UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

AGKI
artificial intelligence research koblenz
AG Künstliche Intelligenz

Profilbasierte Navigation auf einem Mobiltelefon (iPhone)

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Ildar Klassen

Erstgutachter:	Prof. Dr. Ulrich Furbach AG Künstliche Intelligenz
Zweitgutachter:	Dipl.-Inf. Markus Maron AG Künstliche Intelligenz

Koblenz, im Dezember 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen des Projektes NAPA [3] werden einige Neuerungen in der Fußgängernavigation entwickelt. Als Erstes wäre der Satellitennavigationssystem Empfänger zu nennen, der eine Positionierungsgenauigkeit von 1 Meter erreichen soll. Dies ermöglicht eine genauere Navigation, die für Fußgänger geeignet ist. Aufgrund der höheren Genauigkeit werden genauere Karten für die Fußgänger gebraucht. Diese neuen Karten sind ebenfalls ein Teil des Projektes NAPA. Ein Navigationssystem, das diese Technologien nutzt, wird ebenfalls in dem Projekt entwickelt. Abgerundet wird das Ganze mit einem Informationssystem, das lokale Informationen online abrufen und dem Navigationssystem bereitstellen soll. Alle diese Technologien sollen für den Gebrauch in Mobiltelefonen entwickelt werden. Als Referenzplattform wurde das iPhone ausgewählt.

Der Empfänger ist eine Neuentwicklung und soll das neue Satellitennavigationssystem Galileo und das bisherige Global Positioning System (GPS) benutzen. Die Genauigkeit von GPS ist ca. 10m [18], die von Galileo ca. 4m [18]. Durch den gleichzeitigen Empfang beider Signale soll, laut den Entwicklern des NAPA-Empfängers [3], eine Genauigkeit von 1m erreicht werden.

NAVTEQ [5] entwickelt für das Projekt die Kartendaten. Die neuen Karten sollen Fußgänger spezifische Daten enthalten. Man wird nicht mehr wie bisher üblich auf der Straßenmitte navigiert, sondern auf dem Bürgersteig bzw. dem Fußgängerweg. Zusammen mit der höheren Genauigkeit des Empfängers kann also die Straßenseite unterschieden werden. Die Kartendaten enthalten Zebrastreifen, Ampeln, Verkehrsintensität von Fußgängerüberwegen und andere nützliche Informationen. Auf diese Weise ist eine bessere Navigation von Fußgängern möglich.

Die Navigationssoftware für das Projekt wird von Navigon [4] entwickelt.

Die Universität Koblenz beteiligt sich an dem Projekt mit einer Informationsplattform, die mit der Navigationssoftware interagiert.

In dieser Arbeit soll ebenfalls ein Navigationssystem entwickelt werden, das als Alternative zu der Lösung von Navigon zur Verfügung stehen soll. Dabei soll es speziell an die Bedürfnisse des Benutzers angepasst werden. Die Präferenzen

werden in Profilen gespeichert und können *fein* eingestellt werden. Der Benutzer soll sich nicht *nur* strikt entscheiden müssen, ob er einen nicht asphaltierten Weg benutzt oder nicht benutzt. Die Einstellung soll Zwischenwerte erlauben, die eine Tendenz ausrücken. Z.B. würde ein Benutzer eine nicht asphaltierte Strecke benutzen, wenn diese mindestens 20% kürzer als die kürzeste asphaltierte Strecke ist. Für einen anderen Benutzer müsste die nicht asphaltierte Strecke um mindestens 40% kürzer sein, um bevorzugt zu werden. Diese Arbeit soll die Grundlage für ein größeres System bieten, das im Kapitel 1.1 beschrieben ist.

1.1 Vision

Stellen Sie sich folgendes Szenario vor.

Sie sind gerade in einer fremden Stadt angekommen. Sie müssen zu einem Meeting und die Adresse haben Sie in ihrem iPhone gespeichert. Sie fragen einen Passanten nach dem Weg. Er kennt sich nicht aus, aber er sieht, dass Sie ein iPhone besitzen, und empfiehlt Ihnen eine neue Navigations-App. Sie laden sich die App runter und geben das Ziel ein. Die App berechnet einen Weg und Sie folgen ihm. Der Weg führt durch einen Parkplatz mittendurch und dann durch eine Straßenunterführung. Da merken Sie, dass die App speziell für Fußgänger ist. Nach einigen Minuten erreichen Sie ihr Ziel.

Nach dem Meeting brauchen Sie ein Hotel und probieren die neue App aus. Sie geben in die Suche „Hotel“ ein und sehen die Ergebnisse auf der Karte angezeigt. Sie wählen eins aus und die App berechnet den Weg dort hin. Nach dem Sie eine lange Treppe gehen mussten, wollen Sie die Einstellungen ausprobieren. Als Sie die Einstellung der Treppenbenutzung ändern, merken Sie, dass die Einstellung nicht nur an oder aus erlaubt, sondern auch einen gleitenden Wert dazwischen. So stellen Sie den Wert auf „fast aus“. Im Hotel angekommen haben Sie noch Zeit übrig um die Stadt zu erkunden.

Sie haben die App vergessen und spazieren einfach durch die Stadt. Als Sie an einem Imbiss vorbei gehen, meldet sich Ihr iPhone. Es ist die neue App und sie zeigt ihnen ein Angebot des Imbisses. Da Sie nicht gerne in einem Imbiss essen, klicken Sie den „Nicht interessiert“ Knopf und gehen weiter. An einem Restaurant bekommen Sie noch mal eine Info von der App. Es ist ein Menü des Restaurants. Sie entscheiden sich, dort zu essen.

Nach dem Essen wollen Sie gezielt die Sehenswürdigkeiten besuchen. Sie nutzen die App und suchen nach „Sehenswürdigkeit“ in der Karte mit den Ergebnissen klicken Sie mehrere an und setzen Sie auf eine Liste. Dann folgen Sie einfach der Route, die die App berechnet hat. Am Ende der Route angekommen begeben Sie sich in ihr Hotel.

Diese Vision beschreibt ein System, das Navigationssystem und Informationssystem vereint. Diese Arbeit konzentriert sich auf die Navigationskomponente.

1.2 Verwendete Materialien

Um eine App für das iPhone zu entwickeln, werden normalerweise ein Mac und das iOS SDK benötigt. Die Apps für iPhone werden in der Programmiersprache Objective-C geschrieben. Um die Entwicklung von Apps zu erlernen, gibt es viele Ressourcen. Mir hat die Video Kollektion des Onlinekurses von Paul Hegarty von der Stanforduniversität [17] einen schnellen Einstieg in die iPhone-App-Entwicklung ermöglicht.

Um die App zu entwickeln, wurde XCode mit iOS SDK als Entwicklungsumgebung verwendet.

Für die Quellcodeverwaltung wurde Subversion (SVN) [9] verwendet.

Für das Testen wurden Karten von OpenStreetMap (OSM) [6] und Karten von NAVTEQ [5] verwendet.

Sqlite wird als persistenter Speicher der internen Kartendaten benutzt. Ein SQL Wrapper namens FMDB [19] wurde benutzt um den Umgang mit sqlite zu vereinfachen.

Für diese Arbeit wurde eine veränderte Latex Dokumentenvorlage der Arbeitsgruppe Softwareergonomie und Informationretrieval verwendet.

Für den Import von Höhendaten wurde die Google Elevation API [1] verwendet.

Kapitel 2

Grundlagen

Zum besseren Verständnis der Arbeit sind einige Grundlagen notwendig. Zuerst werden die Koordinaten und das Koordinatensystem erklärt. Dann folgt eine kurze Definition von Graphen, die verwendet werden.

2.1 Koordinatensystem

Das Koordinatensystem, mit dem Navigationssysteme arbeiten, ist ein Kugelkoordinatensystem. Dabei werden nur zwei Komponenten benötigt, um einen Punkt auf einer Kugel oder einem Ellipsoid zu beschreiben. Die geografische Länge (Longitude) gibt in Winkelmaß die Abweichung vom sog. Nullmeridian an. Die geografische Breite (Latitude) gibt die Abweichung vom Äquator an. Die Angabe kann in unterschiedlichen Formaten sein. Die menschenlesbare Form ist die Angabe mit Grad, Minuten und evtl. Sekunden in Form `50° 21´ 46.15´´ N, 7° 33´ 31.71´´ O`. Hier ist eindeutig zu sehen, welche Angabe die geografische Länge ist, diese ist mit O für Ost oder W für West versehen. Die geografische Breite ist mit N für Nord oder S für Süd markiert. Bei der maschinenlesbaren Form wird die geografische Länge (Longitude) mit einer Dezimalzahl zwischen -180 und 180 und die geografische Breite (Latitude) mit einer Dezimalzahl zwischen -90 und 90 angegeben. So ist aus der Angabe `50.36282, 7.558808` nicht sofort erkennbar ob es lon,lat oder lat,lon Angabe ist.

Es gibt viele Referenzsysteme und Projektionsverfahren. Als Referenzsystem hat sich WGS84 etabliert. Auf die Navigation wirkt sich das nur über die Distanzfunktion, die verwendet wird, aus.

Chris Veness hat auf einer Webseite [23] eine Übersicht über Distanzfunktionen für Geo-Koordinaten zusammengestellt. Aus dieser Übersicht wurde die Haversine-Formel für die Distanzberechnung in Objective-C übertragen und verwendet. Die Distanzfunktion wurde in SQLite verfügbar gemacht, um sie direkt in SQL Abfragen nutzen zu können. Im Listing 2.1 ist die Implementierung der Distanzfunktion zu sehen.

Listing 2.1: Haversine Distanzfunktion

```
CGFloat DegreesToRadians(CGFloat degrees){
    return degrees * M_PI / 180;
};
CGFloat RadiansToDegrees(CGFloat radians){
    return radians * 180 / M_PI;
};
//earth radius in meters
#define EARTH_R 6371000.785
+ (double) distanceHaversineLon1:(double)lon1 lat1:(double)
    lat1 lon2:(double)lon2 lat2:(double) lat2{
    double dLat = DegreesToRadians(lat2-lat1);
    double dLon = DegreesToRadians(lon2-lon1);
    double a = sin(dLat/2) * sin(dLat/2) +
    cos(DegreesToRadians(lat1)) * cos(DegreesToRadians(lat2))
        *
    sin(dLon/2) * sin(dLon/2);
    double c = 2 * atan2(sqrt(a), sqrt(1-a));
    double d = EARTH_R * c;
    return d;
}
```

2.2 Graphen

Die Kartendaten für die Navigation werden als Graphen modelliert und gespeichert. Die Definition eines ungerichteten Graphen sei die folgende.

$$G = (V, E) \tag{2.1}$$

$$V = \mathbb{N}$$

$$E = \{\{v1, v2\} : v1, v2 \in V \ \wedge v1 \neq v2\}$$

Bei attributierten Graphen werden die Attributinformationen in Form von Funktionen ausgedrückt.

Die Definition wird um die Attribute erweitert. Es gibt boolesche Attribute und nummerische Attribute. Eine Attributfunktion ordnet jeder Kante einen Wert zu.

$$G = (V, E, A) \tag{2.2}$$

$$A = (ANB, AB, ANR, AR)$$

$$ANB = \{name : name \in \mathbb{S}\}$$

$$ANR = \{name : name \in \mathbb{S}\}$$

$$ANB \cap ANR = \varnothing$$

$$AB = (AN \to (E \to \mathbb{B}))$$
$$AR = (AN \to (E \to \mathbb{R}))$$

An dieser Stelle ist nur die Attributierung der Kanten definiert. Die Attributierung der Knoten geschieht analog. Die Attributwerte $AB(attributeBool1, edge1)$ und $AR(attributeReal1, edge1)$ werden für bessere Lesbarkeit abgekürzt durch `edge1.attributeBool1` und `edge1.attributeReal1`. Da die Namensmengen disjunkt sind, ist die Angabe, um welche Funktion es sich handelt, nicht notwendig.

2.3 SQLite

Ich habe mich für SQLite [7] als persistenten Speicher entschieden, weil ein schneller und effizienter Zugriff durch Indizes ermöglicht wird, was ein einfaches Dateiformat wie XML nicht bieten kann. Bei der Arbeit mit XML müssten die gesamten Daten in den Arbeitsspeicher geladen werden, um auf ihnen zu arbeiten oder zu suchen. Deshalb werden mithilfe eines Relationales Datenbank Management System (RDBMS) als Hintergrundspeicher nur die benötigten Daten geladen. In der Mobiltelefonentwicklung hat sich SQLite als RDBMS durchgesetzt, weil das System wenig Arbeitsspeicher verbraucht, die Datenbank als eine einzelne Datei verwaltet und keine Serverinstallation benötigt. Allgemein gilt SQLite als das Referenz-RDBMS für Systeme mit kleinem Arbeitsspeicher.

Eine wichtige Eigenschaft, die zu der Entscheidung beigetragen hat, ist, dass in SQLite eine räumliche Indexstruktur vorhanden ist, das RTree-Modul [8]. Es ermöglicht eine effiziente Suche auf mehrdimensionalen Daten wie den Kartendaten. Man kann Daten aus einem Bereich laden, der sich um den aktuellen Standort befindet. Somit müssen nicht alle vorhandenen Daten geladen werden und man spart Arbeitsspeicher und Ladezeit beim Start. Falls man an der Grenze des geladenen Bereichs Berechnungen durchführen muss, wird der Bereich vergrößert und die Daten nachgeladen.

Kapitel 3

Problemstellung und Analyse

Im folgenden Kapitel wird die Problemstellung beschrieben. Welche Eigenschaften gewünscht sin, welches Kartenmaterial zu Verfügung steht und welche Informationen vom Benutzer benötigt werden.

3.1 Problembeschreibung

Diese Arbeit beschäftigt sich mit der Navigationskomponente der in der Vision (Kapitel 1.1) beschriebenen Anwendung und soll eine Basis für eine spätere Weiterentwicklung bereitstellen.

Die zu entwickelnde Anwendung wird für ein iPhone entwickelt und wird deshalb im Folgenden als App bezeichnet.

Das Hauptziel der Arbeit ist eine Navigation nach einem nicht strikten Benutzerprofil zu ermöglichen. Navigation heißt in diesem Fall eine Planung der Route und eine Neuberechnung der Route wenn nötig. Die berechnete Route soll auf einer Karte angezeigt werden. Nicht striktes Benutzerprofil meint, dass die Einstellungen nicht nur eine binäre Einstellung mit „aus“ oder „ein“ erlauben, sondern auch einen beliebigen stufenlosen Zwischenwert, was der Präferenz des Benutzers entspricht. Wo immer möglich, soll eine Einstellung aus dem geschlossenen Intervall $[0, 1]$ der Einstellung aus der Menge $\{0, 1\}$ vorgezogen werden, wobei der Wert 0 das „aus“ und der Wert 1 das „ein“ repräsentiert. So soll ermöglicht werden, dass die Navigation z.B. nicht nur zwischen Routen mit Treppen und Routen ohne Treppen unterscheidet, sondern auch Routen mit „wenig“ Treppen ermöglicht. Die Einstellung soll stufenlos sein, um eine feinere Einstellung zu ermöglichen.

Da der Galileo-GPS-Empfänger sich noch in der Entwicklung befindet, kann die App nicht dessen Funktionalität benutzen. Weil der Empfänger sich voraussichtlich nur in der Genauigkeit unterscheiden wird, kann während der Entwicklung die vorhandene Schnittstelle des im iPhone eingebauten GPS-Empfängers

benutzt werden. Diese Schnittstelle soll, bei Verbindung mit dem neuen Empfänger, umgeschaltet werden können, um die Genauigkeit zu erhöhen.

Im NAPA Projekt werden neue, feinere Karten mit fußgängerspezifischen Attributen entwickelt, diese Karten sollen für die Navigation verwendet werden können. Da sich diese Karten noch in der Entwicklung befinden, muss eine Möglichkeit gefunden werden, um mit anderen Kartendaten das Konzept zu prüfen.

Die Navigation soll ohne eine bestehende Internetverbindung möglich sein.

3.2 Anwendungsfälle

Um die Vorstellung zu konkretisieren, werden einige Anwendungsfälle beschrieben.

Der einfachste Ablauf ist folgender.

Route berechnen

Der Benutzer startet die App. Die App zeigt eine Karte der Umgebung und die aktuelle Position des Benutzers an. Der Benutzer macht einen langen Tap auf die Karte, worauf eine Markierung erscheint. Der Benutzer tippt auf den Knopf in der Markierung. Die App berechnet mehrere Routen und zeigt diese auf der Karte an.

Weite Route berechnen

Der Benutzer startet die App. Der Benutzer verschiebt die Karte. Um die Karte weit zu verschieben, benutzt der Benutzer die Zoom-Funktion, mit deren Hilfe er die Karte zuerst verkleinert und dann im Zielgebiet vergrößert. Weiterer Ablauf wie in „Route berechnen“.

Route auswählen

Wenn die Routen berechnet wurden, wird eine Auswahl angezeigt. Der Benutzer muss eine Route auswählen. Nach der Wahl der Route wechselt die App in den Routenverfolgungsmodus. Bleibt der Benutzer auf der angezeigten Route, geschieht nichts. Weicht er von der Route ab, wird diese neu berechnet.

Profil ansehen

Der Benutzer tippt auf den Knopf "Profil". Damit gelangt der Benutzer zu den Profil-Einstellungen und das aktuelle Profil wird angezeigt. Hier werden der Profilname, die Routenfarbe und die Präferenzen des Profils angezeigt.

Profildaten ändern

Der Benutzer stellt seine Präferenzen ein, indem er die Einstellungen des verwendeten Profils ändert. Über die Schieberegler kann er die Eigenschaften der berechneten Route einschalten, ausschalten oder so einstellen, dass die Routenberechnung diese Eigenschaften weniger zulässt. Z.B. ergibt die Berechnung einer Route, mit der Einstellung „ein“ (1) für Treppen, eine Route, die Treppen nicht von einfachen Straßen unterscheidet und damit auch möglicherweise mit vielen Treppenpassagen. Nach der Änderung der Einstellung auf „aus“ (0) ergibt die Berechnung eine Route ohne Treppen, die aber länger sein kann als die Route, die Treppen erlaubt. Nach der Einstellung des Wertes auf 0,8, würden Treppenpassagen durch geringfügig längere Strecken ohne Treppen ersetzt, wenn möglich. Treppenpassagen, die eine große Umgehung benötigen, würden nicht ersetzt.

Mehrere Profile

Der Benutzer kann mehrere Profile anlegen und sie benennen. So kann der Benutzer Profile für verschiedene Situationen erstellen. Beim Berechnen der Route werden mithilfe mehrerer Profile verschiedene Routen berechnet und angezeigt. Bei der Auswahl der Route wird deren Profil für die Neuberechnung gemerkt.

3.3 Kartendaten

In diesem Kapitel werden die Kartendaten beschrieben. Zuerst werden die traditionellen Kartendaten beschrieben. Ich nenne Sie traditionell, weil das Modell der Straßenmittellinie in allen aktuellen Navigationssystemen verwendet wird. Danach werden die neuen Kartendaten vorgestellt, die ein anderes Graph-Modell haben.

3.3.1 Traditionelle Kartendaten

Die Kartendaten werden als attributierte Graphen modelliert. Diese Graphen bestehen aus Knoten und Kanten, die mit Attributen versehen sind. Die Kanten repräsentieren die Straßen und Wege, auf denen man sich bewegen kann. Die meisten aktuellen Kartendaten sind für eine Visualisierung und Navigation mit dem Auto gemacht worden. In einer visualisierten Karte wird eine Straße als *eine* Linie dargestellt. In den Daten wird diese Linie als eine Verbindung mehrerer Knoten gespeichert. Die Knoten haben Attribute für die Koordinaten. Mit diesen Informationen kann eine Straße visualisiert werden. Eine Beispieldarstellung der Karte ist in der Abbildung 3.1 links zu sehen. Rechts in der Abbildung ist dieser Kartenausschnitt in dem Karteneditor JOSM [2] etwas aufgehellt zu sehen.

Knoten, die mit mehr als 2 anderen Knoten verbunden sind, stellen Kreuzungen dar. Knoten und Kanten enthalten zusätzliche Informationen in den mit ih-

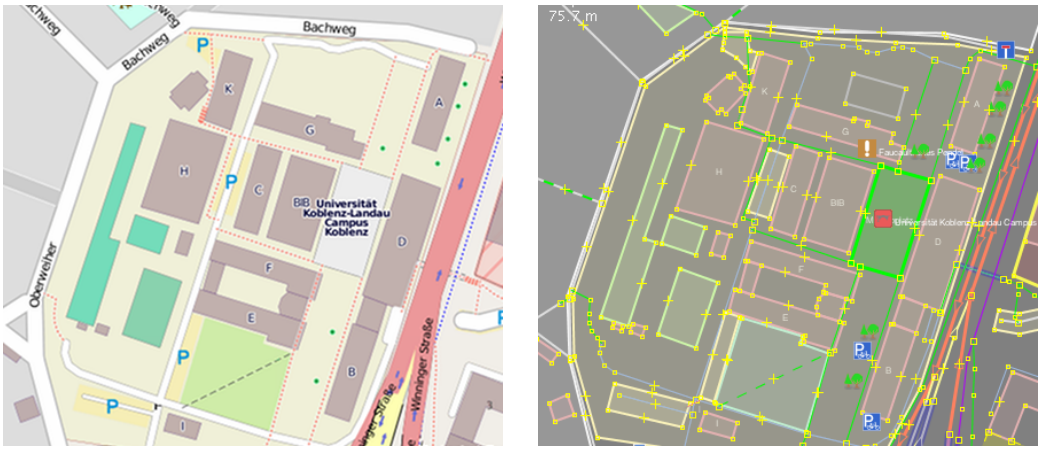


Abbildung 3.1: OpenStreetMap Karte, links normal, rechts im Editor

nen verknüpften Attributen. Die Attribute haben einen Namen und einen Wert. Ein Attribut für einen Knoten wäre z.B. `busstop=true`. Dieses Attribut sagt, dass sich an diesem Punkt eine Bushaltestelle befindet. Ein Beispielattribut für Kanten wäre z.B. `maxspeed=50`, das aussagt, dass die Höchstgeschwindigkeit auf dieser Strecke 50 km/h beträgt.

Gespeichert werden die Daten in relationalen Datenbanken, um diese schnell ändern zu können. Ausgeliefert werden Sie in verschiedenen Formaten. Alle Formate sind Hersteller spezifisch, weil alle Hersteller verschiedene Attribute oder Attributnamen verwenden. Die verbreiteten Arten sind XML-Dateien, Shapefiles und Datenbankexport-Dateien. Die Schnittstelle von OpenStreetMap bietet eine Möglichkeit, Kartenausschnitte als XML-Dateien zu exportieren.

Listing 3.1: OSM-Export

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="50.3000000" minlon="7.5000000" maxlat="
    50.4000000" maxlon="7.6000000"/>
 <node id="30003318" lat="50.3636859" lon="7.5592562" user="
    proedler" uid="383178" visible="true" version="7"
    changeset="6649636" timestamp="2010-12-13T16:55:15Z">
  <tag k="addr:city" v="Koblenz"/>
  <tag k="addr:country" v="DE"/>
  <tag k="addr:housenumber" v="1"/>
  <tag k="addr:postcode" v="56070"/>
  <tag k="addr:street" v="Universitätsstraße"/>
  <tag k="amenity" v="university"/>
  <tag k="name" v="Universität Koblenz-Landau Campus Koblenz"/
     >
  <tag k="note" v="the postal code 56070 is correct for this
```

```
        amenity, tough the postal code area is different"/>
     <tag k="website" v="http://www.uni-koblenz-landau.de/koblenz
        /"/>
14    <tag k="wheelchair" v="limited"/>
    </node>
16  ...
    <way id="4756720" user="egore911" uid="8764" visible="true"
       version="4" changeset="7336873" timestamp="2011-02-19
       T21:44:36Z">
18    <nd ref="30346910"/>
      <nd ref="30346925"/>
20    <nd ref="30346926"/>
      <tag k="highway" v="unclassified"/>
22    <tag k="name" v="Universitätsstraße"/>
    </way>
24  ...
    <relation id="119129" user="Radeln" uid="179033" visible="
       true" version="40" changeset="8090554" timestamp="
       2011-05-09T08:17:03Z">
26    <member type="way" ref="33616446" role=""/>
      <member type="way" ref="40269095" role=""/>
28    <member type="node" ref="485636169" role=""/>
  ...
30    <member type="way" ref="33616441" role=""/>
      <tag k="name" v="Buslinie 8"/>
32    <tag k="network" v="VRM"/>
      <tag k="operator" v="KEVAG"/>
34    <tag k="ref" v="8"/>
      <tag k="route" v="bus"/>
36    <tag k="type" v="route"/>
    </relation>
38  ...
  </osm>
```

Im Listing 3.1 sieht man wie eine Exportdatei aussieht, die von der App verwendet werden kann. In der Exportdatei erkennt man Knoten in Form von `node` XML-Elementen. Diese haben Informationen in Form von XML-Attributen, wie z.B. Latitude (lat) und Longitude (lon), oder in weiteren `tag`-Elementen. Das Element `way` ist eine Liste von Knoten. Auf diese Weise werden mehrere gerichtete Kanten beschrieben und mit Attributen versehen. In `relation` Elementen werden mehrere Knoten und Wege zusammengefasst, die Gemeinsamkeiten haben und werden mit Attributen versehen.

3.3.2 Kartendaten der nächsten Generation

NAVTEQ bietet bereits Karten für die Fußgängernavigation an. Die unter dem Namen „Discover Cities“ angebotenen Zusatzinformationen enthalten einige zu-

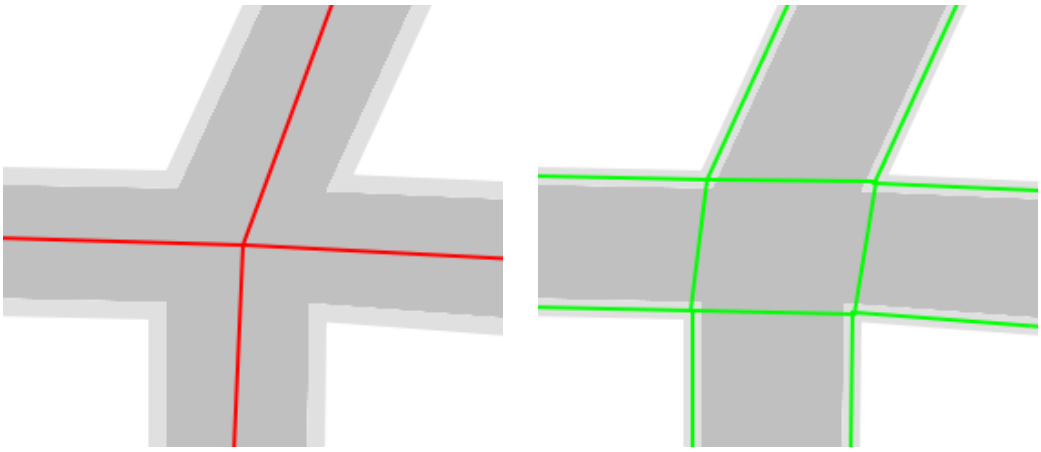


Abbildung 3.2: Gespeicherte Linien - links Straßenmitte, rechts Bürgersteigmitte

sätzliche Wege, die nicht mit der einfachen Karte ausgeliefert werden, weil diese nur für Fußgänger zugänglich sind. Die Daten enthalten Attribute, die speziell für Fußgängernavigation geeignet sind, wie z.B. Treppen, Straßenbeleuchtung, Höhenangaben und Bürgersteige.

Im NAPA Projekt werden die Kartendaten weiter verfeinert. Anstatt die Mittellinie der Straße zu erfassen, werden die Wege der Fußgänger erfasst, also die Bürgersteige und Fußwege. In der Abbildung 3.2 wird der Unterschied zur bisherigen Speicherung deutlich. Links in der Abbildung ist die bisher übliche Speicherart der Linien, diese ist sparsam, weil sie mit nur einer Linie eine Straße beschreibt. Die Kreuzung wird mit nur einem Knoten dargestellt. Rechts in der Abbildung ist die neue Art die Wege für Fußgänger zu modellieren. Hierbei werden Linien mitten auf dem Bürgersteig modelliert. Diese Art erfordert mehr Speicher, ist aber präziser. Für eine Straße werden 2 Linien benötigt. Um diesen Graphen navigationsfähig zu machen, werden zusätzliche Verbindungen an *geeigneten* Stellen benötigt. So wird eine Kreuzung von 2 Wegen, durch 4 Knoten und den zwischen ihnen befindlichen Fußgängerüberwegen modelliert. Die Fußgängerüberwege enthalten weitere Informationen über deren Art, ob Ampel, Zebrastreifen oder einfacher Fußgängerüberweg. Wenn die Stelle nicht geeignet passierbar ist, wie z.B. an einer Schnellstraße, wird diese Verbindung nicht im Graphen gespeichert. Zu den Mittellinien werden auch Umrisse der Wege für die verbesserte Darstellung gespeichert. Diese Umrisse sind für die Navigation irrelevant und deswegen werden sie momentan nicht in die App importiert. Abkürzungen, die keine Wege sind, werden auch gespeichert, in Form von virtuellen Verbindungen (*VirtualLink*). Virtuelle Verbindungen sind beispielsweise eine Abkürzung durch einen Parkplatz oder Stadtplatz.

Listing 3.2: NAPA-Kartendaten

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ped:DiscoverCitiesPedestrian
    xsi:schemaLocation="http://www.navteq.com/ped
        NAPA_Pedestrian_Schema_Version_15.xsd"
    xmlns:ped="http://www.navteq.com/ped"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    areaName="Wurzburg"
    pedVersionNo="1234"
    creationTime="2008-10-29T04:05:07.526-05:00"
    mapVersionNo="6789">
<gml:featureMembers>
    <!-- BOUNDARY CURVES ~~~~~~~~~~~~~~~~~~~~ -->
...
    <!-- ROUTABLE NODES ~~~~~~~~~~~~~~~~~~~~ -->
    <ped:RoutablePedNode gml:id="RN152612307_155944005L
        -155077836R" zStackOrder="0">
        <gml:centerOf><gml:Point srsName="
            urn:ogc:def:crs:OGC:1.3:CRS84" srsDimension="3"><
            gml:pos>9.93005275458 49.7898290377 0.0</gml:pos><
            /gml:Point></gml:centerOf>
        <ped:geometricPedNode xlink:href="#
            GN152612307_155944005L-155077836R"/>
    </ped:RoutablePedNode>
...
    <!-- ROUTABLE LINKS ~~~~~~~~~~~~~~~~~~~~ -->
    <ped:RoutablePedLink gml:id="RL155239724L" linkPVID="
        53111863" stairLocation="MIDDLE" gradeDirection="
        TO_START" streetLight="false" slopeCategory="FLAT"
        pavementType="ASPHALT" trafficDensityCategory="LIGHT">
        <ped:startPedNode xlink:href="#RN152368340_dummy
            -155239724L"/>
        <ped:endPedNode xlink:href="#RN152527934_155239724L
            -155344006L"/>
        <ped:geometricPedLink xlink:href="#GL155239724L"/>
        <ped:shapePoint width="0.75" slope="0"><gml:Point
            srsName="urn:ogc:def:crs:OGC:1.3:CRS84"
            srsDimension="3"><gml:pos>9.93507512164
            49.7928061159 0.0</gml:pos></gml:Point></
            ped:shapePoint>
        <ped:shapePoint width="0.75" slope="0"><gml:Point
            srsName="urn:ogc:def:crs:OGC:1.3:CRS84"
            srsDimension="3"><gml:pos>9.93526345223
            49.7930927574 0.0</gml:pos></gml:Point></
            ped:shapePoint>
    </ped:RoutablePedLink>
...
    <!-- GEOMETRIC NODES ~~~~~~~~~~~~~~~~~~~~ -->
```

```
31 ...
      <!-- GEOMETRIC LINKS ~~~~~~~~~~~~~~~~~~~~ -->
33 ...
      <!-- CROSSWALKS ~~~~~~~~~~~~~~~~~~~~ -->
35    <ped:RoutableCrosswalkLink gml:id="RC152771159_155531648R
         -155398584L" crosswalkType="ZEBRA" curbRamp="true"
         startSideSignage="false" endSideSignage="false"
         linkPVID="0" stairLocation="MIDDLE" gradeDirection="
         TO_START" streetLight="false" slopeCategory="FLAT"
         pavementType="ASPHALT" trafficDensityCategory="LIGHT">
          <ped:startPedNode xlink:href="#RN152771159_155531648R
             -155398584R"/>
37        <ped:endPedNode xlink:href="#RN152771159_155398584L
             -154862875R"/>
          <ped:geometricPedLink xlink:href="#
             GC152771159_155531648R-155398584L"/>
39        <ped:shapePoint width="0.75" slope="0"><gml:Point
             srsName="urn:ogc:def:crs:OGC:1.3:CRS84"
             srsDimension="3"><gml:pos>9.90112421685
             49.7954999552 0.0</gml:pos></gml:Point></
             ped:shapePoint>
          <ped:shapePoint width="0.75" slope="0"><gml:Point
             srsName="urn:ogc:def:crs:OGC:1.3:CRS84"
             srsDimension="3"><gml:pos>9.90112570577
             49.7954347552 0.0</gml:pos></gml:Point></
             ped:shapePoint>
41    </ped:RoutableCrosswalkLink>
  ...
43    <!-- MID-BLOCK CROSSINGS ~~~~~~~~~~~~~~~~~~~~ -->
      <ped:RoutableMidBlockCrossing>
45        <ped:routablePedLink  xlink:href="#RL155714354R" />
          <ped:routablePedLink  xlink:href="#RL155714354L" />
47    </ped:RoutableMidBlockCrossing>
  ...
49    <!-- VIRTUAL CONNECTIONS ~~~~~~~~~~~~~~~~~~~~ -->
      <ped:RoutableVirtualConnection gml:id="VC1" connectionType
         ="LEVEL" connectionAccessRestriction="NOT_APPLICABLE"
         connectionLocation="PLAZA" stairsTraversal="false"
         directionOfTravel="TO_START" grade="TO_START"
         streetLight="false">
51        <ped:connectionPoint percentFromStart="92" >
              <ped:routablePedLinkPoint xlink:href="#
                 RL156018103R" />
53        </ped:connectionPoint>
          <ped:connectionPoint percentFromStart="20" >
55            <ped:routablePedLinkPoint xlink:href="#
                 RL155475376L" />
          </ped:connectionPoint>
57    </ped:RoutableVirtualConnection>
```

```
  ...
59 </gml:featureMembers>
  </ped:DiscoverCitiesPedestrian>
```

Im Listing 3.2 ist eine vorläufige Version der neuen Kartendaten des NAPA Projektes zu sehen, die von der App verwendet werden kann. Eine vollständige Spezifikation des Formats ist in [21] zu finden. Die Datei ist in Abschnitte unterteilt. Im ersten Abschnitt `BOUNDARY CURVES` sind die Umrisse der Wege beschrieben. Diese werden nicht benutzt, da sie für das Routing irrelevant sind. Ebenfalls irrelevant für die Navigation sind die Abschnitte `GEOMETRIC NODES` und `GEOMETRIC LINKS`. Alle routingfähigen Knoten sind im Abschnitt `ROUTABLE NODES` aufgelistet. Die Knoten enthalten lediglich die Koordinaten in einem `gml:pos` Element. Die im Abschnitt `ROUTABLE LINKS` aufgelisteten Links enthalten eine Referenz auf den Startknoten und auf den Endknoten. Die beschreibenden Attribute sind in XML-Attributen verpackt. Mit den Elementen `ped:shapePoint` werden die verfeinernden Punkte beschrieben. Die Fußgängerüberwege aller Art werden im Abschnitt `CROSSWALKS` beschrieben. Durch Attribute kann man z.B. erkennen, ob es sich um einen Zebrastreifen oder um eine Ampel handelt. Im Abschnitt `MID-BLOCK CROSSINGS` werden Bürgersteige beschrieben, zwischen denen der Wechsel problemlos ist. Die virtuellen Verbindungen werden im Abschnitt `VIRTUAL CONNECTIONS` zusammengefasst. Eine virtuelle Verbindung wird nicht mit den Punkten beschrieben, sondern in Prozentangabe der anderen Verbindungen, was den Import dieser Daten erschwert.

3.3.3 Gemeinsamkeiten

Beide Ansätze haben Gemeinsamkeiten. Sie modellieren die Mitte des Pfades, auf dem navigiert werden kann. Beide Ansätze stellen einen Graphen dar auf dem eine Pfadsuche ausgeführt werden kann. Die Knoten und Kanten beider Ansätze haben Zusatzinformationen in Form von Attributen oder auch manchmal Tags genannt. Es wird immer versucht, Speicherplatz zu sparen. In einigen Formaten werden nicht einzelne Kanten gespeichert, sondern direkt ganze Linien, die mehrere Knotenpunkte enthalten, diese werden Links genannt. Ein Link ist also eine Verbindung zwischen zwei Knoten, die mehrere weitere Knoten enthalten kann, die zur verfeinerten Ortung oder Darstellung benutzt werden, aber für die Pfadfindung meist irrelevant sind. Um die Attributinformationen sparsam zu speichern werden Kanten zu Wegen zusammengefasst. Bei dieser Arbeit wurden Karten von OpenStreetMap und die neuen Karten von NAVTEQ verwendet.

Im Kapitel 6.2 ist das Datenmodel der Kartendaten erläutert.

3.4 Profil

Alle benutzerspezifischen Informationen werden in einem Profil gespeichert. Es sollen mehrere Profile möglich sein, um dem Benutzer einen schnellen Wechsel der Einstellungen zu ermöglichen, ohne dabei das Profil ändern zu müssen. Um die Profile zu unterscheiden, wird ein Name vom Benutzer vergeben. Ein Profil enthält die Informationen über die Navigationseinschränkungen der verwendeten Attribute.

Werte die eine feine Einstellung erlauben, werden als Zahl aus dem Intervall $[0, 1]$ gespeichert. Diese stufenlose Einstellung soll eine bessere Einstellung ermöglichen. Hier entspricht der Wert 0 einem Wert „nein“ und der Wert 1 entspricht dem Wert „ja“. Es sind alle Zwischenwerte erlaubt, wie z.B. 0,134756, was das Vorkommen der Eigenschaft stark eingrenzt, aber nicht komplett verbietet. Diese Einstellung soll keine Bevorzugung ausdrücken, das heißt der Wert 1 erlaubt die normale Verwendung der Eigenschaft und es gibt keine höheren Werte als 1. Warum diese Einschränkung nützlich ist, ist im Kapitel 6.3.2 erklärt.

Ein Profil sollte zu allen möglichen Attributen der Karte eine Einstellung speichern. Wenn ein Attribut einen booleschen Wert annehmen kann, wird dazu ein Wert der Verwendungserlaubnis $[0, 1]$ gespeichert. Ist ein Attribut ein Zahlenwert, werden dazu ein Schwellenwert und ein Grenzwert desselben Bereichs wie das Attribut gespeichert.

Ein Profil speichert folgende Informationen:

- Profilname
  `name: string`
- Durchschnittstempo (in km/h)
  `speed: float`
- Letzter Suchbegriff.
  `lastSearchterm: string`
- Verwendung von Treppen $[0, 1]$
  `useStairs: float`
- Verwendung von Brücken $[0, 1]$
  `useBridges: float`
- Weitere Einstellungen für boolesche Attribute
  `use`*`Attribute`*`: float`
- Steigung: Schwellenwert und Grenzwert (in % -Steigung)
  `useSlopeOK: double`
  `useSlopeMax: double`
- Erlaubte Maximalgeschwindigkeit für Autos: Schwellenwert und Grenzwert (in km/h). Wird als Indikator für die Verkehrsintensität benutzt.

`useMaxSpeedOK: double`
`useMaxSpeedMax: double`

- Weitere Einstellungen für Attribute mit Mehrfachwerten
  `use`*`Attribute`*`OK: double`
  `use`*`Attribute`*`Max: double`

Das Profil beeinflusst die Kosten der berechneten Route. Die Berechnung der Routenkosten ist im Kapitel 6.3.1 beschrieben.

Kapitel 4

Anforderungen

An dieser Stelle sollen die Eigenschaften und Funktionen beschrieben werden, die von der App erwartet werden.

Legende:

- MUSS - Anforderung muss erfüllt werden.
- SOLL - Anforderung muss so gut wie möglich erfüllt werden.
- KANN - Anforderung ist optional.

4.1 Routing

1.1. Die App muss eine Route für Fußgänger berechnen.

1.2. Die App muss die berechnete Route auf einer Karte anzeigen.

1.3. Die App soll eine Alternativroute berechnen.

1.4. Die App soll mehrere Routen auf der Karte anzeigen.

1.5. Die App soll ohne Internetverbindung nutzbar sein.

1.6. Die Berechnung einer Route muss durch ein Profil manipuliert werden können.

1.7. Aus der Anzeige der Route muss ersichtlich sein, welches Profil verwendet wurde.

1.8. Das Routing Verfahren muss das neue Kartenmaterial von NAVTEQ verwenden können.

4.2 Profil

2.1. Es muss möglich sein, ein Profil zu erstellen.

2.2. Es muss möglich sein, die Profile zu benennen.

2.3. Es muss möglich sein, ein Profil über eine grafische Schnittstelle das Profil zu bearbeiten.

2.4. Es muss möglich sein, ein Profil zu löschen.

2.5. Es soll möglich sein, die Profile zu sortieren.

4.3 Interaktion

3.1. Es muss möglich sein, ein Ziel direkt über die Anzeige der Karte auszuwählen.

3.2. Es soll möglich sein, ein Ziel über eine Adresssuche auszuwählen.

3.3. Es soll möglich sein, ein Ziel über eine Schnittstelle für andere Apps auszuwählen (AppLink).

4.4 Design

4.1. Die App muss für eine spätere Weiterentwicklung erweiterbar sein.

4.2. Die App muss mindestens ein Kartenformat importieren können.

4.3. Das Modul für die Ortung muss, für die spätere Verwendung mit dem Galileo-Empfänger, angepasst werden können.

4.5 Performanz

5.1. Das Berechnen einer kurzen (100m) Route soll weniger als 1 Sekunde dauern.

5.2. Das Starten der Applikation soll weniger als 1 Minute dauern.

4.6 Rahmenbedingungen

6.1. Das fertige Produkt soll als eine Datei zur Verfügung gestellt werden.

6.2. Das fertige Produkt soll als Download im AppStore zur Verfügung gestellt werden.

6.3. Das Produkt muss auf einem aktuellen iPhone lauffähig sein.

6.4. Das Produkt soll möglichst wenige Abhängigkeiten zu anderen Produkten haben.

6.5. Das Produkt muss in XCode entwickelt werden.

4.7 Dokumentation

7.1. Es muss eine englischsprachige Dokumentation geben.

7.2. Es muss ein englischsprachiges Benutzungshandbuch geben.

7.3. Es muss ein Demonstrationsbeispiel der Benutzung geben.

Kapitel 5

Routingalgorithmen

Im folgenden Kapitel werden noch einige Grundlagen erklärt. Danach werden die einzelnen Routing Algorithmen vorgestellt und verglichen um einen passenden Algorithmus auszuwählen.

Gesucht ist eine optimale Route nach bestimmten Parametern, dies ist ein kürzester Pfad Problem, das in der Literatur als Shortest Path Problem (SPP) bezeichnet wird. Für dieses Problem existieren eine Reihe Algorithmen, die es lösen. Grundlage für die meisten dieser Algorithmen ist eine Suche auf einem Graphen.

Bei meiner Recherche habe ich einige Algorithmen in Betracht gezogen, die dieses Problem lösen.

- 5.2 Dijkstra (5.5 Bidirektionaler Dijkstra)
- 5.3 A-Stern (5.6 Bidirektionaler A-Stern)
- 5.4 Bellman-Ford
- 5.7 Potentialfelder
- 5.7 Hill Climbing
- 5.8 Contraction Hierarchy (CH)
- 5.9 A-Star Landmark Triangle Inequality (ALT)

Die beiden Letzten (ALT und CH) sind als Beschleunigungsalgorithmen von Basisalgorithmen wie Dijkstra und A-Stern zu verstehen.

5.1 Weitere Grundlagen

Um die Pfadrepräsentation, die in den Algorithmen verwendet wird, zu verstehen, werden an dieser Stelle dessen Details erläutert.

Die Verfahren berechnen Pfade in einem Graphen. Ein Pfad ist eine Folge von Knoten. Anstatt alle Knoten vollständig in einem Pfadobjekt zu speichern, werden aufeinander aufbauende Pfade verwendet. Jeder Pfad enthält nur den Pfad, den er erweitert, und den zuletzt hinzugefügten Knoten.

Die Pfade werden in der Prioritätswarteschlange nach den Kosten f sortiert. f setzt sich aus den Kosten des Pfades und den geschätzten Kosten vom Pfad zum Ziel zusammen. Diese Kosten f werden im Pfad gespeichert. Die Berechnung von f ist die folgende:

$$f(path) = g(path) + h(path.lastNode) \tag{5.1}$$

Die Berechnung der Pfadkosten ist trivial. Es ist die Summe der Kosten der enthaltenen Kanten.

$$g(path) = \sum_{i=0}^{n-1} cost(path.node_i, path.node_{i+1}) \tag{5.2}$$

Da der Pfad aus einem Teilpfad und einer Kante zusammengesetzt ist, kann man diese Formel in folgende Formel umformen.

$$g(path) = g(path.extendedPath) + \tag{5.3}$$

$$+cost(path.extendedPath.lastNode, path.lastNode)$$

Auf diese Weise kann der bereits berechnete Wert g vom Teilpfad `extendedPath` verwendet werden, um die Berechnung zu beschleunigen. Dieser Wert g wird ebenfalls im Pfad gespeichert.

Ein Beispiel ist in der Abbildung 5.1 zu sehen. In dieser Abbildung repräsentiert jedes `GraphPath` Objekt einen Pfad im Graphen. Der erste Pfad besteht aus dem Startknoten, dessen Werte g und f gleich 0 sind. Die Berechnung des Wertes f und damit auch h ist nicht erforderlich, da der Startknoten der einzige Knoten in der Prioritätswarteschlange ist und deshalb keinen Prioritätswert braucht.

Der Wert f ist die Summe aus g und h. Deshalb ist bei Verfahren ohne eine Heuristikfunktion der Wert f und g identisch. Bei der Ausführung des Dijkstra Algorithmus wird deshalb nur ein Wert benötigt.

5.2 Dijkstra Algorithmus

Der Algorithmus von Dijkstra [11] führt eine Suche auf einem gerichteten Graphen mit gewichteten Kanten aus. Die Kosten der Kanten dürfen dabei nicht negativ sein. Der Algorithmus findet einen optimalen Pfad von einem Knoten zu einem anderen Knoten des Graphen, wenn es einen solchen Pfad gibt. Der Algorithmus arbeitet wie eine Breitensuche auf einem Graphen, mit dem Unterschied, dass er eine priorisierte Warteschlange benutzt. Eine Implementierung in

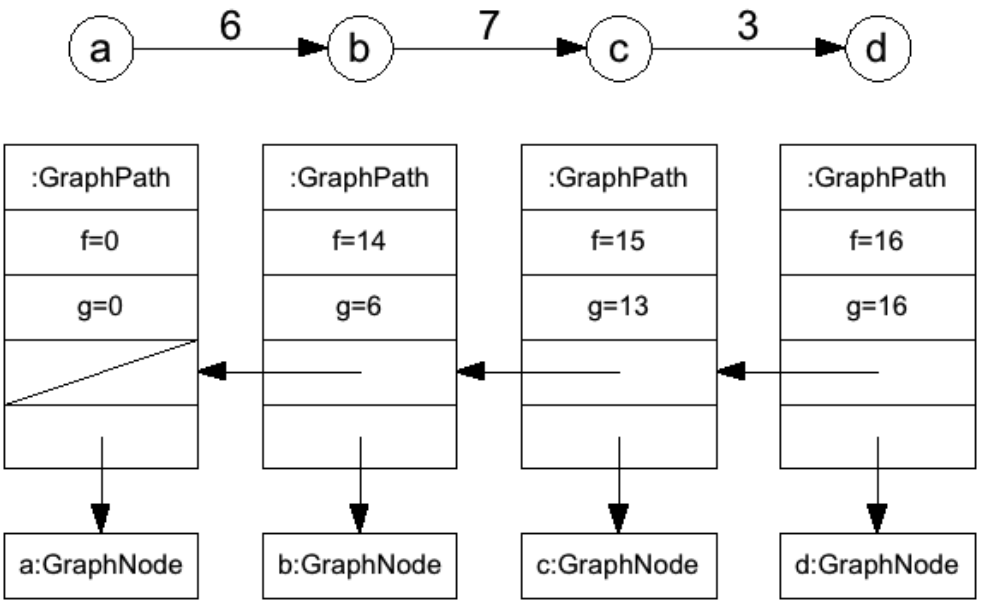


Abbildung 5.1: GraphPath

Objective-C ist im Listing 5.1 zu sehen.

Der Algorithmus arbeitet wie folgt. Alle Knoten bekommen eine Markierung $Node.label$ mit den minimalen Kosten vom Startpunkt aus. Für jeden Knoten wird der Vorgängerknoten $Node.pred$ gemerkt, von dem aus man diesen am schnellsten erreicht. Dann werden alle markierten Knoten untersucht/gescannt. Der Startknoten wird mit Kosten 0 markiert. Es wird ein Knoten mit den niedrigsten Kosten ausgewählt, der noch nicht gescannt wurde. Ist es der Zielknoten, dann ist der Pfad gefunden. Aus den gespeicherten Vorgängern kann der Pfad rekonstruiert werden.

Listing 5.1: Dijkstra Algorithmus

```
//erstelle Prioritätswarteschlange
BinaryHeap *queue = [BinaryHeap heap];
GraphPath *startPath = [GraphPath pathWithNode:startnode];
[queue add:startPath];
NSMutableSet *scanned = [NSMutableSet set];
while (!queue.isEmpty) {
    GraphPath *gp = [queue removeFirstObject];
    if(gp.lastNode == goalNode){
        return gp;
    }else if(![scanned containsObject:gp.lastNode]){
        //untersuche/scan node
        for(Edge *neighbourEdge in [gp.lastNode outgoingEdges
```

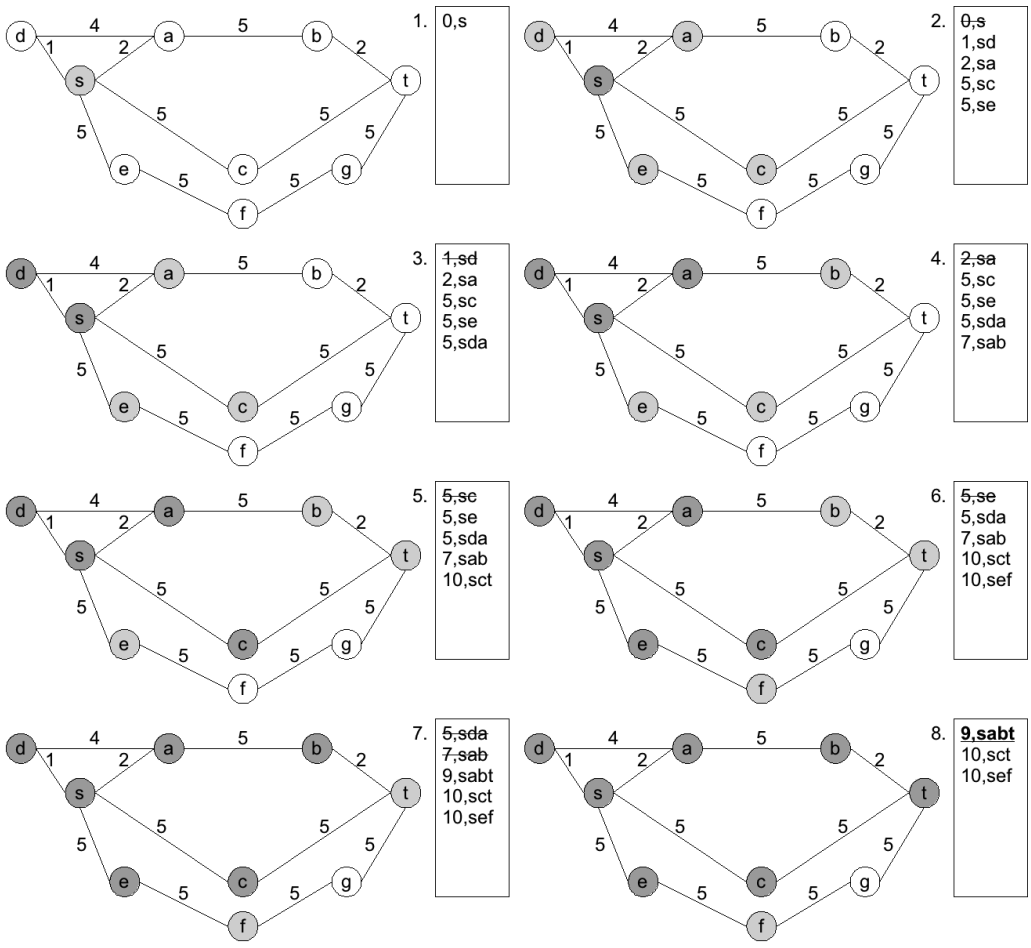


Abbildung 5.2: Dijkstra Algorithmus Ausführung

```
            ]){
             if(![scanned containsObject:[neighbourEdge omega
                 ]]){
                 //markiere/label node
                 GraphPath *neigPath =
                     [GraphPath pathWithPath:gp
                     node:[neighbourEdge omega]
                     weightG:(gp.weightG + [neighbourEdge cost
                         ])];
                 //wird mit Priorität neigPath.weightG
                     eingefügt
                 [queue add: neigPath];
             }
        }
        [scanned addObject:node];
    }
}
return nil; //keinen Pfad gefunden
```

In der Abbildung 5.2 ist eine Ausführung des Algorithmus zu sehen. Es wird ein kürzester Pfad von s zu t gesucht. Im Schritt 7 wurden 2 Schritte zusammengefasst, da der Knoten d keine Nachbarknoten hat, die in die Warteschlange aufgenommen werden.

5.3 A-Stern Algorithmus

Der A*-Algorithmus (A-Stern) arbeitet ähnlich wie der Dijkstra Algorithmus, mit dem Unterschied, dass die Auswahl der Knoten welche als Nächstes untersucht werden sollen, zielgerichtet geschieht. Bei A-Stern heißt das, dass die Knoten nicht nur nach der geringsten Entfernung vom Startpunkt bewertet werden, sondern auch nach der geschätzten minimalen Entfernung zum Zielknoten. Auf diese Weise werden bevorzugt Knoten näher an den Zielknoten untersucht. Dies führt dazu, dass weniger Knoten insgesamt untersucht werden müssen.

Bei diesem Verfahren ist die Funktion, die die restlichen Kosten zum Ziel schätzt, die Heuristikfunktion, entscheidend. Sie bestimmt, wie viel schneller die Suche verläuft. Setzt man diese Funktion auf 0, so arbeitet der A-Stern Algorithmus die gleichen Knoten wie Dijkstra ab.

Listing 5.2: A-Stern Algorithmus

```
//erstelle Prioritätswarteschlange
BinaryHeap *queue = [BinaryHeap heap];
GraphPath *startPath = [GraphPath pathWithNode:startnode];
[queue add:startPath];
NSMutableSet *scanned = [NSMutableSet set];
while (!queue.isEmpty) {
```

```
    GraphPath *gp = [queue removeFirstObject];
    if(gp.lastNode == goalNode){
        return gp;
    }else if(![scanned containsObject:gp.lastNode]){
        //untersuche/scan node
        for(Edge *neighbourEdge in [gp.lastNode outgoingEdges
            ]){
            if(![scanned containsObject:[neighbourEdge omega
                ]]){
                //markiere/label node
                GraphPath *neigPath =
                    [GraphPath pathWithPath:gp
                    node:neighbourEdge.omega
                    weightG:(gp.weightG + [neighbourEdge cost
                        ])
                    weightH:[neighbourEdge.omega heuristicTo:
                        goalNode]];
                //wird mit Priorität f=g+h eingefügt
                [queue add: neigPath];
            }
        }
        [scanned addObject:node];
    }
}
return nil; //keinen Pfad gefunden
```

Um die Mächtigkeit der Heuristikfunktion zu demonstrieren, kann man Folgendes machen. Nach der ersten Berechnung eines optimalen Pfades speichert man die Entfernung zum Zielpunkt in allen Knoten des Pfades zwischen. Man berechnet einen Pfad zum selben Zielknoten von einem der Punkte auf dem optimalen Pfad. Nun kann die Heuristikfunktion für diese Knoten die Entfernung exakt schätzen. Es wird immer ein Knoten aus der Prioritätswarteschlange entnommen, der auf dem optimalen Pfad liegt, da die Priorität, die Zusammensetzung der Entfernung von Startknoten und der geschätzten Entfernung zum Zielknoten, für diese Knoten immer exakt und *minimal* ist. Die Priorität der anderen Knoten, ob geschätzt oder exakt, ist schlechter, d.h. höhere Kosten. Bei einer erneuten Berechnung eines optimalen Pfades von einem Knoten auf demselben Pfad würde nun der A-Stern Algorithmus nur noch die Knoten auf dem optimalen Pfad abarbeiten. Die sogenannte Effizienz, (Knoten auf dem resultierenden Pfad / abgearbeitete Knoten) wie sie in [14] eingeführt wird, um Algorithmen zu vergleichen, liegt dann bei 100%.

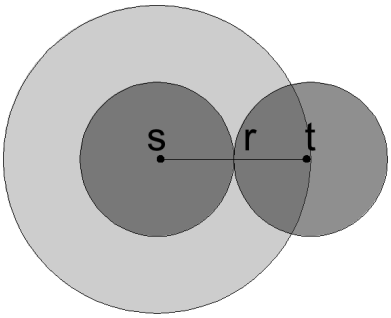


Abbildung 5.3: Suchraumhalbierung

5.4 Bellman-Ford-Algorithmus

Der Vorteil vom Bellman-Ford-Algorithmus ist, er erlaubt negative Kantengewichte im Graphen, die keine Kreispfade mit negativen Kosten erzeugen. Der Nachteil ist, dass der Algorithmus eine schlechtere Laufzeit als Dijkstra hat. Auf diesen Algorithmus greift man zurück, wenn man negative Kantengewichte im Graphen nicht vermeiden kann. Da in diesem Fall keine negativen Kantengewichte vorhanden sind, wird auf weitere Untersuchungen verzichtet.

5.5 Bidirektionaler Dijkstra Algorithmus

Wenn ein Suchalgorithmus aus beiden Richtungen, vom Startpunkt und vom Zielpunkt aus, ausgeführt wird, spricht man von einem *bidirektionalen* Algorithmus. Man erwartet eine Halbierung der Knotenmenge, die untersucht werden muss. Dies ist schematisch in Abbildung 5.3 abgebildet. In der folgenden Formel ist der Faktor $1/2$ auf die Flächenreduzierung bezogen. Bei einer gleichmäßigen Verteilung bedeutet das auch eine Halbierung der Knotenmenge.

$$\pi r^2 \cdot x = 2 \cdot \pi(\frac{r}{2})^2 \tag{5.4}$$

$$r^2 \cdot x = 2 \cdot \frac{r^2}{4}$$

$$x = \frac{1}{2}$$

Sei die Länge des kürzesten Pfades $cost(st) = r$. So werden in Dijkstra alle Pfade untersucht, die kürzer sind als r. Man kann diese Menge als ein Kreis um den Startknoten visualisieren. Im Idealfall wird ein Knoten v, der auf dem Pfad st liegt, von beiden Teilsuchen nach der Hälfte von r gefunden. Die Visualisierung des bidirektionalen Dijkstra Algorithmus wird fast immer mit zwei sich berührenden Kreisen dargestellt. Besser wäre eine Visualisierung mit zwei sich

geringfügig überschneidenden Kreisen, da der erste Knoten, der von beiden Suchen abgearbeitet wird, sich nicht immer auf dem optimalem Pfad befindet.

Im Listing 5.3 ist ein naiver bidirektionaler Dijkstra Algorithmus zu sehen. Dieser Algorithmus sucht immer von der Seite aus, auf der das Element aus der Prioritätswarteschlange den kleineren Wert g hat, also die höhere Priorität. Der Algorithmus bricht die Suche ab, wenn er einen Knoten findet, der von beiden Seiten abgearbeitet wurde. Jede Seite speichert zu einem abgearbeiteten Knoten den optimalen Pfad vom Startknoten der jeweiligen Seite. Beim Aufeinandertreffen werden diese Pfade nachgeschlagen, verbunden und als Ergebnis geliefert. Dieser Algorithmus ist *nicht korrekt*, da der gefundene Pfad nicht garantiert optimal ist.

Listing 5.3: Naiver bidirektionaler Dijkstra Algorithmus

```
1  //erstelle Prioritätswarteschlange für beide Richtungen
   BinaryHeap *queueA = [BinaryHeap heap];
3  BinaryHeap *queueB = [BinaryHeap heap];
   BinaryHeap *queue = nil;
5  GraphPath *startPath = [GraphPath pathWithNode:startnode];
   [queueA add:startPath];
7  GraphPath *goalPath = [GraphPath pathWithNode:goalnode];
   [queueB add:goalPath];
9  NSMutableDictionary *scannedNodesA = [NSMutableDictionary
       dictionary];
   NSMutableDictionary *scannedNodesB = [NSMutableDictionary
       dictionary];
11 NSMutableDictionary *scannedNodesThis = nil;
   NSMutableDictionary *scannedNodesOponent = nil;
13 GraphPath *gp = nil;
   while (!queueA.isEmpty && !queueB.isEmpty ) {
15     GraphPath *gpA = [queue firstObject];
       GraphPath *gpB = [queue firstObject];
17     //Richtungswechsel, suche aus der Richtung mit kleineren g
       if([gpA.weightG compare:gpB.weightG] ==
           NSOrderedDescending){
19         //suche aus B / Rückwärts
           queue = queueB;
21         scannedNodesThis = scannedNodesB;
           scannedNodesOponent = scannedNodesA;
23     }else{
           //suche aus A / Vorwärts
25         queue = queueA;
           scannedNodesThis = scannedNodesA;
27         scannedNodesOponent = scannedNodesB;
       }
29     gp=[queue removeFirstObject];
       if ([scannedNodesThis objectForKey:[curNode getId]] != nil
           ) {
```

```
31          //dieser Knoten wurde bereits abgearbeitet
            continue;
33      }
        //hat die Gegenseite diesen Knoten bereits abgearbeitet?
35      GraphPath *scannedNodePathOponent = [scannedNodesOponent
            objectForKey:[curNode getId]]
        if (scannedNodePathOponent != nil) {
37          //einen Pfad gefunden
            return [gp copyMergeWithReversePath:
                scannedNodePathOponent];
39      }
        //untersuche/scan node
41      for(Edge *neighbourEdge in [gp.lastNode outgoingEdges]){
            if(![scanned containsObject:[neighbourEdge omega]]){
43              //markiere/label node
                GraphPath *neigPath =
45                  [GraphPath pathWithPath:gp
                    node:[neighbourEdge omega]
47                  weightG:(gp.weightG + [neighbourEdge cost])];
                //wird mit Priorität neigPath.weightG eingefügt
49              [queue add: neigPath];
            }
51      }
        [scannedNodesThis setObject:gp forKey:[curNode getId]];
53  }
    return nil; //keinen Pfad gefunden
```

Man kann nicht die Suche abbrechen, wenn ein Knoten v von beiden Seiten abgearbeitet ist. Zwar ist der Pfad sv der optimale Pfad von s zu v und der Pfad vt ist der optimale Pfad von v zu t, aber es ist nicht garantiert, dass der Pfad svt der optimale Pfad von s zu t ist. Das lässt sich am besten an einem Beispiel erläutern. In der Abbildung 5.4 ist eine Ausführung des Algorithmus zu sehen. Im Schritt 7 wird ein Knoten von beiden Seiten abgearbeitet, aber er ist nicht auf dem optimalen Pfad. Der optimale Pfad wird erst im nächsten Schritt gefunden. Die Schwierigkeit bei den bidirektionalen Algorithmen besteht darin, eine gute Abbruchbedingung zu finden, die die Menge der zu untersuchenden Knoten reduziert, aber auch die Korrektheit des Algorithmus bewahrt. Bricht man zu früh ab — ist der Pfad möglicherweise nicht optimal. Bricht man zu spät ab — werden unnötige Pfade untersucht, nachdem man den optimalen Pfad bereits gefunden hat. Es muss eine gute Abbruchbedingung gefunden werden.

Es wird eine Variable μ eingeführt, die der Länge des bisher besten gefundenen Pfades entspricht. Am Anfang ist der Wert von μ gleich ∞. Sobald ein Knoten v von beiden Seiten abgearbeitet wird, ist ein Kandidat svt als optimaler Pfad gefunden. μ speichert nun die Gesamtkosten des gefundenen Pfades. Der Knoten $Node_\mu$, über den der beste μ Wert gefunden wurde, wird ebenfalls gemerkt. Wird ein Pfad kürzer als μ gefunden, so werden μ und $Node_\mu$ aktualisiert. Die Suche

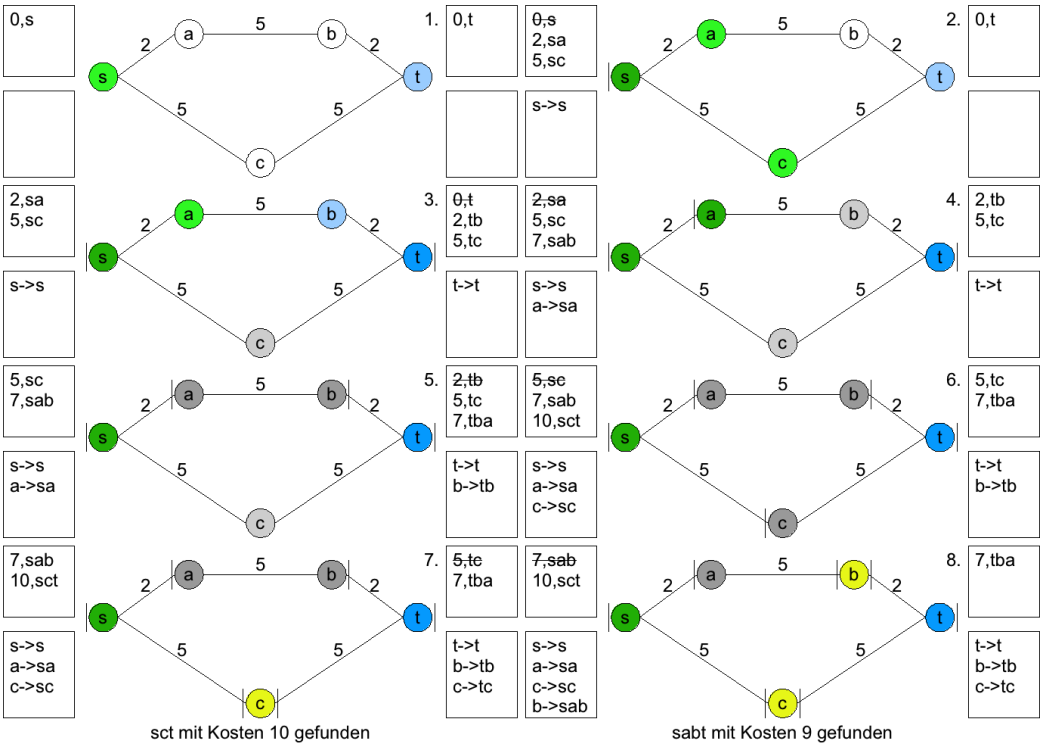


Abbildung 5.4: Bidirektionaler Dijkstra Algorithmus Ausführung

wird abgebrochen, wenn die Kosten eines Pfades aus der Prioritätswarteschlange den Wert μ übersteigen. Diese Kosten sind die minimalen geschätzten Kosten aller nachfolgend untersuchten Pfade und deshalb ist keine Verbesserung von μ möglich. Um den Suchraum weiter zu reduzieren, werden die Pfade, die nicht mehr optimal sein können, abgeschnitten. D.h., bei den abgeschnittenen Knoten werden die Nachbarknoten nicht in die Warteschlange eingefügt und damit auch nicht weiter untersucht.

Die Bedingung für das Abschneiden eines Pfades ist die folgende.

$$gp.extendedGraphPath.weightG + gpOpon.weightG \geq \mu \tag{5.5}$$

$$gp = min(queue) \wedge gpOpon = min(queueOpon)$$

Am Anfang gilt diese Bedingung nicht, weil $\mu = \infty$. Wenn $\mu < \infty$, dann wurde auch mindestens ein Knoten $Node_\mu$ gefunden, der in der Menge der abgearbeiteten Knoten vom Start $scannedA$ und in der Menge der abgearbeiteten Knoten von Ziel $scannedB$ vorhanden ist.

5.6 Bidirektionaler A-Stern Algorithmus

Der bidirektionale A-Stern Algorithmus führt, wie der bidirektionale Dijkstra Algorithmus, zwei Suchen aus, vom Start- und vom Zielknoten aus. Die Schwierigkeit bei allen bidirektionalen Suchalgorithmen ist: eine gute und korrekte Abbruchbedingung zu finden. Auch hier wird eine Variable μ verwendet, die den bisher gefundenen kürzesten Pfad speichert. Wird ein Knoten auf einer Seite untersucht, dessen geschätzte Kosten höher sind als μ, wird es übersprungen. Sind die Kosten beider Seiten über μ, wird die Suche mit dem aktuellen Ergebnis abgebrochen. Diese schwache Abbruchbedingung führt dazu, dass die bidirektionale A-Stern-Suche mehr Knoten als die einfache A-Stern-Suche bearbeitet. Man kann nicht die gleiche Abbruchbedingung wie bei bidirektionalem Dijkstra benutzen, weil für die Mengen der Knoten in den Prioritätswarteschlangen andere Bedingungen gelten.

In der Abbildung 5.5 ist zuerst die beispielhafte Darstellung des Suchraums von A-Stern in einfacher Ausführung zu sehen. Im mittleren Teil der Abbildung sieht man, den momentanen Zustand des bidirektionalen A-Stern Algorithmus, wenn der erste, möglicherweise optimaler, Pfad gefunden wird. In diesem Bild berühren sich die Mengen der abgearbeiteten Knoten nur. Um garantieren zu können, dass der gefundene Pfad optimal ist, muss die Suche fortgesetzt werden bis alle Kandidaten mit Kosten f niedriger als die Kosten des gefundenen Pfades μ untersucht wurden. Dies resultiert dann in einem größeren Suchraum als bei dem einfachen A-Stern Algorithmus, was man im unteren Teil der Abbildung 5.5 sieht.

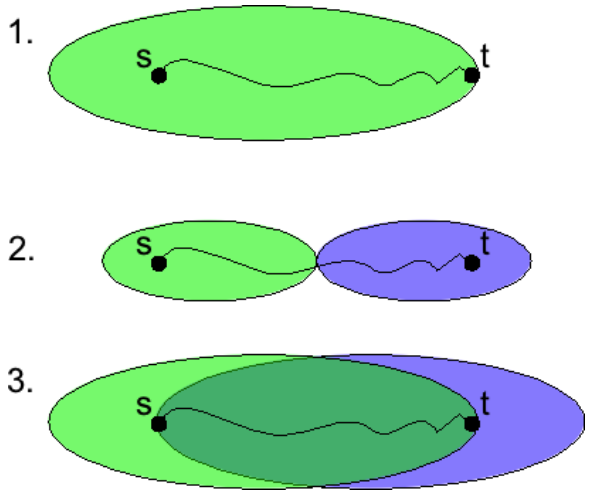


Abbildung 5.5: Suchraum von A-Stern

5.7 Potentialfelder und Hill Climbing

Potentialfelder und der Hill Climbing Algorithmus verfolgen ein anderes Konzept. Diese Verfahren brauchen nicht zwingend einen Graphen, wie Dijkstra oder A-Stern, können jedoch auch mit einem Graphen verwendet werden.

Das Prinzip der Potentialfelder wird am einfachsten erklärt durch eine unebene Fläche und eine Kugel, die auf der Fläche platziert wird. Höher gelegene Stellen haben ein hohes Potential, niedrig gelegene Stellen haben ein niedriges Potential. Die Kugel rollt nach unten, also immer zum niedrigeren Potential. Das Potentialfeld wird durch das Ziel, wo das niedrigste Potential herrscht, vorgegeben. Mögliche Hindernisse stellen hohe Potentiale dar, die als Hügel interpretiert werden können, und wirken abstoßend. Durch die Bestimmung des Gradienten des Feldes ist die Richtung mit dem niedrigeren Potential bekannt. Diese Richtung wird als Bewegungsrichtung bestimmt. Der Algorithmus berechnet also nicht den kompletten Pfad, sondern nur die Richtung, in die man sich bewegen soll. Auf einen Graphen ausgeführt, wird für einen Knoten der nächste Knoten auf dem Pfad zum Ziel berechnet. Auf diese Weise kann man den kompletten Pfad berechnen. Der Hill Climbing Algorithmus arbeitet ähnlich, nur dass dieser sich zum höheren Potential bewegt.

Beide Algorithmen haben gleiche Vor- und Nachteile.
Vorteile:

- Sie sind sehr schnell, wenn man sie nur für eine Richtungsbestimmung verwendet.
- Sie benötigen keinen Graphen, können aber damit verwendet werden.

Nachteile:

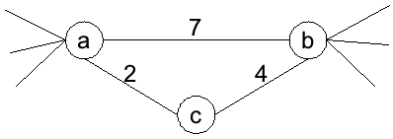


Abbildung 5.6: Unnötige Kante

- Diese Algorithmen berechnen nicht die optimalen Pfade.
- Lokale Minima (bzw. Maxima bei Hill Climbing) lassen den Algorithmus keine Lösung finden. Kann durch Backtracking umgangen werden, verlangsamt aber den Algorithmus.

Diese Algorithmen werden besonders in Computerspielen bevorzugt eingesetzt, wo es mehr um die Geschwindigkeit als um die Optimalität geht. Dort werden sie auf offenes Gelände mit Hindernissen angewendet und nicht auf einen Graphen.

5.8 Contraction Hierarchies

Contraction Hierarchies wurden an der Universität Karlsruhe entwickelt [13]. Das Verfahren ist eine Beschleunigungstechnik der bidirektionalen Suche.

Alle Knoten werden nach einer bestimmten Reihenfolge sortiert und mit einer Zahl (*order number*) versehen. Die Knoten werden nach dieser Reihenfolge abgekürzt. D.H. Kantenkombinationen, die durch diesen Knoten gehen, werden als Abkürzungskanten zusammengefasst. Anschließend wird dieser Knoten, als kontrahiert markiert. Wenn alle Knoten als kontrahiert markiert wurden, ist die Vorberechnung abgeschlossen. Das Verfahren arbeitet mit vorberechneten Kantengewichten, um den maximalen Effizienzgewinn zu erzielen. So werden beispielsweise Kanten, die in keinem kürzesten Pfad vorkommen können, wie die Kante ab in Abbildung 5.6, aus dem Graphen entfernt. Eine Änderung der Kostenfunktion erfordert deshalb eine neue Berechnung der Contraction Hierarchy.

Ein Knoten wird kontrahiert, indem man alle eingehenden Kanten mit allen ausgehenden Kanten verbindet und die Verbindungen, die kürzeste Verbindungen darstellen, als Abkürzungen in den kontrahierten Graphen einfügt. In der Abbildung 5.7 ist eine Kontraktion eines Graphen demonstriert. Die Kontraktion wird nach folgendem Schema gemacht.

1. Berechne alle Gewichte der Kanten.
2. Sortiere alle Knoten und weise ihnen eine Nummer in der Sortierreihenfolge (*order number*) zu.
3. Entferne unnötige Kanten.

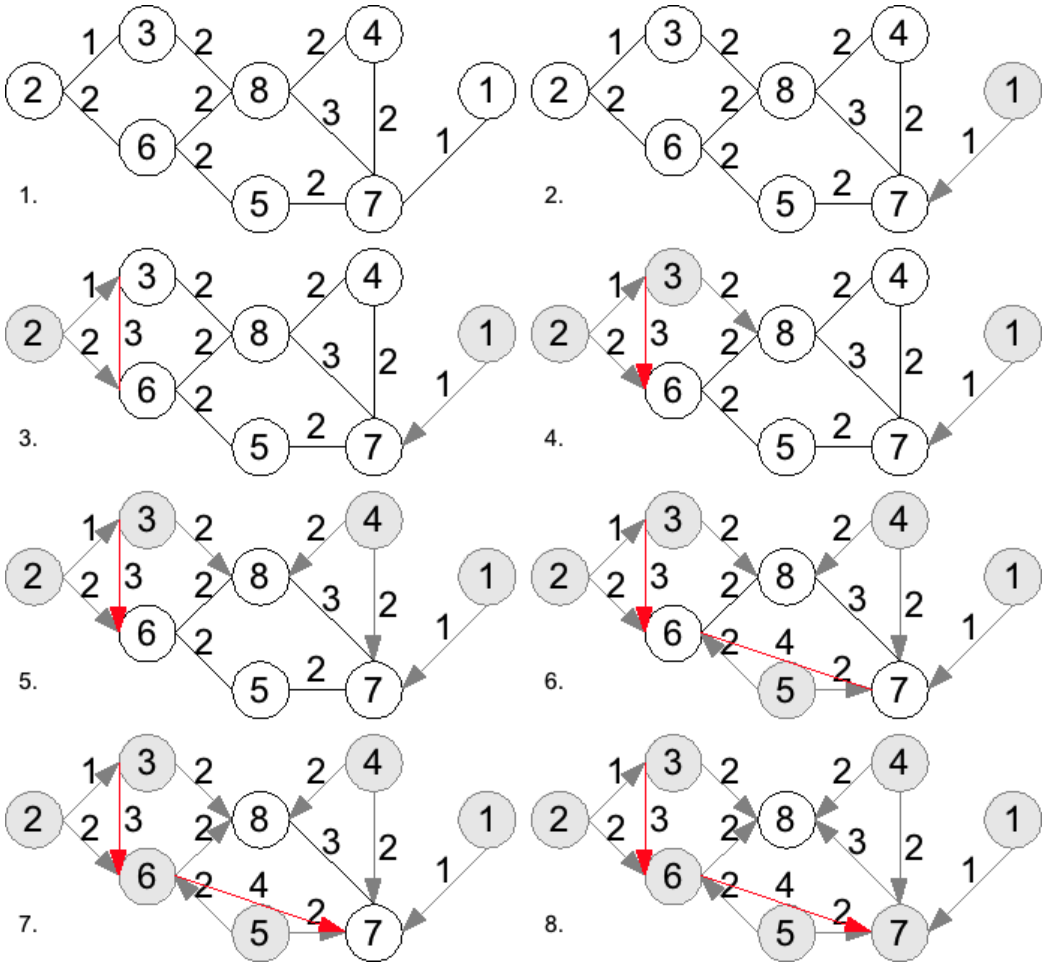


Abbildung 5.7: Graphkontraktion

4. Kontrahiere alle Knoten nach der Sortierreihenfolge der *order number*.
   (a) Betrachte alle Kombinationen aus eingehenden und ausgehenden Kanten als Abkürzungsverbindungen.
   (b) Ist so eine Verbindung der kürzeste Pfad, füge sie in den Graphen ein.
   (c) Markiere den Knoten als kontrahierten Knoten.

Bei der Suche werden nur Kanten betrachtet, die zu einem Knoten mit einer höheren *order number* führen. Es muss eine bidirektionale Suche verwendet werden, da sonst nicht immer ein Pfad gefunden werden kann, weil auf einem reduzierten Graphen gesucht wird, dessen Kanten nur in eine Richtung vorhanden sind. Diese Richtung ist nicht als mögliche Bewegungsrichtung zu verstehen, sondern als eine Weise, wie die Kanten im Graphen gespeichert sind. Nur Knoten, die mit ausgehenden Kanten verbunden sind, werden als Nachbarknoten

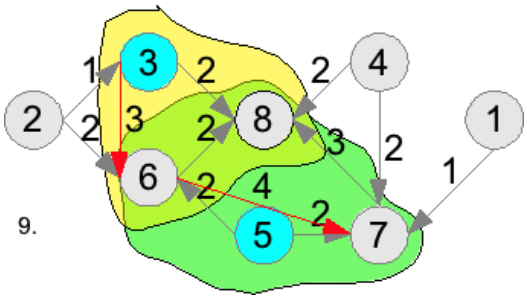


Abbildung 5.8: CH-Suchmenge

gespeichert. Als Beispiel sehen Sie sich den Knoten 8 in der Abbildung 5.7 im fertig kontrahierten Graph an. Es gibt keine Kanten, die vom Knoten 8 ausgehen. Im Suchalgorithmus würde die Methode `getNeighbours` keine Nachbarknoten liefern. Nachbarknoten von Knoten 3 wären 6 und 8. Die Kante von 3 zu 6 ist eine Abkürzungskante und müsste am Ende des Algorithmus entpackt werden. Sie würde im berechneten Pfad durch die beiden Kanten 3-2 und 2-6 ersetzt werden.

Der Algorithmus arbeitet wie folgt.

1. Erstelle *einmalig* die *Contraction Hierarchy*, den veränderten Graphen.
2. Benutze bidirektionales Dijkstra zum Finden des kürzesten Pfades.
3. Entpacke den berechneten Pfad. Ersetze alle Abkürzungskanten durch die Originalkanten.

Nun kommt die Frage nach der Sortierung der Knoten. Nach welchen Kriterien werden die Knoten sortiert? In [13] werden 2 Strategien für die Sortierung der Knoten vorgestellt. Es wird auch erwähnt, dass das Verfahren auch mit beliebiger Sortierreihenfolge funktioniert, obwohl bessere Sortierung eine starke Steigerung der Performanz zu Folge hat. Alle Knoten werden in eine Prioritätswarteschlange eingefügt und beim Entnehmen wird eine fortlaufende Nummer als *order number* zugewiesen. Die Priorität, mit der die Knoten hinzugefügt werden, ist eine Kombination aus verschiedenen Parametern. Bei Interesse lesen Sie die angegebene Literatur. Vereinfacht ausgedrückt, werden die Knoten, bei deren Kontraktion mehr Kanten entfernt werden als neue hinzukommen, bevorzugt. Auf diese Weise werden Knoten, die in Sackgassen sind, zuerst kontrahiert. Danach folgen die Knoten, die nur 2 Nachbarknoten haben, dann mit 3 Nachbarknoten usw.

Eine CH-Suche untersucht deutlich weniger Knoten als eine einfache bidirektionale Suche. Im Graphen aus der Abbildung 5.8 ist zu sehen, dass eine Suche vom Knoten 3 zum Knoten 5 nur die Knoten 3,6,8 in der Hinrichtung durchsuchen kann und aus der Gegenrichtung sind nur die Knoten 5,6,7,8 erreichbar. Zwar gehört der Knoten 2 zu dem kürzesten Pfad 3-2-6-5, doch bei der Suche wird er erst beim Entpacken des fertig berechneten Pfades benötigt. In einem grö-

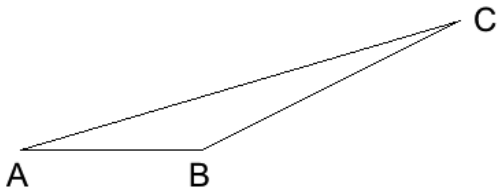


Abbildung 5.9: Dreiecksungleichung

ßeren Graphen ist die Reduzierung der Suchmenge größer, da dort auch Abkürzungen von Abkürzungen existieren und damit über mehrere Hundert Knoten gehen können.

5.9 A-Star Landmark Triangle Inequality (ALT)

Eine weitere Beschleunigungstechnik ist ALT, deren Name eine Zusammensetzung aus den englischen Begriffen für A-Stern, Landmarken und der Dreiecksungleichung ist [14]. Zunächst ein Mal das simple Prinzip der Dreiecksungleichung, das in der Abbildung 5.9 demonstriert ist. Das besagt, dass die Summe der Längen zweier Seiten in einem Dreieck immer mindestens so groß ist wie die Länge der dritten Seite.

$$|\vec{AB}| + |\vec{BC}| \geq |\vec{AC}| \tag{5.6}$$

Daraus folgt:

$$|\vec{AB}| \geq |\vec{AC}| - |\vec{BC}| \tag{5.7}$$

Da die Ungleichung für alle drei Seiten gilt, gilt auch:

$$|\vec{AB}| + |\vec{AC}| \geq |\vec{BC}| \text{ und } |\vec{AB}| \geq |\vec{BC}| - |\vec{AC}| \tag{5.8}$$

Daraus folgt, dass eine Seite immer mindestens so lange ist wie die Differenz der anderen beiden Seiten.

$$|\vec{AB}| \geq ||\vec{AC}| - |\vec{BC}|| \tag{5.9}$$

Diese Regel wird ausgenutzt, um die Heuristikfunktion im A-Stern Algorithmus zu verbessern. Es werden Landmarken auf der Karte verteilt, zu denen in jedem Knoten die kürzeste Entfernung gespeichert wird. Bei der Suche werden eine oder mehrere Markierungen ausgewählt, bei denen die Differenz der Entfernung zum Startknoten und der Entfernung zum Zielknoten am größten ist. Diese Landmarken werden nun für die Berechnung der Heuristik verwendet. Beim Berechnen der Heuristik für einen Knoten A wird die minimale Entfernung für die-

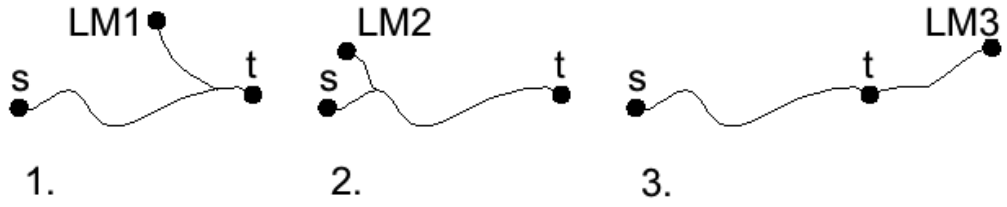


Abbildung 5.10: ALT Landmarken

sen Knoten zum Zielknoten B geschätzt. Aus der Formel 5.9 sieht man, dass diese Entfernung mindestens so Lange ist wie die Differenz der gespeicherten Entfernungen zur Landmarke C. Wenn man mehrere Landmarken verwendet, nimmt man das Maximum aller Schätzungen. Da für alle Schätzungen h_n gilt $|\vec{AB}| \geq h_n$. Würde man eine niedrigere als die maximale Schätzung h_{max} nehmen, würde die Formel $|\vec{AB}| \geq h_n$ nicht erfüllt. So lautet die Heuristikfunktion:

$$h(node) = \max_{i=0}^{n} |dist(node, lm_i) - dist(goalnode, lm_i)| \tag{5.10}$$

In der Abbildung 5.10 kann man einige unterschiedliche Fälle der Landmarkenposition sehen und ihre Effizienz. Die Landmarke $LM1$ ist schlecht geeignet für die Suche zwischen s und t, weil eine Schätzung der Entfernung zu große Abweichung von der tatsächlichen Entfernung ergibt. Bei der Landmarke $LM2$ ist die Überlappung der Pfade s, t und $LM1, t$ groß und die Heuristik der Knoten, die sich darauf befinden, wird exakt geschätzt. Der ideale Fall ist, wenn die Differenz der Pfade LM, s und LM, t genau der Entfernung des Pfades s, t entspricht. Dies ist dann der Fall, wenn der Pfad s, t vollständig auf dem Pfad LM, s oder LM, t liegt. In einem solchen Fall wird der Heuristikwert exakt geschätzt und es werden nur die Knoten auf dem optimalen Pfad abgearbeitet.

In dem Fall 3 der Abbildung 5.10 ist die die Differenz $LM, s - LM, t$ am größten und daher die Landmarke $LM3$ am geeignetsten.

Da zu jeder Landmarke eine Entfernung zu jedem Knoten gespeichert werden muss, sollten wenige Landmarken verwendet werden, die das Kartenmaterial so abdecken, dass zu jedem Knotenpaar mindestens eine Landmarke existiert, die eine gute Schätzung für dieses Knotenpaar erlaubt. Für die Auswahl der Landmarken werden in [14] einige Methoden genannt, die hier nicht konkret erläutert werden. Landmarken am Rand der Karte haben eine hohe Knotenabdeckung, d.h. für viele Knotenpaare ist die Schätzung gut. Deswegen werden 8,16 oder 32 Landmarken verteilt über den Rand der Karte ausgewählt. Je mehr Landmarken, desto besser die Abdeckung, aber auch desto größer der Speicherplatzbedarf.

5.10 Vergleich

Nun werden die Algorithmen verglichen und ein geeigneter Algorithmus ausgewählt. Auswahlkriterien für den Algorithmus sind:

- Laufzeit
- Speicherverbrauch
- Anzahl der untersuchten Knoten

Bellman-Ford-Algorithmus hat seine Hauptstärke darin, dass er mit negativen Kantengewichten umgehen kann. Da wir keine negativen Kantengewichte erlauben, wird dieser Algorithmus nicht weiter betrachtet. Potentialfelder und Hill Climbing Algorithmen haben Schwächen mit lokalen Extremen und der berechnete Pfad wäre nicht optimal.

Auf einem Mobiltelefon stellt meistens der Flash-Speicher den Flaschenhals für die Performanz dar. Deshalb muss man versuchen möglichst wenige Knoten vom Speicher laden zu müssen. Der Dijkstra Algorithmus kann zwar mehr Knoten pro Sekunde abarbeiten als der A-Stern Algorithmus, da er keine Heuristikwerte berechnen muss, aber er untersucht insgesamt mehr Knoten. Insgesamt resultiert das in einer längeren Laufzeit.

Der ALT-Beschleunigungsalgorithmus benötigt viel Speicher für die zwischengespeicherten Distanzen. Jeder Knoten muss die Distanzen zu alle Landmarken speichern. Bei 8 Landmarken werden für 8 Distanzen als 32-bit Fließkommazahl pro Knoten 32 Byte benötigt. Da am Anfang die App nur für bestimmte Städte funktionieren soll, wird wenig Speicher benötigt und der größere Speicherbedarf wäre nicht das Hauptproblem. Allerdings ist der Performanzgewinn auf die kleine Differenz zwischen den tatsächlichen Kosten und den geschätzten Kosten, die mit ALT geschätzt wurden, zurückzuführen. Die zwischengespeicherten Distanzen sind nur für *ein* Profil vorberechnet und würden bei einem abweichenden Profil schlechtere Ergebnisse liefern, aber immer noch besser als die simple Luftliniendistanz-Heuristik. Da die Kostenfunktion, wie im Abschnitt 6.3.1 beschrieben, nie Kosten unter der Distanz berechnet, wäre es möglich ALT als Heuristik einzusetzen.

Die CH ist ein vorberechnendes Verfahren und lässt sich ohne Neuberechnung nicht mit einem anderen Profil benutzen, da beispielsweise Kanten entfernt wurden. CH muss nur sehr wenige Knoten laden, um eine Route zu berechnen, weil viele Knoten bei der Suche übersprungen werden und so nicht in den Speicher geladen werden müssen. Dies hat zwei Vorteile, man benötigt weniger Arbeitsspeicher und das Verfahren hat eine geringe Laufzeit. Leider kann das Verfahren nicht in seiner reinen Form benutzt werden, da in der App mehrere Profile verwendet werden sollen, die sich auch ändern.

Die Benutzung von Abkürzungskanten, wie sie z.B. in CH verwendet werden, kann die Suche enorm beschleunigen. Der Nachteil ist, dass die abgekürzten

Knoten in einer unidirektionalen Suche nicht mehr gefunden werden können. Deshalb muss in dem Fall eine bidirektionale Suche verwendet werden.

Insbesondere weil in den Daten von NAVTEQ Kanten immer aus mindestens 3 Teilkanten bestehen, ist die Beschleunigung durch Abkürzungskanten wie in CH effizienter als die Beschleunigung durch die Heuristikfunktion. Unter anderem auch, weil dadurch weniger Festspeicherzugriffe benötigt werden, deren Reduzierung bei einem Mobiltelefon einen entscheidenden Effizienzgewinn bedeutet.

Es wurde ein bidirektionaler A-Stern Algorithmus mit einigen Eigenschaften von CH, die es dem Algorithmus ermöglichen Abkürzungskanten zu benutzen, implementiert, der im Abschnitt 6.3 beschrieben ist.

Kapitel 6

Entwurf

Im folgenden Kapitel wird der Entwurf der App beschrieben. Am Anfang wird die Architektur der App mit den einzelnen Bausteinen beschrieben. Danach wird das Datenmodel der Kartendaten kurz erläutert. Der wichtigste Teil ist die Beschreibung des Routingalgorithmus mit der detaillierten Erklärung der Kostenfunktion und der Heuristikfunktion.

6.1 Architektur

Die Architektur der Anwendung teilt sich in Folgende Module:

- Graphical User Interface (Graphische Benutzerschnittstelle) (GUI)
  - MapView
  - ProfileView
- Database (DB)
  - MapData
  - ProfileData
- Importer
- CacheGraph
- Positioning
- ProfileManager
- RoutingService

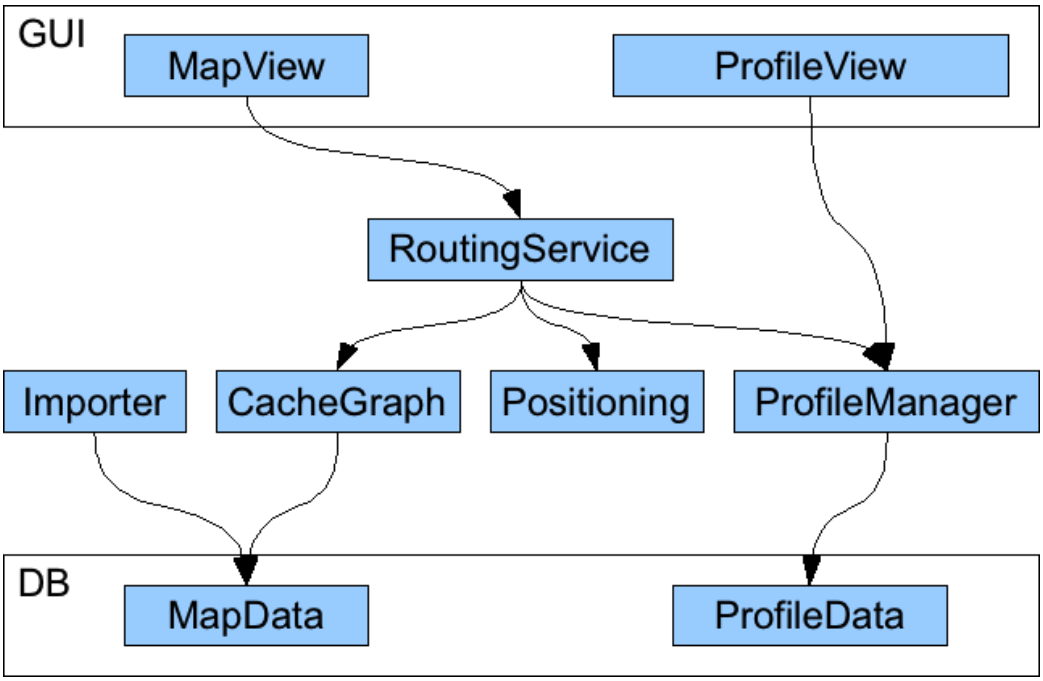


Abbildung 6.1: Architektur

Die Beziehungen der Module sind in der Abbildung 6.1 visualisiert. Die Module in der Kategorie GUI sind für die Anzeige und Interaktion mit dem Benutzer verantwortlich. Die MapView ist für die Anzeige der Karte mithilfe einer externen Application Programming Interface (API) verantwortlich und nimmt Zieleingaben vom Benutzer an. Mit der ProfileView verwaltet der Benutzer seine Profile. Das RoutingService Modul ist für die Planung und Verfolgung einer Route verantwortlich. Dazu verwendet es das Positioning Modul, das später erweitert werden soll, um mit dem Galileo-Satelliten-Empfänger zu arbeiten. Der Datenzugriff auf die Profildaten ist durch den ProfilManager entkoppelt. Die Entkoppelung der Kartendaten ist durch CacheGraph realisiert. Das Importer-Modul importiert die Kartendaten in die Datenbank mit den Kartendaten und sorgt für die nötige Struktur der Daten. Das Hauptmodul ist somit das RoutingService Modul, das die anderen Module dazu verwendet um eine Route zu berechnen und zu verfolgen.

6.2 Datenmodell

Die Abbildung 6.2 zeigt das in der App benutzte Datenmodell für die Speicherung der verwendeten Kartendaten. Da in der App wahlweise die Kartendaten von OSM oder NAVTEQ verwendet werden können, stellt dieses Modell eine

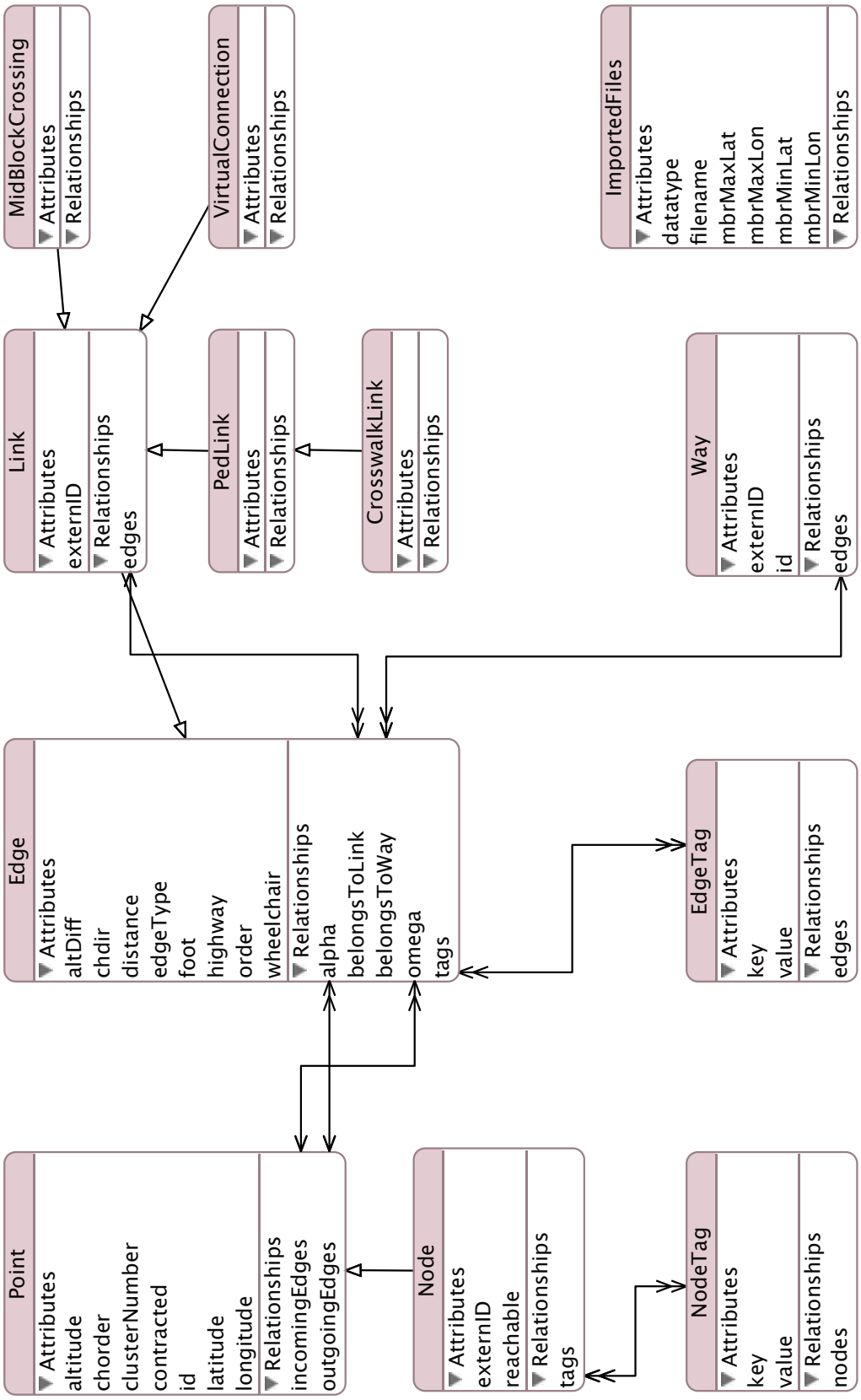


Abbildung 6.2: Modell der Kartendaten

Schnittmenge der importierten Formate dar.

Das Schema wurde mit CoreData modelliert. Es repräsentiert einen attributierten Graph. Da mehr Kontrolle über die Daten benötigt war, wurde ein eigener Mechanismus zum Laden und Verwalten der Graph Daten im Hauptspeicher implementiert. CacheGraph ist die Hauptschnittstelle zum Graphen der Kartendaten.

6.3 Routingalgorithmus

Der Algorithmus, der in der App verwendet wird, ist ein CH-ähnlicher Algorithmus, der einen Wechsel des Profils erlaubt, ohne dass eine neue Vorberechnung benötigt wird. Die Ähnlichkeit mit CH ist, dass Abkürzungskanten bei der Suche genau wie bei CH verwendet werden. Die Abweichung ist in der Erstellung dieser Abkürzungskanten.

Als Erstes werden Abkürzungskanten für Links erstellt, die bereits vorgegeben sind, wie das in NAVTEQ Daten der Fall ist. Dabei wird jedem Zwischenpunkt (shapepoint) eine `order number` von 0 zugewiesen und als kontrahiert markiert. Alle anderen Knoten (node) bekommen eine `order number` von 1. Als Nächstes werden die Abkürzungskanten vom Startknoten (alpha) zum Zielknoten (omega) der Links erstellt. Dabei werden Abkürzungskanten über mehrere Zwischenpunkte hinweg erstellt. Das ist ebenfalls ein Unterschied zu CH, bei dem immer nur 1 Knoten kontrahiert wird. Im nächsten Schritt werden die Sackgassen kontrahiert, indem die Knoten in einer Sackgasse als kontrahiert markiert werden und die `order number` der nicht kontrahierten Knoten um 1 inkrementiert wird. Beim Kontrahieren von Sackgassen ist keine zusätzliche Kante nötig. Schließlich wird jeder Kante eine CH-Kontraktionsrichtung (`chdir`) zugewiesen, die eine Redundanz der `order number` darstellt. Die Kontraktionsrichtung (`chdir`) ist die Differenz der `order number` von omega und alpha.

$$edge.chdir = edge.omega.chorder - edge.alpha.chorder \tag{6.1}$$

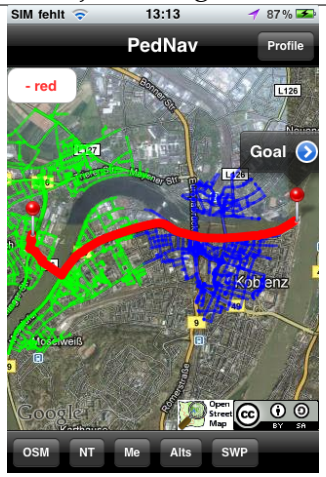
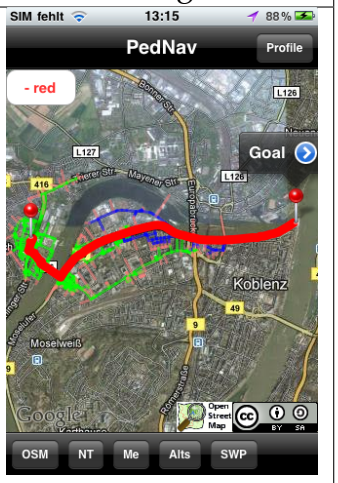
Mit dieser Information ist beim Laden der Kanten kein Laden der zugehöriger Knoten notwendig. So wird das Laden der Kanten beschleunigt. Wie schon im Abschnitt 5.8 erklärt werden nur Kanten geladen, die zu einem Knoten mit höherer `order number` führen. Da hier Knoten auch gleiche `order number` haben können, werden auch Kanten zu Knoten geladen, die die gleiche `order number` haben. Kanten werden für einen Knoten geladen, wenn dieser Knoten in der Kante als `alpha` gesetzt ist und `chorder` größer oder gleich 0 ist oder wenn dieser Knoten in der Kante als `omega` gesetzt ist und `chorder` kleiner oder gleich 0 ist. Auf diese Weise werden nur Kanten geladen, die bei der Suche gebraucht werden und der teuere `TABLE JOIN` mit der Tabelle der Knoten wird vermieden.

Die Suche wird von einem bidirektionalen A-Stern Algorithmus erledigt. Da in der originalen CH der bidirektionale Dijkstra Algorithmus verwendet wird,

musste ein Vergleich gemacht werden, der diese beiden Algorithmen miteinander vergleicht.

Für eine Testroute von der Universität Koblenz (lon=7,559168 lat=50.36287) zum Deutschen Eck in Koblenz (lon=7,605448 lat=50,36448) wurden folgende Daten auf einem iPhone 3GS gemessen, die in der Tabelle 6.1 aufgelistet sind. Die Kosten der resultierenden Route sind 3973,3. Diese Zahl kann als eine Distanzangabe in Metern angesehen werden, wenn alle Parameter im Profil die Distanzfunktion nicht beeinflussen. Diese Route wurde mit den Daten von OSM berechnet und sie enthält 97 Knoten.

Tabelle 6.1: Vergleich der bidirektionalen Algorithmen

Algorithmus	BiDijkstraAlgorithm	BiAStarAlgorithm
Visualisierung		
Abgearbeitete Knoten	3836	1785
Anfragezeit	1,593 s	0,906 s
Effizienz	2,528 %	5,434 %
Knoten/Sekunde	2407,7	1969,2

Da der bidirektionale A-Stern Algorithmus deutlich weniger Knoten untersucht, ist dieser auch besser geeignet. Die Effizienz (Knoten auf dem resultierenden Pfad / abgearbeitete Knoten) von Bi-A-Stern liegt in diesem Fall bei 5,4%, was mehr als doppelt so viel ist, wie bei Bi-Dijkstra. Bei so einem großen Unterschied hilft die Tatsache, dass Bi-Dijkstra mehr Knoten pro Sekunde abarbeiten kann, wenig.

6.3.1 Kostenfunktion

Die Kostenfunktion bestimmt, welcher Pfad die geringsten Kosten aufweist und dadurch als optimaler Pfad berechnet wird. Der Benutzer legt seine Präferenzen im Profil fest und nach diesen Präferenzen werden die Kosten der Kanten berechnet. Je höher die Kosten, desto weniger wird diese Kante bevorzugt. Die Kosten werden auf Basis eines Grundwertes berechnet, der Distanz. Man könnte auch die Reisezeit als Basis verwenden, aber da die Fußgänger durchschnittlich mit dem gleichen Tempo unterwegs sind, sind die beiden Größen austauschbar.

Wenn der Benutzer ein Attribut heruntergestuft hat und dieses Attribut in der Kante vorhanden ist, dann werden die Kosten dieser Kante erhöht. Diese Kostenerhöhung stellt eine Benachteiligung der Kante dar. Eine Bevorzugung würde durch eine Kostensenkung erzielt. Die Kostensenkung würde möglicherweise eine Inkonsistenz mit der Heuristikfunktion hervorrufen. Dann wäre die Berechnung des optimalen Pfades nicht mehr garantiert. Aus diesem Grund wird nur eine Erhöhung der Kosten erlaubt und keine Senkung.

Die Kosten werden mit der Formel $cost(edge)$ berechnet. Die Attributwerte der Kante werden mit $edge.att_n$ definiert, für n Attribute. Diese Werte enthalten für boolesche Attribute Werte zwischen 0 und 1. 0 bedeutet, dass das Attribut auf diesem Streckenabschnitt nicht vorhanden ist. 1 bedeutet, dass es für die gesamte Länge des Streckenabschnitts vorhanden ist. Werte dazwischen sind nur bei Abkürzungskanten zu finden und repräsentieren den Anteil der Abkürzungskante, auf der das Attribut vorhanden ist. 0,34 würde bedeuten, dass 34% der Kante dieses Attribut besitzt. Für numerische Attribute sind die Zahlen gespeichert. Z.B. für die Neigung einer Kante würde 0,07 eine Steigung von 7% bedeuten.

Das Profil speichert für jedes boolesches Attribut einen Benutzungswert zwischen 0 und 1, das $pUse(i)$. Ist der Wert auf 1 gesetzt, was der Standardeinstellung entspricht, dann beeinflusst diese Einstellung nicht die berechneten Kosten. Ein niedriger Wert bedeutet eine Abwertung von Kanten mit diesem Attribut. Für numerische Attribute werden zwei Werte gespeichert: ein Schwellenwert (`threshold`) $pThr(i)$ und ein Grenzwert $pMax(i)$, der stets größer als der Schwellenwert ist. Um den Benutzer nicht durch 2 Einstellungen pro Attribut zu irritieren, wurde der $pMax(i)$ Wert vorerst nicht vom Benutzer einstellbar gemacht. Was später geändert werden kann, wenn ein dafür geeignetes GUI-Element gefunden wird.

Unabhängig vom Attributtyp wird ein Faktor $faktor(edge, i)$ berechnet, um den die Basiskosten erhöht werden. Der Faktor ist ein Wert zwischen 0 und 1 und stellt eine prozentuale Erhöhung der kosten dar. Auf diese Weise wird eine Benachteiligung der Kante bewirkt. Wenn der Faktor auf 0 berechnet wird, gibt es keine Veränderung der Basiskosten. Ergibt der Faktor 1, wird diese Kante für die Berechnung der Route gesperrt, indem die Kosten auf ∞ gesetzt werden.

Bei booleschen Attributen ist der Faktor $faktor(edge, i)$ gleich dem Wert $(1 - pUse(i))$, wenn der Attributwert auf 1 gesetzt ist. Bei numerischen Attributen ist der Faktor das Verhältnis der Differenz von Attributwert und Schwellenwert zu

der Differenz von Grenzwert und Schwellenwert. Ist der Attributwert niedriger oder gleich dem Schwellenwert, dann ist der resultierende Faktor gleich 0. Ist der Attributwert höher oder gleich dem Grenzwert, dann ist der resultierende Faktor gleich 1.

$$cost(edge) = \begin{cases} \infty, & \text{wenn } \max_{i=0}^{n}(faktor(edge,i)) \geq 1 \\ dist(edge) * (1 + \sum_{i=0}^{n} faktor(edge,i)), & \text{sonst} \end{cases} \tag{6.2}$$

$$faktor(edge,i) = \begin{cases} faktorbool(edge,i) & \text{wenn } edge.att_i \text{ boolisch ist} \\ faktornum(edge,i) & \text{wenn } edge.att_i \text{ nummerisch ist} \end{cases} \tag{6.3}$$

$$faktorbool(edge,i) = (1 - pUse(i)) * edge.att_i \tag{6.4}$$

$$faktornum(edge,i) = \begin{cases} 0 & \text{wenn } edge.att_i - pThr(i) \leq 0 \\ \frac{edge.att_i - pThr(i)}{pMax(i) - pThr(i))} & \text{sonst} \end{cases} \tag{6.5}$$

$$pUse(i) \rightarrow [0,1] \tag{6.6}$$

$$pThr(i), pMax(i) \rightarrow dom(edge.att_i) \tag{6.7}$$

Um Abkürzungen (`shortcut`s) nutzen zu können, muss die Kostenfunktion die gleichen Kosten für die Abkürzungskante wie für die Teilkanten berechnen. Es muss also Folgendes gelten.

$$cost(a) + cost(b) = cost(shortcut(a,b)) \tag{6.8}$$

Um Kosten auf einer Abkürzungskante berechnen zu können, müssen die Attributwerte der Teilkante darin so zusammengefasst sein, dass die Berechnung der Kosten möglich ist und die obige Gleichung erfüllt ist.

Beispielrechnung:

$$cost(a) + cost(b) = cost(shortcut(a,b)) \tag{6.9}$$

$$dist(a)f(a) + dist(b)f(b) = dist(c)f(c)$$

$$dist(a)(1 + fak(a.at_1) + fak(a.at_2)) + dist(b)(1 + fak(b.at_1) + fak(b.at_2)) =$$
$$= (dist(a) + dist(b))(1 + fak(c.at_1) + fak(c.at_2))$$

$$dist(a) + dist(a)fak(a.at_1) + dist(a)fak(a.at_2) +$$
$$+dist(b) + dist(b)fak(b.at_1) + dist(b)fak(b.at_2) =$$
$$= dist(a) + dist(b) + (dist(a) + dist(b))(fak(c.at_1) + fak(c.at_2))$$

$$dist(a)fak(a.at_1) + dist(a)fak(a.at_2) + dist(b)fak(b.at_1) + dist(b)fak(b.at_2) =$$

$$= (dist(a) + dist(b))(fak(c.at_1) + fak(c.at_2))$$

$$fak(c.at_1) + fak(c.at_2) =$$

$$= \frac{dist(a)fak(a.at_1) + dist(a)fak(a.at_2) + dist(b)fak(b.at_1) + dist(b)fak(b.at_2)}{dist(a) + dist(b)}$$

$$fak(c.at_1) + fak(c.at_2) =$$

$$= \frac{dist(a)fak(a.at_1) + dist(b)fak(b.at_1)}{dist(a) + dist(b)} + \frac{dist(a)fak(a.at_2) + dist(b)fak(b.at_2)}{dist(a) + dist(b)}$$

In der letzten Gleichung sieht man, dass die Gleichung erfüllt wird, wenn man für $c.at_i$ den Mittelwert der Attribute der Teilkanten $a.at_i$ und $b.at_i$ verwendet. Bei den booleschen Attributen reicht es diesen Mittelwert zu speichern, um die Berechnung zu rekonstruieren. Leider ist es bei den numerischen Attributen nicht möglich, die Berechnung vollständig zu rekonstruieren. Deshalb ist es im Moment nicht möglich, Kanten mit unterschiedlichen numerischen Attributen zu Abkürzungskanten zusammenzufassen.

6.3.2 Heuristikfunktion

Die Heuristikfunktion ist die Komponente, die die infrage kommenden Knoten bei der Suche reduzieren kann. Die Heuristikfunktion soll die restlichen Kosten zum Ziel schätzen. Dabei sollte die Funktion die restlichen Kosten nie überschätzen. Überschätzt die Funktion die restlichen Kosten, so ist es nicht mehr garantiert, dass der berechnete Pfad optimal ist. Verfahren, die absichtlich die Kosten überschätzen, um einen Geschwindigkeitsgewinn zu erzielen, nennt man aggressiv.

$$h(node) \geq 0 \wedge h(node) \leq dist(node, goal) \tag{6.10}$$

Um die Heuristikfunktion besser zu verstehen, werden zwei Extremfälle betrachtet. Wenn es keine geeignete Möglichkeit für die Heuristikfunktion gefunden werden kann, wird die Heuristikfunktion $h(node) = 0$ benutzt. In diesem Fall verhält sich die A*-Suche wie der Algorithmus von Dijkstra und es gibt keine Reduzierung des Suchraums.

Nehmen wir Mal an, dass die Heuristikfunktion immer die exakten restlichen Kosten kennt und diese verwendet werden. Dann reduziert sich der Suchraum auf die Knoten, die auf dem optimalen Pfad liegen. Dies wäre der „perfekte“ Algorithmus.

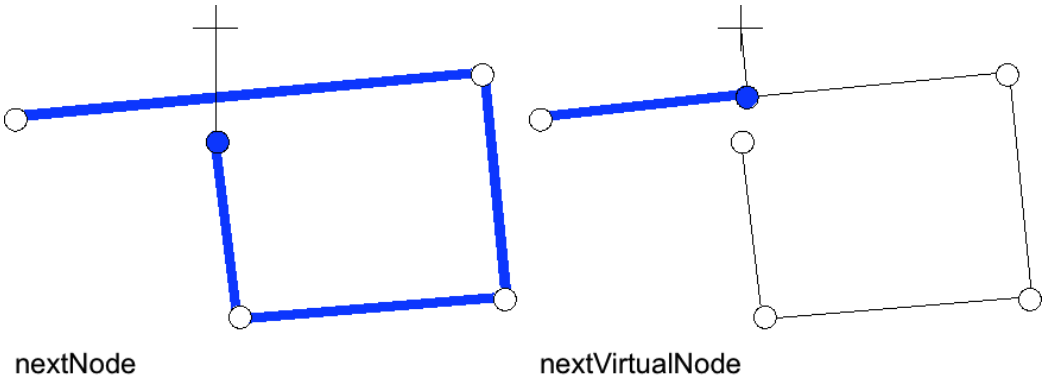


Abbildung 6.3: Knotenauswahl

In der App wird die einfache Luftlinienentfernung als Heuristik verwendet. Diese überschätzt nie die restlichen Kosten, weil die minimalen Kosten, wenn alle Attributfaktoren 0 sind, gleich den Basiskosten sind, diese sind per Definition die Distanz.

Im Abschnitt über ALT 5.9 wird eine effiziente Methode für die Heuristikfunktion mit Vorberechnung erläutert.

6.4 Start- und Zielknoten

Um in großen Datenmengen schnell die passenden Daten zu finden, werden in DBMS Indizes eingesetzt. Z.B. um die Daten eines Knotens mit der $id = 123$ abzufragen, müssen nicht alle Knoten gelesen werden. Allerdings indizieren die normalen Indizes nur 1 Attribut. Bei den räumlichen Daten, wie den Koordinaten von Knoten bzw. Kanten, werden mehrdimensionale Indizes verwendet. Diese mehrdimensionalen Indizes nennt man *spatial Index*, weil sie vorwiegend für die Indizierung von Kartendaten verwendet werden. Hierfür existieren verschiedene Implementierungen. In sqlite wird dies durch das rtree-Modul realisiert, das einen R-Baum-Index implementiert. Die Abfrage geschieht mit der Beschreibung eines Rechtecks, innerhalb dessen die Daten abgefragt werden. Eine Distanzfunktion wird von sqlite nicht bereitgestellt und muss implementiert werden.

Um einen Startknoten für bestimmte Koordinaten auszuwählen, werden in kleinem Radius um diese Koordinaten alle Knoten ausgewählt und die Distanz zu den gesuchten Koordinaten berechnet. Der Knoten mit der kleinsten Distanz könnte dann als Startknoten verwendet werden. Dieses Verfahren ist simpel, hat aber starke Ungenauigkeit, wenn die Knoten weit auseinanderliegen. Dies ist in der Abbildung 6.3 mit dem Namen `getNode` versehen.

Um eine Route präzise zu berechnen, reicht es nicht aus den nächstgelegenen Knoten als Start- bzw. Zielknoten zu nehmen. Diese Randknoten müssen genauer sein. Dafür wird die nächstgelegene Kante ausgesucht und ein virtueller Knoten

erstellt. Dieser Knoten existiert nur für eine einzelne Abfrage und wird nicht gespeichert.

Für gegebene Koordinaten werden alle Kanten durch einen rtree-Index ausgewählt, deren Bounding Box sich mit einem kleinen Rechteck um diese Koordinaten überlappt. Für jede dieser Kanten wird ein Punkt berechnet, der sich auf der Kante befindet und am nächsten zu den gegebenen Koordinaten ist. Mithilfe eines Lots von den gegebenen Koordinaten zu der Kante lässt sich der Punkt berechnen, der am nächsten zu den Koordinaten ist. Dies wird mithilfe einfacher linearer Gleichungen gemacht. Da die gegebenen Koordinaten sich nicht auf einer flachen Ebene befinden, ist dieses Verfahren nicht ganz korrekt, aber es ist eine gute Näherung und lässt sich einfach berechnen. Dadurch werden bessere Randknoten ausgewählt, wie in der Abbildung 6.3 mit dem Namen `getVirtualNode` zusehen ist. Ein virtueller Knoten hat genau 2 Nachbarknoten, die Knoten der Kante auf der er sich befindet. Die virtuellen Knoten sind bei keinen anderen Knoten als Nachbarknoten verknüpft. Die einzige Ausnahme ist, wenn beide virtuelle Knoten sich auf derselben Kante befinden. Aus diesem Grund kann das Verfahren nur mit bidirektionalen Algorithmen verwendet werden, da die unidirektionalen Algorithmen nie den virtuellen Zielknoten finden.

Kapitel 7

Implementierungsspezifische Werkzeuge

In diesem Kapitel werden einige Werkzeuge bzw. Datenstrukturen genannt, die für die Navigationssoftware typisch sind.

7.1 Prioritätswarteschlange (PriorityQueue)

Fast alle genannten Algorithmen, Dijkstra, A-Stern und darauf aufbauende Algorithmen, arbeiten mit einer Prioritätswarteschlange. Die Effizienz einiger Operationen der Prioritätswarteschlange ist maßgebend für die Effizienz des gesamten Algorithmus.

Da man immer vermeiden sollte, etwas zu implementieren, was bereits implementiert ist, habe ich versucht, eine fertige Implementierung einer Prioritätswarteschlange zu verwenden. Nach einigen Testläufen stellte sich heraus, dass diese Implementierung den Speicher nicht korrekt freigegeben hat. Deshalb habe ich selbst eine Prioritätswarteschlange implementiert, die mit einem binären Heap realisiert ist.

7.2 R-Baum (RTree)

Ein R-Baum ist eine Indexstruktur für den effizienten Zugriff auf räumliche Daten. Im Kapitel 6.4 wird die Anwendung des in sqlite eingebauten rtree-Modul beschrieben.

Während der Entwicklung ist es notwendig nicht nur die resultierenden Pfade zu visualisieren, sondern auch den verwendeten Graph. Da die verwendete API für die Karte das Zeichnen darauf nur ausschnittsweise zulässt, mussten auch die Kanten des Graphen ausschnittsweise abgefragt werden können. Da nur die bereits geladenen Kanten angezeigt werden sollten, wäre eine Abfrage

der Datenbank überflüssig. Beim Laden jeder Kante wird sie in einen R-Baum im Speicher eingefügt. Dafür wurde ein RTree nach [16] implementiert. Das Laden des Graphen wird dadurch verlangsamt, deshalb wird es nur während der Entwicklung für die Visualisierung des Kartengraphen verwendet.

7.3 Index Clustering

Index Clustering dient dazu, dass Daten die oft zusammen oder hintereinander aus der Datenbank abgerufen werden, sich möglichst nah beieinander befinden. Das reduziert die Festspeicherzugriffe, weil anstatt mehrerer Seiten (pages) nur eine Seite (page) von der Datenbank geladen wird. Im Fall von räumlichen Daten ist zu erwarten, dass benachbarte Knoten oder Kanten während einer Suche hintereinander abgerufen werden. Das Clustern der Daten würde eine Reduzierung der Speicherzugriffe und damit eine Performanzsteigerung bewirken.

Für das Clustern der zweidimensionaler Daten wurde eine einfache aber wirkungsvolle Methode gewählt. Sqlite clustert alle Daten nach dem `PRIMARY KEY`. Also muss man nur eine passende `id` vergeben um die Daten zu clustern. Die gesamte Menge der Daten wird abwechselnd horizontal und vertikal halbiert. Jede Hälfte für sich wird gruppiert und weiter halbiert. Bei der ersten Halbierung ist die Clusterzahl der ersten Hälfte 0 und die der zweiten Hälfte 1. Bei der nächsten Halbierung wird die Zahl um eine Stelle erweitert, um 0 bei der ersten Hälfte und um 1 bei der anderen Hälfte. Die ersten 4 Schritte sind in der Abbildung 7.1 demonstriert.

Wenn man nun die Zahlen in der Abbildung 7.2 nach Regelmäßigkeiten durchsucht, stellt man fest, dass die Zahlen eine bitweise Zusammensetzung der Koordinaten ist.

Um diese Clusterzahl aus beliebigen zweidimensionalen Koordinaten zu berechnen, werden die Koordinaten in den passenden Zahlenraum interpoliert und dann bitweise zusammengeführt. Um eine 32-bit-Clusterzahl zu berechnen, müssen zwei 16-bit interpolierte Koordinaten zusammengeführt werden. Um Probleme mit Vorzeichen behafteten Zahlen zu vermeiden, wird eine 30-bit Zahl aus zwei 15-bit Zahlen generiert. Die Koordinaten werden aus ihrem normalen Bereich in den erforderlichen Bereich wie folgt interpoliert.

$$x_{int} = \frac{x - x_{min}}{x_{max} - x_{min}} \cdot (2^{15} - 1) \tag{7.1}$$

Beispielrechnung einer 30-bit Zahl für die Koordinaten lon = 9,92669580 und lat = 49,79651345.

$$lon_{int} = \frac{9,92669580 - -180}{180 - -180} \cdot (2^{15} - 1)$$

$$lon_{int} = 0,527574155 \cdot 32767$$

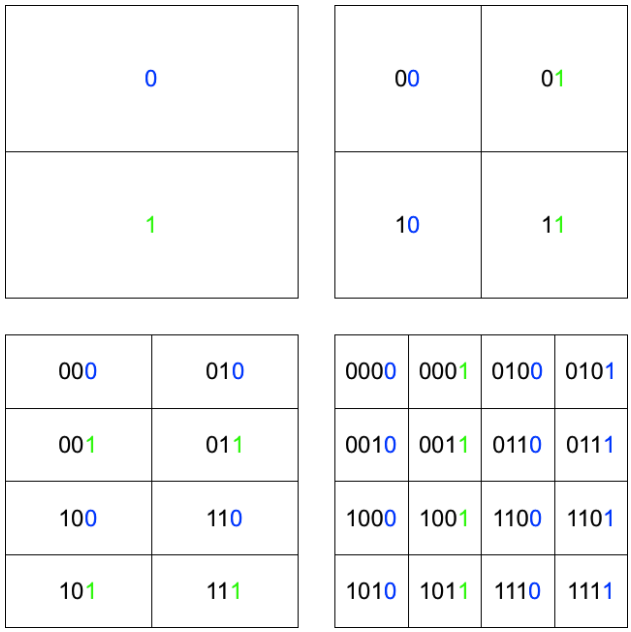


Abbildung 7.1: Clusterzahl - Halbierung

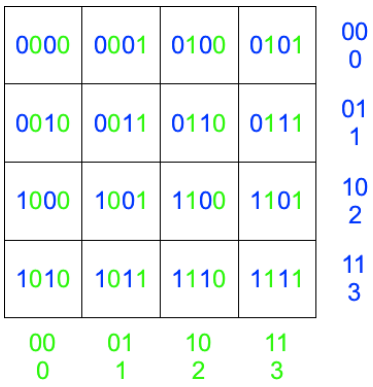


Abbildung 7.2: Clusterzahl - Regelmäßigkeit

$$lon_{int} = 17287$$

$$lon_{bit} = 100\ 0011\ 1000\ 0111$$

$$lat_{int} = \frac{49,79651345 - -90}{90 - -90} \cdot (2^{15} - 1)$$

$$lat_{int} = 0,776647296944444 \cdot 32767$$

$$lat_{int} = 25448$$

$$lat_{bit} = 110\ 0011\ 0110\ 1000$$

$$clusternumber_{bit} = 111000\ 00001111\ 01101000\ 10010101$$

$$clusternumber = 940533909$$

Kapitel 8

Zusammenfassung

Fazit

In dieser Diplomarbeit wurde ein System entwickelt, dass eine Navigation für Fußgänger ermöglicht. Wie beabsichtigt wurde das System für die Benutzung auf einem iPhone realisiert.

Obwohl die Karten der neuen Generation von NAVTEQ sich noch in Entwicklung befinden, konnten erste Eindrücke gesammelt werden, wie die Navigation der Fußgänger in Zukunft aussehen wird. Das System arbeitet aber auch mit Kartendaten von OpenStreetMap, die die klassische Repräsentation der Kartendaten haben.

Die Positionierung kann später, wenn der Galileo-Empfänger zur Verfügung steht, umgestellt werden und Positionsdaten mit höherer Präzision für die Navigation bereitstellen.

Die Routenberechnung konnte mit einem CH-ähnlichen Verfahren durch Vorberechnung beschleunigt werden und erlaubt trotzdem eine Änderung des Profils, ohne dass eine neue Vorberechnung nötig ist. Anders als beim einfachen CH, wo nach einer Änderung des Profils für die Berechnung der Routenkosten eine neue Vorberechnung nötig ist, die im Vergleich zu der Berechnung einer Route aufwendig ist.

In Abbildung 8.1 ist die demonstrative Navigation mit dem fertigen System abgebildet. Diese zeigt eine Berechnung der Routen für 3 unterschiedliche Profile, die unterschiedliche Steigungsgrade bevorzugen. Daraus ergeben sich unterschiedliche Routen.

Ausblick

An dieser Stelle noch einige Ideen, die das Produkt verbessern würden.

Da die App sqlite mit rtree-Modul verwendet, sollte man abwägen, ob die

Abbildung 8.1: Navigation mit PedestrianNavigator

App auf SpatiaLite als Kartendatenspeicher umgerüstet werden soll. Während der Entwicklung der App wurden keine wesentlichen Vorteile entdeckt um SpatiaLite einzusetzen.

Im Moment geschieht die Auswahl des Ziels über die Karte. Ein wichtiger Schritt zum vollständigen Navigationssystem ist die Suche des Ziels über die Eingabe der Adresse. Dies wurde in dieser Arbeit nicht realisiert, aber darüber nachgedacht. Man würde für diese Suche einen Index auf den Adressen hierarchisch aufbauen. Zuerst das Land, dann Stadt und Postleitzahl, dann Straße und wo möglich auch Hausnummer. Diese Einträge würden dann die Koordinate liefern, die als Ziel verwendet werden kann. Man könnte aber auch Online Geocoding Dienste verwenden, um diese Suche zu realisieren.

Ein interessantes Thema ist die *multimodale Navigation*. Navigation mit verschiedenen Transportmitteln auf einer Route. Ein Standardfall einer multimodalen Navigation ist eine gemischte Route aus Fußgängerwegen und den öffentlichen Verkehrsmitteln.

Für den Bereich „Künstliche Intelligenz“ wäre das automatisierte Erlernen der Benutzerpräferenzen und der Anpassung eines dafür vorgesehenen Profils interessant. Die Benutzung von verschiedenen Machine Learning Methoden könnte miteinander verglichen werden.

Glossar

Altitude
Altitude ist die Höhenangabe in Metern über einer festgelegten Nullhöhe. 61, 63

App
Eine Abkürzung für Application (Anwendung/Programm), im modernen Sprachgebrauch ist damit ein Zusatzprogramm für ein Smartphone gemeint. 3, 9, 62

Bounding Box
Eine Bounding Box beschreibt die minimale und maximale Koordinatenwerte jeder Dimension eines Objektes. 54

Galileo
Ein europäisches Satellitensystem, ähnlich dem GPS. 1

Global Positioning System
Ein amerikanisches Satellitensystem, das mit Hilfe eines Empfängers die Ermittlung der Koordinaten ermöglicht. 1, 63

iPhone
Ein iPhone ist ein Smartphone der Firma Apple. 1, 3, 9

Koordinaten
Koordinaten enthalten lat, lon und optional Altitude (alt) und beschreiben einen Punkt auf der Erde. 5

Latitude
Latitude ist die geographische Breite, die Werte zwischen -90° und 90° annimmt und die Entfernung im Winkelmaß vom Äquator angibt. 5, 13, 63

Longitude
Longitude ist die geographische Länge, die Werte zwischen -180° und 180° annimmt und die Entfernung im Winkelmaß vom Nullmeridian angibt. 5, 13, 63

Smartphone
Ein Smartphone ist ein Mobiltelefon, das eine höhere Prozessorleistung hat und mit zusätzlichen Programmen (sog. Apps) ausgestattet werden kann. 61

Tap
Ein Tap ist das gezielte Berühren von berührungsempfindlichen Oberflächen. 10

Abkürzungsverzeichnis

ALT	A-Star Landmark Triangle Inequality. 25, 40, 42, 53
alt	Altitude. 61
API	Application Programming Interface. 46
CH	Contraction Hierarchy. 25, 39, 42, 43
DB	Database. 45
GPS	Global Positioning System. 1, 61
GUI	Graphical User Interface (Graphische Benutzerschnittstelle). 45, 46, 50
lat	Latitude. 13, 61
lon	Longitude. 13, 61
OSM	OpenStreetMap. 3, 46
RDBMS	Relationales Datenbank Management System. 7
SVN	Subversion. 3

Literaturverzeichnis

[1] *Google Elevation API*. http://code.google.com/intl/de-DE/apis/maps/documentation/elevation/

[2] *Java OpenStreetMap Editor Webseite*. http://josm.openstreetmap.de/

[3] *NAPA Projekt Webseite*. http://www.projekt-napa.de/

[4] *Navigon Webseite*. http://www.navigon.com/

[5] *NAVTEQ Webseite*. http://www.navteq.com/

[6] *OpenStreetMap Webseite*. http://www.openstreetmap.org/

[7] *SQLite Documentation*. : *SQLite Documentation*, http://www.sqlite.org/docs.html

[8] *The SQLite R*Tree Module*. : *The SQLite R*Tree Module*, http://www.sqlite.org/rtree.html

[9] *Subversion Webseite*. http://subversion.apache.org/

[10] BAUER, R. ; DELLING, D.: SHARC: Fast and robust unidirectional routing. In: *Journal of Experimental Algorithmics (JEA)* 14 (2009), S. 4

[11] DIJKSTRA, E.W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* 1 (1959), Nr. 1, S. 269–271

[12] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John ; RIEHLE, Dirk (Hrsg.): *Entwurfsmuster*. Addison-Wesley (Deutschland) GmbH, 1996. – ISBN 0–201–63361–2

[13] GEISBERGER, R.: *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks*, Institut fuer Theoretische Informatik Universitaet Karlsruhe (TH), Diploma Thesis, 2008

[14] GOLDBERG, A.V. ; HARRELSON, C.: Computing the shortest path: A* search meets graph theory. In: *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms* Society for Industrial and Applied Mathematics, 2005, S. 156–165

[15] GOLDBERG, A.V. ; KAPLAN, H. ; WERNECK, R.F.: Reach for A*: Efficient point-to-point shortest path algorithms. In: *Proceedings of the eighth Workshop on Algorithm Engineering and Experiments and the third Workshop on Analytic Algorithmics and Combinatorics* Bd. 123 Society for Industrial Mathematics, 2006, S. 129

[16] GUTTMAN, A: R-Trees: A Dynamic Index Structure for Spatial Searching. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD*, 1984, S. 47ff

[17] HEGARTY, Paul: *Developing Apps for iOS*. http://itunes.apple.com/us/itunes-u/developing-apps-for-ios-sd/id395631522. Version: 2010

[18] KAPLAN, E.D. ; HEGARTY, C.J.: *Understanding GPS: principles and applications*. Artech House Publishers, 2006

[19] MUELLER, August: *FMDB Webseite*. https://github.com/ccgus/fmdb

[20] NANNICINI, G. ; DELLING, D. ; SCHULTES, D. ; LIBERTI, L.: Bidirectional A Search on Time-Dependent Road Networks. In: *Journal version of WEA* 8 (2009), S. 110

[21] NAVTEQ: *XML Specification for NAVTEQ Discover Cities Pedestrian, Project NAPA*. 2011

[22] SANDERS, P. ; SCHULTES, D.: Highway hierarchies hasten exact shortest path queries. In: *Algorithms–Esa 2005* (2005), S. 568–579

[23] VENESS, Chris: *Calculate distance and bearing between two Latitude/Longitude points using Haversine formula in JavaScript*. http://www.movable-type.co.uk/scripts/latlong.html