der mit der App mitgelieferte Graph und ein weiterer vom Nutzer installierter Graph.

5.2.5 Track-Aufzeichnung

Nachdem der “Neue Aufzeichnung”-Knopf betätigt wurde, öffnet sich eine neue Ansicht, in welcher einige Basiseinstellungen der zu tätigenden Aufzeichnung festgelegt werden müssen.

In dieser Ansicht muss vom Nutzer ein Name gewählt werden, durch den die Aufzeichnung später identifiziert werden kann. Außerdem kann die Frequenz der GPS-Messungen gewählt werden. Eine höhere Frequenz resultiert in genaueren Aufzeichnungen, verbraucht jedoch auch deutlich mehr Strom.

Nach Eingabe der Einstellungen kann durch den Knopf in der rechten unteren Ecke die eigentliche Aufzeichnung gestartet werden.

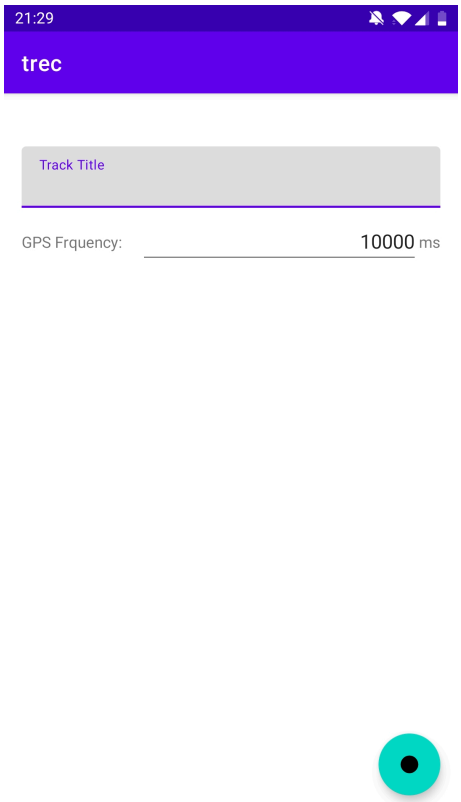


Abbildung 5.7: Einstellungen, die vor Beginn einer Aufzeichnung bestimmt werden müssen.

Die Kartenansicht, welche während der Aufzeichnung angezeigt wird, ist äquivalent zur in Grafik 5.8b dargestellten Ansicht. Der einzige Unterschied beläuft sich darauf, dass die Kartenansicht der laufenden Aufzeichnung automatisch aktualisiert wird, wenn ein neuer Datenpunkt aufgezeichnet wurde.

5.2.6 Detailansicht einer Aufzeichnung

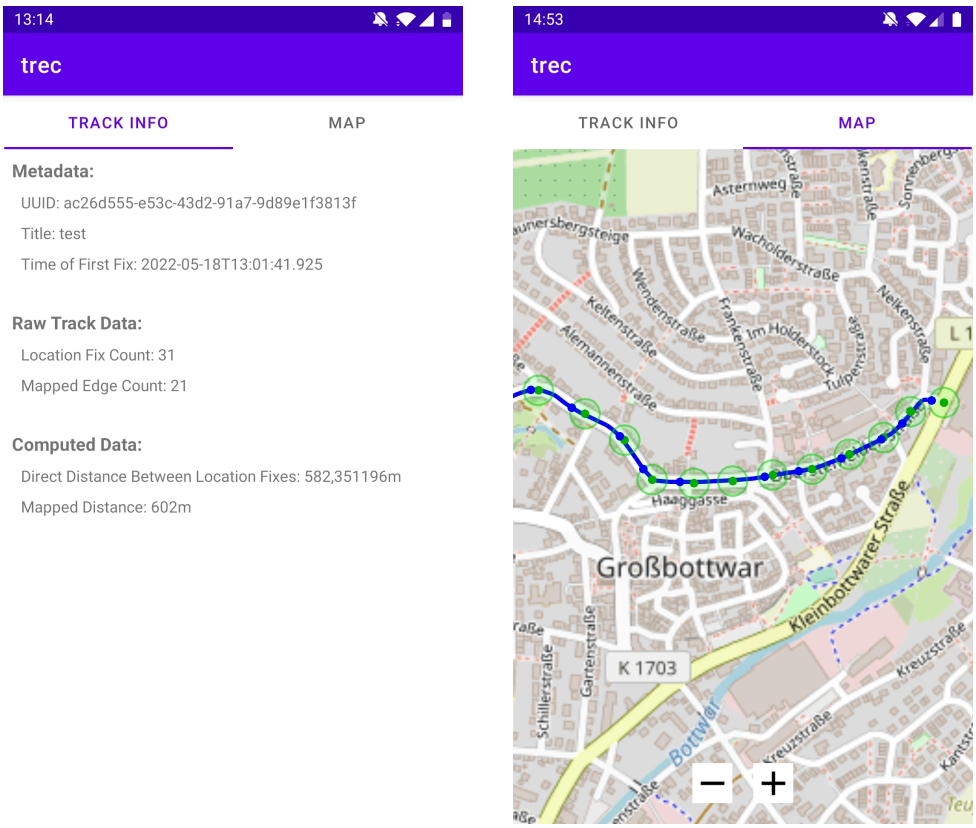
Diese Ansicht erlaubt es dem Nutzer, eine zuvor getätigte Aufzeichnung zu betrachten. Die Ansicht ist in 2 Tabs aufgeteilt.

Allgemeine Daten

In der Ansicht der allgemeinen Daten werden dem Nutzer Basisinformationen über die Aufzeichnung bereitgestellt. So kann der Nutzer den Namen der Aufzeichnung, die einzigartige Identifikationsnummer dieser und der Start der Aufzeichnung in Erfahrung bringen. Weiterhin wird angezeigt, wie viele GPS-Messungen durchgeführt und wie viele Straßensegmente traversiert wurden. Außerdem wird die Länge des zurückgelegten Weges sowohl mittels dem per Map-Matching berechneten wahrscheinlichsten Pfad als auch über die direkte Distanz zwischen den einzelnen GPS-Messungen berechnet.

Detaillierte Kartenansicht einer Aufzeichnung

Die Aufzeichnung wird in dieser Ansicht auf einer Karte dargestellt. Zu sehen sind hier in Grün die GPS-Messungen der Aufzeichnung, deren Ungenauigkeit als semitransparente grüne Kreise dargestellt werden. Weiterhin ist der vom Map-Matching-Algorithmus bestimmte Pfad, welcher basierend auf den GPS-Messungen berechnet wurde, als blaue Linie sichtbar. Die blauen Punkte stellen für jede GPS-Messung den Knoten des Straßennetzes dar, der innerhalb des Fehlerradius liegt und durch den Map-Matcher als Wegpunkt gewählt wurde.



(a) Die allgemeinen Daten einer Aufzeichnung wie sie in der Detailansicht angezeigt werden. **(b)** Detaillierte Kartenansicht einer Aufzeichnung.

Abbildung 5.8: Detailansichten einer Aufzeichnung.

5.3 GPS-Aufzeichnung

5.3.1 GPS-Funktionalität in Android

Mittels der Funktion *requestLocationUpdates* erlaubt Android der Applikation, regelmäßige Positionsaktualisierungen anzufordern. Hierbei können verschiedene Parameter angegeben werden, die bestimmen, unter welchen Umständen eine Aktualisierung erfolgt und welches System zur Positionsbestimmung genutzt werden soll.

Für unsere Zwecke wird GPS als Positionsbestimmungssystem genutzt.

Die Zeit zwischen Aktualisierungen kann vom Nutzer vor Beginn der Aufzeichnung geregelt werden. Eine höhere Frequenz erlaubt hierbei ein präziseres Verfolgen des Bewegungspfades, benötigt gleichzeitig aber auch mehr Energie, was die Batterielebensdauer des Gerätes verringern kann.

Weiterhin kann gewählt werden, welche Distanz von der letzten Aktualisierung mindestens zurückgelegt werden muss, um eine neue Aktualisierung auszulösen. Für unsere Zwecke wird dieser Wert auf 0m gesetzt, da hierdurch in der Aufzeichnung besser nachvollziehbar ist, ob sich das Gerät nur langsam bewegt oder ob es für eine gewisse Zeit stationär war.

Als letzter Parameter muss eine Implementierung des *LocationListener*-Interfaces bereitgestellt werden. Dieses zu implementierende Interface beinhaltet die *onLocationChanged*-Methode, welche durch das Betriebssystem aufgerufen wird, wenn eine neue Aktualisierung zur Verfügung steht.

5.3.2 Android Service

Um die Aufzeichnung von Bewegungen auch mit geschlossener Applikation zu ermöglichen, werden in Android sogenannte *Services* verwendet. Diese erlauben die Ausführung von Programmcode unabhängig von der Hauptapplikation, also auch während diese geschlossen ist. Die Ausführung des Services wird hierbei mit einer Benachrichtigung in der Statusleiste des Betriebssystems signalisiert. Mittels dieser Benachrichtigung können dem Nutzer außerdem Statusinformationen des Services bereitgestellt werden.

Die eigentliche GPS-Aufzeichnung ist mithilfe des Services folgendermaßen umgesetzt: Wenn der Service gestartet wird, was durch die Betätigung der “Aufzeichnen”-Taste in der Applikation geschieht, fordert dieser Positionsaktualisierungen vom Betriebssystem an. Bei jeder neuen Positionsaktualisierung werden die Positionsdaten an eine Map-Matcher-Instanz übergeben, welche mithilfe dieser den nächsten Teil des Pfades berechnen kann.

Andere Applikationsteile können mittels eines *Binders* auf die durch den Algorithmus errechneten Daten zugreifen. Der Binder stellt eine Liste aller bisher aufgezeichneten Positionen und ein *Event* bereit, welches durch andere Programmteile genutzt werden kann, um über Positionsaktualisierungen informiert zu werden.

Wenn der Service gestoppt wird, berechnet dieser noch das Endergebnis des Map-Matching-Algorithmus und speichert dieses als neue Aufzeichnung ab. Weiterhin wird das Beenden der Bereitstellung von Positionsaktualisierungen an das Betriebssystem angefragt.

5.4 Graphen

5.4.1 Ausgangsformat der Graphdaten

Die Graphdaten zur Verwendung in diesem Projekt werden im Format *.fmi* bereitgestellt. Es handelt sich hierbei um ein durch das Institut für Formale Methoden der Informatik der Universität Stuttgart verwendetes Textdateiformat für Graphdaten.[5]

Eine *.fmi*-Datei hat die folgende Struktur, getrennt durch Zeilenumbrüche:

- `Knotenanzahl`
- `Kantenanzahl`
- Für jeden Knoten: `KnotenID OsmKnotenID Latitude Longitude Elevation`
- Für jede Kante: `StartknotenID ZielknotenID Kantengewicht Strassentyp maxGeschwindigkeit`

Die IDs der Knoten sind hierbei aufsteigend und lückenlos, beginnend bei 0. Sie können auch als Index angesehen werden.

5.4.2 Externe Vorverarbeitung

Textdateien besitzen im Allgemeinen eine geringere Informationsdichte als Binärdateien. Hieraus resultiert sowohl ein größerer Speicherplatzbedarf auf dem Gerät als auch eine längere Ladezeit für solche Dateien. Weiterhin müssen zum Beispiel Zahlenwerte, die aus Textdateien eingelesen werden, zuerst validiert und in ihre Binärrepräsentation im Arbeitsspeicher konvertiert werden. Für Binärdateien ist dies nicht notwendig.

Aus diesen Gründen macht das Bereitstellen der Graphdaten in einem Binärdateiformat sehr viel Sinn. Um dies zu bewerkstelligen wird für dieses Projekt ein Kommandozeilenprogramm in der Programmiersprache *Java* entwickelt, welches eine *.fmi*-Datei einließt und die gelesenen Daten in eine *.bfmi*-Binärdatei mit dem in Kapitel 5.4.3 beschriebenen Format schreibt.

Weiterhin erlaubt es das Kommandozeilenprogramm, eine maximale Kantenlänge in Metern zu spezifizieren. Wenn eine Kante diese überschreitet, so wird sie in mehrere Kanten aufgespalten, die dieses Maximum alle unterschreiten. Nötige zusätzliche Knoten werden durch das Programm ebenfalls erstellt. Diese Funktionalität ist nötig, um den in Kapitel 4.3 beschriebenen Lösungsansatz umzusetzen.

Das Kommandozeilenprogramm erwartet folgende Argumente beim Start:

```
<Pfad_zu_Eingabedatei> <Pfad_zu_Ausgabedatei> [max_Kantenlaenge_in_Metern]
```

Argumente in spitzen Klammern sind hierbei verpflichtend, Argumente in eckigen Klammern optional.

Für die Zwecke dieses Projekts wurde eine maximale Kantenlänge von 40m gewählt.

5.4.3 .bfmi-Dateiformat

Es handelt sich um ein für dieses Projekt entwickeltes Binärdateiformat, welches einen Graphen mit Knoten, denen geografische Koordinaten zugeordnet sind, speichern kann.

Alle Datentypen in der Datei werden im Big-Endian-Format gespeichert.

- Integer (4 Bytes): Knotenanzahl
- Integer (4 Bytes): Kantenanzahl
- Für jeden Knoten (je 16 Bytes):
  - Double (8 Bytes): Latitude
  - Double (8 Bytes): Longitude
- Für jede Kante (je 12 Bytes):
  - Integer (4 Bytes): Quellknoten
  - Integer (4 Bytes): Zielknoten
  - Integer (4 Bytes): Kantengewicht

5.4.4 Anordnung im Speicher

Beim Start der Applikation werden die Graphdaten aus der entsprechenden *.bfmi*-Binärdatei in den Arbeitsspeicher des Geräts geladen.

Um eine höhere Performance zu erreichen, wird bei der Repräsentation im Arbeitsspeicher ausschließlich auf die primitiven Datentypen der Programmiersprache Java und Arrays ebenjener zurückgegriffen. Die Daten eines Graphen können folgendermaßen repräsentiert werden:

In zwei Feldern werden die Anzahl von Knoten $|V|$ und die Anzahl von Kanten $|E|$ als Ganzzahlen gespeichert.

Weiterhin beinhaltet das Objekt ein Array aus Ganzzahlen, welches die Daten der Kanten des Graphen hält. Es besitzt $3 * |E|$ Einträge. Für jede Kante hält das Array den Quellknoten, Zielknoten und das Kantengewicht. So können die Daten für einen Knoten mit ID i abgerufen werden, indem das $(3 * i)$-te und die 2 darauffolgenden Elemente ausgelesen werden.

Bei einem weiteren Feld handelt es sich um ein Adjazenzarray. Es besitzt $|V|$ Einträge. Jeder Eintrag des Arrays ist ein weiteres Array und speichert die IDs aller Kanten, die von dem Knoten ausgehen, der denselben Index wie das Array hat.

Für Graphen, deren Knoten geographische Koordinaten besitzen, wird die Struktur dann noch folgendermaßen erweitert:

Zur Speicherung der Latituden bzw. Longituden aller Knoten werden 2 Arrays benötigt, welche jeweils $|V|$ Gleitkommazahlen doppelter Präzision halten.

Gitterstruktur für geographische Daten

Wie in Kapitel 4.1 beschrieben, soll eine Gitterstruktur mit fester Zellengröße genutzt werden, um geographische Abfragen zu beschleunigen.

Um die Knoten des Graphen in dieser Struktur zu speichern, wird zuerst die Ausdehnung des Graphen auf den beiden Achsen bestimmt und gespeichert. Mithilfe dieser Information kann die Anzahl an Zellen bestimmt werden, die der Graph einnimmt. Dies ist abhängig von der Größe einer einzelnen Zelle. Für die Verwendung in der Applikation wurde eine Zellengröße von 0.1° gewählt.

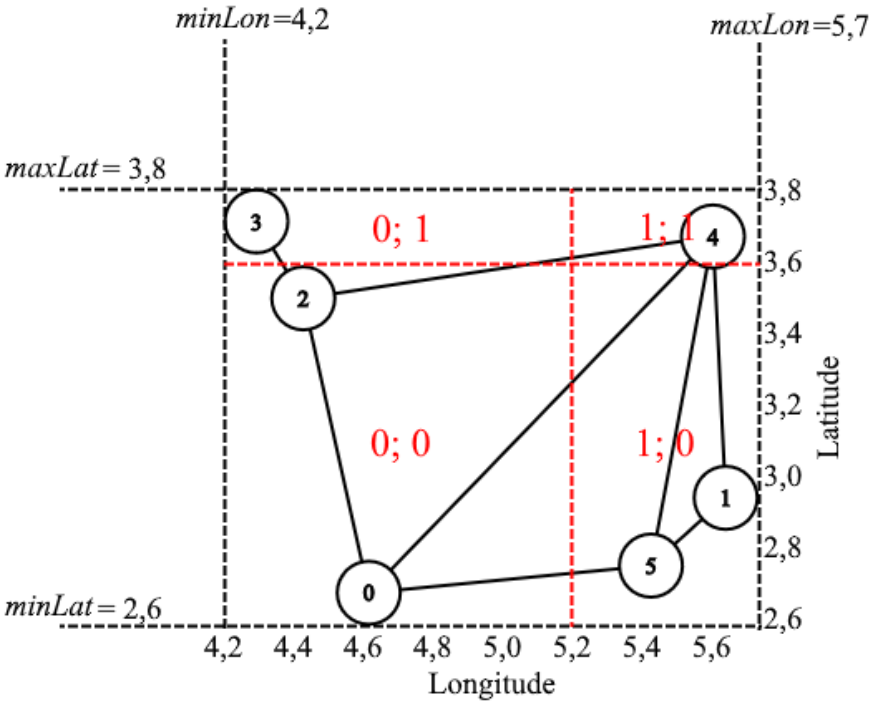


Abbildung 5.9: Beispielhafte Aufteilung eines Graphen in Zellen mit Zellengröße 1.

Daraufhin kann über alle Knoten des Graphen iteriert werden. Es wird hierbei für jede Zelle eine temporäre Liste angelegt, welche Knoten-IDs halten kann. Für jeden Knoten wird bestimmt, in welcher Zelle er liegt und seine ID dann der entsprechenden temporären Liste hinzugefügt.

Nachdem über alle Knoten iteriert wurde, können die temporären Listen zu einem zusammenhängenden Knoten-Array vereinigt werden. Dieses Array sollte dann genau $|V|$ Einträge aufweisen.

Um später den Teil des Arrays zu finden, der die Knoten einer bestimmten Zelle enthält, wird ein weiteres Lookup-Array benötigt, welches einen Eintrag für jede Zelle des Gitters besitzt. Dieser Eintrag entspricht dem Index im Knoten-Array, an dem die Knoten-IDs der entsprechenden Zelle beginnen. Um die Iteration über die Knoten rechtzeitig abzubrechen, kann der darauffolgende Eintrag im Lookup-Array zu Rate gezogen werden. Er signalisiert den Index, ab dem die Knoten-IDs der

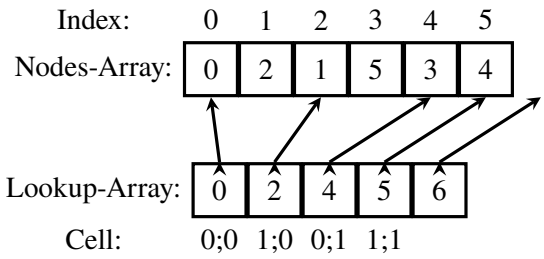


Abbildung 5.10: Knoten-Array und Lookup-Array für die Zellen des Graphen aus Abb. 5.9.

nächsten Zelle folgen. Um sicherzustellen, dass dies auch für die letzte Zelle des Gitters funktioniert, muss am Ende des Lookup-Arrays noch ein zusätzlicher "virtueller" Eintrag hinzugefügt werden, dessen Wert auf den ersten außerhalb des Knoten-Arrays liegenden Index gesetzt wird.

5.5 Map-Matching

5.5.1 Dijkstra-Algorithmus

Zur Implementierung des in Kapitel 2.3 beschriebenen modifizierten Dijkstra-Algorithmus werden zwei Arrays, *distances* und *predecessors* verwendet, deren Größe der Anzahl an Knoten im Graphen entspricht. In diesen Arrays werden für jeden Knoten die errechnete Distanz zum naheliegendsten Startknoten und der Vorgängerknoten auf dem Pfad von diesem Startknoten zum entsprechenden Knoten gespeichert. Diese zwei Arrays bilden nach Terminierung des Algorithmus das Ergebnis der Berechnung.

Die Menge der aktuell betrachteten Knoten W und insbesondere das Entnehmen des Knotens mit der kürzesten Distanz aus ihr muss performant implementiert sein. Hierfür wird die *Java*-Datenstruktur *PriorityQueue* genutzt, welche für genau dieses Nutzungsschema ausgelegt ist. Die Zeitkomplexität des Entnehmens des Elements mit dem kleinsten Schlüssel liegt bei der *Java*-Implementierung der PriorityQueue in $O(log(n))$.

Der implementierte Algorithmus unterscheidet abgearbeitete und nicht abgearbeitete Knoten daran, ob ihr Wert im *distances*-Array ein negatives Vorzeichen hat. Dadurch kann zur Optimierung auf eine explizite Datenstruktur zur Speicherung der Menge abgearbeiteter Knoten U verzichtet werden. Wenn ein Knoten aus W entnommen wird und sein Distanzwert und Vorgänger somit final sind, wird das Vorzeichen des Wertes im *distances*-Array von negativ zu positiv getauscht. Zu beachten ist somit, dass alle mathematischen Berechnungen des Dijkstra-Algorithmus auf das invertierte Vorzeichen angepasst sein müssen.

Eine weitere Eigenheit der Implementierung ist, dass die *PriorityQueue*-Datenstruktur in *Java* kein Verändern, insbesondere kein Verringern, des Schlüssels eines Elements erlaubt. Dies ist aber zwingend erforderlich, da die Distanzwerte von Knoten im Dijkstra-Algorithmus iterativ schrumpfen. Ein einfaches Entfernen und neu hinzufügen des Elements aus der *PriorityQueue* wäre eine mögliche Lösung, die jedoch eine große Leistungseinbuße mit sich bringt, da das Entfernen von Elementen eine Operation mit linearer Zeitkomplexität darstellt. Eine andere simple und performante Lösung ist es, das gleiche Element mit dem aktualisierten, kleineren Schlüssel der

Queue hinzuzufügen, das Element mit dem alten, größeren Schlüssel jedoch nicht zu entfernen. Da sich der Distanzwert eines Elements mit der Entnahme aus der Queue finalisiert und dieses damit zur Menge abgearbeiteter Knoten U zählt, können Elemente mit den veralteten, größeren Schlüsseln, die im späteren Verlauf des Algorithmus aus der Queue entnommen werden, einfach übersprungen werden, sofern der Knoten bereits als abgearbeitet zählt.

5.5.2 Map-Matching-Algorithmus

Zur Implementierung des Map-Matching-Algorithmus werden mehrere Datenstrukturen benötigt, welche Informationen zwischen Iterationen, beziehungsweise, im Fall eines Online-Algorithmus, bis der nächste Datenpunkt zur Verfügung steht, speichern.

In einem Array mit sich in jeder Iteration verändernden Größe werden die IDs der Knoten gespeichert, die als Kandidatenknoten der letzten Positionsmessung ermittelt wurden (also die Knoten, die innerhalb des Fehlerkreises der Messung liegen).
Ein weiteres Array speichert für jeden dieser Knoten die errechnete Länge des kürzesten Pfades zu ebendiesem.
Hinzu kommt eine Liste, die für jeden aktuellen Kandidatenknoten den konkreten kürzesten Pfad speichert. Somit ist jedes Element dieser Liste wiederum ein Array bestehend aus den Knoten-IDs, die traversiert werden müssten, um den Kandidatenknoten auf dem kürzesten Weg zu erreichen.
Als zusätzliche Information, die für die eigentliche Berechnung nicht notwendig ist, werden in einer separaten Liste die Kandidatenknoten der Positionsmessungen gespeichert, die durch den Map-Matcher für den kürzesten Pfad ausgewählt wurden. Diese Information kann später durch den Nutzer auf einer Karte angezeigt werden.
Außerdem wird zur Berechnung des Schwellenwerts zum frühzeitigen Abbruch der Dijkstra-Berechnung der Zeitstempel der letzten Positionsmessung benötigt.

Der Map-Matcher führt dann für jede neue Positionsmessung den in Kapitel 2.4 beschriebenen Algorithmus aus.

Hierzu wird zuerst ein Suchradius um die per GPS bestimmte Position bestimmt. Es werden mithilfe der zuvor beschriebenen Gitterstruktur alle Knoten in diesem Suchradius ermittelt. Diese sind die Kandidaten für die weitere Berechnung. Der Suchradius wird folgendermaßen ermittelt: Sofern die durch das GPS bestimmte Ungenauigkeit multipliziert mit einem Sicherheitsfaktor (Standardmäßig 1,5) geringer als der minimale Suchradius (Standardmäßig 40m) ist, wird der minimale Suchradius verwendet. Ansonsten die GPS-Ungenauigkeit multipliziert mit dem Sicherheitsfaktor verwendet. Der minimale Suchradius ist, wie im Kapitel 4.3 beschrieben, nötig, um sicherzustellen, dass immer mindestens ein Knoten der Straße im Suchradius liegt. Der Sicherheitsfaktor ist nötig, da die durch das GPS bestimmte Ungenauigkeit unter Umständen selbst ungenau sein kann.

Dann wird der Dijkstra-Algorithmus genutzt, um die kürzesten Pfade von den Kandidatenknoten der letzten Positionsmessung zu denen der neuen Positionsmessung zu finden. Diese (Teil-)Pfade und deren Länge werden in den zuvor beschriebenen Variablen gespeichert. Die bereits in den Variablen gespeicherten (Teil-)Pfade aus vorangegangen Iterationen des Algorithmus werden hierzu mit den neuen Pfaden konkateniert.

Wenn alle Positionsmessungen durch den Map-Matcher verarbeitet wurden, kann der finale Pfad ermittelt werden. Hierzu wird aus der letzten Positionsmessung der Kandidatenknoten ermittelt, dessen Pfad der kürzeste ist. Dieser Pfad ist letztlich das Ergebnis des Map-Matching-Prozesses und kann aus der entsprechenden Liste ausgelesen und zurückgegeben werden.

Frühzeitiger Abbruch

Wie in Kapitel 4.2 beschrieben, ist zur Beschleunigung der Berechnung in bestimmten Fällen ein frühzeitiger Abbruch des Dijkstra-Algorithmus vonnöten, wenn die Länge des Pfades zum vom Algorithmus betrachteten Knoten zu den Startknoten einen Schwellenwert übersteigt.

Dieser Schwellenwert wird aus der verstrichenen Zeit Δt zwischen der letzten und der aktuellen Positionsmessung berechnet.

Zu beachten ist, dass die Kantengewichte beziehungsweise Distanzen im Graphen nicht äquivalent zu Distanzen in der realen Welt sind.
Die Kosten einer Kante im für dieses Projekt verwendeten Graphen basieren stattdessen auf der voraussichtlich benötigten Reisezeit der Kante, welche sich wiederum aus der Höchstgeschwindigkeit v_{max}, die auf der entsprechenden Straße gestattet ist, und der Länge der Kante l ergibt:

Kantengewicht $c = \frac{l*360}{v_{max}}$ mit v_{max} in $\frac{\text{km}}{\text{h}}$ und l in m.

Somit kann das maximal zurücklegbare Kantengewicht in einer Zeitspanne folgendermaßen berechnet werden:

$c_{max} = 100 * \Delta t$ mit Δt in s

c_{max} sollte dann noch mit einem Sicherheitsfaktor multipliziert werden. In unserem Fall wählen wir einen Sicherheitsfaktor von 4. Dies mag äußerst großzügig erscheinen, jedoch muss bedacht werden, dass die in den Kartendaten für eine Straße eingetragene Höchstgeschwindigkeit nicht immer korrekt ist und zwischen einer Straße in einem Wohngebiet (30 km/h) und einer Landstraße (100 km/h) bereits ein Faktor von 3.3 liegt. Hinzu kommt noch die Tatsache, dass es nicht ungewöhnlich ist, dass Autofahrer die zugelassene Höchstgeschwindigkeit überschreiten.

Um nun für eine Iteration des Map-Matching-Algorithmus den Schwellenwert zum Abbrechen der Dijkstra-Berechnung zu ermitteln, wird folgendermaßen vorgegangen: In der letzten Iteration wurde für alle Kandidatenknoten der letzten Positionsmessung ein kürzester Pfad ermittelt. Von diesen wird der Knoten ausgewählt, welcher den längsten Pfad besitzt. Die Länge dieses Pfades addiert mit dem zuvor berechneten c_{max} ergibt den neuen Schwellenwert für diese Iteration des Algorithmus.

5.6 Speichern und Laden von Aufzeichnungen

5.6.1 Dateiformat

Aufzeichnungen der Applikation werden im *.trk*-Binärdateiformat gespeichert, welches den folgenden Aufbau hat. Alle Datentypen in der Datei werden im Big-Endian-Format gespeichert.

- UUID (16 Bytes): Einzigartige Identifikationsnummer dieser Aufzeichnung
- String[1]: Name des aktiven Graphen während der Aufzeichnung
- SHA-256 Prüfsumme (32 Bytes): Prüfsumme des aktiven Graphen während der Aufzeichnung
- String[1]: Titel der Aufzeichnung
- Long (8 Bytes): Unix Zeitstempel des Beginns der Aufzeichnung
- Double (8 Bytes): Hüllrechteck nördliche Koordinate
- Double (8 Bytes): Hüllrechteck östliche Koordinate
- Double (8 Bytes): Hüllrechteck südliche Koordinate
- Double (8 Bytes): Hüllrechteck westliche Koordinate
- Integer (4 Bytes): Zurückgelegte Distanz entlang Kanten in Metern
- Integer (4 Bytes): Anzahl GPS-Messungen
- Integer (4 Bytes): Anzahl traversierte Kanten im Graphen
- Level-Of-Detail-Daten zur Darstellung der Aufzeichnung: Variable Größe, siehe 5.6.2
- Für jede GPS-Messung (in Reihenfolge):
  - Integer (4 Bytes): ID des vom Map-Matcher gewählten Knotens innerhalb des GPS-Fehlerradius
- Für jede traversierte Kante (in Reihenfolge):
  - Integer (4 Bytes): ID der Kante
- Für jede GPS-Messung (in Reihenfolge):
  - Double (8 Bytes): Latitude des Mittelpunktes der Messung
  - Double (8 Bytes): Longitude des Mittelpunktes der Messung
  - Float (4 Bytes): Fehlerradius der Messung in Metern
  - Long (8 Bytes): Zeitstempel der Messung

[1] Es handelt sich hierbei um einen Datentyp variabler Länge. Das genaue Format des String-Datentyps kann der Erklärung zur Methode DataOutputStream.writeUTF() in der *Java*-Dokumentation entnommen werden.[2]

5.6.2 Level-Of-Detail-Daten für Aufzeichnungen

Die Level-Of-Detail-Daten bestehen aus einer Liste von Knoten-IDs, die auf dem durch den Map-Matcher bestimmten Pfad der Aufzeichnung liegen. Die IDs werden hierbei in der Liste in einer bestimmten Reihenfolge angeordnet, die es erlaubt, einen beliebigen Detailgrad des Pfades zu laden, indem ein Präfix der Folge der Knoten geladen wird. Für eine genauere Erklärung siehe Kapitel 4.4.

In der *.trk*-Datei wird für die Level-Of-Detail-Daten zuerst die Anzahl an Knoten n, die in der Liste enthalten sind, als Integer (4 Bytes) abgespeichert. Die Anzahl an Knoten in der Level-Of-Detail-Liste kann geringer sein als die Anzahl aller Knoten in der Aufzeichnung, da die Implementierung des Level-Of-Detail-Systems nur für eine Anzahl an Knoten ausgelegt ist, die $2^{l+1} + 1, l \in \mathbb{N}$ entspricht. Wenn der Pfad der Aufzeichnung mehr Knoten hat, werden überschüssige Knoten beim Generieren der Level-Of-Detail-Liste ignoriert.

Daraufhin folgen n Integer (je 4 Bytes), die die Knoten-IDs der Knoten in der Liste enthalten.

6 Evaluation von Laufzeiten und Speichernutzung

Alle Praxistests wurden auf einem Smartphone des Typs *OnePlus 5* aus dem Jahr 2017 unter Verwendung von *Android*-Version 10 durchgeführt. Das Gerät besitzt eine CPU des Typs *Qualcomm Snapdragon 835* und 6 GB Arbeitsspeicher.

6.1 Graph-Daten

Region	Knoten	Kanten	Dateigröße (max. Kantenlänge 40m)	tatsächliche Arbeitsspeichernutzung der App	Ladezeit
Reg.-Bez. Stuttgart	1 132 113	2 292 887	61,4 MB	143 MB	2,1s
Baden-Württemberg	3 600 520	7 300 290	193,8 MB	380 MB	5,7s
Deutschland	25 115 477	50 790 030	1,5 GB	-	-

Tabelle 6.1: Für die Performance-Analyse verwendete Graphdaten und Testergebnisse. Der Graph von Deutschland konnte nicht geladen werden, da seine Größe die Arbeitsspeicherlimits überschreitet. Die Ladezeit ist der Durchschnitt aus drei Messungen.

6.1.1 .bfmi-Dateiformat

Das in Kapitel 5.4.3 beschriebene *.bfmi*-Dateiformat benötigt pro Knoten 16 Bytes und pro Kante 12 Bytes an Speicherplatz.

Vor der in Kapitel 5.4.2 beschriebenen Aufspaltung von zu langen Kanten ergibt sich so für die Graphdaten des Regierungsbezirks Stuttgart eine Dateigröße von ca. 46 MB. Bei einer Aufspaltung von Kanten mit einer Länge von über 40 Metern, so wie sie in der App verwendet wird, wächst die Dateigröße auf ca. 61 MB an. Zum Vergleich wächst die Dateigröße bei einer maximalen Kantenlänge von 30 Metern sogar auf 70 MB, während sie bei einer maximalen Kantenlänge von 50 Metern nur ca. 55 MB beträgt.

Eine *.bfmi*-Datei mit Daten für Baden-Württemberg liegt im Vergleich bei ca. 194 MB und eine Datei mit Daten für ganz Deutschland bei knapp 1,5 GB (beide mit Kantenlängen auf 40m begrenzt).

6.1.2 Arbeitsspeicherverbrauch

Mit der in Kapitel 5.4.4 beschriebenen Datenstruktur erzeugt ein in den Arbeitsspeicher geladener Graph einen theoretischen Speicherverbrauch von in der Größenordnung von ca. 28 Bytes pro Knoten und 16 Bytes pro Kante (wenn nur die reinen Daten betrachtet werden, ohne Berücksichtigung von Overhead verschiedener Quellen).
Für den Graphen des Regierungsbezirks Stuttgart würde dies somit einen theoretischen Arbeitsspeicherverbrauch von ca. 68 MB ergeben.

Bei einer Ausführung unter realen Bedingungen verbraucht die gesamte App im Leerlauf mit geladenem Graphen des Regierungsbezirks Stuttgart 143 MB, mit Graphen von Baden-Württemberg ca. 380 MB.
Das Laden des Graphen von Deutschland war indes aufgrund einer Out-Of-Memory-Exception nicht möglich.

An dieser Stelle sei anzumerken, dass *Android* strenge Limits auf die Speicherallokation von Apps ausübt. Das genaue Limit ist stark intransparent und geräteabhängig, scheint aber in unserem Fall in der Größenordnung von 1 GB zu liegen. Diese Limits können nur umgangen werden, indem speicherintensive Teile der App mithilfe des *Android NDK* in nativem Code verfasst werden, wovon jedoch für dieses Projekt abgesehen wurde.

6.1.3 Ladezeit

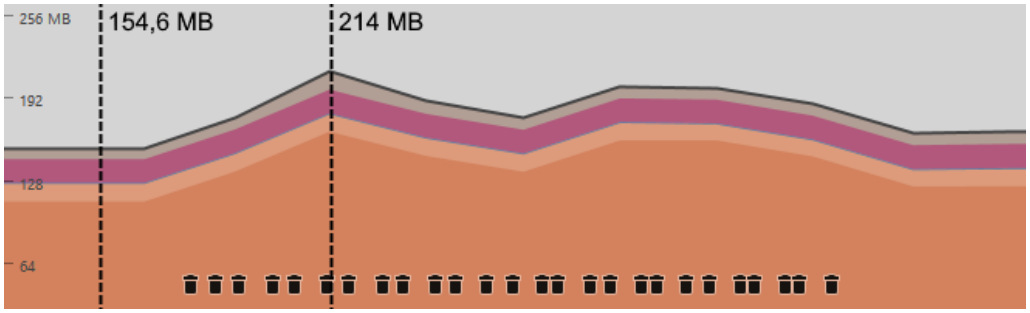
Beim Start der App oder wenn der aktive Graph geändert wird, müssen die Daten aus der *.bfmi*-Datei in den Arbeitsspeicher geladen werden. Die Ladezeit ist abhängig von der Größe des Graphen. Für den Regierungsbezirk Stuttgart und Baden-Württemberg liegen die Ladezeiten bei 2,1 beziehungsweise 5,7 Sekunden, was noch in einem akzeptablen Bereich liegt. Bei größeren Graphen könnte die Ladezeit jedoch auf über 10 Sekunden ansteigen, was die allgemeine Nutzerfreundlichkeit der App einschränken würde.

6.2 Map Matching

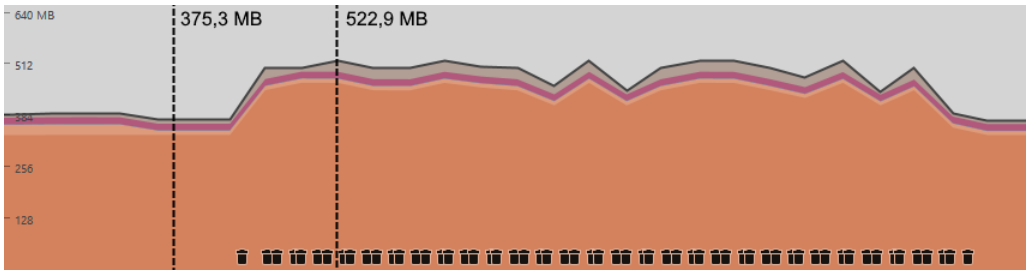
6.2.1 Arbeitsspeicherverbrauch

Der Arbeitsspeicherverbrauch des Map-Matchers wird vor allem durch den Dijksta-Algorithmus dominiert. Dieser allokiert zwei Arrays mit einer Größe von je $4 * |V|$ Bytes. Der Speicherverbrauch ist also direkt abhängig von der Größe des aktiven Graphen.
Hinzu kommt die PriorityQueue, welche die aktuell betrachteten Knoten hält. Sie weist eine dynamische Größe auf, die von der Dichte des Graphen beeinflusst wird.

Weiterer Arbeitsspeicher wird durch den Map-Matcher selbst verbraucht, der für jeden Kandidatenknoten der aktuellen Positionsmessung den kürzesten Pfad im Speicher halten muss. Die Größe dieser Datenstruktur wird also maßgeblich von der Anzahl an Kandidatenknoten der letzten Messung (welche auch von der Präzision der GPS-Messung abhängt) und von der Länge des Pfades beeinflusst. Dieser Speicherverbrauch ist jedoch mit 4 Bytes pro traversierter Kante vergleichsweise gering.



(a) Map-Matching unter Verwendung des Graphen des Regierungsbezirks Stuttgart. Gut erkennbar ist der deutlich geringere Speicherverbrauch sowohl vor als auch während dem Map-Matching im Vergleich zu Figur (b).



(b) Map-Matching unter Verwendung des Graphen von Baden-Württemberg. Gut erkennbar ist der deutlich höhere Speicherverbrauch und die wesentlich aggressivere Garbage Collection durch das System im Vergleich zu Figur (a). Weiterhin ist sichtbar, dass der Speicherverbrauch im Verlauf des Map-Matchings nicht ansteigt, sondern die lokalen Maxima alle einen ähnlichen Wert aufweisen.

Abbildung 6.1: Arbeitsspeicherverbrauch im Zeitverlauf während dem Map-Matching von 78 Positionsmessungen, die in einem Abstand von 10 Sekunden aufgenommen wurden. Die “Mülleimer”-Symbole signalisieren die Ausführung einer Garbage Collection.

So ergibt sich für einen Pfad mit einer Länge von 1000 traversierten Kanten (Eine Pfadlänge in der Größenordnung von 20 km) und einer letzten Positionsmessung, in deren Fehlerkreis 20 Kandidatenknoten lagen, ein Speicherverbrauch von $4 * 1\,000 * 20 = 80\,000$ Bytes.
Nach Abschluss der Aufzeichnung wird nur der eine Pfad zu dem Kandidatenknoten behalten, der minimal ist. Somit beläuft sich der Speicherverbrauch dann nur noch auf $4 * 1\,000 = 4000$ Bytes.

6.2.2 Laufzeit

Für die Laufzeit des Map-Matching-Algorithmus existieren im Wesentlichen zwei relevante Faktoren. Einerseits ist das die Ausführungsdauer der Dijkstra-Operation, welche für jede Positionsmessung einmal ausgeführt werden muss.
Der andere relevante Faktor ist dementsprechend die Anzahl der Positionsmessungen. Bei der Ausführung des Map-Matchers als Online-Algorithmus ist es hier besonders wichtig, dass die Ausführungsdauer zur Verarbeitung einer Positionsmessung nicht die Dauer zwischen Positionsmessungen überschreitet, da der Algorithmus sonst in Verzug gerät.
Die Laufzeit des abschließenden Schritts des Map-Matching-Algorithmus, in welchem der Kandidat mit dem kürzesten Pfad ausgewählt wird, ist hingegen so gering, dass diese für die Laufzeit des gesamten Prozesses vernachlässigt werden kann.

Die Laufzeit einer Dijkstra-Operation ist hierbei davon abhängig, ob Pfade zu allen Zielknoten gefunden werden können oder ob der Schwellenwert zum frühzeitigen Abbruch, welcher in Kapitel 4.2 beschrieben wird, erreicht wird.
Weiterhin ist sie davon abhängig, wie weit die Positionsmessungen voneinander entfernt sind, da der Dijkstra-Algorithmus bei weiter voneinander entfernten Start- und Zielknoten mehr Iterationen durchführen muss, bis er das Ziel erreicht.

Unter Verwendung der Graphdaten für den Regierungsbezirk Stuttgart liegt die Ausführungsdauer einer Dijkstra-Operation bei eng beieinanderliegenden Positionsmessungen (Abstand von 10 Sekunden zwischen Positionsmessungen) in der Größenordnung von 6 ms. Selbst bei einer Terminierung durch Erreichen des Schwellenwerts liegt die Laufzeit hier nicht wesentlich höher.
Bei wesentlich größeren Messabständen von 3 bis 6 Minuten liegt die Berechnungsdauer der kürzesten Pfade zwischen zwei Messungen bei 20 bis 80 ms, wenn alle Knoten erreicht werden können und bei bis zu 830 ms, wenn die Berechnung durch Erreichen des Schwellenwerts abgebrochen wird.

Wenn die wesentlich größeren Graphdaten verwendet werden, welche ganz Baden-Württemberg abdecken, wächst die Ausführungsdauer einer Dijkstra-Operation auf ca. 20 ms an. Dies ist auf die wesentlich größeren Arrays zurückzuführen, die durch den Dijkstra-Algorithmus allokiert und mit Standardwerten initialisiert werden müssen.
Durch die großen Mengen an Speicher, die allokiert und auch wieder freigegeben werden müssen, macht sich auch die Garbage Collection bemerkbar. Für Dijkstra-Ausführungen, in denen der Garbage Collector ausgeführt wird, steigt die Laufzeit auf bis zu 250 ms an. Der Garbage Collector wird auch wesentlich öfter ausgeführt, wie in Grafik 6.1 erkennbar ist, da das System öfter Platz für die großen Arrays des Algorithmus schaffen muss.

Um die Performance-Auswirkungen der Garbage-Collection und Initialisierung der Arrays mit Standardwerten zu reduzieren, könnten spezifische Optimierungen durchgeführt werden. So könnten die Arrays zwischen Dijkstra-Ausführungen wiederverwendet werden, wodurch sie nicht jedes Mal neu allokiert und wieder freigegeben werden müssten.
Da die meisten Elemente der Arrays ohnehin nie genutzt werden, müsste dann noch ein Weg geschaffen werden, nur die Elemente mit neuen Standardwerten zu initialisieren, die in der letzten Ausführung verändert wurden.

Tiefschlaf (Deep Sleep) des Geräts während Aufzeichnungen

Für eine akzeptable Akkulaufzeit von Android-Geräten ist die Reduzierung des Energieverbrauchs des Prozessors sehr wichtig. Wenn der Bildschirm ausgeschaltet ist und keine aufwendigen Berechnungen durchgeführt werden, versetzt das System die CPU in einen Modus, der als Tiefschlaf oder Deep Sleep bezeichnet wird, um so Energie einzusparen. Die CPU wird aus diesem Modus automatisch aufgeweckt, wenn mehr Leistung benötigt wird. Wenn von Apps verhindert wird, dass die CPU in den Tiefschlaf versetzt wird, kann dies einen starken, negativen Einfluss auf die Akkulaufzeit haben, selbst wenn keine komplexen Berechnungen durchgeführt werden sondern die CPU sich im “Leerlauf” befindet. So verbraucht das Testgerät im Tiefschlaf ca. 1.5% der Akkukapazität pro Stunde, während der Wert bei blockiertem Tiefschlaf im Leerlauf auf 6% pro Stunde ansteigt.

Zur Ermittlung der Energieeffizienz der entwickelten App wurde der Anteil der Zeit während einer Aufzeichnung gemessen, den der Prozessor im Tiefschlaf verbringt. Bei einer Periode von 10 Sekunden zwischen Positionsmessungen verbrachte das Gerät hierbei 36% der Zeit im Tiefschlaf. Bei einer Messperiode von 30 Sekunden hingegen waren es bereits 87%. Ohne aktive Applikationen verbringt das Gerät hingegen 95% der Zeit im Tiefschlaf.

6.3 Gespeicherte Aufzeichnungen

Nr.	Pfadlänge	Anzahl Pos.messungen	Anzahl traversierter Kanten	Zeit zw. Positionsmessungen	Dateigröße
1	9,1 km	78	415	10s	5,32 KB
2	26,4 km	152	995	10s	11,02 KB
3	34 km	178	1183	10s	14,65 KB
4	25,3 km	18	991	60s	6,72 KB

Tabelle 6.2: Beispielhafte Aufzeichnungen und ihre Dateigröße im *.trk*-Format.

Der größte Speicherverbrauch des in Kapitel 5.6.1 beschriebenen *.trk*-Binärdateiformat zur Speicherung von getätigten Aufzeichnungen entsteht vor allem aus den Level-Of-Detail-Daten und den traversierten Kanten. Hinzu kommen noch die Daten der einzelnen Positionsmessungen und einige Metadaten.
Trotzdem bleibt die Dateigröße selbst von längeren Aufzeichnungen so klein, dass diese in keinem Fall ein Problem darstellen sollte.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine voll funktionstüchtige *Android*-App entwickelt, welche das kontinuierliche Aufzeichnen der Position des Gerätes und Map-Matching dieser Positionsdaten auf ein Wegenetz erlaubt.
Die App bietet somit die Funktionalität einer typischen GPS-Tracker-App, jedoch mit der zusätzlichen Funktionalität der Zuordnung von GPS-Tracks zu Bewegungen auf dem Straßennetz.

Es wurde ein effizienter Algorithmus präsentiert, welcher performantes und somit energiesparendes Map-Matching direkt auf dem Mobilgerät erlaubt. Weiterhin wurden mehrere Hürden, welche einer performanten Implementierung der Algorithmen im Weg standen, identifiziert, beschrieben und Lösungen für diese erarbeitet und implementiert.

7.1 Zukünftige Arbeiten

In der entwickelten App existieren noch einige Einschränkungen, die eine vollständig im Hintergrund ablaufende Aufzeichnung unmöglich machen.

Während das Laden einer Graph-Repräsentation des Straßennetzes der gesamten Erde aufgrund von Speicherbeschränkungen nicht realistisch ist, macht das aktuelle System des manuellen Auswählens einer Graph-Repräsentation der Region, in der die Aufzeichnung stattfinden soll, die komplett automatisierte Durchführung einer Aufzeichnung unmöglich.
In dieser Hinsicht wäre die Entwicklung eines Systems, welches Teile eines Graphen dynamisch laden kann, ein wichtiger Schritt. Nur so können die Laufzeit- und insbesondere Speichergrenzen, die der App zur Verfügung stehen, eingehalten werden, wenn eine nahtlose, weltumspannende Aufzeichnung möglich sein soll.

Ein weiteres Problem stellt die Situation dar, die auftritt, wenn das Gerät das Straßennetz verlässt. Die aktuelle Implementation ist darauf ausgelegt, dass das Gerät das Straßennetz nicht verlässt. Eine App, die kontinuierlich im Hintergrund aufzeichnet, muss in der Lage sein, nahtlos zwischen dem Folgen eines Straßennetzes und freier Bewegung ohne Grenzen überzugehen.

Literaturverzeichnis

[1] . *calimoto*. URL: https://calimoto.com/ (besucht am 04. 07. 2022) (zitiert auf S. 10).

[2] *DataOutputStream writeUTF (Java Platform SE 7)*. URL: https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html#writeUTF(java.lang.String) (besucht am 30. 05. 2022) (zitiert auf S. 43).

[3] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische mathematik* 1.1 (1959), S. 269–271 (zitiert auf S. 14).

[4] J. Eisner, S. Funke, A. Herbst, A. Spillner, S. Storandt. „Algorithms for Matching and Predicting Trajectories“. In: *ALENEX*. SIAM, 2011, S. 84–95 (zitiert auf S. 16).

[5] FMI Univ. Stuttgart. *Useful Stuff*. URL: https://fmi.uni-stuttgart.de/alg/research/stuff/ (besucht am 10. 05. 2022) (zitiert auf S. 37).

[6] Google LLC. *Google Maps*. URL: https://play.google.com/store/apps/details?id=com.google.android.apps.maps (besucht am 17. 06. 2022) (zitiert auf S. 9).

[7] OSM Mitwirkende. *OpenStreetMap Urheberrecht und Lizenz*. URL: https://www.openstreetmap.org/copyright (besucht am 03. 05. 2022) (zitiert auf S. 13).

[8] OSM Mitwirkende. *Über OpenStreetMap*. URL: https://www.openstreetmap.org/about (besucht am 03. 05. 2022) (zitiert auf S. 13).

[9] *Osmdroid*. URL: https://github.com/osmdroid/osmdroid (besucht am 16. 05. 2022) (zitiert auf S. 27).

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