Institut für Formale Methoden der Informatik

Abteilung Formale Konzepte

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Masterarbeit Nr. 90

OSM-Navigationssystem für Android Watch

Denis Lukenich

Studiengang:	Informatik
Prüfer:	Prof. Dr. S. Funke
Betreuer:	Prof. Dr. S. Funke
begonnen am:	12.10.2015
beendet am:	12.04.2016
CR-Klassifikation:	G 2.2

Inhaltsverzeichnis

Abkürzungsverzeichnis

3G Third Generation

A* A-Stern

ADB Android Debug Bridge

API Application Programming Interface

ART Android Runtime

BW Backward

CH Contraction Hierarchies

CPU Central Processing Unit

ETL Extract Transform Load

FTP File Transfer Protocol

FW Forward

GB Gigabyte

GPS Global Positioning System

GPU Graphics Processing Unit

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

ID Identifikationsnummer

IO Input-Output

JNI Java Native Interface

KB Kilobyte

LRU Least Recently Used

MB Megabyte

NDK Native Development Kit

OpenCL Open Computing Language

OpenGL Open Graphics Library

OpenGL ES Open Graphics Library Embedded Systems

OSM Open Street Map

PBF Protocolbuffer Binary Format

RAM Random Access Memory

SDK Software Development Kit

USB Universal Serial Bus

WLAN Wireless Local Area Network

XML Extensible Markup Language

Kapitel 1

Einleitung

Smartphones sind ein wichtiger Teil unseres Lebens geworden. Während früher ein Handy nur zum Telefonieren da war, bestimmt es heute mit seinen vielseitigen Funktionen große Teile unseres Lebens. Eine der nützlichsten Anwendungen ist die mobile Routenplanung. Damit ist es stets möglich sich auf direktem Wege an bisher unbekannte Orte zu begeben. Neben den Smartphones erhöht sich aktuell auch die Bekanntheit der Smartwatches, welche dazu dienen die Funktionalität des Smartphones zu ergänzen und über die direkte Verfügbarkeit am Arm die Nutzung zu vereinfachen. Android Smartwatches sind dabei nicht nur ein Unterstützer für die Anwendungen des Telefons, sondern besitzen ein vollständiges Betriebssystem mit ausreichend Datenspeicher und Verarbeitungskapazität für komplexe Aufgaben. Auch die Anzahl an Sensoren, welche direkt auf der Uhr vorhanden sind, erhöht sich ständig. Somit werden die Smartwatches immer eigenständiger.

Die bekannteste Navigationsanwendung auf Smartphones ist Google Maps, welches beinahe auf jedem Android Smartphone installiert ist. Es besitzt eine graphische Oberfläche auf der Uhr. Dabei berechnet die Uhr die Daten allerdings nicht selbst, sondern dient nur als Demonstration der Daten die das Mobiltelefon liefert. An dieser Stelle besteht zurzeit eine Marktlücke. Es gibt keine Anwendung, welche unabhängig vom Smartphone eine Route zu einem beliebigen Ziel berechnen kann. Diese Masterarbeit hat das Ziel hierfür eine geeignete Anwendung zu konzipieren und zu entwickeln.

Dabei soll das Mobiltelefon unterstützend zu den eigentlichen Routingaufgaben zur Verfügung stehen. Es dient zum Aufspielen der Daten auf die Uhr und kann verwendet werden, um Routingziele vorab festzulegen. Das Bestimmen der aktuellen Position und die Berechnung der Wege muss die Anwendung lösen können, ohne dabei auf das Smartphone zurückzugreifen.

Ohne die direkte Verbindung mit dem Smartphone, kann für viele Smartwatches auch keine Verbindung zum Internet aufgebaut werden. Viele aktuelle Navigationslösungen basieren auf diesem Internetzugriff. Speziell für Jogger oder Wanderer kann es interessant sein, ohne das Smartphone unterwegs zu sein. Die Uhr am Handgelenk ist einfacher mitzuführen als ein Smartphone in der Tasche. Im Gegensatz zum Telefon besteht bei der Uhr nur ein geringes Risiko, dass sie verloren geht. Somit besteht der Bedarf für eine vom Mobiltelefon unabhängige Lösung.

Technisch sind in dieser Arbeit verschiedene Schwierigkeiten zu erwarten.

Einerseits ist der Speicher der Smartwatches begrenzt. Aktuelle Modelle haben zwischen 2 und 8 GB an Flash-Speicher und ca. ein halbes GB an Arbeitsspeicher, wovon nur ein kleiner Teil verwendbar ist, da auch andere Apps und das Betriebssystem diesen benutzen. Des Weiteren ist die Rechenleistung geringer als bei Smartphones oder Desktop Computern bzw. Servern. Und zuletzt ist die Benutzerschnittstelle von Smartwatches sehr klein und meist nur wenige Zentimeter hoch und breit. Entsprechend unterscheidet sich die Art und Weise der Eingaben stark von klassischen Systemen.

Diese Arbeit wird unterstützt von Wearable Software, einem Pionier in der Entwicklung für Smartwatch Anwendungen aus Karlsruhe. Von ihnen wurden mehrere Anwendungen, welche teilweise sehr erfolgreich sind, seit der Anfangszeit von Android Wear veröffentlicht. Die Unterstützung beschränkt sich auf beratende und gestaltende Aktivitäten in der Entwicklung dieser Anwendung.

Um eine geeignete Lösung entwickeln zu können, erörtert diese Arbeit zuerst verschiedene Grundlagen. Dabei wird zuerst die Smartwatch definiert und ihre aktuelle und zukünftige Entwicklung wird vorgestellt. In diesem Zuge wird auf das Android-Betriebssystem als Basis dieser Arbeit eingegangen und auf die Softwareentwicklungsmöglichkeiten, die es bietet. Neben diesen Kenntnissen der Entwicklung benötigt die Anwendung verschiedene kartographische Daten. Hierfür wird das OSM-Projekt eingeführt, welches global Straßenkarten anbietet. Da diese Kartendaten viel Speicherplatz benötigen, werden für ein platzsparendes Speichern der Daten verschiedene Basics der Datenkompression vorgestellt. Für die Berechnung von kürzesten Routen innerhalb des Straßennetzes werden verschiedene Algorithmen zur effizienten Routenplanung erläutert und gegenübergestellt.

Basierend auf diesen Grundlagen wird die Problemstellung analysiert. Dies ist notwendig, um genau bestimmen zu können, welche Aspekte in der weiteren Entwicklung genauer betrachtet werden müssen und welche weniger wichtig sind. Zuerst werden die Anforderungen an die Anwendung herausgearbeitet und die Hauptprobleme werden analysiert. In diesem Zusammenhang werden die Stärken und Schwächen von Smartwatches beleuchtet und deren Auswirkungen auf die Problemstellung untersucht. Als Basis dieser Arbeit dient eine Sony SWR 50, auch Sony Smartwatch 3 genannt, diese wird mit ihren technischen Daten vorgestellt. Neben den technischen Herausforderungen werden auch die Herausforderungen in der Benutzerinteraktion betrachtet, da sich die Bedienung von Uhren unterschiedlich im Vergleich zu der Bedienung von Smartphones gestaltet. Google gibt hierzu entsprechende Empfehlungen heraus. Das Android Wear System beinhaltet auch viele Application Programming Interfaces (APIs), die die Entwicklung der Anwendung unterstützen können, von denen ausgewählte an dieser Stelle vorgestellt werden.

Mit den Erkenntnissen der Problemanalyse wird das Systemdesign erstellt, um eine Übersicht über die einzelnen zu entwickelnden Komponenten zu bekommen. Für die erkannten Aufgaben werden verschiedene Lösungsmöglichkeiten erwogen und über Analysen und Testaufbauten werden für die einzelnen Probleme die Lösungen für die finale Implementierung ermittelt. Dies umfasst ein Management des vollständigen Datenflusses von der Extraktion aus den OSM-Dateien über die Übermittlung zur Uhr, die Speicherung auf dieser und ein effizientes Einlesen in den Arbeitsspeicher, welches für eine kurze Laufzeit des Routingalgorithmus unerlässlich ist. Ebenso werden die vorgestellten Algorithmen im Problemzusammenhang analysiert und in Anbetracht der Eigenschaften

der Smartwatch gegenübergestellt. Aus den gewonnenen Informationen wird der am besten geeignetste Algorithmus ausgewählt. Zum Abschluss der Designanalyse werden betriebswirtschaftliche Gedanken zur Anwendung vorgestellt. Dies werden Aspekte zur Vermarktung der Anwendung sein und Möglichkeiten ein finanzielles Ergebnis zu erzielen.

Daraufhin beschäftigt sich die Implementierung mit der Umsetzung der vorbereiteten Lösungen und beschreibt verschiedene zentrale Aspekte der Implementierung mit den verwendeten Technologien. Dazu gehört der vollständige Ablauf der Vorverarbeitung der Daten. Auch der zuvor ausgewählte Routingalgorithmus benötigt verschiedene Anpassungen um die korrekten Daten einzulesen und Zwischenergebnisse zu speichern. Dieses Vorgehen wird erläutert. Das dritte vorgestellte Thema betrifft das Rendern der Karte.

Zum Ende der Arbeit wird das Ergebnis präsentiert. Hierbei werden die User Experience und das Handling beschrieben und bewertet. Des Weiteren werden Zeitmessungen des Algorithmus und der Kartenrenderung durchgeführt. Ebenfalls wird eine erste Analyse des Erfolgs der Anwendung vorgestellt.

Die Arbeit schließt mit einer Zusammenfassung.

Kapitel 2

Verwandte Arbeiten

Es gibt verschiedene Ansätze Navigationssysteme auf mobile Geräte zu bringen, wobei die spezialisierten Navigationssysteme in dieser Arbeit nicht berücksichtigt werden. An dieser Stelle werden Umsetzungen und Lösungsansätze aus dem Open Source Umfeld, aus kommerziellen Anwendungen und aus dem wissenschaftlichen Bereich vorgestellt.

Der bekannteste Vertreter mobiler Navigationssoftware ist Google Maps. Bald nach Erscheinen der Smartwatches wurde eine Oberfläche für Smartwatches angeboten. Die Grundidee, welche Google hierbei verfolgt, ist die reine Ansicht der Karte und Navigation auf der Smartwatch. Das Mobiltelefon wird vollständig zur Berechnung und Eingabe der Ziele verwendet.[23] Interaktion mit der Smartwatch lässt den Kartenausschnitt verschieben und eine textuelle Anzeige der verbleibenden Route wird ermöglicht. Seit 2015 ist ebenfalls ein Download von Kartenausschnitten auf das Mobiltelefon möglich. Somit wird dem Benutzer erlaubt die Navigation ohne Internetverbindung durchzuführen. Ein weiterer Kartendienst Here ist bisher nicht für Android Uhren verfügbar. Er war Vorreiter im Download von Kartendaten und setzt auf eine internetunabhängige Navigation des Telefons.[30]

Im Open Source Bereich sind zwei bekannte Vertreter zu erwähnen. Einerseits gibt den Graphhopper, eine Implementierung, welche verschiedene Routingalgorithmen beinhaltet und auch für Android verfügbar ist. Allerdings nicht speziell auf Uhren abgestimmt ist. Des Weiteren gibt es mit dem Tourenplaner eine Umsetzung der Universität Stuttgart. Er beinhaltet eine Web-Oberfläche und eine Android Implementierung. Letztere wurde von Stefan Bühler als Bachelorarbeit erstellt und somit besteht eine wissenschaftliche Quelle hierzu.[8] Hierbei werden die Herausforderungen von Android bei der Implementierung eines Routenplaners beschrieben.

2011 schrieb Ildas Klassen an der Universität Koblenz seine Bachelorarbeit über profilbasierte Navigation auf einem Mobiltelephon. Diese Implementierung wurde speziell für das iPhone entwickelt und beinhaltet ebenfalls eine Offlinefunktionalität und legt den Fokus auf die Nutzung verbessertem und detaillierterem Kartenmaterial in Kombination mit einer verbesserten Positionsbestimmung. Damit soll die Routenfindung detaillierter sein und beispielsweise für Fußgänger die korrekte Straßenseite abbilden.[27]

Einen weiteren Artikel über mobiles Routing schrieben Sanders und Schultes. Sie nutzen Contraction Hierarchies (CH) als Basis und erarbeiten eine Daten-

struktur, welche eine möglichst hohe Datenlokalität anstrebt, um die Anzahl der langsamen Input-Output (IO)-Operationen zu reduzieren. Somit kann die kürzeste Route in Sekundenbruchteilen gefunden werden.[34]

Weitere thematische Überschneidungen bestehen mit der Publikation von Tobias Bagg. Er untersucht verschiedene Kompressionsverfahren auf deren Anwendbarkeit bei der Komprimierung von Graphen.[7]

Kapitel 3

Grundlagen

Dieses Kapitel liefert allgemeine Informationen, welche dem Verständnis dieser Arbeit dienen. Zu Beginn wird ein Überblick über die historische Entwicklung und die aktuelle Verbreitung und den Möglichkeiten von Smartwatches gegeben, auch erwartete zukünftige Entwicklungen werden erwähnt. Daraufhin wird das Betriebssystem Android eingeführt. Dabei werden sowohl der grundsätzliche Aufbau als auch verschiedene Tools zur Softwareentwicklung wie die Entwicklungsumgebung Android Studio vorgestellt.

Im dritten Abschnitt wird das OSM-Projekt als freie Datenquelle für Kartendaten vorgestellt. Dabei werden mögliche Bezugsquellen für die Daten und der Aufbau der entsprechenden Formate benannt.

Der folgende Abschnitt beschäftigt sich mit einer allgemeinen Einführung in die Datenkompression und verschiedenen Ansätzen hierzu. Ausgewählte Codierungen werden detailliert erläutert.

Der fünfte Abschnitt führt Algorithmen, welche den kürzesten Pfad bestimmen, ein. Zuerst liefert er allgemeine theoretische Informationen über das zu lösende Problem und beschreibt daraufhin den Dijkstra-Algorithmus und dessen Verbesserung mittels Heuristiken zu dem A*-Algorithmus. Beide werden auch in der bidirektionalen Variante vorgestellt. Zuletzt wird Contraction Hierarchies eingeführt, welche im Gegensatz zu den vorherigen Methoden speziell vorbereitete Daten benötigt.

3.1 Smartwatches

Als Smartwatch wird eine Uhr bezeichnet, welche durch einen internen Computer unterstützt wird, um neben dem Anzeigen der aktuellen Uhrzeit weitere Funktionalität zu ermöglichen, wie das Anzeigen von Notifications und der Verbindung zu Smartphones.[9] Häufig werden sie für gesundheitsbezogene Anwendungen oder zur Unterstützung der Funktionalitäten des Smartphones verwendet.

Im Weiteren befasst sich dieser Abschnitt mit der Entwicklung von Smartwatches und schafft einen Überblick über vorhandene Modelle und deren Umsetzungen, dabei wird auf die aktuelle Reichweite der Uhren eingegangen, um die Möglichkeiten von Software auf diesen Systemen abschätzen zu können. Spätestens seit Veröffentlichung der Apple Watch ist der Markt für Smartwatches stark

am Wachsen und seitdem sind die digitalen Uhren auch aktiv im Bewusstsein der Bevölkerung. Dabei existiert die Idee programmierbare Uhren zu erstellen weit länger. Die vermutlich erste Smartwatch wurde bereits im Jahre 1972 entwickelt und konnte 24 Nummern speichern. Sie wurde von der Hamilton Watch Company entwickelt und hieß Pulsar.[10]

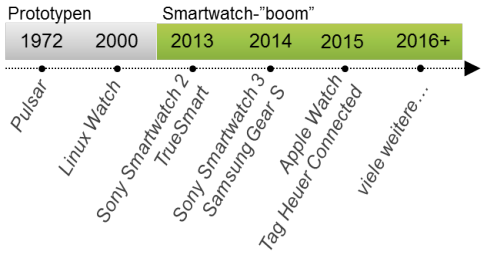


Abbildung 3.1: Entwicklung von Smartwatches

Nach weiteren Entwicklungen stellte IBM im Jahr 2000 einen Prototyp vor, welcher auf Linux basierte und Linux Watch hieß.[25] Die erste Uhr, welche nach heutiger Definition als Smartwatch bezeichnet wird, stammt aus einem Kickstarter Projekt aus dem Jahre 2013. Ihr Name lautet TrueSmart. Wie in Abbildung 3.1 erkennbar ist, startet in diesem Jahr eine verstärkte Entwicklung von Smartwatches, an dieser Stelle entsprechend Smartwatch-"boom" genannt.[31]

Spätestens mit der Veröffentlichung der Apple Watch im April 2015 sind intelligente Uhren ins Blickfeld der allgemeinen Öffentlichkeit geraten. Der Marketingspezialist Apple platzierte seine Uhr mit der gewohnten Werbepräsenz und Medienaufmerksamkeit, sodass die allgemeine Bekanntheit der Uhren deutlich gesteigert wurde.[5] Insgesamt wird der Markt der Uhren nicht nur von den üblichen Smartphone- und Hardwareherstellern bevölkert, sondern auch von den traditionellen Uhrenherstellern. Beispielsweise positionierte TAG Heuer eine hochpreisige Smartwatch im Jahre 2015 am Markt, deren Erstauflage innerhalb weniger Tage ausverkauft war.[42] Weitere bekannte Hersteller sind Samsung,[39] LG[29] und Pebble.[35] Insgesamt gibt es aktuell geschätzt 15 ernstzunehmende Modelle, wobei viele weitere Modelle angekündigt sind.[28]

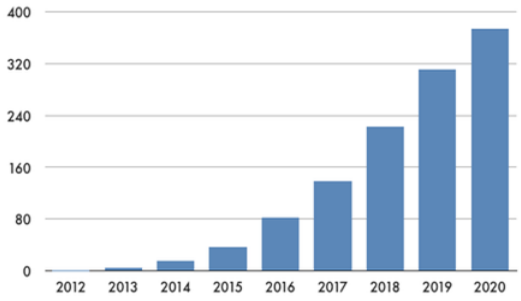


Abbildung 3.2: Übersicht geschätzte Android Smartwatch Verkäufe

Die bisherige Reichweite der Uhren kann nur geschätzt werden. Die produzierenden Unternehmen veröffentlichen nur wenige aktuelle Zahlen, entspre-

chend sind die angegebenen Werte auf Markforschungsanalysen zurückzuführen. Abbildung 3.2 zeigt das erwartete Wachstum der Wearableverkäufe und es ist erkennbar, dass die Smartwatchverkäufe den größten Teil hiervon ausmachen und in den nächsten Jahren stark ansteigen werden. Im Jahre 2019 soll für Smartwatches ein Markt von 400 Millionen Geräten pro Jahr vorhanden sein, entsprechend groß sind die potentiellen Erfolgschancen sowohl für Hardware- als auch für Softwarehersteller. Einen Teil des überproportionalen Anstiegs zwischen 2015 und 2018 lässt sich durch das Erscheinen der Apple Watch und deren Marketingeffektes erklären. Im Zuge dessen ist auch eine deutliche Erhöhung der Android-Verkaufszahlen,[41] gegenüber den ca. 80 Millionen verkauft Smartwatches im Jahre 2015 zu erwarten.[44]

Insgesamt konkurrieren im Smartwatchmarkt drei große Betriebssysteme. iOS,[4] Android[16] und Tizen.[13] Die Herangehensweise an die Funktionen der Uhren unterscheidet sich hierbei. Während Android und Tizen seit längerem Programmierern Zugriff auf die Rechenkapazität und Sensoren bieten, bot Apple bis Oktober 2015 die Smartwatch nur als Anzeige von auf dem iPhone berechneten Daten an. Erst seit dem Update auf Watch OS 2 im Oktober 2015 ist es dort möglich auf Sensoren direkt zuzugreifen, sodass Watch-eigene Berechnungen durchgeführt werden können. Entsprechend eng ist die Apple Watch an das iPhone gekoppelt. Eine Verwendung allein oder mit Smartphones anderer Hersteller ist nicht möglich, während konkurrierende Geräte unabhängig von bestimmten Smartphones sind und auch eigenständig verwendet werden können. Des Weiteren unterscheiden sich alle drei Anbieter auch in ihren Bedienkonzepten. Während Android einen starken Fokus auf Touch-Funktionen legt, versuchen Tizen und IOS mit mechanischen Knöpfen an den Seiten der Uhr die Interaktivität zu verbessern.[28]

Bisher sind die meisten Uhren eng mit dem Mobiltelefon verbunden. Speziell für die Internetfunktionalität und für eine Tonausgabe werden Smartphones häufig benötigt. Es gibt allerdings erste Versuche Smartwatches mit eigener SIM-Karte auf den Markt zu bringen,[43] dies würde die Smartwatch als vollständig eigenständiges elektronisches Gerät erscheinen lassen. Vermutungen zur zukünftigen Entwicklung gehen sogar soweit, dass eine Kombination aus Smartwatch und Tablet das Smartphone ersetzen wird, da zwischen den beiden Geräten keine weitere Produktgruppe mehr benötigt werden könnte.[28]

Weitere Indizien, dass die Uhren schnell an Popularität gewinnen, lassen sich bei den Netzanbietern und den Krankenkassen finden. So gibt es bei der Vertragsverlängerung für einen Handyvertrag auch Uhren, welche preisgünstig oder gar kostenfrei erhalten werden können.[40] Auch einige Krankenkassen haben den Wert von Smartwatches als Gesundheitsgeräte erkannt. So bezuschusst beispielsweise die Techniker Krankenkasse den Kauf von Smartwatches, sofern die Übertragung verschiedener Daten an die Krankenkasse gestattet wird.[6] Ebenfalls werden in naher Zukunft weitere unterschiedliche Gerätetypen und zusätzliche Funktionen hinzukommen, auch dies wird die Popularität weiter unterstützen.[44]

3.2 Android Betriebssystem und Entwicklung

Android ist ein Betriebssystem, welches von Google in erster Linie für mobile Geräte entwickelt wurde, und heute auf über einer Milliarde Geräten zum Einsatz kommt. Dazu gehören Mobiltelefone, Tablets, Smartwatches aber auch Fernsehgeräte und weitere Embedded Systems.[16]

Dieses Betriebssystem basiert auf einem Linux-Kernel und wurde erstmals im Jahre 2008 veröffentlicht. Mit der Einführung von Android 4.4 „Kit Kat“ im Oktober 2013 werden auch Smartwatches unterstützt.[17]

Die Entwicklung für Android findet in erster Linie in der Programmiersprache Java mittels dem Android Software Development Kit (SDK) statt. Dabei handelt es sich um eine Abspaltung von Java, welche in seiner aktuellen Form kompatibel zu Java 7 ist. Entwicklern stellt Google die Integrated Development Environment (IDE) Android Studio bereit. Diese basiert auf der Entwicklungsumgebung IntelliJ und ist auf Android spezialisiert.[18] Entsprechend viele hilfreiche Tools für die Appentwicklung sind bereits integriert: Gradle wird als Build-System verwendet, um das Packen und Erstellen der Anwendungen automatisiert zu ermöglichen. Des Weiteren ist ein Layout Manager integriert, ebenso das Tool Lint für Erkennung von Performance-, Usability-, Kompatibilitäts- oder anderen Problemen. Zusätzlich können Anwendungen einfach signiert werden. Dies wird für eine Veröffentlichung in Googles internem App Store, dem PlayStore, benötigt.

Wichtig für das Entwickeln von leistungskritischen Anwendungen ist der Zeitpunkt und die Art der Kompilierung. Seit Android 5 wird dies bei Installation der Anwendung durchgeführt. In älteren Versionen wird die Anwendung bei jeder Ausführung also Just in Time erneut kompiliert, wie beispielsweise in der Oracle Java Laufzeitumgebung. Durch das neue Verfahren kann die Anwendung speziell für das Mobiltelefon angepasst werden und der Overhead der Laufzeitkompilierung entfällt. Die neue Engine hierfür heißt Android Runtime (ART) und ersetzt die bis Android 4.4 verwendete Dalvik Laufzeitumgebung.[20]

Neben der Entwicklung in Java kann seit Android Studio 1.3 auch das Native Development Kit (NDK) verwendet werden. Dies ermöglicht die Entwicklung von vorkompilierten Programmen in C++. Diese können unter Umständen schneller ausgeführt werden. Spätestens mit dem ebenfalls vorkompilierten Java-Anwendungen mit der ART entfällt der größte Teil der höheren Performance auf die größeren Freiheiten im Speichermanagement. Mit entsprechendem Aufwand können hierbei anwendungsspezifisch optimierte Lösungen erstellt werden. Zur Verbindung mit Java Quellcode wird das Java Native Interface (JNI) unterstützt.[24]

Zusätzlich zu den beiden Programmiersprachen für die Central Processing Unit (CPU) unterstützt Android beispielsweise auch Renderscript. Dies ist eine Google entwickelte plattformunabhängige Berechnungsengine, welche die auszuführenden Aufgaben automatisch über alle verfügbaren Kerne verteilt. Damit können alle Ressourcen der CPU oder Graphics Processing Unit (GPU) verwendet werden.[21]

Des Weiteren ist Open Graphics Library (OpenGL)-Unterstützung grundsätzlich vorhanden. Das sogenannte Open Graphics Library Embedded Systems (OpenGL ES) wird hierbei verwendet. Ab Android API Level 8 ist OpenGL ES 2.0 vorhanden. Da das Betriebssystem für Android Wear auf einem höheren API Level aufsetzt, kann OpenGL ES 2.0 immer verwendet werden. Dies er-

möglicht hoch performante zwei- und dreidimensionale Graphikberechnungen. Ab API Level 18 wird OpenGL ES 3 unterstützt und als aktuellste Ausführung wird Version 3.1 ab API Level 21 unterstützt. Für jedes Telefon muss überprüft werden, ob die Unterstützung durch die Hardware gegeben ist. Die reine Unterstützung des Betriebssystems bedeutet allerdings noch nicht, dass es auch einsetzbar ist.[32]

Die Liste an Programmiermöglichkeiten ist hiermit noch nicht vollständig. Es gibt weitere Frameworks, welche beispielsweise die Programmierung in PHP[26] oder anderen Webtechnologien[3] ermöglichen. Auf diese wird in dieser Arbeit nicht weiter eingegangen.

Ein wichtiger Aspekt in der performanten Android Entwicklung ist die Handhabung des Arbeitsspeichers. In den nativen Entwicklungsmöglichkeiten ist vergleichsweise einfach zu bestimmen, wie viel Arbeitsspeicher eine Datenstruktur nutzt. Im NDK wird pro Klasseninstanz genau der Speicherplatz benötigt, welchen die Variablen innerhalb der Instanz nutzen. Im SDK ist der Speicherverbrauch komplexer zu bestimmen. Ein Objekt belegt hierbei mindestens 32 Bit, wovon 8 Bit nicht zur freien Verwendung verfügbar sind. Sollten eine Instanz mehr als 32 Bit benötigen, so findet eine Erhöhung des allokierten Speicherbedarfs in 16 Bit Schritten statt.

Neben dem Speicherverbrauch einzelner Objekte kann die Gesamtmenge an verfügbarem Speicher abgefragt werden. Zusätzlich lässt sich über das Attribut „largeHeap“ die verfügbare Arbeitsspeichergröße auf das maximal Mögliche des Gerätes erhöhen. Dies kann allerdings Nebenwirkungen haben: Android beendet Anwendungen, welche viel Arbeitsspeicher belegen schneller als andere. Dies kann den Benutzer verärgern, da auch die aktuell ausgeführte Anwendung beendet werden kann.

Dieses Attribut wird in der Manifestdatei hinterlegt. Diese ist für jede Anwendung vorhanden und liefert dem System bestimmte Informationen die zur Ausführung der Anwendung benötigt werden, beispielsweise sogenannte Permissions. Also die Anfrage zu Rechten, die der Benutzer bei Installation der Anwendung bestätigen muss, oder eben Attribute zur Implementierung der Anwendung.

Die Installation und das Debugging von Android Anwendungen werden mittels der Android Debug Bridge (ADB)-Schnittstelle ermöglicht. Diese verbindet Android-Geräte über ein Universal Serial Bus (USB)-Kabel mit einem Computer und gewährt somit verschiedene Steuerungsoptionen des Gerätes.[2] Diese Funktionalitäten werden von den Debugging Tools genutzt. Einerseits lassen sich Anwendungen über den Android Studio-internen Debugger testen. Zusätzlich existiert mit dem Android Device Manager ein Tool, um den Zustand des Mobiltelefons detailliert auszulesen und Sensoren wie den Global Positioning System (GPS) Sensor mit vorbereiteten Daten zu belegen.

3.3 Das Open Street Map Projekt

Um eine geeignete Navigation zur Verfügung zu stellen, muss zuerst eine Möglichkeit gefunden werden entsprechende Kartendaten zu erhalten. Hierzu wird das Open Street Map (OSM) Projekt vorgestellt, welches Kartendaten öffentlich für Webseiten und weitere Anwendungen bereitstellt.[33] Das Projekt wurde 2004 initiiert und sammelt weltweit Kartendaten, um mit diesen beispielsweise

Forschung und Lehre zu unterstützen oder günstige Navigationsgeräte zu ermöglichen. Dabei werden sowohl die Rohform der Kartendaten, als auch vorverarbeitete Karten und APIs angeboten. Insgesamt haben in dem Projekt bisher ca. 2,4 Millionen Benutzer fast 5 Milliarden GPS Punkte hinzugefügt.[33]

Lizenzrechtlich befinden sich die OSM-Daten unter der Open Database Lizenz. Diese erlaubt die Nutzung der Datenbank selbst nur unter Weitergabe derselben Lizenz, weiterverarbeitete Daten können allerdings unter Nennung der Quelle weiterverbreitet werden. Dabei besteht die Auflage, dass eine abgeleitete Datenbasis ebenfalls unter der Open Database Lizenz frei zur Verfügung stehen muss.

Das OSM-Projekt beinhaltet Kartendaten für die ganze Welt. Verschiedene Anbieter bieten hierfür Auszüge oder vorverarbeitete Daten wie geographische Regionen oder gerenderte Karten an. Ein Beispiel hierfür ist das Unternehmen Geofabrik.[15] Auf deren Webseite können Kartendaten, die nach Kontinenten, Ländern oder Regionen aufgegliedert sind, heruntergeladen werden.

Die Daten selbst stehen in zwei Formaten zur Verfügung: Einem komprimierten Extensible Markup Language (XML)-Format als `.xml.bz2` und dem Binärformat `.pbf`. Die beiden Formate beinhalten dieselben Daten, besitzen allerdings eine unterschiedliche Struktur. Während das XML-Format alle Daten textuell repräsentiert, nutzt das Protocolbuffer Binary Format (PBF) Googles Protobuf-Bibliothek zur plattformunabhängigen Speicherung von Binärdaten.

Unabhängig von dem verwendeten Format sind die Daten in Knoten, Relationen und Wege eingeteilt. Knoten stellen Punkte auf der Landkarte dar. Das können Gebäude und andere nennenswerte Orte, oder schlichte Punkte auf der Landkarte sein, welche von den anderen beiden Objektarten verwendet werden können. Ein Weg ist eine geordnete Menge von bis zu 200 Knoten. Dies können Straßen oder andere Strukturen sein. Auch die Formen von Gebäuden können mit den Wegen modelliert werden. Zuletzt stellen die Relationen weitere Strukturen dar. Grenzen oder Flüsse werden mit ihnen abgebildet. Intern bestehen Relationen aus einer Menge von Wegen, Knoten oder weiteren Relationen. Zusätzlich besitzen die Dateien einen Header, welcher Daten zur Struktur des Datensatzes beinhaltet.

Zur Erweiterung der Kartenfunktionalität werden den Objekten Tags hinzugefügt. Diese sind als sogenannte Key-Value Paare implementiert. Das bedeutet, dass zu Schlüsselwerten wie beispielsweise „Straßenart“ zu jedem Datensatz ein entsprechender Wert wie „Autobahn“ hinterlegt werden kann. So kann bei Wegen angegeben werden, ob es sich um Straßen handelt und sofern das der Fall ist, kann auch der Straßentyp oder Straßenname vermerkt werden. Ebenfalls sind zusätzliche Eigenschaften, wie die befahrbaren Richtungen oder Höchstgeschwindigkeiten enthalten. Bei Knoten kann beispielsweise der Gebäudenamen vermerkt sein, sofern es sich bei diesem Punkt um ein Gebäude handelt. In diesem Falle können auch detailliertere Angaben erfolgen, wie die Art der Küche von Restaurants. Ebenso können die Relationen mit Informationen versehen werden. Beispielsweise können die Namen von Ländern, Staaten, Regierungsbezirken oder weiteren kartographischen Strukturen beschrieben werden.

3.4 Datenkompression

Da diese Kartendaten für Gesamtdeutschland sehr groß werden können, werden verschiedene Kompressionstechniken benötigt, um diese effizient übermitteln und speichern zu können. Dieses Kapitel erklärt die Grundlagen, wie Daten komprimiert abgespeichert werden können. Der erste Abschnitt behandelt die Grundbegriffe und die Basisinformationen, während der zweite Abschnitt verschiedene Ideen und jeweils beispielhaft einige Algorithmen, wie Daten in der Praxis komprimiert werden, einführt.

3.4.1 Grundbegriffe

Unter Datenkompression wird ein Vorgang beschrieben, welcher einen Datensatz neu codiert, um ihn in einer geringeren Menge an benötigtem Speicherplatz darzustellen. Die wichtigsten Ziele hierbei sind die Entfernung von Redundanz und Irrelevanz. Das Entfernen von Redundanz zielt darauf doppelte Informationen zu vermeiden. Ein einfaches Beispiel ist die Behandlung von zwei Straßen mit demselben Namen. Anstatt den Straßennamen zweimal zu speichern, kann bei einer Straße die Namensgleichheit vermerkt werden und somit kann die doppelte Information vermieden werden. Die Entfernung von irrelevanten Informationen beschreibt das Entfernen von Daten, welche nicht benötigt werden. Beispielsweise kann eine Anwendung, welche eine graphische Karte aus Straßeninformationen zeichnen soll, auf Attribute wie das Erbauungsdatum der Straße oder die erlaubte Höchstgeschwindigkeit verzichten, ohne an Qualität zu verlieren.

Unter den Kompressionsalgorithmen gibt es zwei grundsätzliche Arten: die verlustfreien und die verlustbehafteten Verfahren. Verlustfrei bedeutet, dass die Originaldaten exakt wiederhergestellt werden können. Bei der verlustbehafteten Kompression ist das nicht vollständig möglich. Dies eignet sich für Medieninhalte wie Videos, Musik und Bilder, da bei diesen durch die Kompression nur kleine im Optimalfall nicht wahrnehmbare Qualitätsunterschiede auftreten, während die Datenmenge erheblich reduziert werden kann. Die verlustfreie Kompression kann die Originaldaten wiederherstellen und eignet sich somit beispielsweise für Texte. Allerdings ist auch die Reduktion der Datenmenge in der Regel erheblich geringer.[11]

Vorteile der Datenkompression sind der geringere Platzbedarf der Daten und die daraus folgenden Effekte. Daten können kostengünstiger gespeichert werden und schneller übermittelt werden. Dabei sinkt auch der Energieverbrauch für Speicherung und Übermittlung. Auf der anderen Seite besteht zusätzlicher Aufwand in der Kompression und der Dekompression der Daten und zusätzlich liegt ein Nachteil, der nur bei der verlustbehafteten Kompression auftritt, in der geringeren Datenqualität.

Ein Grundbegriff der Datenkompression ist der Code.[38] Dieser bezeichnet eine feste Zuordnungsvorschrift einer Ausgangssymbolmenge zu einer Zielsymbolmenge. Dabei wird die Ausgangssymbolmenge Alphabet genannt, sie beinhaltet alle möglichen Symbole, welche auftreten können. Die Zuordnung aller Symbole zu deren Endzuständen, den Codewörtern, bilden zusammen das Wörterbuch.[11]

Codes unterscheiden sich in ihren Eigenschaften. Ihre Codewörter können dabei variable oder fixe Längen haben. Ein Beispiel für einen Code mit einer

festen Länge ist der ASCII Code. In seiner Grundform werden immer 7 Bit benötigt, um einen Buchstaben oder ein anderes Zeichen darzustellen. Ein Beispiel für einen Code mit variabler Länge sind Telefonnummern. Sie bezeichnen eindeutig welches Endgerät erreicht wird, unterschiedliche Nummern können aber unterschiedlich viele Stellen haben. Diese Codes haben den Vorteil, dass häufig vorkommende Symbole mit weniger Bits codiert werden und somit effizienter übertragen werden können.

Bei Codes mit unterschiedlicher Länge werden weitere Einteilungen vorgenommen. Einige der Codes sind eindeutig decodierbar. Das ist der Fall, sofern es ohne Längenangabe möglich ist, aus einer Sequenz mit mehreren Eingaben eindeutig auf die ursprüngliche Eingabe zurückzuschließen. Beispielsweise wäre bei der Sequenz `„24681357"` nicht eindeutig, ob es sich um zwei kurze Telefonnummern `„2468"` und `„1357"` oder um eine lange Telefonnummer handelt. Codes mit fester Länge hingegen sind immer eindeutig decodierbar, sofern die Länge der Codewörter und der Beginn der Sequenz bekannt sind.

Eine Teilmenge der eindeutig decodierbaren Codes sind die sogenannten Präfixcodes. Zu deren Erklärung bietet sich wieder ein Beispiel mit Telefonnummern an. Angenommen internationale Telefonnummern starten immer mit einem `„+"` in diesem Falle kann aus der Reihe `„+49123456+496543210"` die erste Telefonnummer erkannt werden, sobald das Pluszeichen der zweiten Telefonnummer erscheint. Die zweite Telefonnummer kann noch nicht sicher erkannt werden, da weitere Zeichen folgen können. Es muss auf die Übermittlung des nächsten Pluszeichens gewartet werden. Präfixcodes besitzen dieses Problem nicht. Ihre Definition sagt aus, dass kein Codewort der Präfix eines weiteren Codewortes sein darf. Es kann gezeigt werden, dass dies impliziert, dass ein Codewort sofort erfasst werden kann, sobald es übermittelt ist. Obiges Beispiel kann einfach in einen Präfixcode überführt werden, indem die Telefonnummern umgekehrt übermittelt werden. Somit würde das Pluszeichen immer am Ende einer Telefonnummer übermittelt werden und die Vollständigkeit dieser ist bei Abschluss einer Übertragung sofort sichergestellt.[12]

Ein zentraler Bestandteil der Kompressionstheorie ist die Entropie einer Nachricht. Diese Angabe bezeichnet, wie viel Information eine Nachricht besitzt. Sie kann in Bits ausgedrückt werden.

$$H = \sum_{i \in Z} p_i \cdot \log_2 p_i \tag{3.1}$$

In Formel 3.1 berechnet sich die Entropie aus der Summe des Produktes der Wahrscheinlichkeit eines Zeichens und dessen Informationsgehalt. In der Praxis stellt das die Mindestanzahl an Bits dar, die die Nachricht benötigt, um eindeutig rekonstruierbar zu sein, unter der Annahme, dass zwar die Wahrscheinlichkeit der Zeichen feststeht, sich über deren Abfolge aber keine weitere Aussage treffen lässt. Die Herkunft der Wahrscheinlichkeitswerte ist unterschiedlich. Einfach ist die Bestimmung, wenn die vollständige Eingabe bereits vorhanden ist. In diesem Fall können alle Symbole analysiert werden.

Rein zufällige Folgen von Symbolen sind in der Praxis wiederum selten, häufig sind Strukturinformationen über die Eingabedatenmenge vorhanden. In diesem Fall können nicht Entropie basierte Verfahren stärkere Kompressionen erlauben als durch eine Berechnung nach obiger Formel möglich wäre.

Bereits kurz angesprochen wurden die verlustbehafteten Kompressionsver-

fahren. Bei diesen kann der Ausgangsdatensatz nicht vollständig wiederhergestellt werden. Video-, Audio- und Bildkompressionsverfahren nutzen diese häufig. Die Idee ist das Weglassen von Informationen, die für die menschlichen Sinnesorgane nicht erfassbar sind. Die Kompressionsmöglichkeiten hierbei sind deutlich höher, als in der verlustfreien Kodierung. Beispielsweise kann eine verlustfreie Bildkomprimierung den Speicherbedarf in der Regel nur halbieren, hingegen kann eine verlustbehaftete Kodierung einen Kompressionsfaktor 10 erreichen, ohne dass Qualitätsunterschiede wahrnehmbar sind. Auch die Genauigkeit der Koordinaten eines Knotens zu reduzieren und dabei gegebenenfalls Rundungsfehler hinzunehmen, ist eine Art der verlustbehafteten Datenkomprimierung.[45]

3.4.2 Ausgewählte verlustfreie Codierungen

In diesem Abschnitt werden ausgewählte Kompressionsverfahren vorgestellt. Das erste Verfahren ist die Huffman Codierung. Sie wurde im Jahre 1952 entwickelt und garantiert optimale Codes für bekannte Alphabete mit feststehenden Wahrscheinlichkeiten.[38] Dies bedeutet, dass für eine Nachricht, deren Inhalt eine zufällige Anordnung feststehender Symbole ist, kein Code existiert, welcher die Nachricht in einer geringeren Anzahl an Bits repräsentieren kann. Jeder Huffman Code ist ein Präfixcode und benötigt zur Decodierung das zur Codierung verwendete Wörterbuch. Huffman Codes sind allerdings nicht eindeutig. Es kann weitere Codes geben, welche ebenfalls optimal sind. Die durchschnittliche Länge $\bar{l}$ eines Codewortes kann dabei mittels der Entropie H eingegrenzt werden. Formel 3.2 zeigt, dass im Durchschnitt mindestens die Entropie pro Codewort benötigt wird, jedes Codewort aber nie mehr als ein Bit über der Entropie ist.

$$H < \bar{l} < H + 1 \tag{3.2}$$

Eine Codierung mit dem Huffman Code hat den Nachteil, dass das Wörterbuch beiden Seiten bekannt sein muss. Somit eignet sich der Code besser, je mehr Informationen für ein fixes Wörterbuch übertragen werden können. Sollte es sich ändern, muss diese Änderung entsprechend ebenfalls übertragen werden und sorgt somit für einen gewissen Overhead.

Algorithmus 1 : Aufbau des binären Baumes im Huffman Algorithmus

```
1 Ermittle für jedes Symbol die relative Häufigkeit.
2 Erstelle für jedes Symbol einen Knoten und notiere dort die Häufigkeit.
3 while Anzahl der Bäume >= 2 do
4     Wähle die 2 Teilbäume, mit der geringsten Häufigkeit in der Wurzel.
5     Fasse diese Bäume zu einem neuen Baum zusammen.
6     Notiere die Summe der Häufigkeiten in der Wurzel.
```

Algorithmus 1 zeigt den Aufbau des Wörterbuches in der Huffman Codierung. Für die Erstellung des Wörterbuches wird zuerst ein Baum erstellt. Dieser Vorgang ist in Algorithmus 1 beschrieben. Die Symbole werden zuerst nach ihrer relativen Häufigkeit geordnet, daraufhin wird der Baum von den Blättern aus aufgebaut. Es werden immer die beiden Teilbäume mit der kleinsten Wahrscheinlichkeit zusammengesetzt, bis ein vollständiger Baum erstellt ist. Im fertigen Baum kann Blatt eine 1 oder eine 0 zugeordnet werden. Daraufhin kann

das Wörterbuch erstellt werden, indem der Weg von der Wurzel zu jedem Blatt abgelesen wird. Abbildung 3.3 zeigt ein Beispiel zur Codierung eines Textstrings mit dem Huffman Code.

Für die eigentliche Kodierung kann dieses Wörterbuch verwendet werden. Die Symbole müssen nur nachgeschlagen und ersetzt werden. Selbiges gilt für die Dekodierung, in welcher das Wörterbuch in die entgegengesetzte Richtung angewendet werden kann.

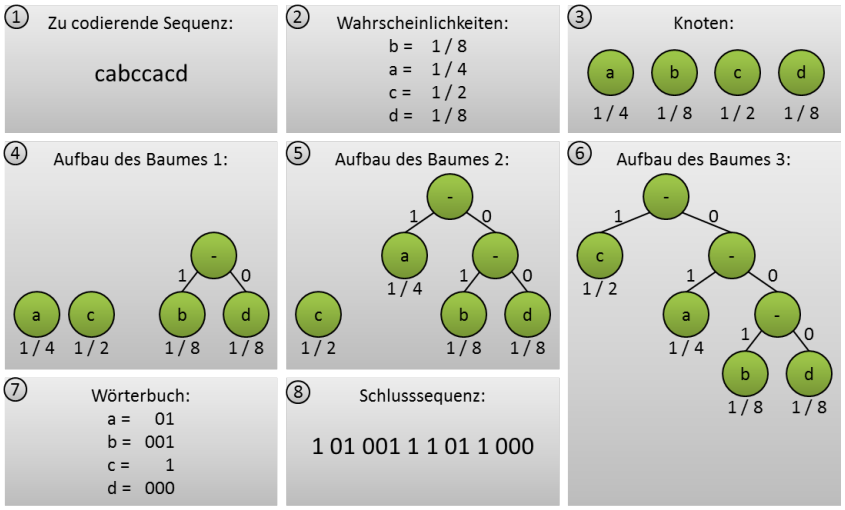


Abbildung 3.3: Ablauf der Huffman Codierung

Einen anderen Ansatz verfolgen die Universal Codes. Diese sind feststehende Codierungsvorschriften für positive, ganzzahlige Werte. Ihr Vorteil ist, dass keine Wörterbücher übermittelt werden müssen, sie können unabhängig von diesen verwendet werden. Demhingegen garantieren sie keine optimale Codierung. Jeder Universal Code hat seine eigenen Eigenschaften und codiert unterschiedliche Datensätze in unterschiedlicher Qualität. Gemeinsam für Universal Codes ist, dass für größere zu codierende Zahlen größere Datenmengen benötigt werden.[37]

Im Folgenden werden zwei Beispiele für Universal Codes vorgestellt. Der Unary Code und der Golomb Code. Der Unary Code ist der einfachste der Universal Codes. Die zu codierende Zahl wird als Anzahl von Nullen übermittelt und die Folge wird mit einer 1 abgeschlossen. Somit kann leicht erkannt werden, dass es sich hierbei um einen Präfixcode handelt, da die 1 immer das Ende einer Zahl bestätigt. Während die Implementierung sehr einfach ist und sehr kleine Zahlen effizient codiert werden können, benötigt der Unary Code sehr große Bitlängen für große Zahlen. Beispielsweise benötigt die Zahl 100 bereits 101 Bits. Somit eignet er sich selten für einen direkten Einsatz. Er wird allerdings von komplexeren Codierungen als Basis verwendet. Der zweite hier vorgestellte Code ist einer dieser: der Golomb-Code. Dieser Code wird beispielsweise bei der verlustfreien Bildkompression verwendet. Sein Ansatz besteht in der Aufteilung der zu komprimierenden Zahl in zwei Zahlen q und r. Hierzu wird ein Parameter m benannt. Damit werden die Ergebnisse der Division durch m und der Rest dieser Division einzeln codiert, wie in Gleichungen bei 3.3 erkennbar.

Diese beiden Werte q und r werden einzeln codiert. Für q wird eine Unary Codierung angewendet, während die Länge von r feststeht und der Wert somit in seiner Binärrepräsentation geschrieben werden kann. Die Länge b bestimmt sich aus nach der Formel $b = \log m$. Für m ergibt sich wiederum, dass es am sinnvollsten ist Zweierpotenzen auszuwählen, da die Bitlänge für r eine ganze Zahl ergibt. Über eine Mappingvorschrift können auch rationale Längen adäquat repräsentiert werden, indem einige Zahlen kürzer Codiert werden als andere. In der Praxis sind meistens nur Codes, deren m eine Zweierpotenz ist, relevant. Es kann erwähnt werden, dass ein Golomb Code mit $m = 1$ äquivalent zu einem Unary Code ist. Der Golomb Code bietet den Vorteil, dass er sowohl kurze Zahlen als auch längere effizient mit wenigen Bits codieren kann. Die exakte Bitlänge hängt dabei stark vom gewählten Parameter m ab. Folgende Graphik 3.4 stellt die beiden Universal Codes gegenüber und bietet Beispiele zu deren Effizienz.

$$\begin{aligned} q &= \lfloor n/m \rfloor \\ r &= n - qm \end{aligned} \tag{3.3}$$

Abbildung 3.4 zeigt 4 Beispiele für die beiden Codes. Es kann erkannt werden, dass der Golomb Code für höhere Zahlen weniger Zeichen benötigt, während für die Zahlen 0 und 1 der Unary Code mit weniger Bits auskommt. Es gibt viele Universal Codes und die Erkenntnis, das bestimmte Codes bestimmte Zahlenbereiche besser codieren als andere, lässt sich auf alle Codes übertragen.

Da die Universal Codes selbst nur ganzzahlige positive Zahlen codieren, müssen negative Zahlen über ein Mapping angegeben werden. Algorithmus 2 zeigt die Codierungs- und Algorithmus 3 zeigt die Decodierungsvorschrift für dieses Mapping.

Zu codierende Zahl	Unary Codierung		Golomb Codierung (m=4)	
0	Nullen		q	0 → 1
	Endung	1	p	0 → 00
	Ergebnis	1	Ergebnis	**1 00**
1	Nullen	0	q	0 → 1
	Endung	1	p	1 → 01
	Ergebnis	0 1	Ergebnis	1 01
5	Nullen	00000	q	1 → 01
	Endung	1	p	1 → 01
	Ergebnis	00000 1	Ergebnis	01 01
10	Nullen	00000 00000	q	2 → 001
	Endung	1	p	2 → 10
	Ergebnis	00000 00000 1	Ergebnis	001 10

Abbildung 3.4: Beispiele des Unary Codes und des Golomb Codes

Ebenfalls ein Problem stellt bei der Verwendung Universal Codes die 0 dar, da einige Codes erst ab der Zahl 1 definiert sind. Sollte sie dennoch benötigt sein, schafft eine Verschiebung der gesamten Zahlenreihe um 1 beim Codieren Abhilfe. Bei der Dekompression muss die Zahlenreihe entsprechend um -1 zurückgeschoben werden.

Alle bisher vorgestellten Codierungsvarianten sind feste Algorithmen, welche keinerlei strukturelle Informationen über die zu codierenden Werte besitzen und somit keine Muster erkennen und nutzen können. Einen einfachen Ansatz zum Ausnutzen dieser liefert das Delta-Coding. Dieses ist dabei nicht mit dem Delta-Code, einem weiteren Universal Code, zu verwechseln. Das Delta-Coding wird vorrangig für auf- oder absteigende Zahlenreihen angewendet. Dabei werden anstatt der Zahl selbst nur die Unterschiede zur jeweils nächsten Zahl gespeichert. Dies soll die zu komprimierenden Zahlen einerseits verkleinern und andererseits sollen auch häufiger dieselben Werte komprimiert werden. Somit kann die Entropie gesenkt werden und die bisherigen Codes können mit weniger Speicherplatz auskommen. Im Delta-Coding muss die vollständige Ziffernfolge seit Beginn des Codierungsvorgangs bekannt sein, ansonsten kann der aktuelle Wert nicht bestimmt werden.

Algorithmus 2 : Codierung

```
1 if n >= 0 then
2     return n × 2 + 1
3 else
4     return n × -2
```

Algorithmus 3 : Decodierung

```
1 if (n mod 2) = 1 then
2     return (n-1) / 2
3 else
4     return n / -2
```

Die wörterbuchbasierten Methoden nutzen ebenfalls Informationen über die Struktur der Symbolanordnung aus. Sie sind speziell für Text geschaffen, da sich in diesem häufig dieselben Buchstabenkombinationen befinden. Texte folgen bestimmten Regeln, diese können durch die Algorithmen ausgenutzt werden um hohe Kompressionsraten zu erreichen. Ein bekannter Vertreter ist der GZIP–Algorithmus. Dieser ist eine Kombination verschiedener Verfahren. Eines dieser Verfahren komprimiert, indem die letzten Zeichen in einem Puffer gehalten werden, welcher nach folgenden Buchstabenkombinationen durchsucht wird. In unten stehendem Beispiel 3.5 kann somit anstatt dem zweiten Hello, eine Referenz gespeichert werden, dass diese Kombination bereits 13 Zeichen früher vorkam und die erkannte Folge dabei 6 identische Zeichen am Stück lang ist.

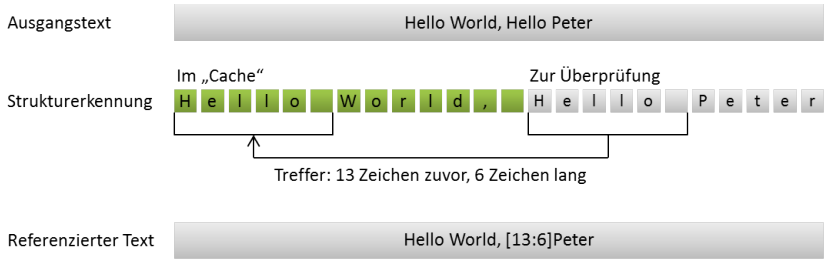


Abbildung 3.5: Einfache Idee der Textkompression

Die Entropie stellt für diese Algorithmen keine Grenze dar. Da Text keine zufällige Anordnung von Symbolen ist, sondern bestimmten Regeln folgt, können diese ausgenutzt werden, um die Dateien stärker zu komprimieren.

3.5 Routenplanung

Zentral für jedes Navigationssystem ist die Berechnung der jeweiligen Route, also des Pfades von einem Startpunkt zu einem Zielpunkt. Im Laufe der Jahre wurden viele Navigationssysteme veröffentlicht. Diese berechnen die Routen nicht immer exakt, da die Berechnung der exakten Route auf längere Distanzen ein nicht zu unterschätzender Aufwand ist. Dieses Kapitel legt die Grundlagen für das Shortest-Path-Problem dar und führt verschiedene Algorithmen ein, welche exakte Lösungen für die zu findenden Wege liefert.

3.5.1 Allgemeine Information und Datenstrukturen

Das Finden der kürzesten Route vom aktuellen Aufenthaltsort zu einem Ziel ist ein Vertreter des Shortest-Path Problems. Durch die steigende Relevanz und dem erhöhten Wunsch nach Navigationssystemen wurde für dieses Problem in den letzten Jahren viel Forschungs- und Entwicklungsarbeit geleistet.

Für die formale Definition ist zunächst die Definition von Graphen allgemein hilfreich. Im Allgemeinen werden die Kartendaten hierbei nach in Gleichung 3.4 vorgestellter Definition als Graphen abgebildet.

$$\begin{aligned} &G = (V, E) \\ &V = \mathbb{N} \\ &E = v_1, v_2 : v_1, v_2 \in V \wedge v_1 \neq v_2 \end{aligned} \tag{3.4}$$

Das Shortest-Path-Problem ist ein Problem der Graphentheorie und sucht den kürzesten Weg zwischen zwei Knoten in einem Graphen. Es ist das Ziel einen Pfad P zu finden, dessen Länge l des Pfades minimal ist. Gleichung 3.5 zeigt die dazugehörigen Formeln zur Bestimmung von P und l, wobei $f(e_i)$ die Länge der Kante ausgibt.

$$\begin{aligned} P &= (e_1, e_2, \ldots, e_n) \in (E_1 \times E_2 \times \cdots \times E_n) \\ l &= \sum_{i=1}^{n-1} f(e_i) \end{aligned} \tag{3.5}$$

Wichtig für die Effizienz vieler Navigationsalgorithmen ist die unterliegende Datenstruktur. Diese wird benötigt, um die Distanz zu verschiedenen Knoten zu halten und das Minimum zu extrahieren. Die relevanten Funktionen hierbei sind:

- Add(Knoten, Priorität) : Void
- ExtractMin() : Knoten
- *DecreasePriority(Knoten, Priorität): Void (Optional)*

Die angewendeten Datenstrukturen sind sogenannte Vorrangwarteschlangen (engl: Priority Queues). Anhand des Namens lässt sich die Aufgabe der Funktion bereits vermuten: Die Add-Funktion fügt ein Knoten mit einer Priorität zu der Warteschlange hinzu. Die ExtractMin-Funktion gibt das Minimum aus und entfernt es aus der Warteschlange. Zuletzt reduziert die DecreasePriority-Funktion

die Priorität des angegebenen Knotens. Diese Funktion ist optional. Sollte sie eine schlechte Performance liefern, oder gar nicht abgebildet werden können, kann diese Funktion auch ignoriert werden und durch ein weiteres Add(Knoten, Priorität) ersetzt werden. Dabei wird der Inhalt der Warteschlange größer und es werden häufig Elemente zurückgegeben, welche zuvor bereits bearbeitet wurden. Für die Funktionsweise von Navigationsalgorithmen ist dies unkritisch, es muss lediglich direkt eine neue ExtractMin-Operation durchgeführt werden. Sie wird in der Folge also häufiger aufgerufen. Hierbei müssen die Kosten, die durch die erhöhte Größe der Warteschlange entstehen, mit der Laufzeit der Funktion abgewogen werden.

Eine häufig verwendete Vorrangwarteschlange ist ein binärer Heap. Seine Idee liegt in einer Struktur eines Binärbaumes, welcher mit Ausnahme der letzten Schicht vollständig aufgefüllt ist. Die letzte Schicht ist linksbündig aufgefüllt. Abbildung 3.6 verdeutlicht in der oberen Hälfte den Aufbau des Baumes. Die Grundbedingung des Heaps ist, dass der Schlüssel jedes Elementes größer oder gleich dem Schlüssel des nächsten Elementes ist. Der Baum ist partiell geordnet. Das bedeutet es besteht von allen inneren Knoten zu deren Wurzeln einen Ordnung, unter den Wurzeln jedoch nicht.[36]

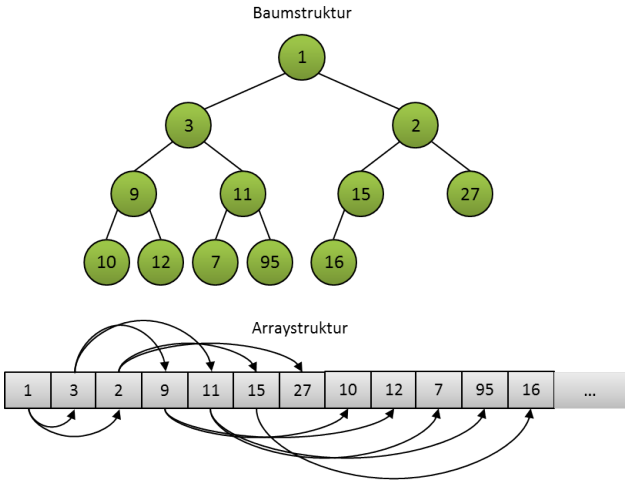


Abbildung 3.6: Beispiel eines Binärbaumes

Um den Zugriff effizient zu gestalten wird ein binärer Heap häufig als Array im Speicher abgebildet. Abbildung 3.6 zeigt im unteren Teil, wie der Baum eines binären Heaps in einem Array abgebildet werden kann. Dabei werden die Positionen der Blätter eines Knotens im Array nach den Formeln $b1 = i * 2$ und $b2 = i * 2 + 1$ berechnet, wobei b1 und b2 die Indexe der Blattknoten sind und i der Index des Ausgangsknotens ist.

- Add : O(log n)
- ExtractMin : O(log n)
- DecreasePriority : O(n)

Vorherige Auflistung zeigt die Laufzeiten der einzelnen Funktionen. Die Add- und die ExtractMin-Methoden benötigen im binären Baum jeweils logarithmische Laufzeit, da lediglich ein Ast des Baumes verändert werden muss. Für die DecreasePriority-Funktion ist lineare Laufzeit notwendig, da zuerst der passende Knoten gefunden werden muss. Da keine strukturellen Informationen, in welchem Ast sich dieser befindet, vorhanden sind, müssen eventuell alle Knoten einzeln durchsucht werden.

3.5.2 Dijkstra-Algorithmus

Als Standard-Algorithmus zur Lösung des Shortest-Path Problems kann der Dijkstra-Algorithmus bezeichnet werden. Sein Prinzip basiert auf dem Überprüfen aller Knoten, welche aus dem bekannten Suchraum erreicht werden können, in der Reihenfolge des nächsten Knotens zum Ausgangspunkt. Dies wird durchgeführt, bis der Zielknoten erreicht ist. Der Suchraum des Algorithmus wächst damit kreisförmig um den Ausgangspunkt an.

Algorithmus 4 : Dijkstra

```
1  Q ← new MinHeap()
2  Q.add(s)
3  while Q ≠ ∅ and Q.peekMin() ≠ t do
4      u ← Q.extractMin()
5      foreach v with (u,v) ∈ E do
6          distance = u.dist + v.length
7          if distance < v.distance then
8              v.distance ← distance
9              v.precedingNode ← u
10             Q.add(v)
```

Algorithmus 4 beschreibt den Dijkstra. Nach einer Initialisierung mit dem Startknoten wird der Algorithmus solange durchgeführt bis das Ziel gefunden ist oder die Vorrangwarteschlange Q leer ist. In letzterem Fall wurde kein Weg gefunden. Für jedes u, also jeden untersuchten Knoten, wird überprüft ob zu den Endknoten seiner angrenzenden Straßen ein kürzerer Weg besteht. Ist das der Fall werden die Ziele zur Vorrangwarteschlange hinzugefügt. Dabei werden alle Knoten in aufsteigender Distanz zum Start bearbeitet, die Distanz jedes untersuchten Knotens ist entsprechend größer oder gleich zum vorherigen untersuchten Knoten.

Der Dijkstra liefert garantiert eine kürzeste Route und hat zusätzlich die weitere Eigenschaft, dass zu jedem Knoten, welcher vollständig bearbeitet wurde, die kürzeste Route gefunden ist. Die Laufzeit des Dijkstra-Algorithmus hängt von der verwendeten Datenstruktur der Priority Queue ab. Mit einem Binärbaum liegt die Zeitkomplexität des Algorithmus bei $O((n + m) * \log(n))$. n bezeichnet die Anzahl an Knoten und m bezeichnet die Anzahl an Kanten. Die Dauer des Algorithmus steigt also überlinear an, das bedeutet beispielsweise, dass für ein Straßennetz mit doppelt so vielen Straßen mehr als die doppelte Zeit benötigt wird.

Der Dijkstra lässt sich auch bidirektional ausführen. Das bedeutet, dass die Suche parallel vom Startknoten und vom Endknoten aus gestartet wird. Der

Algorithmus kann beendet werden, sobald die Summe der Mindestdistanzen eine höhere Distanz liefert als der kürzeste gefundene Weg. Als Mindestdistanz wird in diesem und den folgenden Abschnitten die Distanz bezeichnet, welche in der Vorrangwarteschlange an erster Stelle steht. Der kürzeste Pfad geht dann garantiert über den gefundenen Knoten, an dem sich die beiden Algorithmen treffen, und über dessen Vorgänger zum Startknoten bzw. Nachfolger zum Endknoten lässt sich der kürzeste Weg berechnen. Die Laufzeit ist geringer als beim unidirektionalen Dijkstra. Allerdings müssen während der Ausführung mehr temporäre Daten verwaltet werden.[1]

Algorithmus 5 : Bidirektionaler Dijkstra (gekürzt)

```
1  [...] (Initialisierung)
2  shortestDistance ← ∞
3  while Q1 ≠ ∅ or Q2 ≠ ∅ do
4      u1 ← Q1.extractMin()
5      foreach v with (u1,v) ∈ E do
6          distance ← u1.dist + v.length
7          if distance < v.distanceToStart then
8              v.distanceToStart ← distance
9              v.precedingNode ← u1
10             Q1.add(v)
11             if v.distToT + v.distToS < shortestDistance then
12                 expectedMeetingNode ← v
13                 shortestDistance ← v.distToT + v.distToS
14     [...] (Suche vom Endpunkt)
15     if u1.distToS + u2.distToT > shortestDistance then
           /* Shortest path has been found                    */
16         return expectedMeetingNode
17 return no Result
```

In Algorithmus 5 ist der bidirektionale Dijkstra beschrieben. Es ist erkennbar, dass die einzelne Suche fast gleich wie beim Dijkstra abläuft. Allerdings muss die Terminationsbedingung wie oben beschrieben angepasst werden. Mit fortlaufendem Fortschritt des Algorithmus wird diese Distanz immer größer. Ein Vorteil der bidirektionalen Suche ist die Möglichkeit der Parallelisierung. Beide Suchen können unabhängig voneinander ihre Berechnungen durchführen. Lediglich für die Bestimmung der Termination ist eine Kommunikation nötig. Die größte Verbesserung wird aber erreicht, indem die Anzahl der untersuchten Knoten sinkt. Ein Ansatz hierzu wird im folgenden Abschnitt genauer erläutert.

3.5.3 A*-Algorithmus

Eine ähnliche Idee verfolgt der A-Stern (A*)-Algorithmus. Über eine Heuristikfunktion wird die ungefähre Distanz zum Ziel bestimmt. Anstatt der aktuellen Distanz zum Startknoten wird die geschätzte Gesamtdistanz zum Ziel in die Priority Queue eingefügt. Somit werden vermehrt Knoten verarbeitet, welche in Richtung des Ziels liegen. Knoten in Gegenrichtung werden niedrig priorisiert und somit spät oder im Optimalfall gar nicht abgearbeitet. Algorithmus 6

verdeutlicht die Ähnlichkeit der Dijkstra- und A*-Algorithmen, welche sich in diesem Falle nur in der hinzugefügten Heuristik in Zeile 6 unterscheiden.

Algorithmus 6 : A*

```
1  Q ← new MinHeap()
2  Q.add(s)
3  while Q ≠ ∅ and Q.peekMin() ≠ t do
4      u ← Q.extractMin()
5      foreach v with (u,v) ∈ E do
6          distance = u.dist + v.length +h(v,t)
7          if distance < v.distance then
8              v.distance ← distance
9              v.precedingNode ← u
10             Q.add(v)
```

Die Optimalität des A* ist garantiert, sofern die Heuristikfunktion die eigentliche Distanz nie überschätzt. Mathematisch gesehen ist der Dijkstra ein spezieller A*, bei dem die Heuristikfunktion immer 0 liefert.

In der Praxis ist der A* mit einer geeigneten Heuristik somit in den meisten Graphen schneller als der Dijkstra. Folgende Graphik 3.7 verdeutlicht dies, indem die Suchräume der beiden Algorithmen verglichen werden. Während der A*-Algorithmus sich direkt in Zielrichtung bewegen kann und im Beispiel nur wenige Knoten unnötig verarbeitet, sucht der Dijkstra-Algorithmus alle Knoten kreisförmig ab und bearbeitet in der Graphik alle Knoten. Der bidirektionale Dijkstra ist besser als der reine Dijkstra, allerdings werden weiterhin viele Knoten unnötigerweise berechnet.

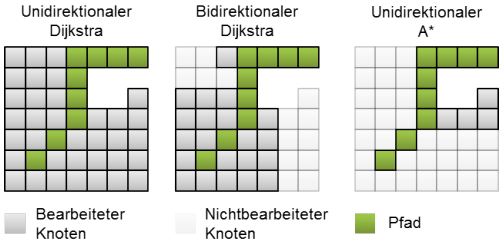


Abbildung 3.7: Vergleich der verschiedenen Suchräume

Die verwendete Heuristik ist ein zentraler Bestandteil des A*-Algorithmus. Die Formeln 3.6 bis 3.11 zeigen die verschiedenen Möglichkeiten auf der Erde die entsprechenden Distanzen zu berechnen. In der Graphik ist a die Distanz zwischen den Latituden und b die Distanz zwischen den Longituden. Die Genauigkeit ist in absteigender Reihenfolge angegeben. Die Qualität der Heuristik wirkt sich direkt auf die Laufzeit des Algorithmus aus. Je näher die ausgegebene Distanz an der tatsächlichen benötigten Distanz liegt, desto weniger Knoten müssen insgesamt überprüft werden, bis die optimale Route gefunden ist. Ergebnisse, welche die Distanz als zu kurz schätzen, sind zwar für die Optimalität erlaubt, sorgen allerdings gegebenenfalls für zusätzliche Knoten, welche überprüft werden müssen.

Alle Heuristiken haben gemeinsam, dass sie die Luftlinie zwischen zwei Knoten schätzen. Die Luftlinie hat die Eigenschaft, dass sie immer kleiner oder gleich dem kürzest möglichen Weg ist. Wie oben beschrieben, ist dies für die Optimalität des A*-Algorithmus notwendig. Diese erste Heuristik berechnet die Distanz der Luftlinie exakt unter Einbeziehung des Erdumfangs. Diese Haversine genannte Funktion ist sehr rechenintensiv, schätzt dafür die Distanz aber sehr gut. Da Straßen nicht der Luftlinie folgen, sondern eventuell Kurven und weitere Umwege beinhalten, kann auch mit dieser Formel die Distanz zum Ziel nicht exakt berechnet werden.

$$\text{Exakt:} h = haversine(punkt_1, punkt_2)^1 \tag{3.6}$$

$$\text{Hypotenuse:} h = \sqrt{a^2 * b^2} \tag{3.7}$$

$$\text{Approximation 1:} h = b + a^2 * 0,428/b \tag{3.8}$$

$$\text{Approximation 2:} h = max(b, 0.918 * (b + a * 0.5)) \tag{3.9}$$

$$\text{Approximation 3:} h = b + a * 0,414 \tag{3.10}$$

$$\text{Manhattan:} h = (a + b)/1,414 \tag{3.11}$$

Die zweite Heuristik vernachlässigt die Erdkrümmung und berechnet die Hypotenuse der beiden Koordinaten. In einem zweidimensionalen Raum wäre dies gleich mit der ersten Heuristik. Die Vernachlässigung der Krümmung verkürzt den Weg zwischen den beiden Koordinaten etwas. Die Abweichung der optimalen Lösung hängt von der Distanz zum Ziel ab. Je größer die Distanz ist, desto höher fällt der Distanzunterschied durch die fehlende Erdkrümmung aus. Für kurze Distanzen ist diese Abweichung vernachlässigbar.

Die folgenden drei Heuristiken sind Approximationen der Hypotenuse. Da das genaue Berechnen dieser mit zwei Quadrat- und einer Wurzelfunktion weiterhin aufwendig ist und viel Rechenleistung benötigt, können diese Schätzungen die Geschwindigkeit deutlich erhöhen. Die Abweichung zur optimalen Hypotenuse lässt die Distanz immer größer schätzen, als die Optimale. Das bedeutet, um sicherzustellen, dass die Distanz nie überschätzt wird, muss durch eine Division durch $(1 + maxAbweichung)$ angepasst werden. Daraufhin liefern diese drei Optimierungen verwendbare Ergebnisse, während die Geschwindigkeit einer einzelnen Berechnung deutlich erhöht wird.

Die letzte Distanz ist die sogenannte Manhattandistanz von der Start- zur Zielkoordinate. Dabei werden die horizontalen und vertikalen Abstände einfach addiert. Um zu garantieren, dass die Distanz nicht überschätzt wird, muss in diesem Fall durch $\sqrt{2}$ dividiert werden.

Für alle Heuristiken gilt zu beachten, dass mit erhöhter Gesamtdistanz des zu berechnenden Weges bei Abweichungen von der korrekten Distanz überproportional mehr Knoten überprüft werden müssen. Die konstante Zeitersparnis einer verbesserten Berechnung, wirkt sich unter Umständen entsprechend gleich mehrfach negativ aus. Einerseits erhöht sich die Anzahl der Knoten wie erwähnt überproportional, durch die ungenaueren Schätzungen und andererseits müssen

[1] Die Haversinefunktion ist eine trigonometrische Funktion zur genauen Berechnung zweier Distanzen auf dem Erdball. Dabei wird die Erdkrümmung miteinbezogen. Nicht in Betracht gezogen werden beispielsweise zusätzliche Distanzen durch mögliche Höhenunterschiede.

die nicht zeitkonstanten Funktionen der Vorrangwarteschlange häufiger aufgerufen werden.

Algorithmus 7 : Bidirektionaler A* (gekürzt)

```
1  [...] (Initialisierung)
2  while Q1 ≠ ∅ or Q2 ≠ ∅ do
3      u1 ← Q1.extractMin()
4      foreach v with (u1,v) ∈ E do
5          distance ← u1.dist + v.length + h(v,t)
6          if distance < v.distanceToStart then
7              v.distanceToStart ← u.dist + v.length
8              v.precedingNode ← u1
9              Q1.add(v)
10             if v.distToT + v.distToS < shortestDistance then
11                 expectedMeetingNode ← v
12                 shortestDistance ← v.distToT + v.distToS
13     [...] (Suche vom Endpunkt)
14     if u1.distToS > shortestDistance and u2.distToT > shortestDistance
        then
         /* Shortest path has been found                              */
15       return expectedMeetingNode
16 return no Result
```

Auch zum A* existiert eine bidirektionale Variante. Diese ist in Algorithmus 7 dargestellt und ähnlich dem bidirektionalen Dijkstra aufgebaut. Sie beinhaltet zusätzlich jeweils die Heuristikfunktion in der Distanzberechnung. Im Gegensatz zum bidirektionalen Dijkstra kann kein vorzeitiges Ende der Suchen bestimmt werden. Es muss gesucht werden, bis beide Suchradien größer sind, als die kürzeste gefundene Distanz. Der bidirektionale A* ist im Gegensatz zum bidirektionalen Dijkstra nicht zwangsläufig schneller als die unidirektionale Variante.

Zwischen dem Dijkstra und dem A* liegen die Unterschiede hauptsächlich in den Suchräumen, während der Dijkstra mit einem runden Suchraum sich gemäßigt dem Ziel nähert, kann der A* durch die direkteren Suchen sich viel Arbeit sparen. Dass die Untersuchung eines Knotens durch die zusätzliche Berechnung der Heuristik mehr Zeit benötigt fällt hierbei kaum ins Gewicht.

Die bidirektionalen Varianten haben Vor- und Nachteile. Während der bidirektionale Dijkstra den Suchraum im Straßennetz praktisch immer deutlich verbessert, kann beim bidirektionalen A* diese Aussage nicht getroffen werden. Hier ist es möglich, dass ein unidirektionale A* weniger Knoten abarbeiten müsste. Vorteil einer bidirektionalen Variante ist immer die mögliche Parallelisierung der beiden Suchvorgänge.

3.5.4 Contraction Hierarchies

Ein weiterer Algorithmus sind die Contraction Hierarchies (CH). Diese verfolgen eine andere Idee als die bisherigen beiden. Während einer Vorbereitungsphase wird der Straßengraph analysiert und mit sogenannten Shortcuts versehen. Wie der Name vermuten lässt sind das Abkürzungen mittels denen zwischen Orten eine direktere Route möglich ist. Die Distanz ist aber weiterhin die des kürzest möglichen Weges. Zusätzlich werden den Knoten dabei Level zugeordnet. Vereinfacht ausgedrückt entspricht ein hoher Level einer hohen Wichtigkeit des Knotens.

Algorithmus 8 : Vereinfachte Vorberechnung des CH

Data : $G = (V, E), <$
foreach *u 2 V aufsteigend nach* $<$ *geordnet* **do**
  foreach *(v, u) 2 E mit* $v > u$ **do**
    foreach *(u,w) 2 E mit* $w > u$ **do**
      if h_v, u, w_i *der einzige kürzeste Pfad von v nach w ist* **then**
        E:= E [(v,w) (mit Länge $w(v, w) := w(v, u) + w(u, w)$)

CH ist ein hierarchiebasierter Navigationsansatz. Über ein Ordnen aller Knoten in einer eindeutigen Ordnung werden diese nacheinander einzeln kontrahiert. Wie im Algorithmus 8 beschrieben, werden dabei alle umliegenden Knoten verglichen, ob ein Shortcut eingefügt werden kann. Aus diesem Vorgehen leitet sich der Name „Kontraktionshierarchie“ (engl. Contraction Hierarchy) ab. Die verwendete Ordnung ist für die Korrektheit von CH irrelevant, hat aber einen Einfluss auf die Performance.[14]

$$G_{\uparrow} = (V, E_{\uparrow}) \; with \; E_{\uparrow} := (u, v) \in E : u < v \tag{3.12}$$

$$G_{\downarrow} = (V, E_{\downarrow}) \; with \; E_{\downarrow} := (u, v) \in E : u > v \tag{3.13}$$

Mit dieser Vorberechnung kann auf dem entstandenen Graphen ein abgewandelter bidirektionaler Dijkstra-Algorithmus angewendet werden. Durch die angewendete Ordnung, kann ein Teil des bidirektionalen Algorithmus nur auf dem Vorwärtsgraphen, und der zweite Teil kann nur auf dem Rückwärtsgraphen ausgeführt werden. Die beiden Graphen werden im Folgenden auch als Forward- (FW-) und Backward- (BW-) Graph bezeichnet.

Wie in Formel (3.12) ersichtlich, ist der Vorwärtsgraph definiert als alle Kanten, bei denen der Startknoten in der Ordnung vor dem Endknoten auftritt. Und der Rückwärtsgraph in Formel (3.13) ist entsprechend als Teilmenge des Gesamtgraphen, bei dem der Endknoten nach dem Startknoten auftritt, definiert.

Der bidirektionale Dijkstra-Algorithmus ist in Algorithmus 9 dargestellt und kann auf diesem Graphen fast wie üblich ablaufen. Lediglich die Terminationsbedingung muss abgeändert werden. Die Suche kann nicht beendet werden, sobald sich die beiden Mindestdistanzen überschneiden. Sie muss durchgeführt werden, bis beide Suchen die Distanz des kürzesten gefundenen Weges überschreiten. Zuvor ist die Optimalität nicht garantiert.

Der A* for CH–Algorithmus in Algorithmus 10 basiert auf dem bidirektionalen A* und benötigt dieselbe Anpassung wie der bidirektionale Dijkstra-

Algorithmus. Dabei kann die Terminationsbedingung sogar belassen werden, es müssen lediglich die Überprüfungen der Level eingefügt werden.

Algorithmus 9 : Dijkstra for CH (gekürzt)

```
 1 [...] (Initialisierung)
 2 while Q1 ≠ ∅ or Q1 ≠ ∅ do
 3     u1 ← Q1.extractMin()
 4     foreach v with (u1,v) and u1.level >= v.level ∈ E do
 5         distance ← u1.dist + v.length
 6         if distance < v.distanceToStart then
 7             v.distanceToStart ← distance
 8             v.precedingNode ← u1
 9             Q1.add(v)
10             if v.distToT + v.distToS < shortestDistance then
11                 expectedMeetingNode ← v
12                 shortestDistance ← v.distToT + v.distToS
13     [...] (Rückwärtssuche)
14     if u1.distToS > shortestDistance and u2.distToT > shortestDistance
        then
         /* Shortest path has been found                              */
15       return expectedMeetingNode
16 return no Result
```

Beide Algorithmen beinhalten in der Ausgabe des kürzesten Pfades möglicherweise Shortcuts. Da diese keine realen Straßen sind, muss überprüft werden, ob einer oder mehrere solcher Shortcuts in der Ausgabe vorhanden ist. Ist dies der Fall, müssen diese durch die Kanten ersetzt werden, aus denen die Abkürzung entstanden ist. Dies können wiederum Shortcuts sein. Entsprechend muss diese Aktion wiederholt werden, bis alle Shortcuts eliminiert sind.[14]

Die Nachteile dieser CH-Algorithmen liegen in den benötigten Daten. Außer den Daten, welche auch zum Rendern der Karte genutzt werden, verwenden die zuvor vorgestellten Algorithmen keine zusätzlichen Informationen. Somit kann der benötigte Speicherplatz so gering wie möglich gehalten werden. Demgegenüber steht die Laufzeit bei der sehr viele Knoten betrachtet werden müssen und die mit größeren Distanzen überproportional stark ansteigen. Auch ist es notwendig, dass der gesamte Graph im Voraus bekannt sein muss, so dass dieser berechnet werden kann. Da sich Straßen nicht häufig ändern, ist die im Prinzip unkritisch, allerdings erhöht sich der Speicherbedarf der Navigationsinformationen um die vorberechneten Daten. Des Weiteren muss beachtet werden, dass für verschiedene Prioritäten in der Navigation unterschiedliche Datensätze benötigt werden. So muss ein Datensatz für den kürzesten Weg nach Distanz und ein Datensatz für den schnellsten Weg bereitgestellt werden.[2] Ebenfalls ist die Möglichkeit, dynamisch auf Stau oder gesperrte Straßen zu reagieren, eingeschränkt.

Der Vorteil der Algorithmen mit vorbereiteten Daten ist die deutlich beschleunigte Laufzeit. Die Routen können in wenigen Millisekunden berechnet

[2]Dies trifft in erster Linie auf Kraftfahrzeuge zu, da deren Bewegungsgeschwindigkeit vom Tempolimit abhängig ist. Fußgänger oder Fahrradfahrer sind hiervon weniger betroffen.

werden und sie beschleunigen hierbei die bisherigen Algorithmen um ein Vielfaches. Speziell bei CH ergibt sich durch Einführung des Hierarchiekonzeptes ein zusätzlicher, immenser Vorteil. Dabei lassen sich die Daten passend für den Algorithmus anordnen, so dass selbst mit begrenztem Hauptspeicher und der Notwendigkeit auf den externen Speicher zurückzugreifen, effiziente Ergebnisse erreicht werden können.

Algorithmus 10 : A* for CH (gekürzt)

```
1  [...] (Initialisierung)
2  while Q1 ≠ ∅ or Q1 ≠ ∅ do
3      u1 ← Q1.extractMin()
4      foreach v with (u1,v) and u1.level >= v.level ∈ E do
5          distance ← u1.dist + v.length + h(v,t)
6          if distance < v.distanceToStart then
7              v.distanceToStart ← distance
8              v.precedingNode ← u1
9              Q1.add(v)
10             if v.distToT + v.distToS < shortestDistance then
11                 expectedMeetingNode ← v
12                 shortestDistance ← v.distToT + v.distToS
13     [...] (Rückwärtssuche)
14     if u1.distToS > shortestDistance and u2.distToT > shortestDistance
        then
          /* Shortest path has been found                          */
15        return expectedMeetingNode
16 return no Result
```

Kapitel 4

Problemanalyse

Im Fokus des vierten Kapitels steht die Analyse, welche Anforderungen an die Anwendung für die Smartwatch gestellt werden und welche Teilaufgaben dabei anfallen, um eine gutes Ergebnis zu erreichen. Hierzu wird die Problemstellung selbst analysiert und alle benötigten Anforderungen werden herausgearbeitet. Diese werden mit den Stärken und Schwächen, welche sich aus der Nutzung der Smartwatch ergeben, abgeglichen. Des Weiteren werden die vorhandenen Best Practices für Benutzerschnittstellen und APIs vorgestellt und dabei für die Problemstellung passende Kandidaten präsentiert, welche die Umsetzung erleichtern können.

4.1 Problemstellung und wichtige Komponenten

Dieser Abschnitt teilt die Zielstellung der Routenplanung in Teilprobleme ein und analysiert die funktionalen und nicht-funktionalen Anforderungen. Graphik 4.1 listet eine Übersicht über die wichtigsten Anforderungen für die Anwendung auf.

Die Hauptaufgabe der Anwendung ist eine Punkt zu Punkt Navigation von der aktuellen Position zu einem beliebigen Ziel innerhalb Deutschlands mittels Android Smartwatches. Ebenfalls muss dies unabhängig vom Mobiltelefon möglich sein. Das bedeutet, die Verbindung zwischen den Geräten kann getrennt sein und eine Navigation muss trotzdem durchgeführt werden können, sofern die Uhr einen GPS-Empfänger besitzt. Ist dies nicht der Fall, sollte eine Verbindung mit dem Mobiltelefon aufgenommen werden, um die Nutzung der Anwendung zu ermöglichen. Wie erwähnt ist eine Navigation innerhalb von Deutschland ausreichend, trotz allem soll es möglich sein jederzeit weitere Länder nach Bedarf hinzuzufügen.

Der Benutzer soll verschiedene Möglichkeiten zur Zielauswahl haben. Es müssen sowohl Spracheingaben auf der Uhr als auch textuelle Eingaben über das Mobiltelefon möglich sein. Bei diesen Eingaben muss nach Straßennamen und Orten gesucht werden können. Ebenfalls muss zwischen gleichnamigen Orten und Straßen adäquat unterschieden werden können. In anderen Worten der Benutzer muss sein Ziel einfach finden können. Weitere Erkennungsmerkmale wie Hausnummern oder Sehenswürdigkeiten sind hilfreich aber nicht notwendig.

Zeitmäßig muss die Anwendung in weniger als 5 Sekunden Routen im Umkreis von 20 Kilometern berechnen können. Für Distanzen bis 50 Kilometer sollten 10 Sekunden nicht überschritten werden. Längere Distanzen können optional unterstützt sein, sofern der Benutzer in der Usability der Anwendung hierdurch einen Vorteil erhält. Das bedeutet: Es ist nicht zielführend Routen über große Distanzen zu erlauben, wenn der Benutzer hierbei minutenlang warten muss. Als Basis für diese Zeitmessungen gilt die SWR 50, welche im folgenden Abschnitt vorgestellt wird. In diesem Zusammenhang sei erwähnt, dass andere aktuelle Smartwatchmodelle ähnlich leistungsfähig sind.

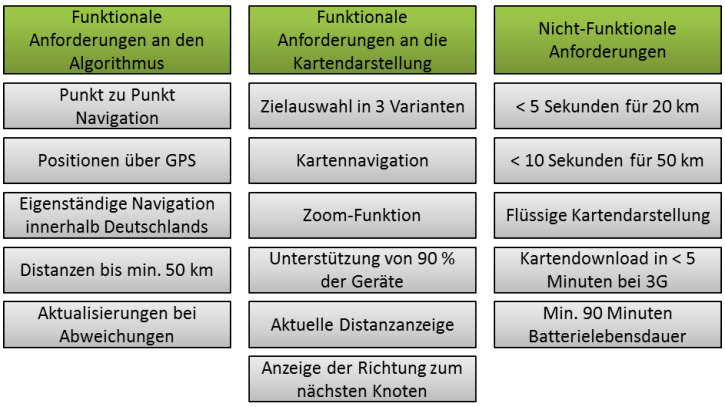


Abbildung 4.1: Übersicht über die Anforderungen

Sollte der Anwender von der vorgeschlagenen Route abweichen, so muss der angezeigte Weg eventuell entsprechend angepasst werden, so dass es weiterhin der kürzeste ist. Der Zeitrahmen sollte die Dauer des initialen Berechnens der Route nicht überschreiten.

Zusätzlich müssen auf graphischer Ebene verschiedene Elemente implementiert werden. Für eine gute Usability wird eine Kartendarstellung benötigt, welche Positionen, Straßen und Straßennamen beinhaltet. Diese Darstellung der Straßen muss nicht exakt sein, es sollten allerdings optisch keine auffälligen Unterschiede zu Originalkarten bestehen. Ebenfalls sind Kanten bei der Abbildung von Kurven prinzipiell erlaubt, sie müssen allerdings in einem akzeptablen Rahmen bleiben. Diese bewusst offene Definition der Anforderung ermöglicht es durch gezielte Vereinfachungen der Karte eine höhere Performance zu erzielen. Neben den Straßen müssen keine weiteren Objekte auf der Karte erkennbar sein.

Dementsprechend ebenfalls offen ist die Definition des erlaubten Speicherplatzes. Eine fixe Anzahl an erlaubten Megabyte (MB) ergibt wenig Sinn, da dies einen starken Einfluss auf die Optik hätte. Es gilt hier den Speicher möglichst gering zu halten, um dem Benutzer zu erlauben in akzeptabler Wartezeit mittels des Mobiltelefons neue Karten zu laden, während er unterwegs ist. Bei Vorhandensein eines 3G-Telefonnetzes müssen die größten vorhandenen Karten in höchstens 5 Minuten geladen werden können.

Die Straßen werden in fünf Kategorien eingeteilt: Autobahnen, Bundesstraßen, Landstraßen, Wege und Feldwege. Die Basis für diese Einteilung sind die Straßenkategorien aus den OSM-Daten. In Verbindung mit der Ermöglichung einer Zoom Funktion können die kleineren Straßen entsprechend bei höheren

Distanzen nicht dargestellt werden, um die Übersichtlichkeit zu gewährleisten.

Es müssen dem Benutzer Informationen über die verbleibende Distanz zum Ziel angezeigt werden. Ebenfalls müssen die Straßen, welche auf dem berechneten Weg liegen, hervorgehoben werden und das Scrollen auf der Karte muss dem Benutzer möglich sein. Dabei ist zwischen zwei Modi zu unterscheiden. In einem Modus verfolgt die Kartenansicht die aktuelle Position des Benutzers, diese wechselt beim Scrollen automatisch in eine fixe Anzeige der ausgewählten Position. Ferner wird eine Möglichkeit gefordert, zurück zum Ausgangsmodus zu wechseln.

Eine weitere nicht-funktionale Anforderung betrifft die Batterielebensdauer. Bei ständiger Bewegung und gewöhnlich häufiger Bedienung der Anwendung muss die Navigation mindestens 90 Minuten lang nutzbar sein.

Die letzte abgebildete Anforderung, dass 90 % der Geräte abgedeckt werden müssen, bedeutet in erster Linie, dass alle offiziellen Android Wear Versionen, und ebenso rechteckige, als auch runde Bildschirme unterstützt werden müssen. Es ist typisch für die Entwicklung für Androidgeräte, dass nicht 100 % als Ziel angegeben wird, da die Geräte teilweise sehr unterschiedlich sind und mit sogenannten Custom Modifications auch sehr unterschiedliche Gestaltungen des Betriebssystems möglich sind. Diese Anforderung ist nicht sofort bei einer ersten Veröffentlichung der Anwendung erforderlich. Aufgrund mangelnder Testgeräte muss es zu Beginn auf allen vorhandenen Geräten lauffähig sein und kann nachträglich bei auftretenden Fehlern korrigiert und an weitere Geräte angepasst werden.

4.2 Eigenschaften der SWR 50

Mit den Kenntnissen über die Anforderungen kann zuerst die Uhr auf ihre Eigenschaften analysiert werden, um im Folgenden geeignete Algorithmen und Vorgehensweisen für die speziellen Gegebenheiten angehen zu können. Im Vergleich zu Computern oder Smartphones ist die Performance der Uhren in Batteriekapazität und Leistung geringer. So ist der Prozessor trotz vorhandener Multicorefähigkeit geschwindigkeitstechnisch deutlich langsamer als bei den verwandten Geräten. Ebenfalls ist der Speicherplatz deutlich geringer bemessen und speziell der Random Access Memory (RAM), auf welchen eine Anwendung zugreifen darf, ist unter Umständen sehr gering. Auch der interne Speicher ist auf wenige Gigabyte (GB) beschränkt.

Dafür ist die Portabilität von Smartwatches größer. Sie kann genutzt werden, ohne hierbei eine Hosentasche oder etwas Ähnliches zum Transport zu benötigen.

Weitere Schwächen der Uhr sind die Abhängigkeit vom Mobiltelefon. Trotz eigener Hardware und vieler eigener Sensoren sind nicht alle Funktionen selbst vorhanden. Beispielsweise sind die Uhren von der SIM-Karte des Telefons abhängig, um Telefonie- und Internetfunktionen aufrecht zu halten. Auch die Wireless Local Area Network (WLAN)-Funktionalität kann nur mithilfe eines Smartphones genutzt werden, selbst wenn eine eigene WLAN-Karte verbaut ist.

Wie bei jedem mobilen Gerät ist auch die Batterielebensdauer ein beschränkender Faktor. Während im reinen Uhrzeitanzeigemodus die Uhr ein bis zwei Tage ohne zwischenzeitlichem Aufladen der Batterie genutzt werden kann, so verkürzt sich dieser Zeitraum deutlich, sobald weitere Funktionen und spezi-

ell rechenintensive Aufgaben durchgeführt werden. Die SWR 50 (auch Sony Smartwatch 3 genannt) ist in Abbildung 4.2 wurde 2014 auf der IFA von Sony präsentiert. Sie bietet vergleichsweise viele Funktionen und eine gute Basis, um unabhängig des Telefons zu arbeiten. Graphik 4.3 liefert eine Übersicht über die Fähigkeiten dieser Smartwatch. Basierend auf Android 4.3, wobei es mittlerweile ein Update für Android 5 gibt, ist ein Vier-Kern Prozessor mit einer Taktrate von 1,2 Gigahertz eingebaut. Der interne Speicher beträgt insgesamt 4 GB, hiervon bleiben neben dem Betriebssystem ca. 2,5 GB übrig.

Abbildung 4.2: Aussehen der SWR 50

Der Arbeitsspeicher beträgt insgesamt 512 MB. Eine Anwendung kann im Normalfall 64 MB hiervon nutzen. Über das „largeHeap“ Attribut kann der verfügbare Arbeitsspeicher auf 128 MB erweitert werden, dies sorgt in Android 5.1 allerdings für eine regelmäßige automatische Beendigung der Anwendung während deren Nutzung. Diese Einstellung ist also zumindest zurzeit nicht zu empfehlen.

An weiteren Funktionen bietet die SWR 50 sowohl eine WLAN als auch eine Bluetoothverbindung um in erster Linie mit dem Mobiltelefon zu kommunizieren. Ebenfalls sind verschiedene Sensoren in die Uhr eingebaut, unter anderem ein Beschleunigungsmesser, ein Kompass und ein GPS Empfänger. Dieser ist zentral für eine eigenständige Navigationsanwendung ohne Mobiltelefon.

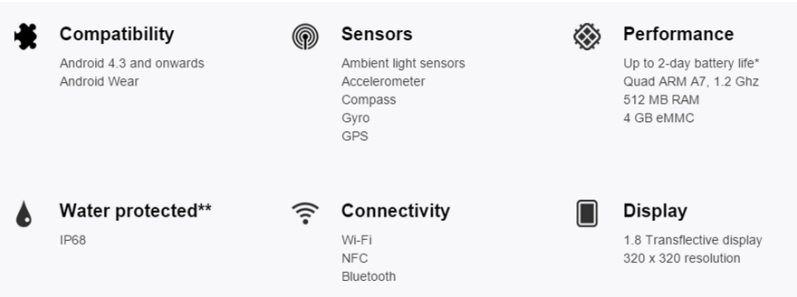


Abbildung 4.3: Funktionen der SWR 50

Das Display ist 1,8″ groß und bietet eine Auflösung von 320 * 320 Pixeln. Es besitzt die übliche Multitouchfunktion und kann somit die Eingaben mehrerer Finger parallel erkennen. Zusätzlich befindet sich ein mechanischer Knopf an der rechten Seite des Displays, welcher in seiner Funktionalität vorbelegt ist und nicht zur freien Programmierung zur Verfügung steht.

Der folgende Abschnitt führt verschiedene Möglichkeiten und Standards ein wie die Funktionalität der Uhr zur Benutzerinteraktion verwendet werden kann.

4.3 Standards für Smartwatch Benutzeroberflächen

Zentral für die Benutzerakzeptanz einer Anwendung ist eine gute Usability und eine intuitive Benutzerführung. Entsprechend werden an dieser Stelle die wichtigsten graphischen Funktionen analysiert und die entsprechenden Entwicklungsempfehlungen auf der Smartwatch nahegelegt. Dabei wird von Uhren ausgegangen, wie sie im Androidumfeld üblich sind: mit reiner Touchfunktion. Das Userinterface einer Uhr besteht aus dem Bildschirm mit einer Größe von wenigen Quadratzentimetern. Entsprechend sind die Anzeige- und Eingabemöglichkeiten eingeschränkt. Ein mechanischer Knopf, welcher in der Regel an einer Seite des Gehäuses befindet, führt zurück zur Anzeige der Uhrzeit.

Hauptbenutzeraktion auf der Uhr ist, neben dem Klicken auf einzelne Felder, das sogenannte Wischen. Also das Bewegen des Fingers von einer Seite des Gerätes zur Nächsten und damit das Verschieben des Inhaltes auf der Uhr. In dem Android SDK sind die meisten Funktionen für die Smartwatch auf dieses Wischen ausgelegt, welches sowohl in horizontaler, als auch in vertikaler Richtung durchgeführt werden kann. Beispielsweise ist die vollständige Menüsteuerung entsprechend angepasst: Aus dem Hauptbildschirm kann über ein Wischen nach links die Anwendungsübersicht aufgerufen werden, während eine Bewegung nach oben aktuelle Benachrichtigungen von Anwendungen an den Benutzer anzeigt, um zwei Beispiele zu nennen. Neben der Steuerung zwischen den Menüs sind auch die Einstellungen selbst größtenteils mit dieser Geste zu bedienen. Über das Navigieren von oben nach unten können die einzelnen Menüpunkte angezeigt und ausgewählt werden. Dies gilt grundsätzlich für alle Arten von Listen, wie sie bei Drop-Down-Menüs oder anderen Auswahlmöglichkeiten erscheinen. Wobei hierfür zusätzlich die WearableListViews vorhanden sind, welche für Smartwatchanwendungen optimierte Auswahlmenüs sind.

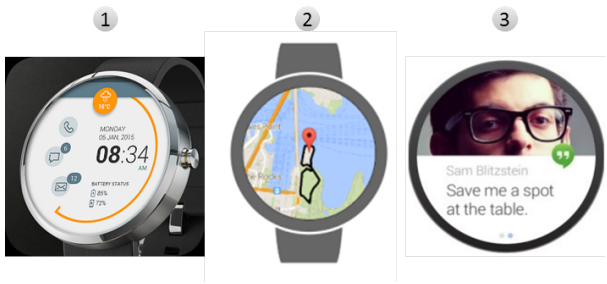


Abbildung 4.4: Beispiel der Wear Oberflächenarten

Um Anwendungen für Smartwatches zu gestalten, gibt es verschiedene Möglichkeiten. Sehr wichtig für die Uhren sind die Watchfaces. Sie sollen die Uhrzeit darstellen und werden in der Smartwatch somit dauerhaft angezeigt. In aktuellen Anwendungen bestehen diese aus einem Canvas, einer Zeichenfläche auf welchen beliebige Elemente gezeichnet werden können. Es ist möglich mit diesen Watchfaces interaktive Anwendungen zu entwickeln, wobei deren Hauptaufgabe die Anzeige der Uhrzeit ist. Dies ist in Abbildung 4.4 zu erkennen. Unter Nummer 1 ist dort ein Watchface zu erkennen, welches verschiedene alltägliche Funktionen bietet.

Die zweite Möglichkeit Daten anzuzeigen, sind Notifications. Dabei handelt es sich um Informationen, die dem Benutzer bei bestimmten Ereignissen angezeigt werden. Diese Funktion wird beispielsweise von Nachrichtendiensten zur Anzeige empfangener Nachrichten verwendet. Sie kann aber auch für Wetterinformationen oder weitere für Anwender interessante Informationen verwendet werden. Ein Beispiel hierfür ist in zuvor genannter Abbildung unter Nummer 3 zu erkennen.

Die dritte Möglichkeit sind traditionelle Anwendungen, welche auf der Uhr laufen. Diese können dort gestartet und ausgeführt werden. Hierbei handelt es sich um vollwertige Applikationen, denen grundsätzlich dieselben Möglichkeiten zur Verfügung stehen wie bei anderen Android-Anwendungen. Eine Abhängigkeit vom Telefon besteht nicht direkt. Trotzdem bieten die Google-Bibliotheken bei vielen Funktionen ein Fallback auf das Telefon an. Beispielsweise kann das GPS des Telefons verwendet werden, da Smartwatches häufig keinen eigenen Empfänger besitzen. Dies ist für den Entwickler mit ähnlichem Aufwand zu realisieren, wie Zugriff auf den Empfänger des eigenen Gerätes. Diese Bibliotheken werden im folgenden Abschnitt 4.4 detailliert gesprochen. Die bereits erwähnte Google Maps Oberfläche ist eine solche Anwendung. Sie ist in Nummer 2 der Abbildung erkennbar.

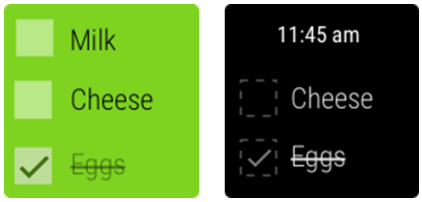


Abbildung 4.5: Beispiel einer Liste im normalen und im Ambientmodus

Diese Anwendungen werden im Vollbildmodus ausgeführt. Das graphische Element, auf welchem die Oberfläche platziert werden kann, heißt Activity. Diese Activities können auf zwei Arten gestaltet werden. Im Standardverhalten werden Activities mit einem Wischen nach rechts beendet und die Anzeige der Uhr wird wieder aufgerufen. Anwendungen werden ebenfalls beendet, sobald für eine bestimmte Zeit keine Benutzerinteraktion stattfindet. Sollte die Möglichkeit beispielsweise für die Kartennavigation erwünscht sein, alle Benutzerinteraktionen durch die Anwendung zu verwenden, kann dies angegeben werden, so dass die Anwendung durch längeres Drücken des Fingers stattdessen geschlossen werden kann. Diese sogenannten „Always-On“ Activities ermöglichen vollständige Touch-Interaktion wie das Verschieben der Karte und zwei-Finger-Zoom-Gesten.

Ein zentrales Element zur Steuerung von Wear-Aktivitäten sind Auswahllisten, diese treten bei der Bedienung entsprechend häufig auf. Abbildung 4.5 zeigt eine solche in verschiedenen Modi. Grundsätzlich können beliebig viele Elemente in einer Liste angeboten werden. Für eine optimale Übersicht sollte die Anzahl an Elementen möglichst gering gehalten werden, da der Benutzer in der Regel nur drei parallel auf dem Bildschirm erkennen kann. Die angesprochenen Modi bilden eine zusätzliche Funktion der Android Wear ab. Die angesprochenen „Always-On“ Aktivitäten sollten nach einer kurzen Zeit ohne Aktivität des Benutzers in einen Modus übergehen, welcher Energie spart und die aktuelle Uhrzeit anzeigt. Dies ist in der Regel eine spezielle Oberfläche, welche die eige-

nen Informationen im Hintergrund darstellt und zusätzlich die Uhrzeit anzeigt. Der Benutzer kann mit einem einfachen Klick oder je nach Systemeinstellung auch durch eine Bewegung der Uhr wieder in den normalen Modus wechseln.[19]

4.4 Analyse nach verwendbaren APIs

Mit dem Finden der Problemstellen in der zu entwickelnden Software kann nach unterstützenden APIs gesucht werden, welche einen Teil der Funktionalität bereitstellen und die Entwicklung damit vereinfachen können. Google hat im Betriebssystem Android verschiedene solcher APIs integriert. An dieser Stelle sind verschiedene Kandidaten aufgeführt, welche für eine Verwendung in der Anwendung geeignet sind, und eine Abwägung deren Nutzens.

Für das Bestimmen von GPS Positionen ist die für Android Mobiltelefone übliche Herangehensweise über einen direkten Zugriff auf den LocationProvider nicht empfohlen. Da nur wenige Smartwatches überhaupt ein GPS System besitzen wird stattdessen der Fused GPS Provider vorgeschlagen. Dieser ermöglicht automatisch die Nutzung der vorhandenen Positionsbestimmungen von Smartwatch und Smartphone. Somit kann ohne gesonderte Betrachtung die Smartwatch die Navigation alleine ausführen, sofern GPS eingebaut ist und ansonsten wird auf das Smartphone zurückgegriffen.

Ein zentraler Bestandteil vieler Wearableanwendungen ist die Kommunikation zwischen Smartphone und Smartwatch. Eine eindeutige Empfehlung von Google[22] sagt aus, dass eine der drei bereitgestellten Bibliotheken verwendet werden sollen, welche für verschiedene Anwendungsfälle optimiert sind. All diese APIs sind in den Google Play Services enthalten und können kostenfrei genutzt werden.[1]

Die erste Bibliothek ist die DataAPI. Sie ist für die Synchronisation zwischen den beiden Geräten gedacht. Dabei übernimmt die API die sichere Zustellung von Nachrichten. Ist eins der beiden Geräte nicht erreichbar, wird die Übermittlung automatisch zu einem späteren Zeitpunkt durchgeführt. Dabei können bis zu 100 Kilobyte (KB) große Anhänge an die Nachricht geschrieben werden. Diese API ist damit sehr gut für das Übermitteln von Einstellungen und anderen wichtigen Informationen geeignet. Die garantierte Zustellung benötigt allerdings zusätzliche Ressourcen der Geräte. So müssen alle gesendeten Nachrichten für eine gewisse Zeit auf dem sendenden Gerät gepuffert werden, bis die Zustellung bestätigt ist.

Die zweite Bibliothek ist die MessageAPI. Diese kann verwendet werden um bis zu 100 KB große Anhänge zwischen den Geräten zu versenden. Im Gegensatz zur DataAPI ist die Ankunft einer Nachricht hierbei allerdings nicht garantiert, dafür kann die MessageAPI performanter für den Austausch vieler Nachrichten sein.

Die dritte Bibliothek ist die ChannelAPI. Gegenüber den beiden vorherigen Bibliotheken unterstützt sie das direkte Verwenden von Streams anstatt von Anhängen. Somit können beliebige Informationen über einen Channel ausgetauscht werden. Dabei ist es ebenfalls möglich den Channel in beide Richtungen zu verwenden. Die API bietet auch eine systemgesteuerte Möglichkeit zur Dateiüber-

[1] Auch wenn es aktuell bei den hier vorgestellten Funktionen nicht zu erwarten ist: Google behält sich das Recht vor für viele seine APIs jederzeit Kosten zu erheben, wie es beispielsweise bei der Google Maps API mit kurzer Ankündigungsfrist geschehen ist.

tragung. In der Implementierung ist sie am aufwendigsten, da alle Fehlerfälle wie beispielsweise Verbindungsabbrüche korrekt abgefangen werden müssen. Sie ist speziell zur Dateiübertragung und zum Austausch von allen anderen großen Nachrichten geeignet.

Für die Anwendung sind somit die DataAPI und die ChannelAPI geeignet. Während die reinen Steuerungsnachrichten wie die Auswahl neuer Ziele über die sicher übermittelten Nachrichten der DataAPI übermittelt werden können, ist für das Senden der Dateien die ChannelAPI am sinnvollsten anzuwenden. Die Entscheidung mittels welcher API Dateien entfernt werden sollen, ist schwieriger zu treffen. Hier würde eine einfache Nachricht mit der DataAPI genügen, um allerdings die Konsistenz in der Dateihandhabung zu wahren wird auch hier die ChannelAPI verwendet.

Ein weiteres Problem ist die Suche nach Straßennamen und Orten. Mit der PlacesAPI bietet Google eine zurzeit kostenfreie Lösung an, welche es ermöglicht über einen Aufruf die vollständige Oberflächen- und Logikverarbeitung an die API abzugeben und am Ende der Benutzerinteraktion eine Position mit deren Koordinaten zu bekommen. Dabei sieht der Benutzer ein Textfeld, in welches er einen Text eingeben kann und ähnlich wie in der Google-Suche werden schon nach den ersten Zeichen Vorschläge eingeblendet. Es ist somit sowohl für den Entwickler als auch für den Benutzer eine komfortable Lösung, um Positionen mit den entsprechenden GPS-Koordinaten zu erhalten. Eine Internetverbindung am Mobiltelefon wird hierbei vorausgesetzt. Wie erwähnt ist diese API für beliebig viele Zugriffe kostenfrei nutzbar. Allerdings hat Google bestimmte Grenzen an täglichen API Abrufen bestimmt, ab welchen zusätzliche Maßnahmen ergriffen werden müssen. Beispielsweise muss bei über 1000 täglichen Abrufen zu Verifikationszwecken eine Kreditkartennummer hinterlegt sein.

Die Nutzung dieser Programmierschnittstellen erleichtert die gesamte folgende Entwicklung erheblich, da diese Funktionen bereits vollständig verwendet werden können und schon in der Entwurfsphase für viele Vereinfachungen sorgen.

Kapitel 5

Systementwurf

Der Fokus dieses Kapitels liegt auf der Architektur des Systems und der Herangehensweise bei der Auswahl verschiedener Designentscheidungen. Abschnitt 5.1 beschreibt die Systemarchitektur im Gesamten. Es bietet einen Überblick über alle Komponenten der Anwendung und deren Aufgaben.

Drei zentrale Aspekte, die für diese Arbeit entscheidend sind, werden im Folgenden erörtert. Abschnitt 5.2 geht auf das Management der Kartendaten ein. Diese benötigen verschiedene Arbeitsschritte von der Extraktion aus dem OSM-Daten über deren Kompression und Übermittlung bis zum Zeichnen auf die Karte und der Nutzung als Basis für die Navigationsalgorithmen. Sowohl die Gesamtübersicht als auch den Aufbau der verschiedenen komprimierten Datenformate werden dargestellt.

Abschnitt 5.3 befasst sich mit der Auswahl des optimalen Routingalgorithmus. Es werden verschiedene Alternativen mittels eines Testaufbaus gegenübergestellt, womit die beste Variante ermittelt werden kann. Dieser Abschnitt erörtert auch die Art, wie die Daten während der Berechnung im Arbeitsspeicher gehalten werden und geht dabei sowohl auf den verwendeten Cache, als auch die Datenstruktur aller für den Algorithmus benötigten Dateien ein.

Der letzte Abschnitt analysiert finanzielle Möglichkeiten und verschiedene Alternativen die Anwendung zu vermarkten. In diesem Zusammenhang werden auch die Gedanken zur Veröffentlichung im Google PlayStore vorgestellt. Diese sind wichtig, da ein möglicher Benutzer sich auf dieser Webseite entscheidet, ob er die Anwendung herunterlädt.

5.1 Gesamtsystemarchitektur

Um die Gesamtanwendung effizient entwickeln zu können, muss die gesamte Architektur in kleinere Teilsysteme eingeteilt werden. Die entstehenden einzelnen Komponenten sollten dabei so weit wie möglich unabhängig voneinander sein, um deren eigenständige Entwicklung zu ermöglichen und die Komplexität damit zu senken. Die Einteilung in drei große Komponenten lässt sich leicht vornehmen: Smartwatch, Smartphone, Server.

Die Smartwatch selbst übernimmt die Navigations- und Anzeigeaufgaben. Dies ist die Hauptkomponente, da sich der Routingalgorithmus und die Anzeige der Karte hier befinden. Zusätzlich müssen die Daten aus dem entwickelten

Format ausgelesen werden.

Das Smartphone übernimmt die Verwaltung der Anwendung und ermöglicht dem Benutzer die Installation und Löschung von Kartendaten auf der Uhr. Ebenso übernimmt das Smartphone das Managen vorbereiteter Ziele auf der Uhr. Die hierfür benötigte Funktionalität bietet die Uhr nicht selbst beziehungsweise nur eingeschränkt, entsprechend wird das Mobiltelefon an dieser Stelle unterstützend einbezogen.

Der Server übernimmt das Auslesen und Vorbereiten der Navigationsdaten und stellt diese zum Download bereit. Diese Funktionen sind weder auf der Uhr, noch auf dem Mobiltelefon sinnvoll. Die originalen Kartendaten sind deutlich zu umfangreich für eine direkte Verarbeitung auf den mobilen Endgeräten, so dass diesen der Zugriff auf aufbereitete Daten zur Verfügung gestellt wird. Im Gegensatz zu den anderen Komponenten hat der Benutzer keine direkte Interaktion mit dem Server.

Abbildung 5.1 zeigt die Hauptfunktionen, die sich innerhalb der einzelnen Komponenten befinden. Eine Verbindung zwischen zwei Komponenten bedeutet, dass zwischen diesen ein direkter Kontroll- oder Datenfluss stattfindet.

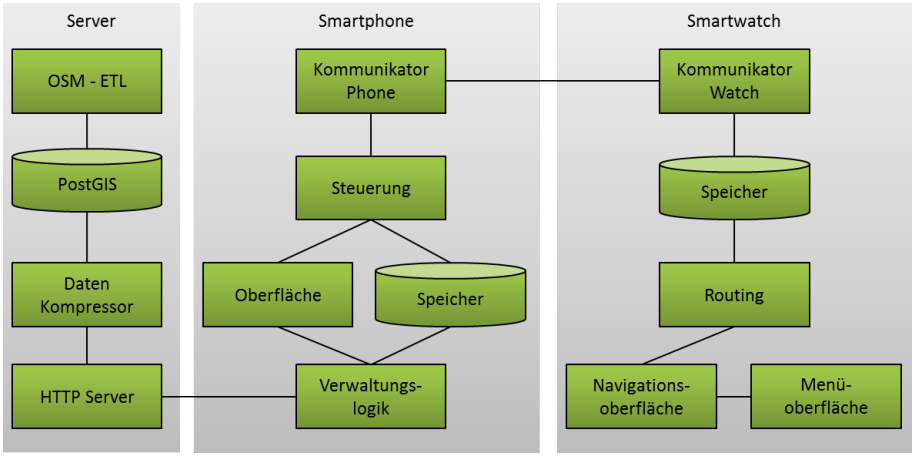


Abbildung 5.1: Gesamtsystemarchitektur

Der Server beinhaltet als erste Komponente den OSM - ETL. Unter dem Begriff Extract Transform Load (ETL) wird die Verarbeitung von Quelldaten verstanden, welche angepasst werden und dann in erwünschter Form in ein Zielsystem geladen werden. Entsprechend liest dieser die OSM-Dateien ein, um die benötigten Kartendaten zu gewinnen. Dabei werden die Straßendaten und Regionsdaten extrahiert und bearbeitet. So werden die Straßen den Regionen zugewiesen und die Knoten, Kanten und Wegeverbindungen aus den OSM-Daten müssen zueinander hergestellt werden. Zuletzt lädt dieser Prozess die Daten in eine Datenbank. Bei dieser handelt es sich um eine PostgreSQL-Datenbank mit PostGIS Erweiterung. Diese Erweiterung erlaubt das effiziente Speichern und Verarbeiten von geometrischen und somit auch von geographischen Informationen in einer Datenbank. Zur Ermöglichung einer CH-Funktionalität ist auch deren Vorberechnung der Kartendaten in diesem Modul enthalten.

Der nächste Block ist der Daten Kompressor. Zweck dieses Moduls ist die möglichst starke Verkleinerung der Kartendaten. Dies ermöglicht einen schnel-

len Download der Daten auf das Mobiltelefon und einen schnellen Transfer vom Mobiltelefon zur Uhr. Die komprimierten Daten werden einerseits nach Ländern aufgeteilt und zusätzlich nach Bundesstaaten. Diese werden auf einem Webserver platziert, welche die einzelnen Bundesstaaten über das Hypertext Transfer Protocol (HTTP)-Protokoll zum Download anbietet.

Im Mobiltelefon ist eines der zentralen Elemente die Anzeige. Die Entkopplung von Anzeigefunktion und Aufgabenfunktion ist geeignet, um mögliche Anpassungen an der Anzeige ohne zusätzlichen Aufwand vornehmen zu können. Es müssen Anzeigen für den Datendownload und -transfer und für die Zielauswahl vorhanden sein.

Das zweite Modul ist die Steuerung der Anwendungslogik. Diese benötigt eine Kommunikation mit dem Webserver und lädt Dateien auf das Smartphone und verwaltet diese. Dabei muss es die Funktionalität für die Anzeige bereitstellen. Ebenso verwaltet es die Ziele und speichert die getroffene Auswahl des Benutzers sowohl auf dem Mobiltelefon als auch auf der Uhr.

Das Smartphone beinhaltet einen lokalen Speicher. Da ein Mobiltelefon nicht zwangsweise mit der Smartwatch verbunden ist oder nachträglich mit weiteren verbunden werden kann, werden alle Daten für eine spätere Synchronisation vorrätig gehalten. Somit können diese durchgeführt werden, ohne dass eine zusätzliche Internetverbindung benötigt wird.

Das letzte Modul auf dem Mobiltelefon übernimmt die Kommunikation mit der Uhr. Es beinhaltet die Verwaltung der Kartendaten, welche das Senden und Löschen dieser handhabt.

Passend zu diesem Modul besitzt die Uhr einen Kommunikator, der die Nachrichten des Mobiltelefons empfängt und verarbeitet. Er kümmert sich direkt um das Speichern und Löschen sowohl von Kartendaten als auch von Zieldaten. Diese Daten werden ebenfalls in einem Speicher gespeichert. Wie auf dem Mobiltelefon wird der Speicher die Kartendateien auf dem Filesystem ablegen.

Das Routingmodul übernimmt das Auslesen der gespeicherten Kartendaten und die Berechnung der schnellsten Wege. In diesem Zusammenhang werden Cachingalgorithmen verwendet, um die Effizienz zu optimieren.

Die Anzeige der Benutzeroberfläche ist wiederum in zwei Module eingeteilt. Das erste Modul übernimmt die Kartenrenderung und die Anzeige des Overlays auf die Karte innerhalb der Navigation. Auch müssen die Benutzereingaben geeignet abgefangen werden und gegebenenfalls in Koordinaten umgerechnet werden.

Außerhalb der Navigation übernimmt das zweite Modul, die Menüoberfläche, die Führung des Benutzers zur Auswahl des erwünschten Ziels.

5.2 Design der Datenverarbeitung und des Datenfluss

Das Thema der folgenden Abschnitte sind die Daten der Anwendung. Zu Beginn wird festgelegt, welche Daten für die Anwendung benötigt werden. Daraufhin wird die grundsätzliche Struktur der Kartendaten bestimmt. Das beinhaltet die Zuordnung, welche Inhalte in welchen Dateien gespeichert werden und es umfasst ebenfalls grundsätzliche Strukturen innerhalb dieser Dateien. Der dritte Abschnitt erörtert die Kompression der Daten und beschreibt detailliert den Aufbau, so dass möglichst geringe Datenmengen entstehen.

5.2.1 Festlegung benötigter Daten

Um die vorgenommenen Datentransformationen zu analysieren, werden zuerst die benötigten Daten erfasst. Folgende Graphik liefert hierzu eine Übersicht. Hierbei werden alle für das Routing und für das Rendering benötigten Daten aufgelistet, ebenso wie einige Metadaten. Eine weitere Aufgabe welche Informationen benötigen könnte, ist das Suchen nach Orten. Hierfür wurde in Abschnitt 4.4 festgelegt, dass die Google PlacesAPI verwendet wird. Somit werden keine zusätzlichen Gebiets- und Metainformationen benötigt.

Abbildung 5.2: Darstellung der benötigten Daten

Die Informationen, welche für das Rendern benötigt werden, liefern die Basis für den Datensatz. Zwingend benötigt werden die Koordinaten aller Knoten und die Start- und Endpunkte der zu zeichnenden Kanten. Des Weiteren werden auch die Straßennamen dem Benutzer angezeigt, entsprechend müssen diese in der Datenbasis vorhanden sein. Auch die Kategorie eines Weges, also ob es sich beispielsweise um eine Autobahn handelt oder um eine lokale Straße, ist abzuspeichern. Diese Informationen werden zur passenden Darstellung der Wege benötigt.

Als nächstes werden den gesammelten Daten die Informationen hinzugefügt, die die Navigationsalgorithmen benötigen. Alle Algorithmen haben gemeinsam, dass zusätzlich die Information benötigt wird, welche Straßen in welche Richtung befahrbar sind. Ansonsten kommen sowohl die Dijkstra, als auch die A*-Algorithmen ohne zusätzliche Daten aus. Hingegen benötigt CH zusätzliche vorberechnete Kanten, die Shortcuts. Diese müssen entsprechend dem Datensatz hinzugefügt werden, sofern dieser Algorithmus verwendet wird. Die hierfür benötigten Informationen sind einerseits die zusätzlichen Kanten und andererseits

muss zu jedem Knoten das dazugehörige Level angegeben sein.

Es existieren weitere Anforderungen an den Datensatz, die keine zusätzlichen Daten benötigen, aber Einfluss auf die Struktur der Daten haben. Der Datensatz muss die Möglichkeit bieten effizient den nächsten Knoten zu einer Koordinate zu finden. Dies wird beispielsweise zur Bestimmung des Startknotens des Navigationsalgorithmus benötigt. Des Weiteren muss die Welt in einzelne Staaten und in einer zweiten Granularität in einzelne Bundesländer eingeteilt sein.

Sowohl die Informationen für das Rendern, als auch die Informationen für die Algorithmen haben gemeinsam, dass nicht davon ausgegangen werden kann, dass alle in den RAM passen. Entsprechend muss eine Möglichkeit vorhanden sein, um einzelne Blöcke von Daten einzulesen. Dies muss eine geeignete Struktur sein, um weiterhin effizient suchen zu können.

5.2.2 Grundsätzliche Struktur der Kartendaten

Die gesamten Daten sind in verschiedene Datensätze eingeteilt. Jeweils ein Datensatz pro Bundesland besteht aus drei verschiedenen Dateien. Die erste Datei speichert die eigentlichen Karteninformationen, die zweite Datei speichert die dazu gehörigen Straßennamen und die dritte Datei speichert Metadaten, welche für einen effizienten Zugriff auf die beiden ersten Dateien benötigt werden. Zuletzt gibt es zusätzlich einen globalen Metadatensatz, welcher Information über den gesamten Datenbestand des Servers enthält.

Zentral gibt es eine Einteilung der gesamten Weltkarte mittels eines Gitternetzes eingeteilt in einzelne Zellen. Die Rastergröße wird dabei so gewählt, dass pro Breitengrad am Äquator jeweils 20 Zellen möglich sind und die Zellen sollen in etwa quadratisch sein. Das bedeutet, dass eine Zelle etwas mehr als 5 km auf 5 km groß ist. Die Zellengröße erweist sich als guter Mittelweg zwischen geringer Größe, für schnelles Einlesen einzelner Zellen und großer Größe, um wenig Metadaten verwalten zu müssen. Jede Zelle muss ihre Identifikationsnummer (ID) aus einer Position statisch berechnen können, dafür wird folgende Formel verwendet.

$$ID = \lfloor 20 * lat \rfloor * (360 * 20) + \lfloor 20 * lon * \cos(lat) \rfloor \tag{5.1}$$

Der Wert der Latitude muss entsprechend um die Zellengröße von 20 multipliziert werden. Zusätzlich müssen alle Änderungen der Latitude eine größere Auswirkung haben als jede mögliche Änderung der Longitude, welche von -180° bis 180° reichen und ebenfalls mit Faktor 20 multipliziert wird. Somit muss die Latitude mit dem Wert (360 * 20) multipliziert werden. Für die Berechnung der Zelle der Longitude ist dies nicht notwendig, dafür muss beachtet werden, dass mit größerer Entfernung vom Äquator würde eine Zelle nach obiger Definition immer kleiner werden. Somit muss die Anzahl an Zellen entsprechend verringert werden. Hierfür wird zusätzlich die Korrektur um den Kosinus der Latitude benötigt. Dies führt dazu, dass alle Zellen ungefähr gleich groß sind, während einige IDs ungenutzt bleiben. Es sei erwähnt, dass an der (theoretischen) Datumsgrenze einige Zellen ebenfalls etwas kleiner sein können. Da diese mitten im Pazifischen Ozean verläuft, ist dies irrelevant.

Der globale Metadatensatz hat die Aufgabe, dass sowohl aus der ID eines Knotens, als auch aus einer Position bestimmt werden kann, in welcher Datei sich dieser Knoten befindet. Da sich in einer Datei immer ein Bundesland

befindet, reicht es aus das entsprechende Bundesland zu bestimmen.

Um dieses Ziel zu erreichen, wird eine Liste aller vorhandenen Länder und Staaten mit deren Namen und deren kleinster und größter Knoten–ID abgespeichert. Jedes Bundesland bekommt einen eigenen ID-Bereich zugewiesen und zwischen den einzelnen Bereichen dürfen keine Überschreibungen existieren. Dies muss bei der ID-Zuweisung beachtet werden. Somit kann auf diesem Metadatensatz das Bundesland, in welchem sich ein Knoten befindet, anhand dessen ID einfach bestimmt werden. Es ist lediglich eine binäre Suche notwendig, um innerhalb der heruntergeladenen Bundesländer das korrekte finden zu können, da die Daten sortiert hinterlegt werden können.

Für die Suche nach Koordinaten wäre es am naheliegendsten über eine Geometriebibliothek ein Multipolygon zu benutzen. Da die Polygone für einzelne Länder aber teilweise aus mehreren Tausend einzelnen Punkten bestehen, benötigt dies sowohl sehr viel Speicherplatz als auch vergleichsweise lange, um die einzelnen Punkte zu berechnen. Dabei genügt es, die Koordinaten einzelnen Zellen zuzuordnen, in welchen alle Knoten nach dem nächstliegendsten durchsucht werden können. Dies sorgt dafür, dass zusätzlich im Metadatensatz alle Zellen mitgespeichert werden, welche ein Land beinhaltet oder schneidet.

Diese Zellen-IDs werden ebenfalls im Speicher für die Wege verwendet. Für jede Zelle wird hierbei ein entsprechender Datensatz mit den beinhalteten Wegen angelegt. Dabei gibt es Wege, welche die Grenze zwischen Zellen überqueren. Diese werden immer in die Zelle des Startknotens gelegt. Die Ordnung der Knoten-IDs innerhalb eine Staates erfolgt damit nach Zellen-ID. Die Start- und Endknoten-ID jeder Zelle werden in der Metadatendatei des einzelnen Bundesstaates abgelegt. Somit kann aus einer Knoten-ID einfach die entsprechende Zelle bestimmt werden. Innerhalb einer Zelle sind die Knoten in einem Array gespeichert, sodass durch einen Abzug der Startknoten-ID der Zelle der Eintrag gefunden werden kann. Dieser Eintrag beinhaltet direkt alle benötigten Informationen über die Straße außer den Straßennamen.

Da das Auslesen der Metadaten und das Einlesen der zu verwendenden Codierungen Zeit benötigt werden hiervon 25 Elemente in einem Least Recently Used (LRU)-Cache gehalten. Wie zuvor erläutert, existiert ein Metadatensatz pro Bundesland. Somit sind 25 Metadatensätze bei weitem genug, um lokale Anfragen abzudecken, während gleichzeitig nur wenig Arbeitsspeicher verbraucht wird.

Da die Straßennamen häufig dieselben sind, werden diese nur als Referenz in den Straßenspeicher hinterlegt. Somit muss ein Straßenname in jeder Zelle nur einmal gespeichert werden. Diese Namen sind in einer zusätzlichen Datei pro Bundesland abgelegt.

5.2.3 Möglichkeiten zur Kompression von Kartendaten

Die in den Anforderungen spezifizierte möglichst geringe Übertragungsdauer kann durch eine auf dieses Problem optimierte Kompressionsstrategie in Angriff genommen werden. Durch die Verwendung von verschiedenen Komprimierungsverfahren unter Ausnutzung bekannter Muster innerhalb der Daten lässt sich die Datenmenge stark reduzieren. Die Kompression hilft einen Datensatz für ein einzelnes Bundesland im ein oder zweistelligen MB-Bereich zu halten und somit schnelle und effiziente Downloads zu ermöglichen. Der größte Teil der Speichererspарnis kann dabei in der Datei mit den Kartendaten und in der Datei mit

den Straßennamen erreicht werden. Entsprechend wird an dieser Stelle die Komprimierung dieser beiden Dateien erläutert. In den folgenden Paragraphen wird zusätzlich die Kompression der Auflistung von Zellen, über welche sich ein Land erstreckt, erläutert.

Eine reine Auflistung hierfür würde ebenfalls sehr viel Platz und bei einer binären Suche entsprechend viele Schritte benötigen. Eine einfache Komprimierung dieser Daten liegt darin, dass bei aufeinanderfolgenden Zellen-IDs nur die erste ID gespeichert wird und daraufhin die Anzahl der direkt folgenden Zellen. Am Beispiel von Deutschland kann die Datenmenge somit von 12 009 Integer Werten auf 448 Integer Werte reduziert werden. Das bedeutet eine Ersparnis von über 96 % und gegenüber der unkomprimierten Speicherung der Liste ist das eine Ersparnis von über 99,6 % an Werten gegenüber den 124 615 Koordinaten welche sich in dem Multipolygon befinden, wobei eine Koordinate sogar noch komplexer abzuspeichern ist, als einfache Integerwerte. Diese Integerwerte können wiederum mittels einer bitweisen Codierung um weitere 75 % der Datenmenge reduziert werden. Eine große Kompression an dieser Stelle ist sehr hilfreich, da diese Daten immer vorhanden sein müssen, um die jeweils korrekte Datei auswählen zu können und dem Benutzer gegebenenfalls vorschlagen zu können, welches Land er herunterladen muss. Vor der Benutzung der Navigationsanwendung muss diese zentrale Datendatei heruntergeladen werden.

Die dazugehörige detaillierte Kompressionsvorschrift lässt sich aus Abbildung 5.3 ablesen. Es werden zwei unabhängige Coder verwendet. Als Coder wird in diesem Abschnitt ein Algorithmus zur Codierung von Daten bezeichnet. Der erste betrachtet die Information der Zellen-ID und der zweite betrachtet jeweils wie viele Zellen am Stück vorhanden sind. Für beide wird eine Delta-Codierung verwendet, es werden also nur die Unterschiede zwischen den Werten codiert. Während die Zellen-IDs stetig aufsteigend sind und hier nur positive Unterschiede codiert werden müssen, müssen bei den Längen auch negative Zahlen berücksichtigt werden. Die Differenzcodierung lohnt sich trotzdem, da Länder in der Regel im Groben rechteckige oder runde Formen haben und somit die Zahlen außen klein sind und ansteigen und ca. ab der Mitte wieder kleiner werden. Der `WrapperDeltaCoder` übernimmt die Analyse der Daten ob negative Werte vorhanden sind und passt die Codierung an.

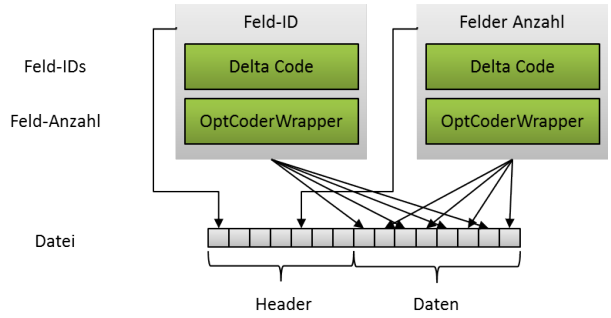


Abbildung 5.3: Kompression der Zelleninformationen

Nach der Differenzcodierung müssen die Unterschiede in Bits codiert werden, hierfür wird der `OptCoderWrapper` verwendet. Dieser beinhaltet eine Analyse der eingehenden Werte und wählt den optimalen Präfixcode aus. In der Auswahl

hiervon befinden ein Huffman Coder mit verschiedenen Varianten das Wörterbuch abzuspeichern, mehrere Universal Codes, ein `OptGolombCodeWrapper` und zuletzt ein Coder mit feststehender Bitlänge, welcher für jeden Wert die minimale Anzahl an Bits verwendet, um die größte Zahl zu codieren. Der genannte `OptGolombCodeWrapper` wählt den bestmöglichen m-Wert für einen möglichen Golomb Code aus. Somit ist mittels dieser Wrapperkombination eine platzeffiziente Speicherung der Daten garantiert.

Da die Datenmenge feststeht, kann während der Kompression in einer vorbereitenden Analyse die am besten geeignetste Codierungsvorschrift festgelegt werden. Für unterschiedliche Daten können unterschiedliche Coder geeignet sein. Somit ist es notwendig in einem Header am Anfang der Datei die verwendeten Coder zu beschreiben. Dieser Header beinhaltet alle Informationen darüber, welche Coder verwendet wurden und zusätzlich alle Informationen, die dieser Coder benötigt, um die Daten korrekt wiederherstellen zu können. Beispielweise speichert der Huffman Coder an dieser Stelle das verwendete Wörterbuch.

In der Metadatendatei werden die Offsets für die Startpunkte der einzelnen Zellen gespeichert. Dies ist nötig, dass direkt zum Startbyte einer Zelle gesprungen werden kann. Die Codierung muss entsprechend beim Codieren sicherstellen, dass die einzelnen Zellen in sich geschlossene Informationsblöcke beinhalten. Zum Beispiel kann es vorkommen, dass das letzte Byte eventuell nicht vollständig gefüllt ist. In diesem Fall müssen die verbleibenden Bits mit Nullen aufgefüllt werden.

Für eine geeignete Komprimierung ist ein zentrales Kriterium die Sortierung. Der Wert nach welchem sortiert ist, kann deutlich platzsparender codiert werden. Ein angewendetes Delta-Coding auf diesen Wert erzeugt kleine Zahlen, die somit nur wenige Bits benötigen. Die geeignetste Sortierung ist zuerst nach der Zelle und daraufhin nach dem Wert der Latitude. Die Knoten-IDs können nach dieser Sortierung neu vergeben werden. Somit sind gleich zwei Werte sortiert und profitieren von den Vorteilen.

Abbildung 5.4 zeigt die verschiedenen Kompressionsvorschriften für die einzelnen Datentypen. Jeweils bekannt an dieser Stelle ist die jeweilige Zellen-ID, somit muss diese nicht extra gespeichert werden. Begonnen wird mit den Startknoten. Diese werden mit zwei Codern codiert. Diese codieren jeweils den ersten Wert einer Zelle mit einem anderen Coder als alle folgenden Werte. Dies ist notwendig, da der erste zu codierende Wert nicht von vorherigen abhängig sein darf. Entsprechend ist hier keine Differenzcodierung möglich, sondern die Zahl muss vollständig geschrieben werden. Alle folgenden Werte können mit einer Differenzcodierung gespeichert werden. Dieses Kriterium erfüllt der Differenzcoder zwar alleine, allerdings sind die Startknoten sehr strukturiert, indem der Wert immer um kleine Werte ansteigt. Ein Huffman Coder kann diese Struktur optimal abbilden. Entsprechend ist diese Aufteilung getätigt, um diesen effizienten Huffman Coder nutzen zu können, während die Offsetwerte in einem extra Code geschrieben werden können, welche weniger Strukturinformationen besitzen und somit ein Universal Code diese geeigneter codieren kann.

Der Endknoten wird ebenfalls auf zwei Coder verteilt. Je nachdem, ob der Endknoten innerhalb der aktuell codierten Zelle liegt oder in einer anderen. Für alle die in der aktuellen Zelle liegen, werden häufig kurze Zahlen erwartet, während für alle anderen Endknoten vermutlich lange Zahlen codiert werden müssen. Somit kann mehr Speicherplatz gespart werden, als das zusätzliche Bit benötigt, welches darauf hinweist, welcher Coder angewendet wird.

Die Codierung für die Länge eines Weges benötigt nur eine Codierung, für diesen Wert gibt es keine Strukturen, welche ausnutzbar wären. Ebenfalls wird an dieser Stelle keine Differenzcodierung angewendet. Dieser erreicht die hohe Streuung der Weglängen keinen Vorteil.

Die Koordinaten werden einzeln nach Latitude und Longitude codiert. Dabei werden die Fließkommazahlen zuerst in Ganzzahlen umgewandelt. Dies übernimmt der `GeoToIntCoder`. Während die Umrechnung bei der Longitude eine einfache Multiplikation mit anschließender Rundung ist, findet bei der Latitude zusätzlich eine Anpassung zu der Erdkrümmung statt. Würde diese immer mit demselben Multiplikationsfaktor multipliziert, wie die Longitude, würde bei höheren Longituden die Latitude immer genauer werden und somit Speicherplatz verschwendet werden. In der Anwendung ist ein Meter als Genauigkeit ausgewählt.

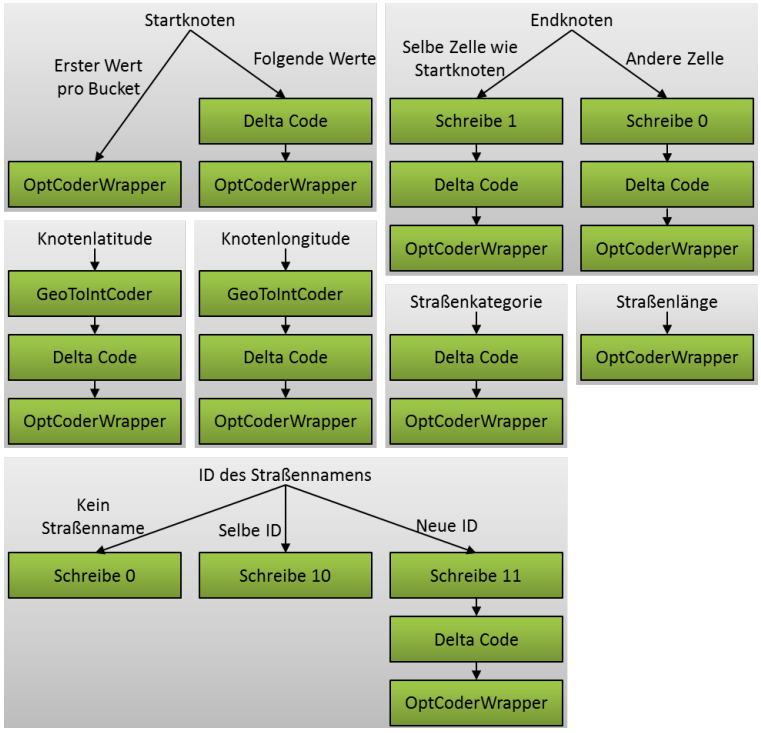


Abbildung 5.4: Kompressionsvorschrift der Kartendaten

Zuletzt wird eine Referenz auf den Namen der Straße codiert. Dies ist die komplexeste Codierungsvorschrift, da zwei häufige Optionen direkt verarbeitet werden. Sollte kein Straßenname vorhanden sein, wird eine 0 geschrieben und sollte der Straßenname mit dem vorherigen übereinstimmen wird eine 1 und eine 0 geschrieben. Sollten beide nicht zutreffen, sondern es ist ein neuer Straßenname, wird eine 11 geschrieben und anschließend wird die entsprechende Nummer codiert. Diese ist häufig aufsteigend, deshalb lohnt sich der Delta-Coder an dieser Stelle, allerdings ist diese komplexe Umsetzung an dieser Stelle hilfreich, da somit diese beiden sehr häufig vorkommenden Sonderfälle abgefangen werden können und mit wenig Bits abgebildet werden.

Bei allen genannten Codern kann der zuvor vorgestellte OptCoderWrapper verwendet werden, um die bestmögliche Codierung zu finden.

Die Straßendateien werden einfacher codiert. Der Coder ist ebenfalls dynamisch gehalten. Es ist zu erwarten, dass der OptCoderWrapper häufig den Huffman Code auswählen wird, da ein kleines Wörterbuch für viele Straßen verwendet werden kann. Der Einsatz der zeitintensiven Suche nach dem optimalen Code ist weiterhin sinnvoll, da diese Kosten einmalig beim Vorberechnen entstehen. Es wird also die Übertragung eines kleinen Wörterbuches benötigt, um viele Daten optimal übertragen zu können. Das Startbyte zu jeder Straße befindet sich in der Metadatendatei und der Name einer Straße endet, sobald das Null-Zeichen decodiert wird.

Die somit erreichte Datenkompression erzielt gegenüber einer einfachen Kodierung, welche die Zahlen jeweils als Integerwerte mit einer Länge von 1 bis 4 Byte je nach Typ speichert, eine hohe Kompressionsrate. Im Vergleich mit den 968 MB, die die unkomprimierte Variante für Deutschland benötigen würde, können über 89 % gespart werden, sodass der vollständige Datensatz für Deutschland nur 104 MB inklusive der Metadaten groß ist. Somit ist erkennbar, dass dies ein zentrales Erfolgskriterium für eine in der Praxis nutzbare Anwendung ist, die unkomprimierten Daten könnten nur mit langen und großem Speicherbedarf genutzt werden. Es ist anzumerken, dass eine größere Komprimierung möglich wäre, dies allerdings die Performance beim Einlesen der Daten negativ beeinträchtigen würde und somit nicht sinnvoll ist. Beispielsweise würde eine Vergrößerung der Zellengröße zwar die Dateigröße verkleinern, dafür einige Berechnungen auf diesem Datensatz verlangsamen. Eine detaillierte Aufstellung der Dateigrößen der einzelnen Bundesländer folgt in Kapitel 7.4.

5.3 Auswahl des Routingalgorithmus

Die Auswahl des richtigen Algorithmus ist wichtig für eine qualitativ hochwertige Anwendung. Mit der richtigen Kombination aus schnellen Berechnungen und wenig benötigten Daten ist die Benutzererfahrung der Anwendung besser.

Ziel des hier angewendeten Testaufbaus ist es die verschiedenen Alternativen abzuwägen und dabei die beste Alternative zu finden. Diese Suche wird in drei Teilen beschrieben. Der erste Abschnitt analysiert die Kriterien unter welchen die Algorithmen verglichen werden. Hierzu wird der grundlegende Aufbau erarbeitet, welcher hinter dem Algorithmus liegt. Der zweite Abschnitt beschreibt die eigentliche Durchführung des Tests während der letzte Abschnitt die gewonnenen Ergebnisse vergleicht und anhand der Kriterien den besten Algorithmus aussucht.

5.3.1 Kriterien zur Algorithmusauswahl

Bevor ein Algorithmus ausgewählt werden kann, nach welchen Kriterien diese Auswahl stattfindet. Zentral dabei ist die Laufzeit des Algorithmus für verschiedene Anwendungsfälle. Hierfür werden drei verschiedene Kategorien festgelegt:

- Kategorie S: Distanzen bis 10 km
- Kategorie M: Distanzen von 10 bis 25 km
- Kategorie L: Distanzen von 25 bis 50 km

Die Distanz bezeichnet in dieser Liste jeweils die Luftlinie zwischen dem Startpunkt und dem Zielpunkt der Routensuche. Diese Einteilung ermöglicht es, kürzere Distanzen stärker zu berücksichtigen als längere Distanzen, da diese für Fußgänger von höherer Bedeutung sind.

Ein zweites Kriterium ist der benötigte Speicherplatz für die Kartendaten. Wie in den vorherigen Kapiteln ausführlich erläutert, haben größere Kartendaten verschiedene Nachteile in der allgemeinen Handhabung der Anwendung. Für diese Algorithmusauswahl im Speziellen stellt sich hierbei die Frage, ob die zusätzlichen Informationen für CH benötigt werden oder nicht.

Zusätzlich wäre es sinnvoll den Batterieverbrauch der einzelnen Algorithmen zu messen. Aufgrund unterschiedlicher Auslastung beispielsweise des Prozessors können sehr unterschiedliche Leistungsprofile anfallen. Allerdings wird der Navigationsalgorithmus nicht ständig ausgeführt und die Messung des Batterieverbrauches ist sehr schwierig, da verschiedenste Faktoren miteinfließen. Selbst bei langen Testläufen der einzelnen Algorithmen und Betrachtung der Laufzeit einer vollen Batterieladung kann sich diese Zeit bei einzelnen Tests abhängig von externen Einflüssen wie beispielsweise der Umgebungstemperatur unterscheiden. Hier eine gesicherte Aussage zu treffen ist somit nicht sinnvoll.

Ebenfalls kein Kriterium ist der Arbeitsspeicherverbrauch, da dieses Kriterium durch die Laufzeit bereits vollständig abgebildet ist.

Somit ist in erster Linie die Laufzeit verschiedener Anfragen relevant für die Qualität eines Algorithmus. Diese Tests werden jeweils drei Mal durchgeführt. Einmal mit vorgewärmtem und einmal mit leerem Cache vor der Berechnung jeder Route. Zusätzlich wird ein Testlauf durchgeführt, welcher reale Bedingungen widerspiegeln soll. In diesem wird die Zelle der Startkoordinate bereits in den Cache geladen, bevor die Routenberechnung beginnt. Da in der eigentlichen Anwendung das Zeichnen stattfindet, bevor die Routenberechnung gestartet wird, ist dies der realen Anwendung am nächsten.

5.3.2 Testkandidaten und -durchführung

Für die Durchführung des Tests ist es zuerst notwendig alle untersuchten Algorithmen und Varianten zu ermitteln. Die Basis bilden der Dijkstra und der A*-Algorithmus. Beide werden jeweils in drei unterschiedlichen Varianten getestet. Einer unidirektionalen und einer bidirektionalen Ausführung ohne vorberechnete Daten und einer CH-basierten Ausführung auf dem entsprechend vorbereiteten Graphen. Zusätzlich werden bei den A*-basierten Algorithmen jeweils verschiedene Heuristiken getestet. Das sind die in Kapitel 3.5.3 in den Formeln ab 3.6 aufgelisteten Heuristiken: Die korrekte Distanzberechnung unter

Einbeziehung der Erdkrümmung, die Hypotenuse, die drei Hypotenusenapproximationen und die Manhattandistanz.

Einen zentralen Einfluss auf die Leistung des Algorithmus hat die Speicherung der Graphdaten im Arbeitsspeicher. Hierfür werden drei Formate angeboten: Arrays, Objekte in verketteten Listen und für Android optimierte Objekte in verketteten Listen. Zur Vereinfachung der einzelnen Beschreibung wird die Einteilung in verschiedene Zellen nicht betrachtet. Diese Einteilung betrifft alle Formate in einer ähnlichen Art und Weise und ist somit an dieser Stelle nicht relevant. Dabei wird zwischen Kanten für den Dijkstra-Algorithmus und Kanten für die A*-Algorithmus unterschieden.

Bei der Speicherung der Kanten in verketteten Listen werden die Kanten selbst in einem Array abgespeichert, die Position innerhalb des Arrays stellt dabei ihre Knoten-ID dar. Innerhalb des Arrays selbst wird dabei die erste Kante gespeichert und alle weiteren Kanten, die an demselben Knoten starten, werden hintereinander angefügt. Die Datenstruktur des Dijkstra ist hierbei etwas einfacher als die des A*. Die Informationen über die Position des Knotens wird nicht benötigt. Somit können alle Kanten die gleiche Struktur haben. Beim A* hingegen muss zusätzlich die Position angegeben werden. Dies ist nur in der ersten Kante erforderlich. Entsprechend können alle zusätzlichen Kanten gleich wie die Kanten für den Dijkstra-Algorithmus aufgebaut sein.

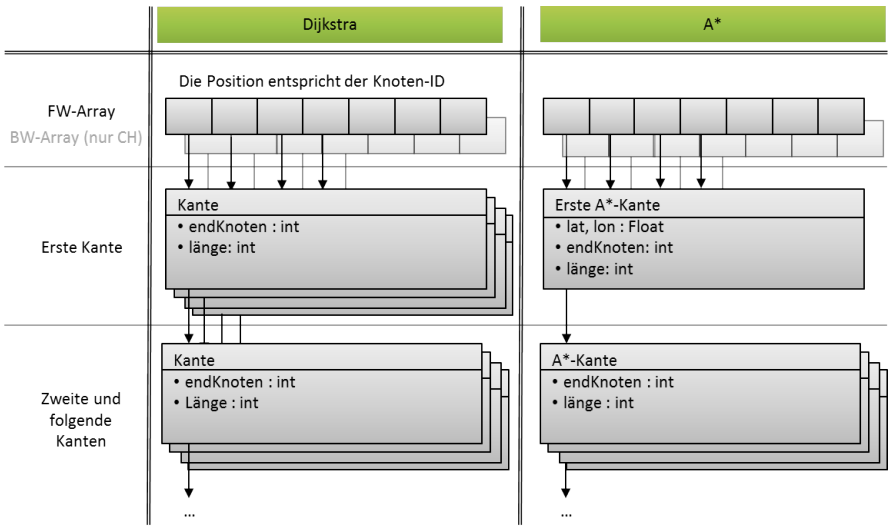


Abbildung 5.5: Darstellung der Kanten als verkettete Listen

Die verketten Listen für Android sind ähnlich aufgebaut. Der Unterschied ist die optimierte Ausnutzung der Objektgrößen im Arbeitsspeicher. Mittels der Beschreibung in Kapitel 3.2 lässt sich berechnen, dass die eine bisherige Kante 20 Byte an Daten benötigt, während das Betriebssystem hierfür 32 Byte allokiert. Somit sind 12 Byte ungenutzt. Graphik 5.6 stellt die optimierten Objekte dar, um keinen Speicherplatz ungenutzt zu lassen. 8 der 12 Bytes können einfach zur direkten Speicherung eines zweiten Ziels verwendet werden. Beispielsweise benötigt ein klassische Kreuzung mit einem Knoten, der Start- oder Endpunkt von 4 Kanten ist, in der nicht-optimierten Variante 4 Objekte, während

in der optimierten Variante nur 2 Objekte benötigt werden. In der Dijkstra-Variante ist dabei sogar noch Platz für einen fünften Knoten, welcher in den passenden Kreuzungen ebenfalls belegt wird. Eine der Optimierungen sind die `endKnotenOderLänge`-Variablen welche aus der Abbildung hervorgehen. Je nach Objekt wird in diesen abwechselnd ein Knoten oder eine Länge gespeichert, somit kann alle zwei Objekte eine zusätzliche Kante untergebracht werden.

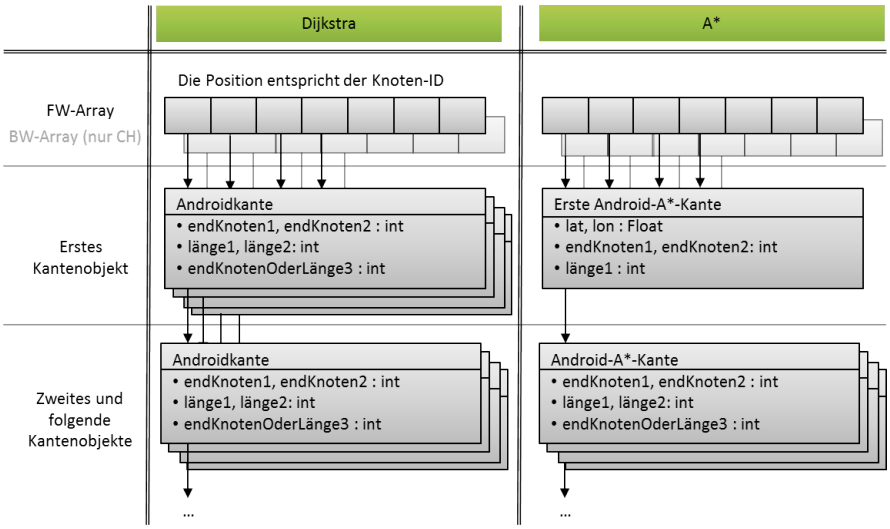


Abbildung 5.6: Darstellung der Kanten als optimierte verkettete Listen

Die dritte Variante ist die direkte Speicherung der Daten in Arrays. Dies spart Arbeitsspeicher, da die Anzahl an Objekten sich mit einer größeren Kantenanzahl erhöht, dafür ist ein einzelner Zugriff teurer, da zusätzliche Arrayzugriffe durchgeführt werden können.

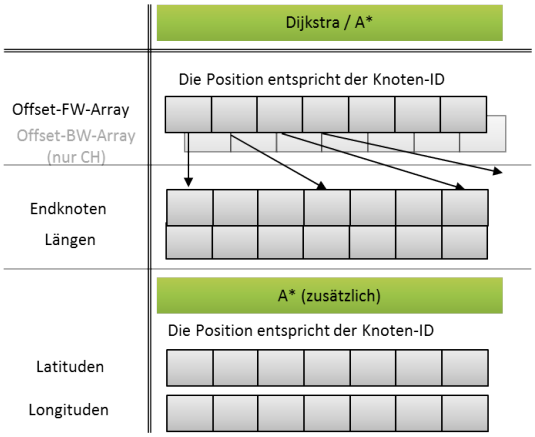


Abbildung 5.7: Darstellung der Kanten in Arrays

Die Darstellung als Arrays ist typisch für Routingalgorithmen. Die Informationen werden nicht nach Straße oder Kante geordnet abgespeichert sondern

nach Typ der Information. Das benötigt nur ein Array pro Information, allerdings wird zusätzlich ein Array benötigt. Dies funktioniert einfach für alle Informationskategorien, die an Knoten gebunden sind. Auf diese kann mit der Knoten-ID zugegriffen werden. Komplexer ist der Zugriff auf Informationen, welche pro Kante existieren. Hierfür muss ein zusätzliches Offset-Array angelegt werden, indem Start- und Endpositionen aller Straßen, die an einer bestimmten Straße starten, gespeichert werden.

Mit den festlegten Testkandidaten wird im Folgenden die weitere Umgebung des Tests geklärt. Die Abbildung 5.8 zeigt nochmals eine Übersicht aller beschriebenen Testaufbauten. In jeder Zeile werden die Kombinationsmöglichkeiten dargestellt und in der Zahl neben der Kategoriebeschreibung ist die Gesamtanzahl an Testmöglichkeiten dargestellt.

Als Entwicklungsbasis wird das Android SDK verwendet. Die Implementierung der Anwendung mittels NDK ermöglicht zwar theoretisch eine höhere Performance, da ein Großteil der Laufzeit allerdings durch Speicherzugriffe beim Laden von Daten zustande kommt ist kein großer Leistungszuwachs zu erwarten. Das SDK bietet dem hingegen viele praktische Vorteile in der Entwicklung durch die große Menge an APIs und allgemeiner Unterstützung. Somit fällt die Wahl auf das SDK.

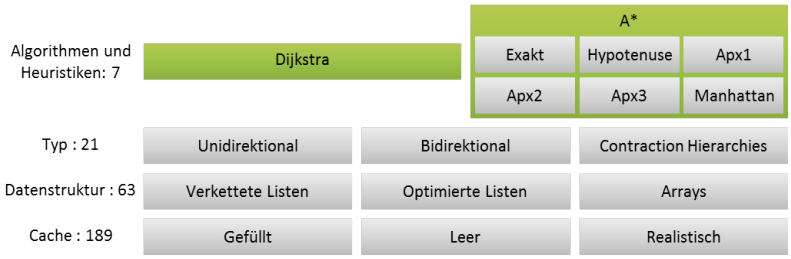


Abbildung 5.8: Darstellung der Testaufbauten

Als Datenstruktur der Vorrangwarteschlange kommt bei allen Tests eine Min-Heap-Implementierung zum Einsatz, welche eine leicht angepasste Version der Implementierung aus dem Tourenplaner ist.[8]

Wie im vorherigen Kapitel beschrieben, findet das Laden der Knoten in Zellen statt. Sobald ein Knoten benötigt wird, wird die gesamte Zelle geladen. Für die meisten der vorgestellten Algorithmen ist das sinnvoll. Sobald ein Knoten gefunden ist, werden speziell beim Dijkstra in den nächsten Schritten auch umliegende Knoten benötigt. Auch der A*, obwohl er sich tendenziell stärker in eine bestimmte Richtung ausdehnt, nutzt in der Regel viele der Knoten aus einer geladenen Zelle.

Für CH stimmt das nur bedingt. Strukturbedingt müssen insgesamt nur sehr wenige Knoten überprüft werden. Somit werden pro Zelle sehr viel mehr Knoten geladen, welche gar nicht benötigt werden und CH ist in diesem Testaufbau benachteiligt. Dies muss in der Ergebnisfindung berücksichtigt werden.

Bei beiden Lösungen werden die vorhandenen Zellen gecached. Grundsätzlich liegen so viele Zellen wie möglich im Arbeitsspeicher. Dabei ist die Cachingstrategie eine Least Recently Used Strategy. Sobald der Speicher voll ist, wird dabei die Zelle entfernt, dessen letzte Verwendung am längsten her ist. Wird eine Zelle daraufhin noch einmal benötigt, muss sie ein weiteres Mal eingelesen werden.

Durch die Zugriffsstruktur sollte dies bei allen Algorithmen selten vorkommen. Sobald Zellen aus dem Cache entfernt werden, liegen sie in der Regel in Regionen, welche kein weiteres Mal benötigt werden.

Die automatisierte Durchführung der Tests arbeitet sequentiell alle Testfälle ab: zuerst in aufsteigender Reihenfolge und daraufhin ein zweites Mal in absteigender Reihenfolge. Um auch eine Fairness vor möglichen Überhitzungen zu garantieren wird zwischen jedem Testlauf eine Pause von zwei Minuten eingelegt. Ein Test wiederum besteht aus jeweils 100 Berechnungen von kurzen, mittleren und langen Distanzen. Alle Navigationsstart- und -endpunkte werden im Voraus festgelegt, um keine Abweichungen durch eine mögliche unterschiedliche Schwierigkeit der zu berechnenden Routen zuzulassen.

Somit sind alle benötigten Informationen gegeben, um die Tests durchzuführen. Die Ergebnisse werden im folgenden Abschnitt vorgestellt.

5.3.3 Algorithmusauswahl

Dieser Abschnitt beschreibt die einzelnen Ergebnisse. Hierfür werden zuerst die Algorithmen verglichen und der beste wird bestimmt. Dabei fällt auf, dass Contraction Hierarchies unerwartet langsam ist. Der Effekt, welcher hierzu führt, ist in einer zweiten Tabelle analysiert. Eine dritte Tabelle bestimmt die optimale Heuristik, bevor in der letzten Tabelle die passende Datenstruktur ausgewählt wird.

In Abbildung 5.9 werden die Algorithmen miteinander verglichen. Dabei werden nur die Tests auf der Uhr mit dem „realistisch“ aufgewärmten Cache verwendet, von den einzelnen Heuristiken und Datenstrukturen ist jeweils die beste Alternative abgebildet. Die detaillierte Auswahl wird später in diesem Abschnitt besprochen.

Algorithmus	Distanz S	Distanz M	Distanz L
Dijkstra Unidir	1.101 ms	2.528 ms	19.773 ms
Dijkstra Bidir	719 ms	4.288 ms	10.977 ms
Dijkstra CH	28.106 ms	-	-
AStar Unidir	458 ms	1.148 ms	3.880 ms
AStar Bidir	546 ms	1.460 ms	5.553 ms
AStar CH	8.425 ms	-	-

Abbildung 5.9: Ergebnisse nach Algorithmus

Abbildung 5.9 zeigt die Ergebnisse nach den einzelnen Algorithmen aufgegliedert. Dabei sind sowohl die Laufzeiten, als auch die Anzahl an überprüften Knoten angegeben.

Die Entscheidung für den Algorithmus fällt nach diesen Tests für den A*-Algorithmus. Dieser erreicht in allen drei Distanzkategorien das beste Ergebnis und bleibt deutlich unter den in Kapitel 4.1 erwarteten Laufzeiten.

Wenig überraschend ist, dass der Dijkstra-Algorithmus langsamer ist als der A*-Algorithmus. Ebenfalls mit der Theorie vereinbar ist die verbesserte Laufzeit des bidirektionalen Dijkstra gegenüber dem unidirektionalen Dijkstra, was auf die geringere Anzahl an untersuchten Knoten zurückzuführen ist, während der bidirektionale A* hier gegenüber dem A* keine Verbesserung bewirkt, allerdings weiterhin besser als die Dijkstra-Varianten ist.

Auffällig hingegen ist die hohe Laufzeit der Contraction-Hierarchies. Trotz der sehr geringen Anzahl an Knoten sind diese im Testaufbau durch die geringe Verfügbarkeit an Arbeitsspeicher sehr benachteiligt. Durch das Speichern von Forward- und Backwardkanten, erhöht sich der benötigte Arbeitsspeicher und es treten deutlich mehr Cachemisses auf. Auch wenn die Anzahl der überprüften Knoten weiterhin gering ist, sind die Suchradien insgesamt am größten, so dass die meisten Zellen eingelesen werden müssen. In Kombination mit dem erhöhten Speicherverbrauch des Kartenmaterials ist CH kein Kandidat für eine praktische Smartwatchanwendung.

Cachegröße	Distanz S	Distanz M	Distanz L
20 MB	196 ms	618 ms	5.511 ms
40 MB	88 ms	199 ms	1.670 ms
60 MB	60 ms	130 ms	705 ms
80 MB	48 ms	103 ms	379 ms
100 MB	42 ms	91 ms	257 ms
150 MB	35 ms	74 ms	170 ms
200 MB	32 ms	64 ms	147 ms
512 MB	31 ms	58 ms	142 ms
1024 MB	30 ms	57 ms	135 ms
2048 MB	32 ms	61 ms	136 ms
4710 MB	36 ms	66 ms	140 ms

Abbildung 5.10: Ergebnisse nach Cachegröße

Es gibt zwar verbesserte Möglichkeiten der Datenstruktur,[34] allerdings ist diese nicht auf eine derart kompakte Speicherstruktur ausgelegt. Graphik 5.10 verdeutlicht diese Ergebnisse und listet die CH-Ergebnisse nach Cachegröße auf. Für diese Untersuchung wurde ein PC mit 16 Gigabyte RAM und Intel Core i7 3610QM Prozessor verwendet. Dabei ist erkennbar, dass mit geringerem Arbeitsspeicher im Cache die Leistungsfähigkeit deutlich vermindert wird. Es ist erkennbar, dass bereits mit leicht erhöhter Cachegröße die Laufzeit stark verbessert wird. Die Tatsache, dass ein zu großer Cache die Laufzeit wieder ansteigen lässt, liegt in der Speicherverwaltung von Java. Diese mit der größeren Menge an zugewiesenem Arbeitsspeicher aufwendiger.

Eine weitere interessante Analyse betrifft die Heuristiken. Diese Ergebnisse sind in Abbildung 5.11 dargestellt. Dabei ist erkennbar, dass die exakte Berechnung der Hypotenuse die beste Laufzeit liefert. Die exakte Berechnung überprüft etwas weniger Knoten, allerdings ist deren Aufwand zu hoch. Dem hingegen benötigen die Approximationen der Hypotenuse mehr Knoten, als die gesparte Rechnungsleistung an Zeitvorteil bringt. Die dritte Approximation ist für lange Distanzen geringfügig besser.

Für eine kombinierte Berechnung, welche ab einer bestimmten Distanz die Heuristik wechselt, lohnt sich das Ergebnis nicht. Die Unterschiede sind gering genug, sodass die Hypotenuse allein als beste Heuristikfunktion ausgewählt wird.

Die letzte Tabelle an dieser Stelle in Abbildung 5.12 betrachtet die verschiedenen Datenstrukturen. Die androidoptimierten Algorithmen verbessern die Laufzeiten der Algorithmen mit unoptimierten verketteten Datenstrukturen. Es können allerdings beide objektbasierten Strukturen nicht mit der Array-

struktur mithalten, so dass diese als beste ausgewählt werden kann.

Heuristik	Distanz S	Distanz M	Distanz L
Haversine	446 ms	1.120 ms	3.858 ms
Hypotenuse	439 ms	1.101 ms	3.722 ms
Hypot. Apx. 1	448 ms	1.126 ms	3.833 ms
Hypot. Apx. 2	453 ms	1.125 ms	3.737 ms
Hypot. Apx. 3	459 ms	1.138 ms	3.685 ms
Manhattan	503 ms	1.275 ms	4.445 ms
(Dijkstra)	1.101 ms	4.288 ms	19.773 ms

Heuristik	Knoten S	Knoten M	Knoten L
Haversine	1.797	7.810	32.496
Hypotuse	1.832	7.927	32.881
Hypot. Apx. 1	1.920	8.328	34.529
Hypot. Apx. 2	1.950	8.411	34.969
Hypot. Apx. 3	1.988	8.603	35.591
Manhattan	2.454	10.173	43.552
(Dijkstra)	9.521	46.562	207.617

Abbildung 5.11: Ergebnisse nach Heuristik

An dieser Stelle nicht speziell beschrieben, ist der Füllgrad des Caches vor dem Starten des Algorithmus. Dieser Füllgrad ist unerheblich für die Auswahl der Algorithmen, da alle Algorithmen, Heuristiken, Datenstrukturen gleichmäßig vom gefüllten Cache profitieren. Für die angegebenen Ergebnisse wurden die Testläufe für alle drei Varianten gleich gewichtet.

Datenstruktur	Distanz S	Distanz M	Distanz L
Verkettete Listen	623 ms	1.704 ms	7.035 ms
Opt. Verk. Listen	592 ms	1.683 ms	6.683 ms
Arrays	467 ms	1.427 ms	5.000 ms

Abbildung 5.12: Ergebnisse nach Heuristik

Im Allgemeinen lassen sich aus diesen Testdaten verschiedene Ergebnisse ziehen. Zusammengefasst fällt die Entscheidung für einen A*-Algorithmus mit Hypotenuse als Heuristikfunktion und Array-Datenstruktur.

5.4 Finanz- und Marketingaspekte

Neben den technischen Details ist für jede Applikation auch der betriebswirtschaftliche Aspekt von großer Bedeutung. In diesem Kapitel werden zuerst Gedanken zur finanziellen Planung getroffen, daraufhin werden Marketingmöglichkeiten dargestellt, welche unter anderem die Gestaltung der Veröffentlichungsseite im Google PlayStore beinhalten. Die Umsatzfrage steht bei dieser Anwendung nicht im Vordergrund. Zu Beginn wird sie vollständig kostenfrei erhältlich sein und auch keine sonstigen Einnahmen generieren.

Grundsätzlich können Kosten für den Download der Anwendung erhoben werden. Alternativ kann die Anwendung kostenfrei angeboten werden. In diesem

Fall ist eine Finanzierung über Werbung oder kostenpflichtige Extrafunktionen denkbar.

Innerhalb dieser Anwendung sind diese beiden Möglichkeiten grundsätzlich umsetzbar. Werbung würde dezent in die Smartphoneanwendung passen. Der Benutzer würde sich hiervon kaum belästigt fühlen, da diese nicht häufig verwendet wird. Allerdings sind die generierten Einnahmen entsprechend gering. Zusätzlich können auf der Smartwatch einzelne Werbeeinblendungen erfolgen kurz bevor die Navigationsanzeige gestartet wird.

Eine weitere Variante ist die Einführung von Extrafunktionen, welche kostenpflichtig sind. Dies können zusätzliche Kartendaten wie Points of Interest oder das Zeichnen exakter Ländergrenzen sein. Solange die bisherige Funktionalität davon unberührt bleibt, sind Benutzer hierfür häufig bereit nach Bedarf Extras zu erwerben.

Auch wenn es, wie erwähnt, aktuell nicht geplant ist Kosten zu erheben. Es kann notwendig werden, wenn die Kosten für die Bereitstellung des Downloadservers aufgrund großer Nachfrage stark ansteigen. In diesem Fall werden kostenausgleichende Einnahmen benötigt.

Um eine Vielzahl an Kunden zu erreichen, müssen diese über die Anwendung informiert werden. Verschiedene Marketingmaßnahmen sollen einen großen Kundenstamm generieren. Eine Standardmöglichkeit hierfür ist die Werbung innerhalb anderer Anwendungen. Dem steht bei geringen Kosten ein guter Nutzen gegenüber. Allerdings sind ohne Einnahmen auch keine Marketingausgaben gewünscht.

Ein weitere Alternative ist die Nutzung der vorhandenen Ressourcen durch den Partner Wearable Software. Diese bieten mit ihrer Webseite[1] einen Informationsquelle für Android Wear Kunden. Über eine Platzierung der Applikation in den „Featured Apps“ kann die Anwendung ohne direkte Kosten beworben werden.

Für die Darstellung im PlayStore werden Screenshots, ein Logo und verschiedene Texte zur Beschreibung der App benötigt. Diese bieten den ersten Eindruck für den möglichen Nutzer der Anwendung. Dort muss attraktiv beschrieben werden, um die Benutzer zum Download der Anwendung zu bewegen.

Wichtig in der Onlinepräsenz ist das Beschreiben der Eigenschaften der Anwendung als Stärken. So wird die Kartendarstellung mit bekannten Konkurrenten wie Google Maps nicht mithalten können. Entsprechend muss diese Kartendarstellung als schlicht angegeben werden, um auf die hieraus resultierenden Vorteile der kleinen Kartendaten und effizienten Renderung schließen zu können. Ansonsten werden die Screenshots die Anwendung umfassen, so dass der Benutzer in kurzer Form einen Überblick über die Anwendung erhält.

[1] `http://www.androidwearcenter.com`

Kapitel 6

Ausgewählte Aspekte der Implementierung

Dieses Kapitel nutzt die Erkenntnisse und Entscheidungen aus dem Systementwurf und beschreibt detailliert verschiedene ausgewählte Lösungen in einzelnen Komponenten der Anwendung. Die erste dieser Funktionen ist die Vorberechnung und Aufbereitung der Kartendaten. Diese läuft in verschiedenen Schritten ab, um eine Entkopplung unterschiedlicher Teilaufgaben zu ermöglichen und die Stärken verschiedener Umgebungen nutzen zu können.

Der zweite Prozess der Anwendung ist die Berechnung der kürzesten Route. Mit der Auswahl des korrekten Algorithmus kann dessen Berechnung optimiert werden, um die Wartezeiten zu verkürzen und die Batterie zu schonen. Dabei muss der knapp vorhandene Arbeitsspeicher möglichst effizient genutzt werden. Vorgestellt an dieser Stelle ist auch der Ablauf zum Finden eines Knotens nach dessen ID.

Des Weiteren ist der Prozess zum Rendern der Karte erläutert. Dieser muss auf schnelle Reaktionszeit und Batterieeffizienz optimiert sein. Zusätzlich gilt wie oben, dass der hierfür benötigte Arbeitsspeicher möglichst gering sein sollte, um die Routenfindung nicht zu beeinflussen.

6.1 Ablauf der Vorverarbeitung

Die Vorberechnung ist in sechs Schritte eingeteilt. Abbildung 6.1 zeigt eine Übersicht dieser Schritte, welche in diesem Abschnitt einzeln beschrieben sind.

Der erste Schritt extrahiert die benötigten Daten aus dem OSM-File. Hierfür wird die Bibliothek „osmosis-osm-binary“ verwendet. Diese ermöglicht das Einlesen von OSM-Dateien und gibt währenddessen Callbacks mit deren Inhalten. Wie im Kapitel 3.3 beschrieben, sind diese eingeteilt nach Knoten, Relationen und Wegen. Ziel der Extraktion ist das Sammeln von Knoten und Kanteninformationen. Die einzigen Relationen die benötigt werden sind die Staats- und Ländergrenzen, welche später einzeln aus einer vorverarbeiteten Quelle eingelesen werden. Für einen effizienten Ablauf muss die Datei insgesamt zweimal gelesen werden.

Das erste Einlesen überspringt zuerst die Knoten. Aus den Wegen werden alle erwünschten Straßen herausgefiltert. Jeder Weg beinhaltet die IDs zu den Kno-

ten, welche ihn beschreiben. Diese werden zwischengespeichert. Beim zweiten Durchgang können beim Einlesen der Knoten zu jeder ID die weiteren Informationen gespeichert werden. Diese zusätzlichen Informationen umfassen in erster Linie ihre Koordinaten. Zuletzt werden die Wege ein zweites Mal verarbeitet. Mit den Informationen der Knoten können die Wege, welche eine Aneinanderreihung von kurzen, geraden Stücken sind, in diese einzelnen Kanten eingeteilt werden.

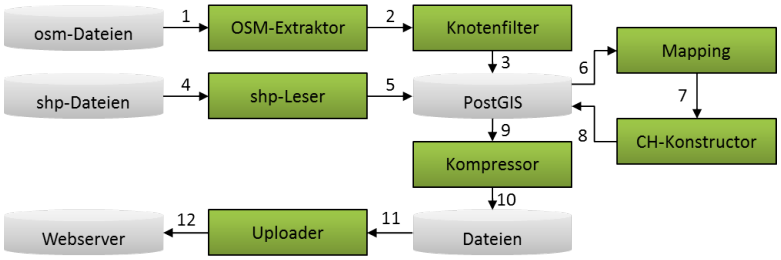


Abbildung 6.1: Darstellung der Vorverarbeitung

Dieses effiziente Vorgehen beim Einlesen der Dateien ist möglich, da die Dateien von Geofabrik immer nach dem Typ der Information geordnet sind. Das OSM-Format erfordert diese Struktur allerdings nicht zwingend. Entsprechend muss das Einlesen den Fall abfangen, dass ein Knoten entdeckt wird nachdem bereits ein Weg gelesen wurde. Sollte das der Fall sein, muss die zweite Phase weiter unterteilt werden, so dass die neue zweite Phase nur alle Knoten einliest und die erstellte dritte Phase die Wege weiterverarbeitet. Alle zwischengespeicherten Informationen sind optimalerweise im Arbeitsspeicher gehalten. Sollte dieser aufgrund der Datenmenge nicht ausreichen, kann hierfür auch auf Festspeicher zurückgegriffen werden.

Bevor diese Daten in der Datenbank gespeichert werden, setzt der zweite Schritt ein. In den Originaldaten werden für eine hohe Genauigkeit Kurven in viele kleine Geraden eingeteilt, da ansonsten keine Möglichkeit Kurven abzuspeichern gegeben ist. Diese kurzen Teilstücke erhöhen die Gesamtdatenmenge erheblich. Zur Begrenzung dieses Anstiegs werden in diesem Schritt Knoten, welche an genau zwei Teilstücke grenzen, überprüft. Bei diesen Knoten handelt es sich immer um reine Informationen welche für die reine Darstellung von Kurven benötigt werden. Kreuzungen sind dort nicht vorhanden. Sie sind für einen optimalen Routingalgorithmus irrelevant. Somit kann gefiltert werden und alle nicht benötigten Informationen können entfernt werden. Es muss ein Kompromiss zwischen einer guten Kartendarstellung und der Datenmenge gefunden werden. Knoten, an denen sich eine Eigenschaft der Straße leicht verändert, wie beispielsweise der Straßenname, dürfen von dieser Reduktion nicht betroffen sein.

Um dies zu erreichen, werden die Winkel zwischen den einzelnen Teilstücken der Straße gemessen. Unterscheidet sich die Neigung von zwei Straßen um weniger als die Toleranzgrenze von 10°, so wird dieser Knoten eliminiert und die angrenzenden Kanten werden zusammengefasst. Zusätzlich werden jeweils die Winkel zum ersten Teilstück nach der vorherigen Kreuzung und zum ersten Teilstück vor der nachfolgenden Kreuzung berechnet. Diese Knoten bleiben ebenfalls erhalten, sobald die Toleranzgrenze zu diesen überschritten ist. Abbildung 6.2

zeigt eine Übersicht, wie viele Knoten hierdurch eingespart werden können. Zusätzlich befinden sich Beispiele über die Qualität der entstehenden Daten in der Abbildung.

Es lässt sich erkennen, dass bei einem Filter von 10° die Datenmenge deutlich reduziert werden kann. Dabei ist die Kartendarstellung weiterhin in guter Qualität. Die einzelnen Teilabschnitte lassen sich leicht erkennen, aber die Kurven sind weiterhin gut sichtbar. Zwischen einer Toleranzgrenze 5° und 10° können ca. 3,5 Millionen Knoten eingespart werden, während es im nächsten Schritt von 10° zu 15° nur knapp über 2 Millionen Knoten sind. Dort werden ebenfalls erste nennenswerte Kanten sichtbar, so dass insgesamt gegenüber den 28,5 Millionen Knoten ohne einen Filter, eine Reduktion um über ein Drittel der Knoten bzw. um über 11 Millionen Knoten durchgeführt wird.

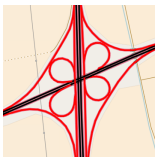
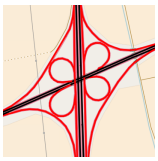
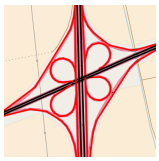
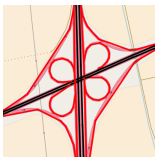
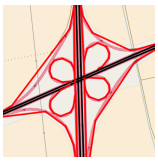
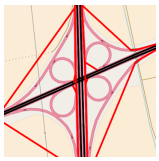
Filter	0°	5°	10°
Anzahl Knoten	28.591.620	21.171.686	17.530.900
Graphik			
Filter	15°	20°	100°
Anzahl Knoten	15.254.925	13.733.068	7.917.423
Graphik			

Abbildung 6.2: Vergleich der Knotenqualität

Somit sind fast alle wichtigen Informationen vorhanden. Es fehlen lediglich die Ländergrenzen. Diese werden im dritten Schritt von einer weiteren Datenquelle eingelesen. `http://www.gadm.org` bietet vorgerenderte OSM-Daten als Shapefiles. Diese Shapefiles lassen sich deutlich einfacher und effizienter einlesen als die rohen OSM-Daten, somit kann sowohl Entwicklungszeit gespart werden, als auch die Dauer der Vorverarbeitung reduziert werden. Die erhaltenen Grenzen werden ebenfalls in der PostGIS - Datenbank gespeichert.

Die Stärken von PostGIS lassen sich im Folgenden vierten Schritt erkennen. Die Zuordnung der Knoten in die jeweiligen Staaten und Länder lässt sich mit einer einzelnen SQL-Anfrage realisieren. Sie ist in Listing 6.1 dargestellt. Die Tabellen und Spaltennamen sind entsprechend ihrer Aufgaben klar benannt. Mittels eines GIS-Index können die Unterabfragen deutlich beschleunigt abgearbeitet werden. Die PostGIS `ST_CONTAINS` – Funktion übernimmt die geometrischen Berechnungen. Ein Nachteil ist die trotz der Indexierung hohe Dauer dieser Abfrage. Für den Kartensatz von Deutschland benötigt diese Abfrage mehrere Stunden. Die Dauer entsteht durch die Tatsache, dass jeder Knoten mit den Polygonen verglichen werden muss, die teilweise aus mehreren zehntausend einzelnen Koordinaten bestehen. Eine Optimierung ist das vorherige Speichern der Quelldaten in einer temporären Tabelle `tempNodes`. Diese befindet

sich nach Möglichkeit vollständig im Arbeitsspeicher und ermöglicht ein sehr schnelles Verarbeiten der Werte.

Listing 6.1: Länderzuordnung (vereinfacht)

```
insert into nodes(stateID, countryID)
 (select coalesce(stateID, -1), coalesce(parentID, -1)
  from tempNodes n inner join
   (select poly, stateID,parentID from StateSimple) j
    on ST_CONTAINS((j.poly),n.position));
```

Eine weitere Komponente übernimmt die Kompression der Daten. Diese komprimiert die Daten wie im Konzept in Kapitel 5.2 beschrieben.

Der letzte Schritt stellt die Daten auf einem Server bereit. Hierfür wird für die Rohdaten ein schlichter FTP-Upload im Binary-Mode verwendet. Nach dem Upload können die Daten über HTTP heruntergeladen werden.

6.2 Probleme und Optimierungen der Routenberechnung

Der verwendete A*-Algorithmus besitzt verschiedene Stellräder, welche sich zentral auf die erzielte Performance auswirken. Mit einer verbesserten Performance sinken sowohl der Batterieverbrauch als auch die Wartezeit, bis der Benutzer eine aktualisierte Route angezeigt bekommt. Somit wirken sich Optimierungen im Algorithmus positiv auf das gesamte Anwendungserlebnis aus.

Bevor der Algorithmus selbst optimiert wird, können dessen Aufrufe minimiert werden. In einer perfekten Navigationssitzung wird nur ein Aufruf benötigt. Beim Starten der Berechnung kann von der aktuellen Position die direkte Route vom Start zum Ziel berechnet werden. Daraufhin kann die Route, die der Benutzer zurücklegt entsprechend mit jedem Knoten gekürzt werden. In der Praxis werden jedoch gelegentlich neue Berechnungen nötig. Dies kann bei Baustellen oder bei abweichend gewählten Routen des Benutzers der Fall sein. Dann muss der Algorithmus neu aufgerufen werden. Die Gesamtanzahl der Routenberechnungen kann jedoch gering gehalten werden.

Eine Herausforderung in der Implementierung des Algorithmus für große Distanzen ist das Zwischenspeichern der Ergebnisse. Es ist nicht möglich alle benötigten Entfernungen zum Startknoten und die Vorgänger der jeweiligen Knoten in zwei großen Arrays zu speichern. Entsprechend müssen auch diese aufgeteilt werden. Es bietet sich an, die einzelnen Teile an die Zellen zu koppeln. Jede Zelle erhält zu jedem ihrer Knoten ein Zwischenergebnis. Wird nun eine Zelle aus dem Cache entfernt, so muss das Zwischenergebnis in eine temporäre Datei übertragen werden. Bei einem erneuten Aufruf der Zelle, können die Daten wieder geladen werden, so dass es dabei zu keinen Abweichungen kommt. Als Dateiname wird die Zellen-ID gewählt. Dies garantiert eine einfache und eindeutige Zuweisung der Zwischenergebnisse zu den Dateien und ermöglicht beim Laden einer Zelle eine effiziente Überprüfung ob Zwischenergebnisse vorhanden sind. Eine Ausnahme bilden Zellen, von denen alle Knoten bereits vollständig bearbeitet sind. Speziell beim Dijkstra-Algorithmus kommt dies durch den runden Suchradius häufig vor. In diesem Fall müssen nur die Vorgängerknoten

gespeichert werden, da diese Teil des Endergebnisses sein können. Die Distanzen sind nicht notwendig und können für die finale Route in kurzer Zeit neu berechnet werden. Im verwendeten Algorithmus ist dies nur bei hohen Distanzen gelegentlich der Fall. Da diese Optimierung keine negativen Auswirkungen hat, ist sie trotz des geringen Nutzens integriert.

Die Heuristik selbst benötigt für die korrekte Berechnung der Distanz eine effiziente Umrechnung der Koordinaten in Distanzen. Wie schon in Kapitel 3.5.3 erörtert ist die Heuristik neben der Vorrangwarteschlange eine der rechenintensivsten Aufgaben im Algorithmus. Deshalb ist es sehr lohnend diese so weit wie möglich zu optimieren. Abbildung 11 zeigt die hierfür verwendete Funktion.

Algorithmus 11 : Optimierte Berechnung der Heuristik

```
1 b ← abs(lat1 − lat2)
2 a ← abs(lon1 − lon2) ∗ CosLookup[min(abs(lat1), abs(lat2))]
3 return hypot(a, b) ∗ (40030173.592F/360F)
```

Die Idee hinter dieser Umsetzung ist der Gebrauch der in Abschnitt 5.3.3 verwendete Heuristik unter Einbeziehung einer annähernd korrekten Distanz der Longituden. Um diese korrekt zu berechnen wird mittels der Formel $deltaLon = cos(lat) * deltaLon$ eine Korrektur der Latitude um den Kosinus der Longitude angewendet. Diese Formel direkt umzusetzen besitzt das Problem, da sich die Latitude beider Werte unter Umständen stark unterscheiden kann. Da die A*-Heuristik kleinere Distanzen erlaubt, ist die einfachste Lösung dieses Problems, das kleinere Ergebnis zu nehmen. Durch die Kosinusberechnung ist das der größere der beiden Latituden-Beträge. Aufwendig ist auch die Berechnung des Kosinus selbst.

Deshalb ist diese Berechnung in eine Lookup-Table ausgegliedert. In einem Array kann für jeden vollständigen Wert von 0° bis 90° der kleinstmögliche Kosinuswert erhalten werden. Dieser unterscheidet sich selbst im schlechtesten Fall um weniger als 2 % vom optimalen Wert und ist somit weiterhin ausreichend genau. Da der Kosinus von 0° bis 90° sich gleich verhält wie von 0° bis -90° kann zur Berechnung beider Hemisphären der Betrag genommen werden. Die Umrechnung der Lat-Lon Unterschiede in Meter wiederum kann über die Multiplikation mit einer Konstante vorgenommen werden.

Eine weiterer wichtiger Aspekt für einen optimierten Routingalgorithmus ist die Abfrage nach einem Knoten. Nach der Knoten-ID wird diese Abfrage sehr häufig innerhalb des Algorithmus selbst eingesetzt. Das Wesentliche in dieser Routine ist die Bestimmung der Zelle, in welcher sich der Knoten befindet. Es wird zunächst überprüft, in welchem Bundesland und damit in welcher Datei der Knoten anzutreffen ist. Da die Bundesländer eigene ID-Bereiche besitzen kann dies über eine binäre Suche bestimmt werden. Zumal in der Regel nur sehr wenig Bundesländer auf der Uhr vorhanden sind, kostet diese Suche kaum Zeit. Nachdem die Datei bestimmt ist, sind innerhalb der Metadatendatei alle ID-Bereiche für die Zellen hinterlegt. Diese überschneiden sich ebenfalls nicht. Somit kann mit einer zweiten binären Suche die korrekte Zelle bestimmt und eingelesen werden. Das Vorgehen wird mit Abfragen, ob sich der Knoten in der aktuellen oder einer anderen bereits vorhandenen Zelle befindet, weiter optimiert.

6.3 Effiziente und energiesparende Renderung der Kartendarstellung

Das Rendern der Karte muss live erfolgen. Das Speichern von Bildern des gesamten Kartenmaterials würde die Möglichkeiten der Smartwatch weit überschreiten. Für eine sinnvolle Qualität wäre deutlich mehr Speicherplatz nötig, als vorhanden ist. Somit ist die Entscheidung das Rendern live auf der Uhr zu übernehmen eindeutig.

Die verwendete Technik ist eine Variation des Double Bufferings. In diesem Verfahren wird ein berechnetes Bild nur gezeichnet, sobald dessen Berechnung vollständig abgeschlossen ist. Dieses wird solange beibehalten, bis das Zeichnen des folgenden Bildes abgeschlossen ist. In dieser Anwendung wird das vorberechnete Bild größer allokiert, als der eigentliche Bildschirm groß ist. Dies erlaubt, dass die rechenintensive Berechnung einer neuen Karte nicht bei jeder Bewegung der Karte notwendig wird. Wie für die Routingalgorithmen ist ein wichtiges Ziel die Häufigkeit der Berechnungen der Kartendarstellung möglichst gering zu halten. In jede Richtung werden 50 Pixel mehr allokiert als benötigt. Nur sobald sich die Karte 10 oder mehr Pixel bewegt hat, wird das Berechnen eines neuen Bildes gestartet.

Auf der anderen Seite darf die Überallokierung auch nicht zu groß sein, da der Speicherbedarf der Bilder im RAM quadratisch ansteigt. Die 50 Pixel in jede Richtung sind ein guter Kompromiss. Weder steigt der Speicherbedarf zu stark an, noch spürt der Benutzer bei normaler Fortbewegung etwas. Sollte der Benutzer zu schnell scrollen, ist für kurze Zeit ein leerer Raum erkennbar. Es ist zu erwarten, dass das nicht als Einschränkung der Anwendung empfunden wird, da die aktualisierte Karte nach kurzer Zeit vorhanden ist.

Die Häufigkeit des Zeichnens ist auf ein Minimum reduziert. Das Zeichnen in das Bitmap wird ohnehin nur bei Veränderung des Position angestoßen, aber auch das Zeichnen des Bitmaps auf die Oberfläche der Uhr kann minimiert werden. Es muss nur durchgeführt werden, sobald sich ein Parameter, welche die Zeichnung beeinflusst ändert. Dazu gehören die Position, der Zoomfaktor und ein aktualisierter Weg.

Eine Alternative wäre das Verwenden von OpenGL gewesen. Dieses erlaubt es parallel sehr viele Knoten mit Hardwarebeschleunigung zu berechnen und damit die Berechnungszeiten zu verkürzen. Die Notwendigkeit hierfür ist nicht gegeben, da auf der Uhr allerdings nicht viele Straßen angezeigt werden. Somit ist kein großer Nutzen durch OpenGL vorhanden. Ebenso ermöglicht das Double Buffering die notwendigen Berechnungen sehr selten durchzuführen. Kein Nutzen erhält OpenGL auch in dem Falle, dass eine neue Zelle in den Cache geladen werden muss. Die Wartezeit hierfür ist unabhängig von der Art zu zeichnen. Des Weiteren bietet Android mit der API zum Zeichnen einen hohen Komfort und viele Funktionalitäten, die ansonsten nicht vorhanden wären. Somit fiel die Entscheidung hier zugunsten der Double Buffering Methode in der ein Bild mit Standardmitteln erstellt wird.

Innerhalb der Berechnung eines Bildes werden zuerst alle Zellen bestimmt, welche im Zeichenbereich ganz oder teilweise enthalten sind. Durch die Größe der Zellen und dem höchstmöglichen erlaubten Zoomfaktor sind dies höchstens vier Zellen. Nord- und Südpol stellen hierbei ebenfalls keine Ausnahme dar, da durch die geringere Gesamtzahl an Zellen bei höherer Latitude an den Polen

dort nur noch jeweils eine Zelle vorhanden ist. Für den Ort einzelner Kanten ist kein Index vorhanden, somit wird über alle Kanten aller Zellen iteriert.

Zu Beginn wird in einem Filter überprüft, ob die Kategorie des Knotens im aktuellen Zoomfaktor überhaupt gezeichnet wird. Ist weit herausgezoomt, werden beispielsweise keine lokalen Straßen gezeichnet. Als nächstes wird gefiltert, ob sich der Knoten im Zeichenbereich befinden kann. Dazu werden alle Knoten, bei denen sowohl Startpunkt, als auch Endpunkt oberhalb des Zeichnungsbereiches liegen ignoriert. Selbiges wird für die verbliebenen drei Richtungen durchgeführt. Nur von den verbliebenen Knoten wird das Zeichnen durchgeführt. Somit kann ein Großteil der Zeichenarbeit erspart werden. Die maximal möglichen Koordinaten, einer Zelle stehen durch die fixe Zuordnung fest. Sind beide Wege vorhanden, kann die Straße über einen schlichten Draw-Befehl im Android-Canvas gezeichnet werden. Dieser erlaubt das teilweise oder vollständige Zeichnen außerhalb des Bildes und kann somit einfach aufgerufen werden. Je nach Kategorie wird dem Aufruf die entsprechende Farbe mitübergeben. Algorithmus 12 zeigt die wichtigsten Filterungen beim Rendern der Karte. Damit wird die effiziente Berechnung sichergestellt. Ebenfalls ist erkennbar, dass jeder Straßenname höchstens einmal gezeichnet wird und zwar an die Kante, die am nächsten an der Mitte der Anzeige liegt. Somit bleibt die Karte übersichtlich und wird nicht von zu vielen Straßennamen überflutet. Die Straßen werden am Ende des Vorganges gezeichnet, indem der Canvas entsprechend der Straßenrichtung gedreht wird, und der Text ausgerichtet wird, dass er mittig über der Straße angezeigt wird. Der Canvas bietet diese bequeme Funktion, dass ein Text mittig gezeichnet werden kann.

Algorithmus 12 : Filterung und Vorgehen beim Zeichnen

```
1  ZeichenBuckets ← berechneBucketsAusPosition(ZeichenBereich)
2  StraßenSet ← ∅
3  foreach Kante in ZeichenBuckets.kanten do
4      if Kante.Endknoten in ZeichenBuckets.knoten then
5          if Kante.Kategorie >= MindestKategorie then
6              if Kante.Koords in Zeichenbereich then
7                  zeichneKante()
8                  NamenDist ← dist(Kante.Mitte zu Zeichenbereich.Mitte)
9                  AlteDist ← StraßenSet.get(Kante.Name)
10                 if NamenDist < AlteDist then
11                     Füge (Kante.Name,NamenDist) zu StraßenSet hinzu
12 foreach Straßenname in Straßenset do
13     zeichneStraßenname()
```

Für das Abfragen des Straßennamen wird ein zusätzlicher Cache verwendet, welcher die letzten 200 Straßennamen bereit hält und bei Bedarf eine nicht vorrätige Straße auf den Dateien nachlädt. Die 200 ergibt sich aus kurzen Effizienztests, mit 200 lässt sich das Scrollen über eine gute Strecke ohne messbare Verzögerungen gestalten, während der Arbeitsspeicher, den 200 Straßen benötigen, in der Regel unter 10 Kilobyte liegen wird.

Neben der Karte wird der berechnete Weg gezeichnet. Hierfür kann über die einzelnen Knoten innerhalb des Weges iteriert werden und sollte sich der Teil einer Straße innerhalb des Zeichnungsbereiches befinden, wird diese Straße mittels blauer Punkte hervorgehoben, so dass sich der Weg von der Kartendarstellung abhebt.

Kapitel 7

Aussehen und Leistung der Anwendung

Der wichtigste Teil jeder Anwendung ist die Bedienbarkeit und eine intuitive Benutzerführung. Dem Nutzer muss klar sein, wie er welche Funktion aufrufen kann. Dieser Abschnitt stellt die geplanten Lösungen in drei Kategorien eingeteilt vor. Das Smartphone ist die Erste hiervon. Die Benutzeroberfläche ist technisch unabhängig von der Uhr und somit eignet es sich, dessen Bedienbarkeit eigenständig zu betrachten.

Innerhalb der Smartwatch wird zwischen der allgemeinen Navigationsführung und der Kartenansicht unterschieden. Diese beiden haben verschiedene Funktionalitäten und somit ist auch hier eine getrennte Betrachtung sinnvoll. Zum Abschluss des Kapitels werden Messungen der Laufzeit des Algorithmus und der Größe des Datensatzes analysiert und vorgestellt.

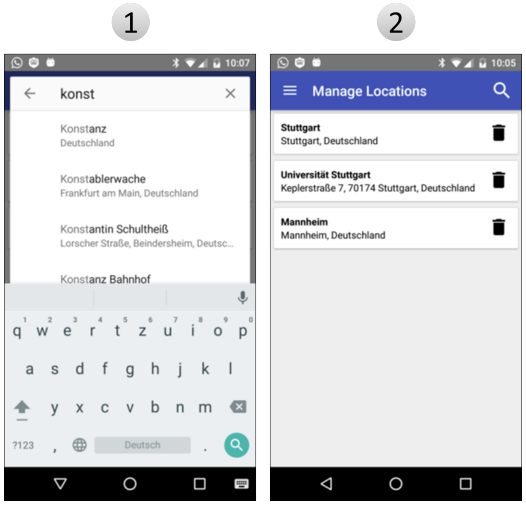


Abbildung 7.1: Oberfläche zur Zielauswahl

7.1 Verwaltungsansichten auf dem Smartphone

Die Oberfläche des Mobiltelefons ist schlicht gehalten. Sie bietet zwei Grundfunktionen: Das Managen vorhandener Routinginformationen auf der Smartwatch und das Vorbereiten von Navigationszielen auf dieser. Das Vorbereiten der Ziele lässt sich über eine Activity mit einem Textfeld und eine Liste der eingegebenen Ziele einfach lösen. Abbildung 7.1 zeigt in Screenshot 1, wie diese aussieht.

Eine der Aufgaben, die dabei übernommen werden muss, ist die Suche eines Ortes aus den getätigten Eingaben. Diese gestaltet sich einfach. Ein Aufruf der Google PlacesAPI übernimmt die vollständige Handhabung. Wie unter Nummer 2 erkennbar sieht der Benutzer ein einfaches Textfeld und kann dort den Anfang seines Ziels eingeben. Daraufhin erhält er mehrere Vorschläge zur Auto-Vervollständigung seiner Eingabe. Wählt der Benutzer ein Ziel aus oder bricht er die Eingabe ab verschwindet die API wieder. Der Eintrag wird gegebenenfalls in die Liste der Ziele übernommen. Zu jedem erstellten Eintrag in dieser Liste wird ein Button zum Entfernen des selbigen angeboten. Somit kann der Benutzer seine erwünschten Ziele verwalten. Jede Änderung wird sofort auf die Uhr synchronisiert.

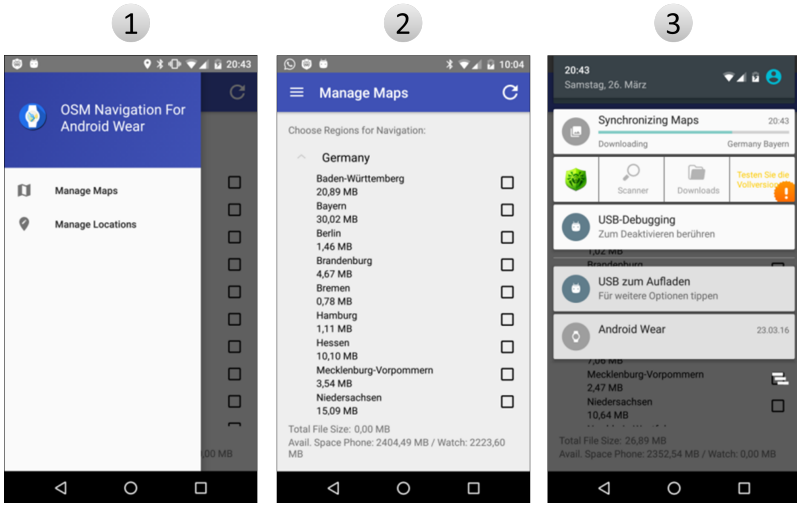


Abbildung 7.2: Dateiauswahloberfläche des Mobiltelefons

Über einen sogenannten Navigation Drawer, ein Menü welches an der Seite der Anwendung geöffnet werden kann und zur Auswahl verschiedener Acitivities dient, kann die Verwaltung der Daten erreicht werden. Die Funktionen hierfür beinhalten eine Liste in der zu Beginn eine Übersicht über alle vorhandenen Länder angezeigt wird. Jedes dieser Länder lässt sich aufklappen und entsprechende Bundesländer werden angezeigt und stehen zur Auswahl bereit. Zu jeder über eine Checkbox getroffene Auswahl wird der benötigte Speicherplatz angezeigt. In Kombination mit einer Anzeige des freien Speicherplatzes auf Uhr und Mobiltelefon kann der Benutzer entscheiden, wie viele Daten er auf die Uhr laden möchte. Bei jedem Klick auf eine der Checkboxen wird automatisch der entsprechende Download beziehungsweise die entsprechende Löschoperation

gestartet. Abbildung 7.2 zeigt in Screenshot 1 den Navigation Drawer und in Screenshot 2 die Auswahl vorhandener Kartendaten.

In einer Notification wird der derzeitige Fortschritt des Datentransfers eingeblendet. Beim Fortschreiten der Dateianzeige wird die bisher heruntergeladene Datenmenge im Verhältnis zur Gesamtdownloadmenge des aktuellen Bundeslandes angezeigt. Android bietet hierfür direkt eine Progressbar in den Notifications an. Ein Beispiel hierfür ist in Screenshot 3 abgebildet.

7.2 Verwaltungsansichten auf der Smartwatch

Wie in Abschnitt 4.3 besprochen, unterscheidet sich die Bedienung einer Smartwatch deutlich von der eines Smartphones. Folglich muss die Handhabung angepasst werden. Die Hauptfunktion der Menüoberfläche ist das Auswählen eines Ziels, damit die Navigation starten kann. Dieses kann der Benutzer auf verschiedene Arten erreichen, dazu erscheint beim Starten der Anwendung der Auswahlbildschirm, welcher sich in Abbildung 7.3 in Bild 1 befindet.

Abbildung 7.3: Menüs der Smartwatchoberfläche

Dieses wird über eine für Smartwatch-Anwendungen übliche Auswahlliste erledigt. Es werden drei Alternativen zur Auswahl angeboten. Die erste angezeigte Funktion öffnet die reine Kartenansicht an der gegenwärtigen Position. Es wird dabei vorerst keine Route berechnet. Dies ermöglicht dem Benutzer die aktuelle Umgebung zu überprüfen und eventuell über die Kartenansicht Ziele direkt einzutippen.

Die zweite Auswahlmöglichkeit ermöglicht die Spracheingabe von Zielen. Hierzu wird der Google Speech Recognizer in Kombination mit der auf dem Mobiltelefon bereits vorhandenen Google PlacesAPI verwendet. Die Durchführung der Spracheingabe wird vollständig der API überlassen. Diese beinhaltet eine standardisierte Oberfläche und die dazugehörige Funktionalität. Nach Abschluss oder Abbruch des Funktionsaufrufes durch den Benutzer wird das Ergebnis als Text zurückgegeben. Dieses wird im Falle einer erfolgreichen Anfrage direkt an das Telefon übermittelt, so dass dieses die Google PlacesAPI ansteuern kann. Die hiervon erhaltenen Zielvorschläge werden an die Uhr zurückgeliefert. Das Navigationsziel wird zur ersten der Alternative, welche Koordinaten beinhaltet, gestartet. Ein Beispiel der Oberfläche bei Eingabe und Auswahl eines Wertes ist in den Bildern 3 und 4 obiger Abbildung dargestellt.

Der dritte Eintrag in der Liste ist in Bild 2 dargestellt und ermöglicht es, die vordefinierten Ziele der Smartphoneanwendung einzusehen und eins dieser Ziele auszuwählen und somit die Navigation dorthin zu starten.

7.3 Navigationsansicht auf der Smartwatch

Wichtigste Anzeige auf der Smartwatch ist die Kartenanzeige und die Anzeige des aktuellen Weges. Die Karte folgt dabei grundsätzlich dem gegenwärtigen Standort. Über Wischen kann es allerdings in verschiedene Richtungen bewegt werden und es kann herein- beziehungsweise herausgezoomt werden. Wie in Abschnitt 6.3 besprochen, werden bei weiterem Herauszoomen kleinere Straßen entfernt. Diese Funktion hilft dem Benutzer die Übersicht zu erhalten und sorgt auch für einen erhöhte Leistung der Kartenrenderung. Die Einstufung der Wichtigkeit erfolgt nach der in Kapitel 4.1 festgelegten Kategorisierung. Ebenfalls werden unterschiedliche Muster für verschiedene Straßentypen dargestellt. Auch die Straßennamen werden gerendert. Hierbei wird zur Erhöhung der Übersichtlichkeit jeder Straßenname höchstens einmal im Bild dargestellt und es wird immer in Richtung der Straße gedreht. Ebenfalls werden unwichtigere Straßennamen eliminiert, sobald weit genug herausgezoomt worden ist. Dabei fällt die Darstellung der Namen etwas früher weg, als die Darstellugn der Wege, um die Abstufung etwas weicher erscheinen zu lassen. Abbildung 7.4 zeigt verschiedene Ansichten der Oberfläche.

Der erste Screenshot zeigt eine übliche Ansicht auf der Smartwatch. Es handelt sich um die Startposition einer Navigation. Dabei sind die lokalen Straßen in der Umgebung in grau gezeichnet. Des Weiteren ist der Standort der Uhr mittels eines blauen Punktes in der Mitte der Karte erkenntlich. Ebenfalls in blau ist der Weg dargestellt, den der Benutzer gehen muss, um sein Ziel schnellstmöglich zu erreichen. Zusätzlich befindet sich am unteren Teil des Bildschirmes eine Anzeige der Distanz bis zum Erreichen des Ziels und ein Kompass, welcher die Richtung zur nächsten Kreuzung anzeigt. Dieser dreht sich dynamisch, sodass direkt erkennbar ist, in welche Richtung zu gehen ist.

Abbildung 7.4: Navigationsansicht der Smartwatch

Der zweite Screenshot zeigt in Bild 1 in obiger Abbildung die Anwendung bei minimaler Zoomstufe. Dabei ist erkennbar, dass für die erhöhte Übersicht nur noch wichtige Straßennamen gerendert werden. Im Allgemeinen versucht die Anwendung Straßennamen möglichst im Zentrum zu platzieren. Der Algorithmus hierfür ist in Kapitel 6.3 beschrieben. Die Abbildung zeigt ebenfalls, dass Straßen in einer wichtigeren Kategorie größer dargestellt werden. Dieses ist verstärkt im dritten Screenshot sichtbar. Bundesstraßen und Autobahnen sind in gelb beziehungsweise blau dargestellt. Somit sind diese für den Benutzer leichter zu realisieren. Der letzte Screenshot zeigt die Oberfläche auf einer runden Uhr.

In allen Bildern sind die Funktionen am rechten Bildschirmrand zu erkennen. Über die '+' und '-'-Buttons kann der Benutzer die Zoomstufe verändern. Der dritte Button startet eine Funktion, welche die Anzeige wieder zentriert. Die Kartenansicht wird dabei neu eingestellt, so dass die Position des Benutzers im

Zentrum erscheint. Der letzte Button ermöglicht es ein neues Ziel auszuwählen. Nach einem Klick auf den Knopf, wird der Benutzer zur Auswahl eines neuen Zieles auf der Karte aufgefordert. Somit kann er kurzfristig seine Meinung ändern und an einen neuen Ort navigieren.

Insgesamt hat der Benutzer alle Funktionen, welche ein einfaches Navigieren erfordert auf einer Ansicht gesammelt und kann mit dieser sein Ziel erreichen.

7.4 Messergebnisse

Dieser Abschnitt stellt verschiedene Übersichten zu den erreichten Ergebnissen dar. Der Fokus liegt in der Datengröße, welcher für die einzelnen Bundesländer innerhalb von Deutschland erreicht werden kann und in der Laufzeit des ausgewählten Routingalgorithmus.

Eine weitere Angabe, welche keiner ausführlichen Analyse bedarf, ist die Dauer des Zeichnen eines Kartenteils. Je nach Zoomfaktor und Straßendichte des aktuellen Kartenausschnittes unterscheiden sich die Ergebnisse hierbei stark. Die benötigte Zeit befindet sich häufig in einem Fenster von ca. 15 ms in ländlichen Gegenden bis zu ca. 100 ms pro Bild in einigen Städten.

Zeilenbeschriftungen	Komprimiert	Kompr. + Filter	Unkomprimiert
Baden-Württemberg	14,28 MB	7,35 MB	132,17 MB
Bayern	20,51 MB	10,39 MB	193,70 MB
Berlin	0,98 MB	0,68 MB	9,14 MB
Brandenburg	3,17 MB	1,85 MB	29,92 MB
Bremen	0,52 MB	0,34 MB	5,08 MB
Hamburg	0,74 MB	0,47 MB	7,06 MB
Hessen	6,87 MB	3,80 MB	63,71 MB
Mecklenburg-Vorpommern	2,39 MB	1,36 MB	22,99 MB
Niedersachsen	10,29 MB	6,10 MB	91,33 MB
Nordrhein-Westfalen	18,75 MB	10,66 MB	169,99 MB
Rheinland-Pfalz	5,77 MB	2,87 MB	53,28 MB
Saarland	1,30 MB	0,66 MB	11,93 MB
Sachsen	6,13 MB	3,22 MB	58,89 MB
Sachsen-Anhalt	2,95 MB	1,66 MB	27,94 MB
Schleswig-Holstein	4,00 MB	2,20 MB	37,02 MB
Thüringen	3,53 MB	1,82 MB	33,81 MB
Gesamtergebnis	102,16 MB	55,43 MB	947,96 MB

Abbildung 7.5: Kompression der einzelnen Bundesländer

Besser bestimmbar ist die Größe der Kartendaten. Diese lässt sich exakt messen. Die Tabelle in Abbildung 7.5 zeigt die Datenmenge nach Bundesländern aufgelistet. Dabei sind drei Spalten an Vergleichswerten angegeben. Zuerst sind die in der Applikation verwendeten Daten in der Spalte „Komprimiert“ dargestellt. In der letzten Spalte „Unkomprimiert“ sind die Daten in ihrer Rohform angegeben. Hierbei werden die Inhalte aller Kanten als Integer-Werte mit 32 bzw. 8 Bit gespeichert. Um darzustellen, wie gut die Kompression werden könnte, worauf allerdings aufgrund einer optimierten Kartendarstellung verzichtet wurde, ist die mittlere Spalte angegeben. Diese filtert alle Kanten heraus, welche für die Navigation unerheblich sind. Es lässt sich feststellen, dass die unkomprimierten Daten um über 85 % reduziert werden können. Eine weitere Halbierung der Datenmenge wäre auf Kosten der Darstellung möglich. Innerhalb der einzelnen Bundesländer sind die Ergebnisse jeweils ähnlich. Die Größe

der Kartendaten hängt in erster Linie von der Anzahl der Straßen innerhalb des Landes ab, die Kompressionsverfahren haben auf diese Relation keinen Einfluss.

Abbildung 7.6 zeigt die Daten für Gesamtdeutschland nach Kategorie aufgeteilt, wobei die vorhandenen Metadaten an dieser Stelle vernachlässigt sind. Es ist erkennbar, dass Startknoten und Latitude gegenüber beispielsweise dem Entknoten und der Longitude deutlich weniger Speicher benötigen. Dies liegt an der vorhandenen Sortierung für diese Eigenschaften. Somit kann die Struktur der Sortierung passend genutzt werden, während für die anderen Werte weniger hilfreiche Strukturinformationen genutzt werden können.

Die weiteren Eigenschaften, einerseits die Kategorie der Wege und die Straßennamen benötigen kaum Speicherplatz. Dies liegt in der geringen Auswahl an möglichen Zeichen, die beide besitzen. So gibt es für die Kategorie nur sechs Möglichkeiten, während die Straßennamen zwar über 50 mögliche Zeichen haben, allerdings wiederholen sich die Straßen wiederum häufig. Die Zuordnung einer Straße zu ihrer ID und die Länge der Straßen sind in der Mitte der angegebenen Werte.

Eigenschaften	Größe
Startknoten	5,37 MB
EndKnoten	18,89 MB
Kategorie	4,55 MB
StraßenID	16,95 MB
Length	16,78 MB
Lat	8,40 MB
Lon	26,47 MB
Straßenname	4,55 MB
Gesamt	102,16 MB

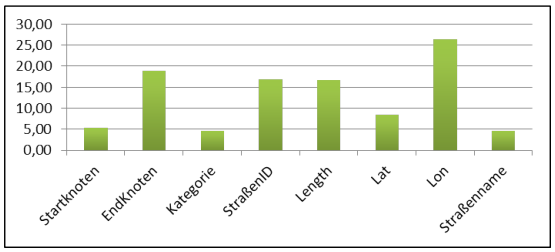


Abbildung 7.6: Kompression der einzelnen Kategorien

Zuletzt werden die Navigationsalgorithmen gemessen. Deren Zeiten sind nach dem Dijkstra-Rang gemessen und eingeteilt. Von einem beliebigen Startpunkt aus kann jedem Knoten ein solcher Rang zugeordnet werden, indem ein Dijkstra gestartet wird, welcher alle Knoten berechnet. Dabei wird aus der Reihenfolge in der die Knoten bearbeitet werden, der entsprechende Rang bestimmt. Dem ersten Knoten der bearbeitet wird, wird Rang 1 zugewiesen, dem zweiten wird Rang 2 zugewiesen, und so weiter.

Rang	Laufzeit
100	177 ms
200	190 ms
500	207 ms
1.000	232 ms
2.000	264 ms
5.000	343 ms
10.000	456 ms
20.000	630 ms
50.000	1.082 ms
100.000	1.764 ms
200.000	2.943 ms
500.000	4.979 ms
1.000.000	12.923 ms

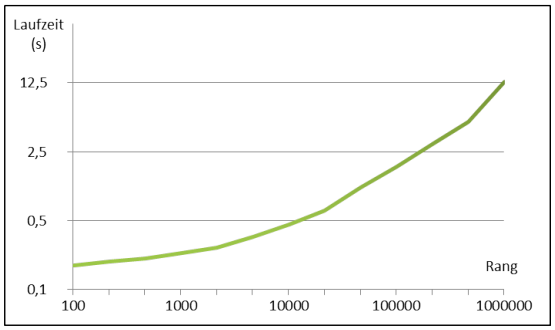


Abbildung 7.7: Laufzeitmessungen des Algorithmus

In Abbildung 7.7 wird die Übersicht der Laufzeiten des Algorithmus in ver-

schiedenen Distanzen dargelegt. Dabei sind die Distanzen als Dijkstra-Rang angegeben und die Laufzeit in Millisekunden. Der Graph zeigt verschiedene Erkenntnisse. Einerseits ist ein Offset bei etwas über 150 ms erkennbar. Dies ist ungefähr die Zeit, die das Einlesen einer Zelle benötigt und kann somit nicht weiter reduziert werden. Des Weiteren ist ebenfalls erkennbar, dass die Laufzeit trotz logarithmischer Skala in x- und y-Achse weiterhin überproportional ansteigt. Dies liegt einerseits am A*-Algorithmus, welcher keine proportionale Laufzeit erreicht und verstärkt wird diese Entwicklung durch die Arbeitsspeicherknappheit, welche dafür sorgt, dass mit höheren Distanzen immer mehr Zellen aus dem Arbeitsspeicher entfernt werden und eventuell noch einmal eingelesen werden müssen.

Kapitel 8

Zusammenfassung und Ausblick

Im Verlauf dieser Arbeit wurden zuerst verschiedene Grundlagen erörtert. Dies waren Informationen über Smartwatches, über deren historische Entwicklung bis zu den erwarteten zukünftigen Verkaufszahlen. Daraufhin wurde mit Android eines der populären Betriebssysteme für Smartwatches eingeführt. Zu diesem wurden der Aufbau und verschiedene Möglichkeiten zur Entwicklung erörtert. Dazu gehören die einzelnen Programmiersprachen, vorhandene Technologien und unterstützende Tools. Ebenso wurde das Arbeitsspeichermanagement von Android beschrieben. Der nächste Abschnitt widmete sich Kartendaten. Dort wurde das OSM-Projekt vorgestellt und der Aufbau und die vorhandenen Informationen in den angebotenen Karten analysiert.

Da diese Daten viel Speicherplatz benötigen, wurden daraufhin verschiedene Möglichkeiten zur Datenkompression vorgestellt. Dabei wurden zuerst verschiedene Begriffe und Kategorisierungen von Codes erläutert und im zweiten Teil wurden der Huffman Code, der Unary Code, der Golomb Code und ein Teil des GZIP-Algorithmus erläutert.

Im letzten Abschnitt des ersten Kapitels wurde das Shortest-Path Problem dargestellt. Ebenfalls wurden verschiedene Algorithmen eingeführt und deren Eigenschaften beleuchtet. So wurde unter anderem der Dijkstra- mit dem A*-Algorithmus verglichen. Dabei wurden jeweils die unidirektionalen und die bidirektionalen Varianten einzeln vorgestellt und gegenübergestellt. Zusätzlich wurden beim A*-Algorithmus verschiedene Heuristiken eingeführt, um dessen Performance optimal nutzen zu können. Des Weiteren wurde mit CH ein sehr schneller Algorithmus erläutert, welcher allerdings einen speziell vorbereiteten Graphen benötigt.

In Kapitel 4 wurde die Aufgabenstellung analysiert und anschließend wurden verschiedene funktionale und nicht-funktionale Anforderungen abgeleitet. Daraufhin wurden Smartwatches nach ihren Stärken und Schwächen analysiert sowie das Referenzmodell, die Sony Smartwatch 3 vorgestellt. Mit den Informationen über technische Eigenschaften und auch Bildschirmgrößen konnten einige Best Practices in der Gestaltung von Benutzerinterfaces auf Smartwatches angewendet werden. Es wurden beispielsweise die verschiedenen Möglichkeiten erläutert, wie Anwendungen auf einer Smartwatch aussehen können. Zuletzt

wurden mehrere Schnittstellen eingeführt, da diese die nachfolgende Entwicklung vereinfachen können. Für Smartwatches steht der Fused Location Provider bereit, welcher das GPS von sowohl Smartphone als auch Smartwatch verwenden kann. Des Weiteren stehen drei APIs zum Austausch von Daten zwischen den Geräten zur Verfügung. Hiervon wurden die ChannelAPI zum Austausch von Dateien und die DataAPI zum Austausch von Steuerungsinformationen ausgewählt. Außerdem wurde die Google PlacesAPI eingeführt, welche für das Smartphone eine komfortable Möglichkeit bietet, um aus Texteingaben Orte zu gewinnen.

Das nächste Kapitel ist das erste, in welchem die Entwicklung der Anwendung direkt thematisiert ist. Zu Beginn wurde dabei eine Architektur des gesamten Systems entwickelt. Dabei wurden verschiedene Komponenten abgeleitet, in welche die Anwendung aufgeteilt wird. Dabei wurden die drei zentralen Komponenten Server, Smartphone und Smartwatch vorgestellt, welche wiederum jeweils aus mehreren Subkomponenten bestehen. Daraufhin wurden die benötigten Informationen detailliert analysiert. Es wurde exakt festgelegt, welche Daten benötigt werden. Dazu gehört der Aufbau von Knoten und Kanten. An den Datensatz existieren verschiedene Anforderungen, um eine gute Leistung zu erzielen.

Um alle Anforderungen effizient implementieren zu können, wurde die Weltkarte mit einem Gitternetz überzogen und die Knoten wurden in Zellen eingeteilt. Ebenfalls wurde festgelegt, dass die Dateien nach Bundesländern gruppiert und komprimiert gespeichert werden. Um eine starke Kompressionsrate zu erreichen, wurden diverse Codes angewendet. Eine komplexe Kompressionsvorschrift wurde entwickelt, welche jede Informationskategorie nach eigenen Vorschriften codiert. Somit konnte die Datenmenge, welche für Gesamtdeutschland benötigt wird, von 968 MB auf 102 MB reduziert werden.

Die Auswahl des besten Algorithmus ist aufwendig gestaltet, um einerseits verschiedene Erkenntnisse über die Performance der vorgestellten Grundlagen zu erhalten und andererseits neben dem Algorithmus auch Erkenntnisse über die beste Heuristik und die beste Datenstruktur zu erhalten. Es wurde ein Testablauf konzipiert, welcher detaillierte Ergebnisse über die einzelnen Komponenten des Algorithmus ergab. Dabei wurden die einzelnen Kandidaten vorgestellt. In diesem Zusammenhang wurden die drei möglichen Datenstrukturen für die Kanten erläutert. Es gibt die Möglichkeit zur Speicherung der Knoten und Kanten in Arrays oder in verketteten Listen, wobei die verketteten Listen sich in eine normale und eine speicheroptimierte Variante aufteilen. Im Ergebnis stellte sich der unidirektionale A*-Algorithmus als am besten geeignet heraus. Als Datenstruktur konnte eindeutig die Arraystruktur bestimmt werden und zuletzt wurde als Heuristik die Berechnung der Hypotenuse ausgewählt, welche sich als geringfügig besser als die Alternativen erwies. Der CH-Algorithmus war durch den Testlauf benachteiligt und ist in der implementierten Form nicht für Smartwatches geeignet.

Im letzten Abschnitt dieses Kapitels wurden betriebswirtschaftliche Aspekte analysiert. Es wurde erörtert, dass beispielsweise das Schalten von Werbung oder der Verkauf von Zusatzfunktionen mögliche Einnahmequellen sind. Allerdings wird die Anwendung zu Beginn ohne direkte oder indirekte Kosten für den Endkonsumenten sein. Um einen großen Kundenstamm zu generieren, kann am besten auf die Infrastruktur von Wearablesoftware zurückgegriffen werden.

Das sechste Kapitel legte ausgewählte Themen der finalen Anwendung vor.

Begonnen wird mit dem detaillierten Ablauf der Vorverarbeitung. Diese wurde in 6 Phasen eingeteilt und an verschiedenen Stellen optimiert. Eine der Optimierungen betrifft die Auswahl, um wie viele Knoten die Darstellung von Kurven reduziert werden kann, ohne dass die Zeichenqualität stark leidet. Daraufhin wurde die Implementierung des Algorithmus genauer betrachtet. Die Berechnung der Hypotenuse als Heuristik benötigt eine Umrechnung von Koordinaten in Meter, welche ebenfalls optimiert ist. Des Weiteren ist das Abfragen der Knoten essentiell für den Algorithmus. Im dritten Absatz wurde das Zeichnen der Kartendarstellung erläutert. Mittels möglichst früher Filterung nicht benötigter Knoten kann bei der Iteration über einzelne Zellen sehr viel Zeit gespart werden. Der Vorgang des Zeichnens von Kanten und Straßennamen ebenfalls sehr performant implementiert.

Im letzten Kapitel wurde das Aussehen der Anwendung vorgestellt. Die einzelnen Oberflächen der Anwendung auf dem Smartphone und der Smartwatch sind mittels Screenshots abgebildet und deren Funktionen sind erläutert.

In dieser Masterarbeit ist somit eine vollwertige und hilfreiche Smartwatch-App entstanden. Nach zusätzlichen Stabilitätstests wird diese zeitnah im Play-Store veröffentlicht und daraufhin von jedem heruntergeladen werden kann.

Im Allgemeinen funktioniert die Bedienung der entstandenen Anwendung in der aktuellen Form bereits sehr gut und ist intuitiv bedienbar. Es sind kaum Verzögerungen bemerkbar und speziell die Kompression der Kartendaten für Gesamtdeutschland auf nur knapp über 100 MB ist ein Herzstück der effizienten Leistung. Über das Herunterladen der einzelnen Karten bis zur Anzeige wird stetig über den Zwischenstand informiert und die Anwendung reagiert schnell auf Eingaben. Durch entsprechende Batteriesparmaßnahmen werden die erwarteten 90 Minuten an Batterielaufzeit gut erreicht und auch die Ausführungsgeschwindigkeit der Algorithmen liegen mit unter 6 Sekunden für lange Distanzen und deutlich unter 2 Sekunden für kurze Distanzen weit unter den erlaubten Laufzeiten.

Sie besitzt die geplante schlichte Bedienoberfläche und kann innerhalb von Deutschland die Navigation übernehmen. Zukünftig ist nach ersten Benutzerrückmeldungen das Hinzufügen weiterer Länder und weiterer Funktionalität geplant. Die Möglichkeiten hierfür sind unbegrenzt. Evaluiert wird zum Beispiel der Nutzen einer erweiterten Sprachsteuerung, mit welcher eine Alternativroute zu einer Baustelle gefunden werden kann.

Abbildungsverzeichnis

Algorithmenverzeichnis

Anhang A

Fachwortverzeichnis

Eine **Activity** ist in Android das Hintergrundelement, auf welchem sichtbare graphische Elemente platziert werden können. Es übernimmt die Steuerung des Anwendungslebenszyklus.

Android ist ein Betriebssystem für mobile Geräte wie Smartwatches, Smartphones oder Tablets.

Der **Android Device Manager** ist ein Tool, welches in der Anwendungsentwicklung verwendet werden kann, um Androidgeräte zu manipulieren und damit unter anderem Tests zu erleichtern.

Android Studio ist die Entwicklungsumgebung welche von Google für die Entwicklung in Android bereitgestellt wird.

Ein **Alphabet** ist in der Datenkompression die Menge aller vorhandenen Symbole, welche codiert werden.

Eine **binäre Suche** ist ein Algorithmus zum Finden eines Wertes in einer sortierten Liste.

Die **ChannelAPI** ist eine Programmierschnittstelle zur Kommunikation zwischen Smartwatch und Smartphone. Sie nutzt Streams zur Kommunikation.

Ein **Code** ist eine Vorschrift, wie Daten abgebildet werden.

Die **DataAPI** ist eine Programmierschnittstelle zur Kommunikation zwischen Smartwatch und Smartphone. Sie nutzt Nachrichten zur Kommunikation und garantiert deren Zustellung.

Double Buffering ist eine Technik um Flackern bei der Anzeige von Bildern zu vermeiden. Dabei wird ein neues Bild erst auf dem Monitor angezeigt, wenn die Berechnung vollständig abgeschlossen ist. Bis dorthin wird das zuletzt berechnete Bild angezeigt.

Die **Entropie** ist in der Datenkompression der Informationsgehalt, den eine Nachricht besitzt. Sie kann in Bits angegeben werden.

Gradle ist das bevorzugte Build-System für Androidanwendungen. Es ermöglicht unter anderem die Verwaltung von verwendeten Bibliotheken und das Packen der Anwendung in installationsfähige Dateien.

Eine **LookUp Table** ist eine Tabelle mit vorberechneten Werten von komplexen mathematischen Operationen. Sie wird verwendet, um eine höhere Leistung zu erzielen, indem die Berechnung vor der Laufzeit des Programmes stattfindet.

Lint ist ein Tool zur Unterstützung in der Softwareentwicklung. Es erkennt typische Probleme im Quellcode und gibt Warnungen aus.

Ein **Least Recently Used Cache** ist ein Speicher mit begrenzter Kapazität, der bei einem Überlauf immer das Element entfernt, welches am längsten nicht verwendet wurde.

Die **MessageAPI** ist eine Programmierschnittstelle zur Kommunikation zwischen Smartwatch und Smartphone. Sie nutzt Nachrichten zur Kommunikation.

Eine **Notification** ist eine Benachrichtigung auf dem Smartphone oder der Smartwatch welche von einer Anwendung mit einem beliebigen Inhalt versehen werden kann.

Die **Google PlacesAPI** ist eine Programmierschnittstelle zum Suchen nach Punkten auf der Landkarte.

Der **PlayStore** ist der AppStore von Google. Dort können Anwendungen für Android gefunden und heruntergeladen werden.

PostGis ist eine Erweiterung für PostgreSQL Datenbanksysteme, welche vielseitige Funktionen für die Handhabung von geographischen Daten bietet.

Renderscript ist eine von Google entwickelte Programmiersprache, um aufwendige Berechnungen parallelisiert zu verarbeiten.

Ein **Shortcut** ist im Algorithmus Contraction Hierarchies ein Weg, welcher durch die Vorberechnung in den Graphen eingefügt wurde und real nicht existiert.

Ein **Symbol** ist in der Datenkompression ein Element, welches codiert wird.

Ein **Watchface** bezeichnet die Oberfläche von Smartwatches auf denen die Uhrzeit angezeigt wird. In der Regel sind diese programmatisch frei gestaltbar und können mit diversen Funktionen versehen werden.

Ein **Wörterbuch** ist in der Datenkompression eine beidseitige Zuordnungsvorschrift zwischen allen Symbolen und allen Codewörtern.

Anhang B

Algorithmen

Algorithmus 13 : Bidirektionaler Dijkstra

```
 1 Q1,Q2 ← new MinHeap()
 2 Q1.add(s)
 3 Q2.add(t)
 4 shortestDistance ← ∞
 5 expectedMeetingNode ← null
 6 while Q1 ≠ ∅ or Q2 ≠ ∅ do

 7     u1 ← Q1.extractMin()
 8     foreach v with (u1,v) ∈ E do
 9         distance ← u1.dist + v.length
10         if distance < v.distanceToStart then
11             v.distanceToStart ← distance
12             v.precedingNode ← u1
13             Q1.add(v)
14             if v.followingNode is set and v.distToT + v.distToS <
                shortestDistance then
15                 expectedMeetingNode ← v
16                 shortestDistance ← v.distToT + v.distToS

17     u2 ← Q2.extractMin()
18     foreach v with (u2,v) ∈ E do
19         distance ← u2.dist + v.length
20         if distance < v.distanceToTarget then
21             v.distanceToTarget ← distance
22             v.followingNode ← u2
23             Q2.add(v)
24             if v.precedingNode is set and v.distToT + v.distToS <
                shortestDistance then
25                 expectedMeetingNode ← v
26                 shortestDistance ← v.distToT + v.distToS

27     if u1.distToS + u2.distToT > shortestDistance then
           /* Shortest path has been found                          */
28         return expectedMeetingNode
29 return no Result
```

Algorithmus 14 : Bidirektionaler A*

```
1  Q1,Q2 ← new MinHeap()
2  Q1.add(s)
3  Q2.add(t)
4  shortestDistance ← ∞
5  expectedMeetingNode ← null
6  while Q1 ≠ ∅ or Q2 ≠ ∅ do

7      u1 ← Q1.extractMin()
8      foreach v with (u1,v) ∈ E do
9          distance ← u1.dist + v.length + h(v,t)
10         if distance < v.distanceToStart then
11             v.distanceToStart ← distance
12             v.precedingNode ← u1
13             Q1.add(v)
14             if v.followingNode is set and v.distToT + v.distToS <
                shortestDistance then
15                 expectedMeetingNode ← v
16                 shortestDistance ← v.distToT + v.distToS

17     u2 ← Q2.extractMin()
18     foreach v with (u2,v) ∈ E do
19         distance ← u2.dist + v.length + h(v,t)
20         if distance < v.distanceToTarget then
21             v.distanceToTarget ← distance
22             v.followingNode ← u2
23             Q2.add(v)
24             if v.precedingNode is set and v.distToT + v.distToS <
                shortestDistance then
25                 expectedMeetingNode ← v
26                 shortestDistance ← v.distToT + v.distToS

27     if u1.distToS + u2.distToT > shortestDistance then
           /* Shortest path has been found                          */
28         return expectedMeetingNode
29 return no Result
```

Algorithmus 15 : Dijkstra for CH

```
1  Q1,Q2 ← new MinHeap()
2  Q1.add(s)
3  Q2.add(t)
4  shortestDistance ← ∞
5  expectedMeetingNode ← null
6  while Q1 ≠ ∅ or Q2 ≠ ∅ do

7      u1 ← Q1.extractMin()
8      foreach v with (u1,v) and u1.level >= v.level ∈ E do
9          distance ← u1.dist + v.length
10         if distance < v.distanceToStart then
11             v.distanceToStart ← distance
12             v.precedingNode ← u1
13             Q1.add(v)
14             if v.followingNode is set and v.distToT + v.distToS <
                shortestDistance then
15                 expectedMeetingNode ← v
16                 shortestDistance ← v.distToT + v.distToS

17     u2 ← Q2.extractMin()
18     foreach v with (u2,v) and u2.level >= v.level ∈ E do
19         distance ← u2.dist + v.length
20         if distance < v.distanceToTarget then
21             v.distanceToTarget ← distance
22             v.followingNode ← u2
23             Q2.add(v)
24             if v.precedingNode is set and v.distToT + v.distToS <
                shortestDistance then
25                 expectedMeetingNode ← v
26                 shortestDistance ← v.distToT + v.distToS

27     if u1.distToS + u2.distToT > shortestDistance then
           /* Shortest path has been found                          */
28         return expectedMeetingNode
29 return no Result
```

Algorithmus 16 : A* for CH

```
1  Q1,Q2 ← new MinHeap()
2  Q1.add(s)
3  Q2.add(t)
4  shortestDistance ← ∞
5  expectedMeetingNode ← null
6  while Q1 ≠ ∅ or Q2 ≠ ∅ do

7      u1 ← Q1.extractMin()
8      foreach v with (u1,v) and u1.level >= v.level ∈ E do
9          distance ← u1.dist + v.length + h(v,t)
10         if distance < v.distanceToStart then
11             v.distanceToStart ← distance
12             v.precedingNode ← u1
13             Q1.add(v)
14             if v.followingNode is set and v.distToT + v.distToS <
                shortestDistance then
15                 expectedMeetingNode ← v
16                 shortestDistance ← v.distToT + v.distToS

17     u2 ← Q2.extractMin()
18     foreach v with (u2,v) and u2.level >= v.level ∈ E do
19         distance ← u2.dist + v.length + h(v,t)
20         if distance < v.distanceToTarget then
21             v.distanceToTarget ← distance
22             v.followingNode ← u2
23             Q2.add(v)
24             if v.precedingNode is set and v.distToT + v.distToS <
                shortestDistance then
25                 expectedMeetingNode ← v
26                 shortestDistance ← v.distToT + v.distToS

27     if u1.distToS + u2.distToT > shortestDistance then
           /* Shortest path has been found                          */
28         return expectedMeetingNode
29 return no Result
```

Literaturverzeichnis

[1] Kozyntsev A.N. Bidirectional search and Goal-directed Dijkstra. http://http://wwwmayr.informatik.tu-muenchen.de/lehre/2010SS/sarntal/07_kozyncev_slides.pdf. Accessed: 2016-03-11.

[2] Android Debug Brigde. http://www.droidwiki.de/Android_Debug_Bridge. Accessed: 2016-03-11.

[3] Apache. Cordova. https://cordova.apache.org. Accessed: 2016-03-11.

[4] Apple. iOS. http://www.apple.com/ios/. Accessed: 2016-03-11.

[5] Apple watch scooped up over half the smartwatch market in 2015. http://techcrunch.com/2016/01/13/apple-watch-scooped-up-over-half-the-smartwatch-market-in-2015/. Accessed: 2016-03-11.

[6] „Apple-Watch-Zuschuss“ startet bei der Techniker Krankenkasse. http://www.mobiflip.de/apple-watch-zuschuss-techniker-krankenkasse/. Accessed: 2016-03-11.

[7] Tobias Bagg. Externe Komprimierte Graphdarstellungen. Master's thesis, Universität Stuttgart, 2014.

[8] Stefan Bühler. Onboard Routenplanung auf dem Smartphone. Master's thesis, Universität Stuttgart, 2013.

[9] Marta E. Cecchinato, Anna L. Cox, and Jon Bird. Smartwatches: The good, the bad and the ugly? In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 2133–2138, New York, NY, USA, 2015. ACM.

[10] Hamilton Watch Company. Pulsar LED Watches. http://www.oldpulsars.com/. Accessed: 2016-03-11.

[11] Datenkompression: Allgemeine Einfürung. https://de.wikibooks.org/wiki/Datenkompression:_Allgemeine_Einführung. Accessed: 2016-03-11.

[12] Datenkompression: Verlustfreie Verfahren: Statistische Verfahren: Präfix Codes. https://de.wikibooks.org/wiki/Datenkompression:_Verlustfreie_Verfahren:_Statistische_Verfahren:_Präfix_Codes. Accessed: 2016-03-11.

[13] Linux Foundation. Tizen. https://www.tizen.org. Accessed: 2016-03-11.

[14] Robert Geisenberger. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. Master's thesis, Universität Karlsruhe, 2008.

[15] Geofabrik. OpenStreetMap Data Extracts. http://download.geofabrik.de. Accessed: 2016-03-11.

[16] Google. Android. https://www.android.com. Accessed: 2016-03-11.

[17] Google. Android History. https://android.com/history/. Accessed: 2016-03-11.

[18] Google. Android Studio Overview. http://developer.android.com/tools/studio/index.html. Accessed: 2016-03-11.

[19] Google. App Structure for Android Wear. http://developer.android.com/design/wear/structure.html. Accessed: 2016-03-19.

[20] Google. ART and Dalvik. https://source.android.com/devices/tech/dalvik/. Accessed: 2016-03-11.

[21] Google. Computation. http://developer.android.com/guide/topics/renderscript/index.html. Accessed: 2016-03-11.

[22] Google. Detecting Location on Android Wear. http://developer.android.com/training/articles/wear-location-detection.html. Accessed: 2016-03-11.

[23] Google. Info - Google Maps. https://www.google.com/maps/about/. Accessed: 2016-03-11.

[24] Google. JNI Tips. http://developer.android.com/training/articles/perf-jni.html. Accessed: 2016-03-11.

[25] IBM. Linux Watch IBM. http://researcher.watson.ibm.com/researcher/view_group.php?id=6101. Accessed: 2016-03-11.

[26] irontec. PHP for Android (PFA). http://phpforandroid.net/doku.php. Accessed: 2016-03-11.

[27] Ildar Klassen. Profilbasierte Navigation auf einem Mobiltelefon (iPhone). Master's thesis, Universität Koblenz Landau, 2011.

[28] The use of smartwatches. http://download.geofabrik.de. Fachgespräch mit Cedrik Larrat am: 2016-01-17.

[29] LG. LG Smart Watches. http://www.lg.com/us/smart-watches. Accessed: 2016-03-11.

[30] Here. https://company.here.com/here/. Accessed: 2016-03-11.

[31] Onemate. Onemate TrueSmart. http://www.omate.com/smartwatch.omate.trueSmart.html. Accessed: 2016-03-11.

[32] OpenGL ES. http://developer.android.com/guide/topics/graphics/opengl.html. Accessed: 2016-03-11.

[33] OpenStreetMap - Deutschland. http://www.openstreetmap.de/faq.html. Accessed: 2016-03-11.

[34] C. Vetter P. Sanders, D. Schultes. Mobile Route Planning. In *ESA 2008*, pages 732–743, Karlsruhe, September 2008.

[35] Pebble. Pebble-Smartwatches. https://www.pebble.com/our_story. Accessed: 2016-03-11.

[36] G. Pomberger and H. Dobler. *Algorithmen und Datenstrukturen: eine systematische Einführung in die Programmierung.* Pearson Studium - IT. Pearson Studium, 2008.

[37] D. Salomon. *Variable-length Codes for Data Compression.* Springer London, 2007.

[38] D. Salomon. *Data Compression: The Complete Reference.* Springer Berlin Heidelberg, 2012.

[39] Samsung. Wearable Tech. http://www.samsung.com/us/mobile/wearable-tech. Accessed: 2016-03-11.

[40] Samsung Gear s2 oder Huawei Watch für 150 Euro bei Smartphone-Kauf mit Vertrag. http://www.areamobile.de/vodafone/vodafone-samsung-gear-s2-oder-huawei-watch-fuer-150-euro-bei-smartphone-kauf-mit-vertrag. Accessed: 2016-03-11.

[41] Smartwatch Uptick in Mobile Slump. http://www.eetimes.com/document.asp?doc_id=1326427. Accessed: 2016-03-11.

[42] Lvmh's tag heuer to step up smartwatch production to meet demand. http://www.bloomberg.com/news/articles/2015-12-04/lvmh-s-tag-heuer-to-step-up-smartwatch-production-to-meet-demand. Accessed: 2016-03-11.

[43] A. Villas-Boas. LG's latest smartwatch can make phone calls without your smartphone. http://www.techinsider.io/lg-watch-urbane-2-2015-11. Accessed: 2016-03-11.

[44] Wearable-Markt wä und wandelt sich. http://www.it-markt.ch/News/2015/12/18/Wearable-Markt-waechst-und-wandelt-sich.aspx. Accessed: 2016-03-11.

[45] S. Wickenburg, A. Rooch, and Groß J. Die JPEG-Kompression. http://mathematik.de/spudema/spudema_beitraege/beitraege/rooch/nkap04.html. Accessed: 2016-03-11.

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.

Worms, 11.04.2016

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Worms, 11.04.2016