Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 00150

Offline Reiseplaner für Bahnverbindungen

Patrik Schäfer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Prof. Dr. Stefan Funke
Beginn am:	5. Juni 2014
Beendet am:	5. Dezember 2014
CR-Nummer:	F.2.2

Kurzfassung

Steht einem Benutzer eines Android-Gerätes keine Internetverbindung zur Verfügung, ist es ihm meistens versagt eine Reiseplanung für eine bevorstehende Zugreise durchzuführen, da die Elektronische Fahrplanauskunft beinahe aller Verkehrsbetriebe ausschließlich über eine Online-Anwendung funktioniert. In dieser Arbeit wurde eine App programmiert, die direkt auf dem Android-Gerät die Wegberechnung offline durchführt. Das Routing in öffentlichen Verkehrsnetzen ist auf einem Android-Gerät eine besondere Herausforderung, da nur begrenzte Ressourcen zur Verfügung stehen. Als Routing-Algorithmen wurden dazu der RAPTOR-Algorithmus, sowie das Routing mithilfe von Transfer-Patterns implementiert und getestet. Als Datensatz dient ein reduzierter Datensatz des Netzes der Deutschen Bahn. Anschließend wurden die Laufzeiten der Algorithmen in der App evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Heutzutage ist es für viele eine Selbstverständlichkeit, sich mobil mithilfe eines Smartphones Informationen beschaffen zu können. Mit dem Smartphone kann ermittelt werden, welches Essen heute in der Cafeteria angeboten wird oder lassen sich die neusten Nachrichten aus aller Welt anzeigen. Ein häufiges Anwendungsgebiet ist auch die elektronische Fahrplanauskunft der Deutschen Bahn oder des lokalen Verkehrsverbundes. Für diese elektronische Fahrplanauskunft gibt es meist sogar eine eigene App. Dort kann dann bequem der Start- und Zielbahnhof sowie die Ab- bzw. Ankunftszeit eingegeben werden und in Sekundenschnelle werden mehrere Vorschläge von Verbindungen für die gewünschte Reise angezeigt. Dabei werden meistens sogar aktuelle Informationen, wie Verspätungen oder Zugausfälle, miteinbezogen. Es macht durchaus Sinn, dass diese Apps eine Internetverbindung benötigen, da der Fahrplan auf diese Weise immer aktuell ist, auf aktuelle Ereignisse eingegangen werden kann und die Routing-Anfrage von einem rechenstarken Server sehr schnell beantwortet werden kann. Es gibt jedoch Situationen, in denen kein Internetzugang möglich ist. Für diesen Fall gibt es momentan kaum Möglichkeiten, um eine Reiseplanung mittels des Smartphones durchzuführen.

Dazu kommt, dass mit einer Reiseanfrage an eine elektronische Fahrplanauskunft Daten über eine geplante Reise des Benutzers an die Fahrplanauskunft übermittelt werden, die der Benutzer möglicherweise nicht preisgeben will. Es kann also interessant sein, ein Routing in öffentlichen Verkehrsnetzen direkt auf dem Smartphone durchzuführen. Dazu müssen die Fahrplandaten auf dem Smartphone gespeichert sein und es muss Routing-Algorithmen geben, die auf den begrenzten Ressourcen des Smartphones schnell die gewünschte Route aus diesen Fahrplandaten berechnen können. In dieser Arbeit wurde eine Offline Reiseplaner-App für Android entwickelt, die offline, also direkt auf dem Gerät, die beste Verbindung berechnet und dem Benutzer anzeigt. Diese App erlaubt es dem Benutzer, analog zu ihren Online-Pendants, den Start- und Zielbahnhof, sowie die Abfahrtszeit einzugeben. Als Routing-Algorithmen wurden eine Implementierung des RAPTOR-Algorithmus sowie eine Implementierung des Routings mit Transfer-Patterns getestet und evaluiert.

Android wurde vor allem deshalb als Zielplattform gewählt, weil es das Betriebssystem ist, mit dem mehr als drei viertel aller verkauften Smartphones ausgeliefert werden [Bei]. Die App benutzt als Fahrplandatensatz einen reduzierten Datensatz des Netzes der Deutschen Bahn im HAFAS Rohdatenformat.

Dieses Datenformat wird in dieser Arbeit, nach Erläuterung einiger Grundlagen, vorgestellt. Im Anschluss daran wird beschrieben, wie diese Daten weiterverarbeitet wurden, um sie zu komprimieren und kompakter darzustellen. Daraufhin werden die Funktionsweisen und Implementierungen der zwei verwendeten Algorithmen geschildert. Um sich ein Bild von der entstandenen App machen zu können, wird in der Folge die implementierte App mit Beispielscreens vorgestellt. Nachdem die App fertiggestellt war, wurden die erwähnten Algorithmen ausgiebig getestet. Die Ergebnisse dieser Tests

werden im folgenden Kapitel präsentiert. Am Ende folgt eine Zusammenfassung, die die wesentlichen Ergebnisse dieser Arbeit zusammenfasst.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Grundlegende Definitionen als Grundlage für die folgenden Kapitel

Kapitel 3 – Fahrplandaten: Beschreibung des HAFAS-Rohdatenformats und der Weiterverarbeitung des selbigen

Kapitel 4 – Routing-Algorithmen: Beschreibung und Funktionsweise der verwendeten Routing-Algorithmen

Kapitel 5 – Vorstellung der App: Vorstellung der implementierten Android-App mit Screenshots

Kapitel 6 – Evaluation: Evaluierung der implementierten Routing-Algorithmen mit einem Testgerät

Kapitel 7 – Related Work: An dieser Stelle wird ein anderes Projekt vorgestellt, dass Speichereffiziente Routing-Algorithmen implementiert

Kapitel 8 – Zusammenfassung und Ausblick: Die Arbeit wird nochmals zusammengefasst und ein Ausblick gegeben.

2 Grundlagen

Klassischerweise werden die Fahrverläufe eines öffentlichen Verkehrsmittels als Fahrplan dargestellt.

Definition 2.0.1
$\tau_{arr}(t, p)$ *beschreibt den Zeitpunkt an dem der Trip t an der Station p ankommt.*
$\tau_{dep}(t, p)$ *beschreibt den Zeitpunkt an dem der Trip t an der Station p abfährt.*
$\tau_{ch}(p)$ *beschreibt die Zeit, die an der Station p mindestens zum Umsteigen zwischen unterschiedlichen Zügen benötigt wird.*
$l(p1, p2)$ *beschreibt die Zeit, die für den Fußweg von p1 nach p2 benötigt wird.*

Ein Fahrplan ist formal ein 5-Tupel $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, hierbei ist $\Pi \subset \mathbb{N}$ die Menge der möglichen Zeitpunkte (Tag mit Uhrzeit als Element einer abzählbaren Menge), $\mathcal{S}$ die Menge der Haltestellen, $\mathcal{T}$ die Menge der Trips, $\mathcal{R}$ die Menge der Routen und $\mathcal{F}$ die Menge der Fußwege. Ein Trip, auch als Fahrt bezeichnet, ist eine Sequenz von Haltestellen, an denen aus- oder zugestiegen werden kann (Startbahnhof bis Endbahnhof). Mit Ausnahme der ersten Station besitzen alle Stationen innerhalb eines Trips eine Ankunftszeit. Analog besitzen alle Stationen, außer der letzten eine Abfahrtszeit. Es gilt $\tau_{arr}(t, p) \leq \tau_{dep}(t, p)$. Jede Route in $\mathcal{R}$ besteht aus Trips, welche die selbe Reihenfolge von Haltestellen gemeinsam haben. Falls zwischen Haltestellen zu Fuß gegangen werden kann, gibt es einen Fußweg in $\mathcal{F}$, der angibt, wie lange man von einer Station p_1 zu einer erreichbaren Station p_2 braucht.

Beispiel:

Trip Die S1, die am Mittwoch um 17:17 Uhr in Herrenberg abfährt und um 18:38 in Kirchheim(Teck) (mit Halt in Nufringen, Gärtringen, ...) ankommt, ist ein Trip.

Route Die Menge aller Fahrten der S1 von Herrenberg nach Kirchheim(Teck) mit entsprechenden Zwischenhalten ist eine Route.

Als Ergebnis eines Reiseplanungsalgorithmus wird eine Menge von Reisen $\mathcal{J}$ ausgegeben. Eine Reise ist eine Abfolge von Trips und Fußwegen. Eine Reise, die k Trips enthält, hat exakt k-1 Umstiege (Fußwege werden in diesem Ansatz nicht als zusätzlicher Umstieg gewertet). Eine Reise J_1 dominiert eine Reise J_2, wenn J_1 unter jedem Aspekt besser oder gleich gut ist wie J_2. Eine Menge, sich nicht gegenseitig dominierender Reisen ist eine Pareto-Menge. Ein Aspekt, der immer betrachtet wird, ist die früheste Ankunftszeit am Zielbahnhof. Darüber hinaus können noch weitere Aspekte, wie Anzahl der Umstiege, die Länge der Wartezeiten oder die Länge der Fußwege betrachtet werden.

Definition 2.0.2 (Pareto-Menge)
Die Pareto-Menge ist die Teilmenge einer Menge A von n-Tupeln, für die gilt:
Ein n-Tupel $x = (x_1, x_2, ..., x_n)$ ist in der Pareto-Menge, wenn es in A kein anderes n-Tupel gibt, dass in allen Parametern mindestens gleich gut und in einem Parameter besser ist. Das heißt insbesondere, dass in der Pareto-Menge für jeden Parameter mindestens ein Tupel enthalten ist, das das Optimum dieses Parameters enthält.

Im einfachsten Falle enthält die Pareto-Menge nur "1-Tupel". Dies entspricht bei einer Routenberechnung der Optimierung nach nur einem Kriterium: Der Ankunftszeit am Zielbahnhof.

Das heißt für einen gegebenen Startbahnhof p_s, einen Zielbahnhof p_t und eine Uhrzeit τ, soll eine Reise zurückgegeben werden, die beim Startbahnhof frühestens zur Uhrzeit τ losfährt und möglichst früh beim Zielbahnhof p_t ankommt. Diese Anfrage kann als $A@\tau \rightarrow B$ notiert werden.

Üblicherweise ist jedoch folgendes gewünscht: Wenn es mehrere schnellste Reisen von p_s nach p_t gibt, die zur gleichen Uhrzeit p_t erreichen, jedoch mit unterschiedlich vielen Umstiegen, sollte als Ergebnis des Algorithmus diejenige dieser Reisen mit den wenigsten Umstiegen zurückgegeben werden. Des Weiteren könnte für den Bahnfahrer auch eine Verbindung interessant sein, bei der er, unabhängig von der benötigten Zeit, am wenigsten umsteigen oder am kürzesten an Zwischenbahnhöfen warten muss.

Je nach Kriterium ist man daher an der gesamten oder einem Teil der Pareto-Menge der Reisen interessiert.

3 Fahrplandaten

Fahrplandaten des deutschen öffentlichen Verkehrsnetzes sind von beinahe keinem Verkehrsverbund in Deutschland erhältlich. Das liegt unter anderem daran, dass auch interne Informationen wie Leerfahrten im HAFAS gespeichert sind, die nicht öffentlich zugänglich gemacht werden sollen [Kau14]. In anderen Ländern ist man weniger zurückhaltend, so veröffentlichen z.B. die Schweizer Bundesbahn und die französische SNCF ihren Fahrplan regelmäßig. Eine gute Möglichkeit seine Fahrpläne zu veröffentlichen, stellt die, von Google entwickelte, General Transit Feed Specification (GTFS), die vor allem für den Endbenutzer gedacht ist und keine internen Informationen enthält, dar. Um diese Arbeit für Deutschland durchführen zu können, wurde mir ein Fahrplan der Deutschen Bahn im HAFAS-Rohdatenformat vom Zeitraum 15.01.2014 - 22.01.2014 zur Verfügung gestellt. Der Datensatz enthält ca. 8000 Haltestellen und ca. 50000 Trips.

3.1 HAFAS Rohdatenformat

Das HaCon Fahrplan-Auskunfts-System (HAFAS) ist eine Fahrplansauskunftsoftware der Firma Hannover Consulting (HaCon). Das System wird von der Deutschen Bahn, der Schweizer Bundesbahn, der Österreichischen Bundesbahn und vielen anderen europäischen Eisenbahngesellschaften für die Fahrplanauskunft benutzt. So basiert z.B. die Reiseauskunft auf bahn.de auf der HAFAS Software.

Um Fahrplanaustausch zu ermöglichen, hat HaCon das HAFAS Rohdatenformat entwickelt. Das Format besteht aus Dateien, die zwingend notwendig sind und Dateien, die optional sind.

Zwingend notwendig sind das Haltestellenverzeichnis BAHNHOF, die Koordinaten der Haltestellen BFKOORD, die Fahrplandaten FPLAN, die Gültigkeitsperiode der Fahrplandaten ECKDATEN, Informationen über die Verkehrstage der Fahrten BITFIELD, die Angabe zur Art der Verkehrsmittel ZUGART, die Fußwege zwischen Haltestellen METABHF sowie die Haltestellen bezogenen Umsteigezeiten UMSTEIGB.

Optional können Dateien wie z.B. die Angaben zu Gleisen GLEISE, Zusätzliche Attribute für einzelne Laufwegabschnitte ATTRIBUT oder Liniendefinitionen LINIE beigefügt sein.

Eine Zeile wird durch das Format zeichengenau festgelegt. Für die Datei BAHNHOF werden die Zeichenangaben beispielhaft angegeben, bei den anderen wird darauf verzichtet.

BAHNHOF
Zeichen 1-7 der Zeile geben die Nummer der Haltestelle an, Zeichen 9-11 den Verkehrsverbund und Zeichen 13-62 den Haltestellennamen. Im Haltestellennamensfeld können optional mehrere Namen, getrennt durch $ Zeichen, stehen, die auch mit einem Sprachenkürzel in eckigen Klammern gekennzeichnet werden können.

Beispiel:

```
8000096 VVS Stuttgart Hbf$S$Stoccarda
8000098 VRR Essen Hbf$E
8000099 VRR Essen-Steele
8000100 AAV Euskirchen$EU-
8000101 VGF Eutingen im Gäu
8000102 VGW Finnentrop
8000103 SHT Flensburg$FL$Flensborg
8000104 NVV Frankenberg(Eder)$FKB
8000105 RMV Frankfurt(Main)Hbf$Francfort (Main)
```

BFKOORD
Jede Zeile beginnt mit der Nummer der Haltestelle, gefolgt vom Längengrad und Breitengrad der Haltestelle. Zur besseren Lesbarkeit kann danach noch der Haltestellenname dazugeschrieben werden.

Beispiel:

```
8000096 9.181635 48.784084 Stuttgart Hbf
8000098 7.014793 51.451355 Essen Hbf
8000099 7.076106 51.450504 Essen-Steele
8000100 6.792193 50.657760 Euskirchen
8000101 8.782236 48.479952 Eutingen im Gäu
8000102 7.964701 51.172635 Finnentrop
8000103 9.436884 54.774092 Flensburg
8000104 8.789181 51.054565 Frankenberg(Eder)
8000105 8.663789 50.107145 Frankfurt(Main)Hbf
```

FPLAN
Die FPLAN Datei ist das Herzstück des HAFAS Datenformates. Hier sind die einzelnen Trips gespeichert. Sie enthält Zeilen, welche die Fahrt kategorisieren, z.B. mit Fahrnummern, Verkehrstagen oder Fahrtkategorie. Diese Zeilen beginnen mit einem *.

Zeilen, die nicht mit einem * beginnen sind Datenzeilen, die den Laufweg der Fahrt beschreiben.

Jede Fahrt muss zwingend folgende Zeilen beinhalten:

- Eine *Z, *KW- oder *T-Zeile, die die Fahrtnummer angibt und den Beginn einer neuen Fahrt kennzeichnet,
- eine *G-Zeile, die das Verkehrsmittel angibt und
- eine *A VE-Zeile, welche die Verkehrstage, an der die Fahrt verkehrt, festlegt

Beispiel:

```
*Z 21206 02____    01                                  %
*G RB  8002553 8000271                                 %
*A VE 8002553 8004819 000124                           %
*A VE 8004819 8000271 000000                           %
*A G  8002553 8000271                                  %
8002553 Hamburg-Altona              00702              %
8004819 Pinneberg             00714 00715              %
8004888 Prisdorf              00718 00718              %
8005887 Tornesch              00721 00722              %
8000092 Elmshorn              00728 00729              %
8003003 Horst(Holst)          00734 00734              %
8001387 Dauenhof              00739 00739              %
8006572 Wrist                 00746 00747              %
8001190 Brokstedt             00753 00753              %
8000271 Neumünster            00804                    %
```

Die *Z, *KW- oder *T-Zeile:

*Z bezeichnet normale Züge, *T bezeichnet Züge bei denen die Taktung, jedoch nicht die genaue Abfahrtszeit bekannt ist und *KW bezeichnet Kurswagen. Da in dem mir vorliegenden Datensatz keine *T-Züge vorkamen und auf Kurswagen nicht gesondert eingegangen wurde, wird an dieser Stelle nur die *Z-Zeile vorgestellt.

Die HAFCON-Spezifikation 5.4, die es seit dem 30.08.2010 gibt, spezifiziert eine *Z-Zeile folgendermaßen:

Zeichen	Bedeutung
1-2	*Z
4-9	6-stellige Fahrtnummer
11-16	6-stellige Verwaltungsnummer
18-22	leer
24-26	Taktanzahl in Minuten
28-30	Zeit zwischen einzelner Takte in Minuten

Dabei ist die Taktanzahl und die Taktzeit optional. Wenn diese Felder frei sind, fährt der Zug nur einmal zur beschrieben Zeit.

In dem mir vorliegenden Datensatz einer älteren HAFCON-Spezifikation sind die *Z-Zeilen jedoch etwas anders. Die Fahrtnummer ist nur 5-stellig und der Bereich zwischen Verwaltungsnummer und Taktanzahl ist nicht leer.

Die *G-Zeile enthält die 3-stellige Art des Zuges, über die in der Datei ZUGART weitere Informationen gefunden werden können.

Die *A VE-Zeile enthält die Verkehrstagenummer mit der in der Datei BITFELD nachgeschaut werden kann, an welchen Verkehrstagen diese Fahrt verkehrt. Wenn die Verkehrstagenummer nicht angegeben oder 000000 ist, verkehrt dieser Zug täglich.

Die anderen *A-Zeilen, die nicht mit *A VE beginnen, sind wie die Zuggattung *G, Richtungsinformationen *R, Linieninformationen *L oder andere *-Zeilen optional und geben weitere Informationen über diese Fahrt an.

Die Laufwegzeilen sind folgendermaßen gestaltet:

Zeichen	Bedeutung
1-7	Haltestellennummer
9-29	der Haltestellenname (optional)
30-35	die Ankunftszeit an der Haltestelle im Format [V]HHHMM
37-42	die Abfahrtszeit an der Haltestelle im Format [V]HHHMM

Bei dem [V]HHHMM steht V für das Vorzeichen. Eine negativer Wert bedeutet, dass der Eintrag zwar für Informationszwecke enthalten ist, jedoch nicht für Ein- oder Ausstieg zugelassen ist. Es gibt 3 Zeichen für die Stundenanzahl, da, wenn die Fahrt über Mitternacht hinausgeht, die Stundenanzahl mit 25 Uhr, etc. weiter gezählt wird. Die Abfahrtshaltestelle hat logischerweise keinen Eintrag in dem Ankunftszeitfeld und die Endstation keinen Eintrag im Abfahrtsfeld.

Die Datei ECKDATEN enthält nur das Datum, ab dem der Fahrplan gültig ist, sowie das Datum, ab dem Fahrplan nicht mehr gültig ist.

Die Datei BITFIELD ordnet jeder 6-stelligen Verkehrstagenummer einen 96 Zeichen langes, aus hexadezimalen Ziffern, bestehendes Bitfeld zu. Jedes Bit steht für einen Tag innerhalb des gültigen Fahrplanzeitraums. Ist das Bit 1, dann findet diese Fahrt an dem Tag statt, ist das Bit 0, dann findet die Fahrt an diesem Tag nicht statt. Jeweils 4 Bit werden zu einer hexadezimalen Ziffer zusammengefasst.

In der Datei ZUGART werden den verschiedenen Zuggattungen Kategorien zugeordnet (siehe Tabelle 3.1 auf der nächsten Seite)

Des Weiteren sind dort auch Angaben zur Tarifgruppe und Zuschlägen der einzelnen Zuggattungen aufgeführt.

In der Datei METABHF sind Fußübergänge zwischen Stationen gespeichert. Diese Datei war bei dem mir zur Verfügung gestellten Datensatz leider nicht enthalten und wurde manuell berechnet. Wie diese Fußwege dann in den Datensatz integriert wurden, wird im nächsten Kapitel beschrieben.

Als zusätzliche Datei in dem Datensatz befand sich noch UMSTEIGB, in der für jeden Bahnhof eine allgemeine Umsteigezeit und eine Umsteigezeit zwischen IC-Zügen angegeben ist.

Außerdem umfasst der Datensatz noch ATTRIBUT (zusätzliche Informationen wie z.B. die Nummer des Ruftaxis an einer Station) und GLEISE, in der die An- bzw. Abfahrtgleise der einzelnen Fahrten an den Stationen notiert sind.

Klasse	Bedeutung
00	ICE-Züge
01	Intercity- und Eurocityzüge
02	Interregio- und Schnellzüge
03	Nahverkehr, sonstige Züge
04	S-Bahnen
05	Busse
06	Schiffe
07	U-Bahn
08	Straßenbahn
09	Anrufpflichtige Verkehre
14	nur Direktverbindungen
15	nur Züge mit Schlafwagen*
16	nur Züge mit Liegewagen*
17	nur Züge mit Fahrradbeförderung

Tabelle 3.1: Verschiedene Klassen von Verkehrsmitteln

3.2 Weiterverarbeitung

Der erste Schritt bei der Weiterverarbeitung war, die 24 FPLAN Dateien, die der Datensatz enthielt zu einer einer großen FPLAN Datei zusammenzufassen, in der alle Züge enthalten sind. Zusätzlich wurden die Fahrten ausgefiltert, die nicht berücksichtigt werden sollten, die also nicht der Kategorie 0-4 angehörten.

Der mir zur Verfügung gestellte Datensatz hat eine Gültigkeit von 15.01.2014 - 22.01.2014. Die Fahrpläne der Deutschen Bahn sind immer circa 1 Jahr gültig (in diesem Jahr vom 15. Dezember 2013 bis 13. Dezember 2014). In diesem Jahr fahren die Züge relativ gleichmäßig und es gibt keine großen Unterschiede zwischen einzelnen Wochen. Das heißt, der Fahrplan von dieser einen Woche ist auch für andere Wochen relevant. Aus diesem Grund habe ich mich entschieden, die Woche von 15.01.2014 - 21.01.2014 zu extrapolieren und auf alle folgenden Wochen zu übertragen. Das heißt wenn eine Fahrt in dieser Woche montags verkehrte, dann wird daraus geschlossen, dass sie jeden Montag des Jahres verkehrt.

Die FPLAN-Datei wurde aufgrund dieser Annahme im nächsten Schritt so weiterverarbeitet, dass für jede Fahrt in der *A VE-Zeile statt der 6-stelligen Verkehrstagenummer ein 7 stelliges Bitfeld eingefügt wurde. Das erste Bit gibt an, ob die Fahrt montags verkehrt (also 1 für JA und 0 für NEIN), das Zweite, ob die Fahrt dienstags verkehrt, usw.

Bei den Fahrten der Kategorien 0-4 wurde im vorliegenden Datensatz von der Möglichkeit, diese mit Taktanzahl und Taktminuten anzugeben, kein Gebrauch gemacht. Jede Fahrt beschreibt also genau eine Fahrt zu einer Uhrzeit. Viele S-Bahn Linien, aber auch andere Züge, verkehren mehrmals am Tag auf der gleichen Route. Es ist leicht ersichtlich, dass man diese Redundanz ausnutzen kann um die Datei zu verkleinern. Die Datei soll dazu genutzt werden um auf einem Android-Smartphone

ein Routing durchzuführen. Dateizugriffe verbrauchen mehr Zeit als Rechenoperationen, deswegen erhofft man sich durch eine kleinere Datei ein schnelleres Routing.

Folgendermaßen wurde vorgegangen: Zuerst wurden die Fahrten der Anzahl der Haltestellen nach in absteigender Reihenfolge sortiert. Die längste Fahrt F_{muster} wird mit allen nachfolgenden Fahrten F_{test}, die durch die Sortierung ja kürzer oder gleich groß sind, zu einer Route zusammengefasst, wenn sie eine Teilmenge von der längsten Fahrt sind.

Die Teilmenge ist in diesem Fall so definiert:

1. Die Reihenfolge der Stationen der Fahrt F_{test} (auch wenn F_{test} kürzer ist) muss der Reihenfolge der Stationen in der Fahrt F_{muster} gleichen
2. Die Startstation der Fahrt F_{test} darf nicht vor der Startstation der Fahrt F_{muster} liegen
3. Die Endstation der Fahrt F_{test} darf nicht hinter der Endstation der Fahrt F_{muster} liegen
4. Die Differenzen zwischen Ankunft- und Abfahrtszeit in jeder Station müssen identisch sein. Ebenso müssen die Differenzen zwischen Abfahrtzeit in Station i und Ankunftszeit in Station j identisch sein.

Beispiel:
Angenommen F_{muster}, F_{test1}, F_{test2}, F_{test3}, F_{test4} seien Fahrten und A, B, C, D, E seien Stationen und die Laufwege sind folgendermaßen gegeben:

Fahrt	Reihenfolge
F_{muster}	A B C D E
F_{test1}	C D E
F_{test2}	Z A B
F_{test3}	C D E F
F_{test4}	A B D E

F_{test1} genügt den Kriterien 1,2 und 3. Wenn die Differenzen zwischen den Ankunfts- und Abfahrtszeiten denen von F_{muster} entsprechen, kann F_{test1} zur selben Route hinzugefügt werden.
F_{test2} genügt nicht dem Kriterium 2.
F_{test2} genügt nicht dem Kriterium 3.
F_{test4} genügt nicht dem Kriterium 1.

Die *Z- und die *A VE-Zeile wurden neu gestaltet: Diese aus mehreren Fahrten bestehende Route bekommt eine 5-stellige Routen-ID zugewiesen. Die *Z Zeile besteht jetzt nur aus *Z und der Routen-ID. Falls man die Möglichkeit, verschiedene Verkehrsmittel beim Routing auszulassen, nutzen möchte, könnte man danach noch die Kategorie der Route notieren.

Die *A VE-Zeile sieht jetzt folgendermaßen aus:

Zeichen	Bedeutung
1-2	*A
3	Anzahl der in Betracht gezogenen Tage
5-7	# Nummer der Station ab der diese Verbindung gilt
9-11	# Nummer der Station bis zu der diese Verbindung gilt
13ff	Liste der Startzeiten der Fahrt bei der Station, ab der diese Verbindung gilt, in Minuten des Tages, mit Leerzeichen getrennt, in aufsteigender Reihenfolge

Anzahl der in Betracht gezogenen Tage: Bei der Suche nach der nächst-möglichen Fahrt darf nicht nur die Fahrt in Betracht gezogen werden, die an demselben Tag startet. Es kann durchaus sein, dass eine Fahrt am Vortag beginnt und über Mitternacht in den nächsten Tag fährt und diese die nächst-mögliche Fahrt ist. Ob das zutrifft, muss bei der Suche überprüft werden. Mit dem Hintergedanken sich diese Überprüfung sparen zu können, falls keiner der Züge in dieser Liste über Mitternacht fährt, und die Suche nach der nächst-möglichen Fahrt zu vereinfachen, wurde hier markiert, ob ein Zug der Liste über Mitternacht fährt. Wenn es in der Liste der Startzeiten keine Fahrt gibt, die über Mitternacht hinaus fährt, steht an dieser Stelle eine 0. Falls es mindestens eine Fahrt gibt, die über Mitternacht fährt, steht hier eine 1. Es gibt in diesem Datensatz sogar eine Fahrt (nach Moskau), die länger als 24 Stunden andauert und dabei die Mitternachtsgrenze zweimal überschreitet. In diesem Fall steht an dieser Stelle eine 2. Diese Markierung ist nicht unbedingt notwendig und bringt auch keinen Performance-Vorteil, vereinfacht aber den Algorithmus im häufigen 0 Fall.

Um die Datei möglichst klein zu bekommen, wurden alle *-Zeilen, außer die *Z und die *A VE Zeile im neuen Format nicht mehr berücksichtigt, da diese keine Relevanz für das Routing besitzen. Um diese Informationen nicht zu verlieren, kann man die Routen-ID mit den zugehörigen alten Zugnummern in einer separaten Datei speichern und sie so nach dem Routing aufrufen.

Die An- und Abfahrtszeiten der einzelnen Stationen werden in Minuten nach Abfahrt bei der Start-station angegeben. So ist die Abfahrtszeit bei der ersten Station immer 0000 Minuten. Diese neu entsandene Datei hat den Namen FAHRPLANKOMPAKT.

Beispiel für eine Route:

```
*Z 07271
*A0 1111110 #00 #02 362
*A0 1111111 #02 #10 370 490
*A0 0000011 #00 #02 482
*A0 1111111 #00 #10 542 602 662 722 782 842 1202
8004332      0000
8000540 0002 0003
8000204 0007 0008
8006266 0014 0014
8002076 0022 0027
8006085 0032 0033
8000269 0045 0048
8001929 0050 0051
8002612 0056 0056
8002794 0064 0065
8000284 0080
```

Aus den Dateien BAHNHOF, BFKOORD und UMSTEIGB wurde eine neue Datei STATIONS geformt. Diese vereinigt den Namen mit den Koordinaten und der Umsteigezeit dieses Bahnhofs. Dabei wurde der Einfachheit halber für jede Station nur ein Name gespeichert.

Zeichen	Bedeutung
1-7	Station-ID
8-17	Längengrad
19-28	Breitengrad
29-30	Umsteigzeit zwischen ICs
32-23	Umsteigzeit allgemein
25ff	Name der Station

Beispiel:

```
8000096 9.181635 48.784084 7 7 Stuttgart Hbf
8000098 7.014793 51.451355 7 7 Essen Hbf
8000099 7.076106 51.450504 4 4 Essen-Steele
8000100 6.792193 50.657760 4 4 Euskirchen
8000101 8.782236 48.479952 3 3 Eutingen im Gäu
8000102 7.964701 51.172635 6 6 Finnentrop
8000103 9.436884 54.774092 5 5 Flensburg
8000104 8.789181 51.054565 3 3 Frankenberg(Eder)
8000105 8.663789 50.107145 8 8 Frankfurt(Main)Hbf
```

Nun wurden noch die Fußwege zwischen Stationen berechnet und der Datei zugefügt. Es wurde ein Fußweg von der Station p_1 zur Station p_2 eingefügt, wenn die beiden Stationen nicht weiter als 500 Meter voneinander entfernt sind. Die Entfernungen wurden mit der Semiversus-Formel berechnet. Für 100m Weg wurde eine Minute Fußweg angenommen. Da manche Stationen, wie z.B. Berlin Hbf

und Berlin Hbf (tief) die exakt selben Koordinaten haben und Fußwege keine Dauer von 0 Minuten haben sollen, wurde sich dazu entschieden den Fußweg als die Summe von der berechneten Dauer des Fußwegs und der Umsteigezeit des Zielbahnhof anzugeben. Beim Routing wird dann darauf geachtet, dass bei der Suche nach der nächsten Abfahrt keine Umsteigezeit mehr auf die Ankunftzeit aufgeschlagen wird, falls die Station mit einem Fußweg erreicht wurde.

Die Fußwege wurden der Station mithilfe von Fußwegzeilen nach der Station, bei der losgelaufen werden kann, hinzugefügt.

Zeichen	Bedeutung
1-2	*W
4-11	Station-ID der Zielstation
13ff	Dauer des Fußwegs

Beispiel:

```
8011160 13.369545 52.525592 8 8 Berlin Hbf
*W 8070952 8
*W 8089021 8
*W 8098160 8
```

Um bei dem Transfer-Pattern Algorithmus schnell die benötigten Routen finden zu können, wurden noch zusätzlich Routenzeilen hinzugefügt. Für jede Route auf der die Haltestelle liegt, gibt es eine Routenzeile mit der Routen-ID der Route und der Position der Station auf der Route.

Zeichen	Bedeutung
1-2	*Z
3-10	Routen-ID der Route
12-14	Position der Station auf der Route

Damit sieht ein Eintrag für eine Station beispielsweise so aus:

```
0710635 9.498559 51.330472 5 5 Hauptfriedhof, Kassel
*Z 0001068 24
*Z 0001069 03
*Z 0001070 03
*Z 0001655 03
*Z 0001656 20
*Z 0001657 03
*Z 0001658 03
*Z 0001659 03
*Z 0001660 03
*W 0710636 5
*W 0711550 5
```

4 Routing-Algorithmen

Die meisten Routing-Algorithmen für öffentliche Verkehrsnetze sehen das Routing in diesen als Graph-Problem an und lösen dieses mit einer Variante des Dijkstra Algorithmus (klassischer (bi-direktionaler) Dijkstra, A*-Algorithmus, Contraction Hierarchies). Um dieses Problem zu lösen, muss aus den gegebenen Routen, Fahrten und Haltestellen ein Graph konstruiert werden. Dafür gibt es mehrere Möglichkeiten:

a Der zeit-expandierte Graph: Jede Haltestelle besitzt für jede Ankunft und Abfahrt eines Zuges an dieser Station einen eigenen Knoten. Der Vorteil hierbei ist, dass jede Kante nach der Konstruktion ein festes Gewicht hat und der Algorithmus auf diesem Graphen ohne weitere Modifikation laufen kann. Der Nachteil ist, dass der entstandene Graph sehr groß ist.
b Der zeit-abhängige Graph: Im *route model* erhält jede Haltestelle für jede Route, die dort hält, einen eigenen Knoten und dazu noch einen Transferknoten. Im *station model* besitzt jede Haltestelle nur einen einzigen Knoten. Die Kosten der Kanten werden zeitabhängig berechnet.

Grundsätzlich ist eine gute Heuristik für den Aufwand des Suchalgorithmus selbst: Je weniger Knoten pro Station im Graphen enthalten sind, desto komplizierter wird der Suchalgorithmus, um Umstiege zwischen Zügen korrekt zu berücksichtigen.

Da der Arbeitsspeicher einer Android-App sehr begrenzt ist, können eventuell für große Verkehrsnetze keine großen Graphen (wie der zeit-expandierte Graph) in diesem gehalten werden, beziehungsweise können nicht alle Fahrten im Arbeitsspeicher gehalten werden, um die Kantenkosten bei einem zeit-abhängigen Graphen schnell zu berechnen. Da bei einem Dijkstra-basierten Algorithmus, die als nächstes zu berechnende Kante auf einer Route liegen kann, die nicht im Arbeitsspeicher ist, kann es dazu kommen, dass viele kostenintensive (langsame) Zugriffe auf sekundären Speicher nötig sind.

Aus diesem Grund wurde in dieser Arbeit der, von Daniel Delling et al. vorgestellte, Round-bAsed Public Transit Optimized Routing (RAPTOR)-Algorithmus verwendet [DPW12]. Dieser berechnet die Pareto-optimalen Reisen nicht auf einem Graphen, sondern rundenbasiert. In jeder Runde i werden alle Reisen berechnet, die genau i Umstiege brauchen um bei einer Haltestelle anzukommen und die schnellste davon gespeichert. Der Vorteil hierbei ist, dass in jeder Runde jede Route nur einmal betrachtet werden muss und man die Routen also sequentiell durchgehen kann und man so die Speicherzugriffe potentiell weniger sind als bei einer Graph-Lösung.

Ein anderer Ansatz, der verwendet wurde, ist das von Hanna Bast et al. vorgestellte Routing mithilfe von Transfer-Patterns [BCE+10]. Hier wird durch Speichern bester Verbindungen erreicht, dass nur auf sehr wenige Routen zugegriffen werden muss und dadurch nicht so viele Speicherzugriffe benötigt werden.

4.1 RAPTOR

4.1.1 Funktionsweise der Basisversion des Algorithmus

Die Invariante für jede Runde i ist, dass alle Verbindungen mit i-1 Umstiegen korrekt berechnet wurden. Für jede Station p sind die frühesten Ankunftszeiten nach 0 bis i-1 Umsteigen als $(\tau_0, \tau_1, ..., \tau_{i-1})$ gespeichert. Bevor die Runde beginnt wird bei allen Stationen $\tau_i = \tau_{i-1}$ als obere Grenze gesetzt. Falls die Station mit k Umsteigen nicht erreicht werden kann, ist τ_k = inf. Anschließend werden alle Routen nacheinander betrachtet. Die Route r besteht aus den Haltestellen $p_1, p_2, ..., p_k$. Die Stationen werden der Reihenfolge nach durchgegangen, bis man eine Station findet, deren τ_{i-1} nicht unendlich ist. Jetzt sucht man die früheste Fahrt der Route, die $\tau_{dep}(t, p_l) >= \tau_{i-1}(p_l) + \tau_{ch}(p_l)$ erfüllt. Also die erste Fahrt der Route, die mit einbezogener Umsteigezeit erreicht werden kann. In der ersten Runde muss keine Umsteigezeit berücksichtigt werden (In dieser Arbeit wird auch keine Umsteigezeit aufgeschlagen, wenn die letzte Verbindung von $\tau_{i-1}(p_l)$ ein Fußweg war, da in den Fußwegen die Zeit, das Gleis am Bahnhof zu finden bereits einkalkuliert ist). Wir nennen diese Fahrt $et(r, p_l)$.

Diese Station wird als Abfahrtsstation zwischengespeichert. Nun traversieren wir diese Fahrt weiter und wenn die Ankunftszeit dieses Trips an einer Station kleiner ist als das τ_i der Station, wird τ_i auf diese Ankunftszeit gesetzt. Als nächstes wird überprüft, ob $\tau_{i-1} + \tau_{ch} < \tau_i$ ist. Ist dies der Fall, wird überprüft, ob es eine frühere Fahrt der Route gibt als die momentan traversierte. Gibt es eine frühere, wird ab jetzt diese traversiert und die Abfahrtsstation ist die aktuelle Station.

Wenn alle Routen traversiert wurden, werden im nächsten Schritt alle Fußwege durchgegangen. Wenn es einen Fußweg zwischen p_k und p_l gibt, dann wird, falls $\tau_i(pk) + l(p_k, p_l) < \tau_i(pl)$ ist, $\tau_i(p_l)$ auf $\tau_i(p_k) + l(p_k, p_l)$ gesetzt. i wird iteriert und es beginnt eine neue Runde. i wird solange iteriert bis alle $\tau_i(p_k) = \tau_{i-1}(p_k)$ sind, also in der Runde durch keine zusätzliche Fahrt eine frühere Ankunftzeit erreicht wurde.

Hierdurch wurde für alle Stationen die vollständige Pareto-Menge berechnet.

4.1.2 Verbesserungen/Vereinfachungen

Es ergibt keinen Sinn, an einer Station in einen Trip einzusteigen, wenn bei diesem $\tau_{i-1} = \tau_{i-2}$ gilt. Diese Fahrt wurde schon in der Runde davor traversiert. Um diese auszulassen und sich auch den ersten Schritt mit dem Kopieren der τ_{i-1} zu τ_i zu sparen, werden in den Stationen nur die τ_k gespeichert, für die es keine τ_l gibt, mit $\tau_k = \tau_l$ und l < k. Stationen, für die in einer Runde ein besseres τ^* gefunden wurde, werden markiert. Beim weiteren Traversieren einer Fahrt wird immer mit dem τ, mit der frühesten Ankunftszeit (das nach Definition auch die meisten Umsteige haben muss) verglichen. Dieses τ wird als τ^* definiert. In der nächsten Runde werden nur diejenigen Routen betrachtet, auf denen eine Station liegt, die in der vorherigen Runde markiert wurden. Dazu werden die markierten Stationen am Anfang der Runde durchgegangen und alle Routen, auf denen diese liegen, zu einer Menge Q von Routen hinzugefügt. Es werden in dieser Runde nur Routen aus der Menge Q betrachtet. Um den Algorithmus zusätzlich zu beschleunigen, kann noch „target pruning“ eingesetzt werden. Das heißt, es werden keine Routen markiert, deren Ankunftszeit über τ^* der Zielstation liegt.

4.1.3 Der implementierte Algorithmus in Pseudocode

procedure RAPTOR(p_s, p_t, τ_{start})
  $\tau_0(p_s) \leftarrow \tau_{start}$
  $i \leftarrow 0$
  markiere p_s
  while change = true **do**
    change $\leftarrow$ false
    Q $\leftarrow$ leere Menge von Routen
    for all markierte Stationen p **do**
      Füge alle Routen zu Q hinzu, auf denen p liegt
    end for
    Lösche Markierungen
    for all Routen r in Q **do**
      for Stationen p_j in r, j in 1..k **do**
        if nicht eingestiegen **then**
          if $\tau^*(p_j)$!= inf and $\tau^*(p_j) = \tau_{i-1}(p_j)$ **then**
            eingestiegen $\leftarrow$ true
            abfahrtbahnhof $\leftarrow p_j$
            trip t $\leftarrow et(r, p_j)$
          end if
        else
          if $arr(t, p_j) < min(\tau^*(p_j), \tau^*(p_t))$ **then**
            $\tau^*(p_j) \leftarrow arr(t, p_j)$
            change $\leftarrow$ true
            markiere p_j
          else
            if $\exists$ trip t_2 mit $et(t2, p_j) < et(t, p_j)$ **then**
              $t \leftarrow t_2$
              abfahrtbahnhof $\leftarrow p_j$
            end if
          end if
        end if
      end for
    end for
    for all Fußwege (p_k, p_l) in $\mathcal{F}$ **do**
      if $\tau^*(p_k) + l(p_k, p_l) < \tau^*(p_l)$ **then**
        $\tau^*(p_l) \leftarrow \tau^*(p_k) + l(p_k, p_l)$
      end if
    end for$i \leftarrow i + 1$
  end while
end procedure

4.1.4 Konkrete Umsetzung

Die Datei FAHRPLANKOMPAKT wird zeilenweise gelesen. Wie in Abschnitt 3.2 beschrieben, ist die Datei so aufgebaut, dass eine Route aus Informationszeilen, die mit einem * gekennzeichnet sind, und Laufwegzeilen besteht. Es wird zuerst die Bezeichnerzeile *Z gelesen. Falls die Routen-ID nicht in Q ist, werden alle folgenden Zeilen dieser Route nicht berücksichtigt. Ist die Routen-ID in Q, werden die Informationen aus den *A-Zeilen in bestehende Arrays gespeichert. Das sequentielle Auslesen der Laufwegzeilen entspricht dem Traversieren der Route. Die Stationen sind in einer HashMap gespeichert und können so sehr schnell mit ihrer Station-ID als Schlüssel gefunden werden. Jede Station speichert ihre Pareto-Menge der τ_i als Liste, die nach den Ankunftszeiten sortiert ist. Zusätzlich zu den τ_i sind in dieser Liste die dazugehörigen Abfahrtsbahnhöfe dieser Verbindung, sowie die Routen-ID mit Abfahrtszeit, die eine Fahrt zusammen eindeutig definieren, gespeichert. Mit diesen Informationen ist ein schnelles Backtracking nach Beendigung des Algorithmus möglich.

4.2 Transfer-Patterns

Ein weiterer Ansatz um nicht so oft auf einzelne Routeninformationen zugreifen zu müssen und schnell routen zu können, sind Transfer-Patterns.

Die Idee dieses Ansatzes ist, dass unabhängig vom Wochentag oder der Uhrzeit der Reise von A nach B, immer eine von wenigen unterschiedlichen Verbindungen die beste ist.

Beispiel: Angenommen man möchte von Stuttgart nach Frankfurt fahren und egal zu welcher Uhrzeit und an welchem Abfahrtstag man fährt, die beste Verbindung ist immer SStuttgart - Frankfurtöder SStuttgart - Mannheim - Frankfurt". Diese besten Verbindungen nennt man Transfer-Pattern. Wenn man überlegt wie viele theoretische Wege es von Stuttgart nach Frankfurt geben könnte, dann sind zwei Transfer-Patterns im Vergleich dazu sehr wenig. Wenn man diese Information gespeichert hat, kann man sehr schnell routen. Man muss sich nur die Routen anschauen in der Stuttgart vor Frankfurt, Stuttgart vor Mannheim und Mannheim vor Frankfurt in der Reihenfolge der Haltestellen liegt. Wenn man die Fahrplaninformationen als Fahrplan gespeichert hat, kann man dann sehr schnell die Ankunft an den Haltestellen bestimmen. So lässt sich schnell die beste Verbindung herausfinden und so ist auch die grundsätzliche Funktionsweise des Algorithmus.

Allerdings müssen dazu die Transfer-Patterns für jedes Stationenpaar vorliegen. Dazu sind einige Vorberechnungen nötig.

4.2.1 Vorberechnungen

Der gängie Ansatz der dazu veröffentlichten Arbeiten um die Transfer-Patterns für eine Startstation A zu berechnen, ist ein mehrfach-Kriterien Dijkstra auf dem zeit-expandierten Graphen. Für alle erreichbaren Stationen B, wird der Arrival-Chain-Alorithmus angewendet um die besten Verbindungen von A nach B zu finden. Der Arrival-Chain-Algorithmus geht alle Ankunftsknoten der Station der Reihe nach, bei der frühesten beginnend, durch und sammelt so die besten Verbindungen.

In dieser Arbeit wurden die Transfer-Patterns jedoch anders berechnet, da der RAPTOR-Algorithmus bereits implementiert wurde:

Der Zeitpunkt t_{start} ist ein beliebiger Tag und eine beliebige Uhrzeit. t wird auf t_{start} gesetzt.

Beginne einen RAPTOR-Query (ohne target pruning) bei Station A mit dem Startzeitpunkt t. Der RAPTOR-Algorithmus berechnet alle optimalen Pareto-Mengen für alle Stationen B. Füge diese Verbindungen der Menge der Transfer-Pattern für die Verbindung von A nach B hinzu (oder nur die schnellste, falls man später nur an der schnellsten Verbindung mit den wenigsten Umstiegen interessiert ist). Hierbei wird die jeweils früheste Abfahrt gespeichert t_{edep} = min(alle Abfahrten von optimalen Verbindungen) von A. Setze t auf t_{et} + 1.

Dieser Vorgang wird wiederholt, bis $t = t_{start} + 1440min * 7$ gilt, da sich dann durch die Periodizität des Fahrplans die Verbindungen wiederholen. Danach sind alle Transfer-Patterns für die Station A berechnet.

4.2.2 Speicherung

Die Transfer-Patterns für die Station A werden in einem gerichteten azyklischen Graphen gespeichert. Es gibt dabei verschiedene Arten von Knoten: Den Quellknoten A, einen Zielknoten für jede erreichbare Station B und Präfix-Knoten.

Der Graph wird folgendermaßen erstellt:

procedure CREATEDAG(Transfer-Patterns für Startbahnhof A)
  Erstelle Quellknoten A
  for all Transfer-Pattern $A, C_1, ..., C_k, B$ **do**
    if Zielknoten B existiert nicht **then**
      Erstelle Zielknoten B
    end if
    Finde die maximale Kette an Präfixknoten in derselben Reihenfolge $C_1, ..., C_i$, $i \leq k$,
    die mit A verbunden ist // kann auch die Länge Null haben

    Erstelle die Präfixknoten $C_{i+1}, ..., C_k$
    Füge die Kante $e(B, C_k)$ hinzu
    for l in k .. i + 1 **do**
      füge Kante $e(C_l, C_{l-1})$ hinzu // die Kanten C_i bis C_1 sind bereits vorhanden
    end for
    if i = 0 **then**
      füge Kante $e(C_1, A)$ hinzu
    end if
  end for
end procedure

Abbildung 4.1 zeigt den fertigen Graphen eines Beispiels.

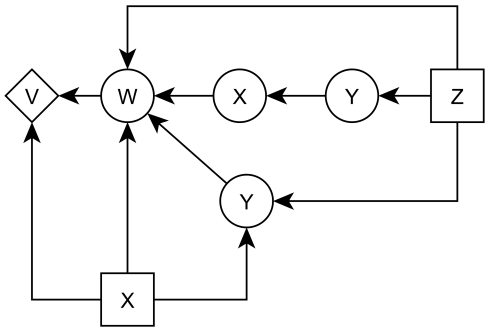
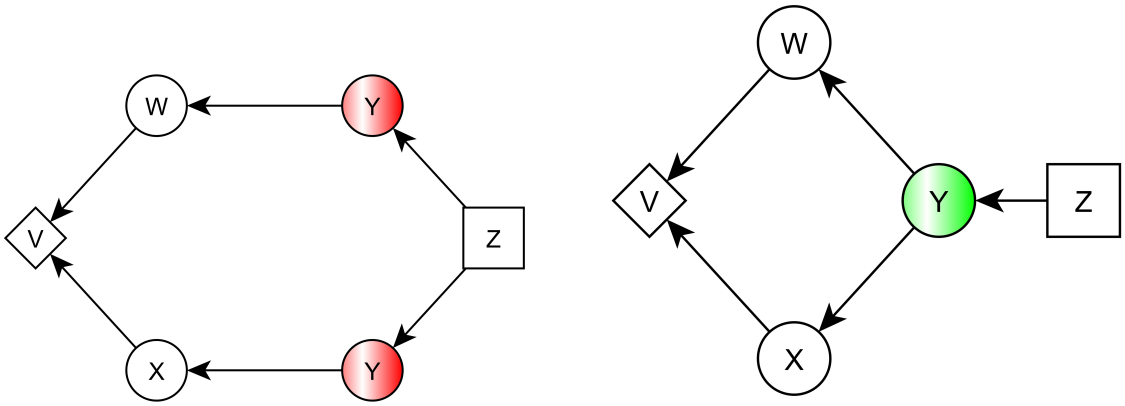


Abbildung 4.1: So sieht der azyklische, gerichtete Graph aus, wenn angenommen wird, dass für V → Z die Transfer-Patterns VWZ, VWYZ, VWXYZ und für V → X die Transfer-Patterns VX, VWX, VWYX berechnet wurden. Der Quellknoten ist die Raute, die Präfixknoten sind Kreise und die Zielknoten sind Rechtecke.

Das sieht auf den ersten Blick etwas seltsam aus, da die Kanten von den Zielstationen in Richtung der Startstation verlaufen, aber das ermöglicht uns später daraus einen Query-Graphen aufzubauen, der alle Transfer-Patterns für die die Verbindung A zu B enthält.

Man kann diesen Graph durch isomorphische Reduktion und gemeinsame Eingangspunkte noch weiter verkleinern. Isomorphische Reduktion wird in Abbildung 5.4 beschrieben. Mehr über kompakte Repräsentation in der Arbeit von Jonas Sternisko [Ste13].



(a) Vor isomorphischer Reduktion

(b) Nach isomorphischer Reduktion

Abbildung 4.2: Isomorphische Reduktion macht im Prinzip das gleiche für Suffixe, was die normale Speicherung für Präfixe macht: Gleiche Suffixe vor dem Zielknoten werden vereinigt.

4.2.3 Query-Graph und Query

Der Query-Graph besteht aus allen Knoten des Teilgraphen, welche beim Traversieren aller Wege vom Zielknoten B zum Quellknoten A besucht werden. Seine Kantenmenge besteht aus den inversen Kanten des Teilgraphen.

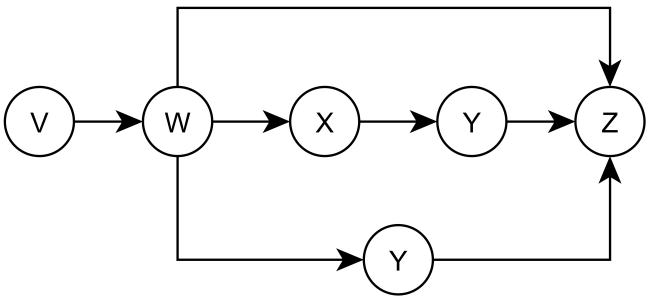


Abbildung 4.3: Der Query-Graph für die Anfrage V → Z aus dem Beispiel der Abbildung 4.1.

Bei der Query $A@\tau \rightarrow B$ werden die Ankunftszeiten bei B für alle Transfer-Patterns, die hier durch die unterschiedlichen Wege von A zu B repräsentiert werden, berechnet. Damit jede Kante nur einmal betrachtet werden muss, auch wenn isomorphische Reduktion angewendet wurde, funktioniert die Suche in dem Graphen folgendermaßen:

procedure QUERY
Q ist eine Menge von Kanten
füge Q alle ausgehenden Kanten von A hinzu

while die Anzahl der Kanten (k_1, k_2) in Q > 0 **do** // k_1 und k_2 sind Knoten des Query-Graphs
nehme die Kante e(k1, k2) aus Q bei der die Knoten-ID von k_2 am größten ist
for all Routen r von p_{k1} zu p_{k2} **do**
// p_{k1} ist die Station, die durch den Knoten k_1 repräsentiert wird
if $arr_r(p_{k2})$ ein $\tau(k_2)$ der Pareto-Menge von k_2 dominiert **then**
$arr_r(p_{k2})$ ersetzt die dominierten τ
end if
if k_2 ist nicht markiert **then**
füge die ausgehenden Kanten von k_2 zu Q hinzu
markiere k_2
end if
end for
end while
end procedure

Die in Betracht kommenden Routen von p_{k1} zu p_{k2} können sehr schnell herausgefunden werden, indem für jede Station alle Routen mit der dazugehörigen Position der Station auf dieser Route gespeichert werden. Um herauszufinden, welche Routen betrachtet werden müssen, wird überprüft,

welche Routen sich bei beiden Einträgen finden und die Postion von p_{k1} bei dieser Route kleiner ist als die Postion von p_{k2} bei der selben Route.

4.2.4 Hubs

Da die zu speichernden Datenmengen für viele Stationen sehr groß werden können, wird sich eine weitere Eigenschaft von öffentlichen Verkehrsmitteln zu Nutze gemacht. Üblicherweise fahren die meisten besten Verbindungen über gewisse Knotenpunkte, im Englischen Hub genannt. Wenn z.B. ein Reisender aus dem Großraum Stuttgart ein überregionales Ziel erreichen will, wird er in den meisten Fällen am Stuttgarter Hauptbahnhof in einen überregionalen Zug einsteigen. Stuttgart ist in diesem Beispiel ein Hub. Die Idee, die zur Verringerung der Speichermenge führen soll, ist, dass man nur die Transfer-Patterns für eine Nicht-Hub Haltestation speichert, die entweder zu einem Hub führen oder in denen kein Hub vorkommt. Sei A, C_1, ... , C_k, B ein Transfer-Pattern, dann wird es in dem Präfix-Graph gespeichert, falls B ein Hub ist oder kein C_i ein Hub ist. Falls es ein C_i gibt, dass ein Hub ist, dann wird nur die Hubstation C_i mit dem kleinsten i als Access-Station von A gespeichert.

Um die Hubs zu bestimmen, kann man einen vereinfachten, zeit-unabhängigen Graphen konstruieren. Jede Station wird durch einen Knoten repräsentiert und die Kanten entsprechen den Routenverläufen. Als Kosten einer Kante wird das Minimum von allen möglichen verschiedenen Kantenkosten, die diese Kante zu verschieden Zeitpunkten haben kann, festgelegt. Auf diesem Graph wird ein Dijkstra-Algorithmus mit limitierter Reichweite von einer Sammlung zufälliger Haltestellen ausgeführt. Die Stationen, die auf den meisten kürzesten Wegen liegen, werden als Hubs ausgewählt.

Die Vorberechnung in der gängigen Dijkstra-Variante wird durch die Hubs auch beschleunigt, da die Knoten, die von Hubs erreicht werden können, als inaktiv markiert werden können und so der Dijkstra-Algorithmus früher abgebrochen werden kann.

Der Query-Graph wird zusätzlich zu B auf die Access-Stationen erweitert. Dann wird der standardmäßige Query-Algorithmus ausgeführt, der in diesem Schritt gleichzeitig die besten Verbindungen zu B ausrechnet, die keinen Hub auf dem Weg besitzen (falls solche existieren), sowie die besten Verbindungen zu den Access-Stationen. Im nächsten Schritt werden dann die berechneten Ankunftszeitpunkte an den Access-Stationen dazu benutzt, die besten Verbindungen von den Access-Stationen ab diesem Ankunftszeitpunkt zu dem Zielbahnhof B zu berechnen. Da die Access-Stationen meist sehr schnell von A erreicht werden können, ist die Berechnung der Ankunftszeit bei diesen meist sehr schnell. Tests haben gezeigt, dass Querys mit Hubs in der Praxis nur minimal langsamer sind als Querys ohne Hubs.

4.2.5 Konkrete Umsetzung

Bei dieser Implementierung wurde dieser Graph in 2 Dateien gespeichert. In der einen Datei befindet sich die Adjazenzliste aller Knoten in topologischer Sortierung. In der Zeile Nr. 0 ist der Knoten mit der Knoten-ID 0. Als erstes steht dort die Station-ID des Bahnhofs, für den dieser Knoten steht. Danach folgt die Adjazenzliste mit den Knoten-IDs der Nachfolger (bei Zielknoten existieren mehrere Nachfolger und bei Präfix-Knoten nur ein Nachfolger). Kanten gehen nur von Knoten kleinerer ID zu Knoten größerer ID. Daraus folgt, dass der Quellknoten A immer die höchste ID hat und in der letzten

Zeile steht. (Graph-Datei) In der anderen Datei sind die Knoten-IDs der Zielknoten gespeichert. Der erste Eintrag in der Zeile ist die Station-ID, der zweite Eintrag ist die zugehörige Knoten-ID. Durch die topologische Sortierung ist es möglich, alle Wege vom Zielknoten B zum Quellknoten A zu finden, in dem man die Datei einmal zeilenweise liest. (Entry-Datei)

Da die Vorberechnungen der Transfer-Patterns eine Menge Zeit benötigen, wurde nur die Transfer-Patterns von einer Auswahl von 40 Stationen berechnet. Diese 40 Stationen sollten für Testzwecke ausreichen. Auf den Einsatz von Hubs wurde dabei verzichtet. Die Größe der entstanden Dateien liegt dabei für die Entry-Datei unter 100kb und zwischen 0,5 und 2MB für die Graph-Dateien.

Die Implementierung des Algorithmus geschieht so, wie man es naiv vermuten würde. Es wird zuerst die Knoten-ID des Zielknoten des Zielbahnhofs in der Entry-Datei gesucht. Im Anschluss wird die Graph-Datei zeilenweise gelesen und dadurch der Query-Graph aufgebaut. Durch die topologische Sortierung der Knoten stehen die Nachfolger eines Knoten immer weiter unten in der Datei als der Knoten selbst.
Ein Knoten ist als ein Objekt implementiert, das einen Verweis auf die repräsentierte Station des Knoten enthält und einen Verweis auf den Nachfolger-Knoten im Query-Graph. Dieser Verweis auf den Nachfolger-Knoten entspricht der Kante zwischen den Knoten. Zusätzlich wird in dem Knoten das Element der Pareto-Menge mit der geringsten Ankunftszeit und der dazugehörige Vorgängerknoten gespeichert, da im Anschluss nur die schnellste Verbindung mit den wenigsten Umstiegen zurückgegeben werden soll. Wenn alle Pareto-optimalen Verbindungen zurückgegeben werden sollten, müsste die gesamte Pareto-Menge gespeichert werden.
In diesem Graphen-Modell wird die in Abschnitt 4.2.3 beschriebene Query ausgeführt. Nach der Ausführung wird ein Backtracking vom Zielknoten durchgeführt und das Ergebnis ausgegeben. Die Routen werden in einer RandomAccessFile gehalten. Zu der FAHRPLANKOMAKT-Datei wurde ein Index für jede Route erstellt, der angibt bei welchem Byte der Datei welche Route beginnt. Mithilfe dieses Index können die benötigten Routen schnell aufgerufen werden.

5 Vorstellung der App

In diesem Kapitel wird die implementierte Android-App mit ihren Funktionen, benutzten Bibliotheken und ihrer Benutzerschnittstelle vorgestellt.

Die App sollte

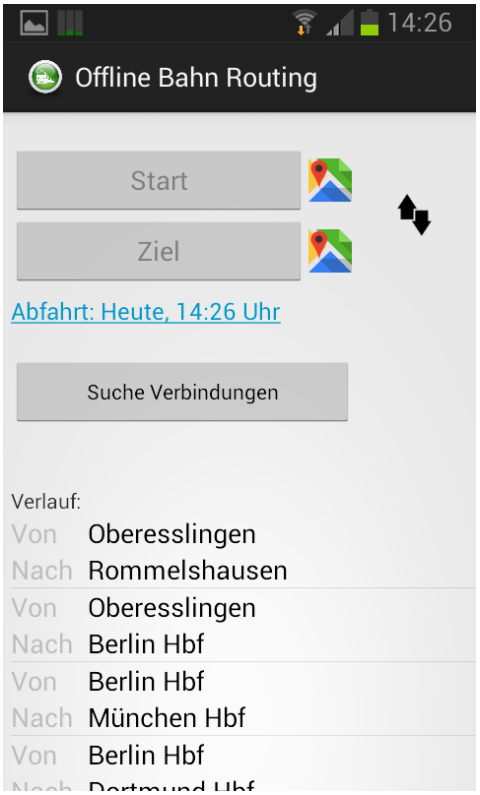
- eine komfortable Auswahl von Start-und Zielbahnhof sowie Abfahrtszeit erlauben
- die schnellste Reise von dem eingegebenen Startbahnhof zum eingegebenen Zielbahnhof zur Abfahrtzeit möglichst schnell berechnen
- die schnellste Reise als Reiseplan mit den einzelnen Verbindungen ausgeben
- die Möglichkeit bieten, die Reise als graphische Ansicht anzeigen zu lassen

5.0.6 App-Aufbau

Android-Apps basieren zumeist auf der Programmiersprache JAVA. Um Apps für Android entwickeln zu können, werden das JDK und das Android SDK, das die Android-Programmierschnittstellen enthält, benötigt. Als Entwicklungsumgebung wurde Eclipse gewählt. Eine Android-App besteht aus Activities, die sichtbare Benutzerschnittstellen repräsentieren und von denen nur eine zur gleichen Zeit aktiv sein kann. Datenaustausch zwischen den einzelnen Activities kann durch Bundles stattfinden, in denen Objekte, die keine Standarddatentypen sind, erst serialisiert oder in Parcels verpackt werden müssen, um dann in der neuen Activity wieder entpackt zu werden. Seit API Level 11 gibt es in Android das Konzept der Fragmente. Fragmente besitzen einen eigenen Lebenszyklus und eine eigene Benutzerschnittstelle, müssen jedoch in einer Acitvity existieren. Es können mehrere Fragmente gleichzeitig angezeigt werden und einzeln ausgetauscht werden. So kann die GUI ohne Activity-Wechsel dynamisch geändert werden und es müssen zum Datentausch keine Bundles erzeugt werden, da der Austausch über die gemeinsame Acitivity realisiert wird.

Die in dieser Arbeit erstellte App besteht aus einer Haupt-Activity und mehreren Fragmenten, die ausgetauscht werden. Beim Start der App wird zunächst ein Startbildschirm angezeigt und im Hintergrund werden die Haltestellen mit den zugehörigen Informationen aus der Datei STATIONS geladen. Diese Informationen werden von den unterschiedlichen Fragmenten benötigt und deswegen in der Activity gehalten, sodass alle Fragmente auf diese Informationen zugreifen können. Der Ladevorgang dauert etwa drei Sekunden und der Fortschritt wird, wie in Abbildung 5.1a zu sehen ist, in einem Ladebalken angezeigt.

(a) Start-Fragment



(b) Auswahl-Fragment

Abbildung 5.1

Nachdem das Laden beendet ist, wird zum Auswahl-Fragment, dessen Screen in Abbildung 5.1b zu sehen ist, gewechselt. Mit einem Klick auf den Start-Button oder Ziel-Button wird zu dem Textauswahl-Fragment, das in Abbildung 5.2 gezeigt wird, gewechselt. In diesem kann der Name des gesuchten Bahnhofs in ein EditText-Feld eingegeben werden. Sobald etwas eingetippt wurde, werden alle Haltestellen, die mit diesem Text beginnen, in einer Liste angezeigt. Mit Klick auf ein Element der Liste wird die Haltestelle als Start- oder Zielbahnhof (Je nachdem über welchen Button der Benutzer zu diesem Fragment gewechselt ist) eingeloggt und zu dem Auswahl-Fragment zurückgekehrt. In diesem wird daraufhin der Name der Haltestelle als Text des Start- bzw. Zielbutton angezeigt.

Der Start- bzw. Zielbahnhof kann alternativ auch über eine Karte ausgewählt werden. Die Möglichkeit dazu, bilden die Karten-Symbole neben dem Start- bzw. Zielbutton. Wird auf eines dieser Karten-Symbole geklickt, öffnet sich das Kartenauswahl-Fragment. Das Aussehen dieses Fragments kann in Abbildung 5.3a betrachtet werden. Das Kartenauswahl-Fragment besteht größtenteils aus einer Mapsforge-Karte (siehe Abschnitt 5.0.7). Die Haltestellen werden an ihren jeweilige Koordinaten als blaue Punkte auf der Karte angezeigt. Falls GPS auf dem Smartphone aktiviert ist, wird die Karte bei der momentanen Position geöffnet, sodass die nächstgelegenen Haltestellen schnell gefunden werden können. Klickt der Benutzer auf einen dieser Punkte, so wird dieser grün statt blau eingefärbt. Der Name der Station wird über der Karte eingeloggt und der Benutzer kann diesen Bahnhof mit Klick

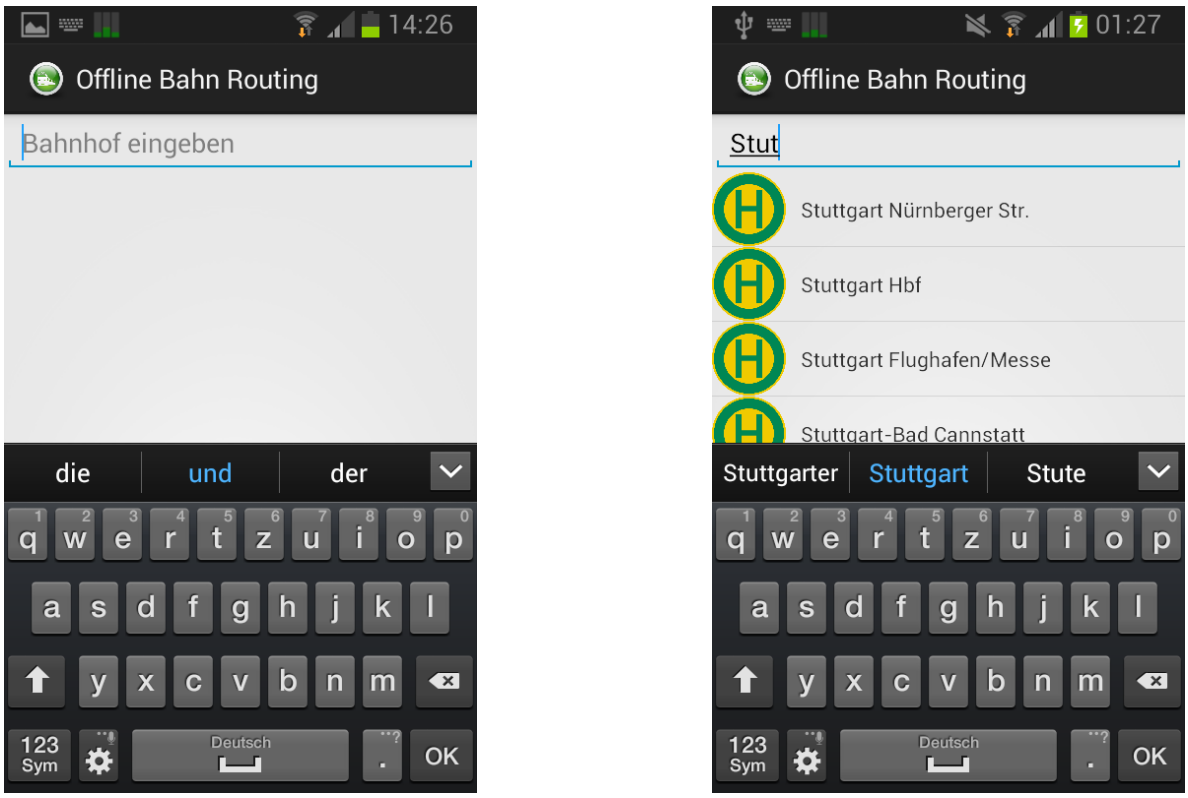


Abbildung 5.2: Das Textauswahl-Fragment

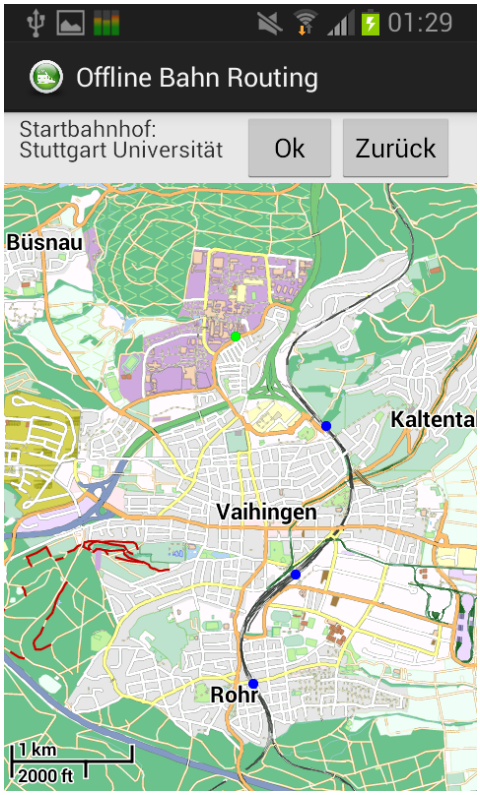
auf den „Ok"-Button bestätigen und zum Auswahl-Fragment zurückkehren. Darüber hinaus hat der Benutzer auch die Möglichkeit mit dem „Zurück"-Button ohne Änderung zu dem Auswahl-Fragment zurückkehren.

Rechts neben den Karten-Symbolen auf dem Auswahlbildschirm befindet sich noch ein Button mit dem der Startbahnhof bequem mit dem Zielbahnhof getauscht werden kann.

Die dritte Möglichkeit Start- und Zielbahnhof einzugeben, stellt der Anfragen-Verlauf dar. Der Verlauf ist im unteren Teil des Bildschirms zu finden. Hier werden die letzten zehn unterschiedlichen Reiseanfragen gespeichert. Mit Klick auf ein Listenelement der Verlaufs, werden der Start- und Zielbahnhof auf die Werte des Verlaufselements gesetzt. Der Verlauf steht auch nach Beendigung der App beim nächsten Start wieder zur Verfügung.

Falls der Nutzer die Abfahrtszeit, die als Text unter den Buttons für die Start- und Zielauswahl angezeigt wird, ändern möchte, kann er auf den Text klicken und es öffnet sich ein PopUp-Fenster, in welchem er den Abfahrtszeitpunkt mithilfe von Rädern ändern kann. Der Standard Wert der Abfahrtszeit ist der Startzeitpunkt der App. Ein Screenshot dieser Auswahl ist in Abbildung 5.3b sichtbar.

Mit Klick auf den Button „Suche Verbindungen" wird die Reiseanfrage gestartet und zum Reise-Fragment gewechselt. Dies ist nur dann möglich, wenn zuvor eine Start und eine Zielstation eingegeben



(a) Kartenauswahl-Fragment

(b) PopUp zur Abfahrtszeitauswahl

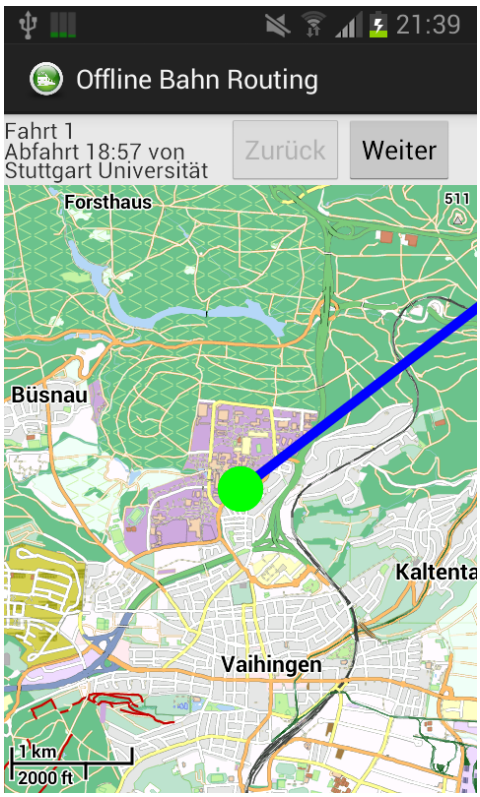
Abbildung 5.3

wurde. Es wird nun einer der in Kapitel 4 vorgestellten Algorithmen ausgeführt. Wenn die Berechnung der Reise abgeschlossen wurde, werden die einzelnen Verbindungen und Fußwege der Reise, beginnend mit der zeitlich ersten Verbindung, als Liste angezeigt (siehe Abbildung 5.4a). Möchte sich der Benutzer die Reise auf der Karte anzeigen lassen, kann er im rechten, oberen Eck auf das Kartensymbol klicken um eine Karte, auf der die Verläufe der einzelnen Verbindungen der Reise als Polygonzug eingezeichnet sind, zu öffnen. Wie auch in Abbildung 5.4b zu sehen ist, werden Stationen an denen um- oder ein- oder ausgestiegen wird, als grüne Kreise kenntlich gemacht. Mit dem Weiter-Button und dem Zurück-Button können die einzelnen Etappen der Reise durchgegangen werden.

5.0.7 Mapsforge

Mapsforge ist eine kostenlose, open-source Bibliothek, die ein schnelles Rendering von Kartendaten auf einem Android-Gerät ermöglicht [Map]. Das Projekt wurde 2008 von Thilo Mühlberg and Jürgen Broß von der Freien Universität Berlin ins Leben gerufen. Diese haben sich von dem Projekt als aktive Entwickler zurückgezogen, jedoch wird das Projekt von anderen Entwicklern weitergeführt. So wurde im Mai 2014 nach einer langen Auszeit die Version 0.4.0 veröffentlicht. Die in diesem Projekt verwendete Version 0.5.0 ist die aktuellste zu diesem Zeitpunkt.

(a) Reiseplan-Fragment



(b) Visualisierung der Reise

Abbildung 5.4

Mapsforge-Karten werden aus OpenStreetMap-Karten erstellt und werden auf dem Android-Gerät gespeichert, sodass kein Internetzugang zur Darstellung der Karten nötig ist. Mapsforge bietet dazu einige Funktionalitäten zum Anzeigen von Overlays auf der gerenderten Karte.

5.0.8 Joda Time

Da die JSR-310 date/time API, die in Java 8 SE integriert ist, nicht für Android zur Verfügung steht und die älteren Klassen java.util.Date and java.util.Calendar einige Design-Probleme besitzen, wurde zur Verwaltung von Zeitpunkten die JodaTime-Bibliothek verwendet [Jod]. JodaTime ist ein Open Source-Projekt und steht unter der Apache 2.0-Lizenz. Nach eigenen Angaben ist Performance der JodaTime-Klassen in nahezu jeder Hinsicht besser als die der java-Klassen Date und Calendar.

6 Evaluation

Der RAPTOR-Algorithmus und der Transfer-Pattern-Algorithmus wurden mit dem Datensatz, der in Kapitel 3 beschrieben wurde, getestet. Als Testgerät wurde das Smartphone Samsung Galaxy S4 mini benutzt.

6.1 RAPTOR

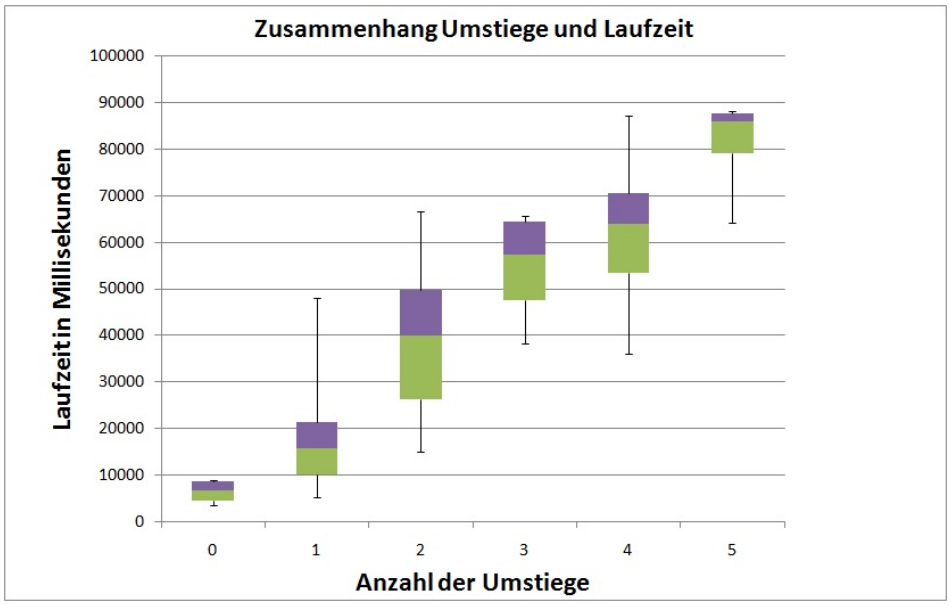
Getestet wurden Verbindungen von einem zufälligen Startbahnhof A zu einem zufälligen Zielbahnhof B zu einem zufälligen Zeitpunkt τ.

Die Ergebnisse, die in Abbildung 6.1 zu sehen sind, zeigen, dass die Laufzeit des Algorithmus mit zunehmender Fahrzeit, sowie mit zunehmender Zahl der benötigten Umstiege auf der optimalen Route zunimmt. Da „target pruning“ eingesetzt wurde, entspricht dieses Verhalten den Erwartungen. Die Ergebnisspanne reicht von ca. fünf Sekunden für Reisen ohne Umstiege bis zu ca. 85 Sekunden für Reisen, bei der die beste Verbindung fünf Umstiege benötigt. Die geringe Geschwindigkeit ist vermutlich dem zeilenweisen Abarbeiten der Datei geschuldet. Das Laden der Trips in den Arbeitsspeicher und das dadurch erreichte, mögliche dynamische Zugreifen auf markierte Routen, sollte den Algorithmus deutlich beschleunigen.

6.2 Transfer-Patterns

Getestet wurden Verbindungen von einem Startbahnhof A aus der Auswahl der Stationen, für die die Transfer-Patterns berechnet wurden, zu einem zufälligen Zielbahnhof B zu einem zufälligen Zeitpunkt τ.

In Abbildung 6.2 sind die Ergebnisse des Tests visualisiert. Man sieht, dass das Routing mit Transfer-Patterns in dieser Implementierung deutlich schneller ist, als der RAPTOR-Algorithmus. Es wird beinahe jede Anfrage unter 5 Sekunden beantwortet. Bei dem Transfer-Pattern-Algorithmus gibt es keinen direkten Zusammenhang zwischen der Zahl der Umstiege und der Laufzeit. Vielmehr spielt in diesem Fall die Größe des Query-Graphs eine entscheidende Rolle. Ist dieser groß, sind viele Kanten bzw. viele verschiedene Transfer-Patterns zu berechnen und die Laufzeit des Algorithmus erhöht sich.



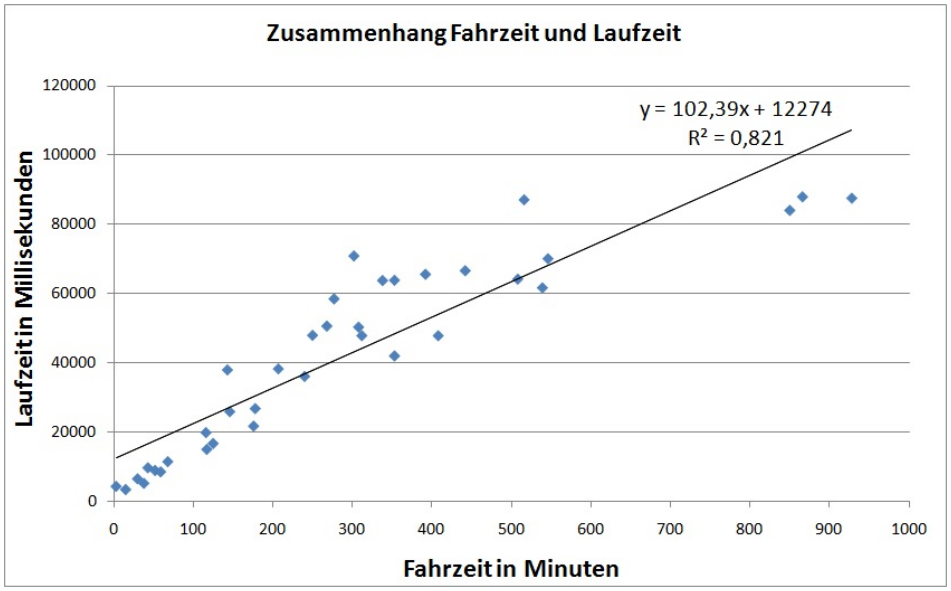


Abbildung 6.1: Ergebisse des RAPTOR-Tests

6.3 Vergleich zwischen RAPTOR und Tranfser-Patterns

Wie in den vorherigen Sektionen berichtet wurde, ist das Routing mit Transfer-Patterns in dieser Implementierung deutlich schneller. Jedoch hat der RAPTOR-Algorithmus auch gewisse Vorteile gegenüber dem Transfer-Pattern-Algorithmus:

- Es sind keine Vorberechnungen nötig

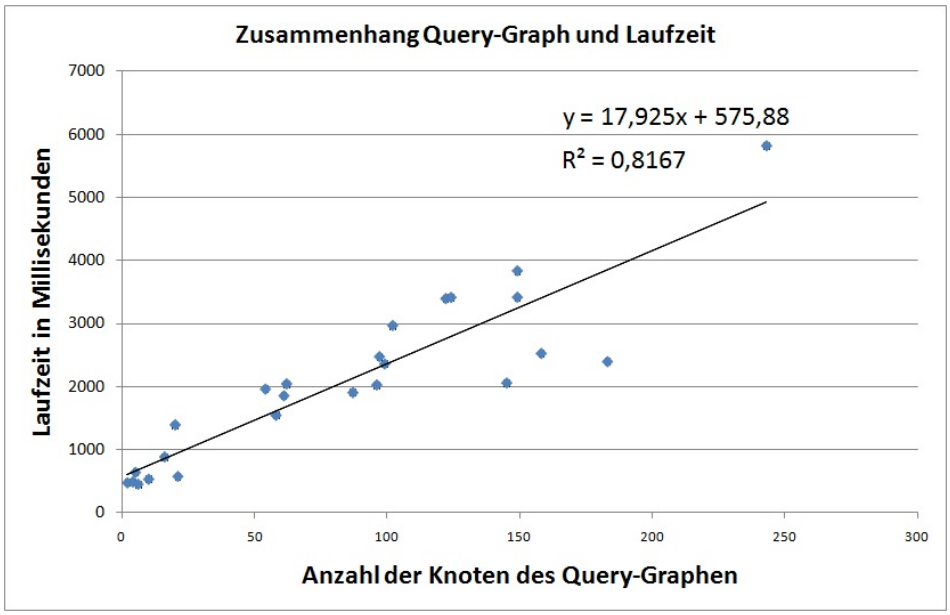


Abbildung 6.2: Ergebisse des Transfer-Pattern-Tests

- Der benötigter Speicher auf dem Telefonspeicher oder der SD-Karte ist deutlich geringer
- Er ist flexibler als die Transfer-Patterns. So kann z.B. der RAPTOR-Algorithmus, wenn als zusätzliches Query-Krierium „keine ICEs“ hinzugenommen wird, die ICE-Trips beim Routing vernachlässigen und liefert das gewünschte Ergebnis. Falls die Transfer-Pattern für diesen Fall nicht vorberechnet wurden, kann der Transfer-Pattern-Algorithmus für dieses Kriterium kein garantiert korrektes Ergebnis liefern. Werden die Transfer-Patterns für viele verschiedene Konstellationen von Kriterien gespeichert, können die Datenmengen für die Speicherung drastisch größer werden.

7 Related Work

An dieser Stelle möchte ich das GraphHopper-Projekt vorstellen, dass sich mit einer ähnlichen Problemstellung wie diese Arbeit beschäftigt: Speicher-effiziente Routing-Algorithmen in Java. Jedoch geht es bei dem Projekt um normales Routing in Straßengraphen und nicht um öffentliche Verkehrsmittel [Pro].

7.1 GraphHopper

Das von dem Entwickler Peter Karich gestartete Projekt GraphHopper steht unter Apache License und bietet einige Speichereffiziente Algorithmen, die, im Gegensatz zu den in dieser Arbeit vorgestellten Algorithmen, auf Graphen basieren. Die Graphdaten werden aus einer OpenStreetMap-Karte extrahiert, dabei werden standardmäßig Contraction Hierarchies berechnet (auch abstellbar). Als Routing-Algorithmen stehen dem Entwickler der uni- und bidirektionale Dijkstra-Algorithmus sowie der uni- und bidirektionale A*-Algorithmus zur Verfügung. Das Routing funktioniert auch unter Android, falls 32MB Arbeitsspeicher alloziert werden können. Es existiert auch eine GraphHopper Demo-App für Android. Mithilfe von MapsForge-Karten ist diese App auch offline vollständig funktionstüchtig. Strecken in Gebieten unter 500km^2 können unter Android in unter 5 Sekunden berechnet werden.

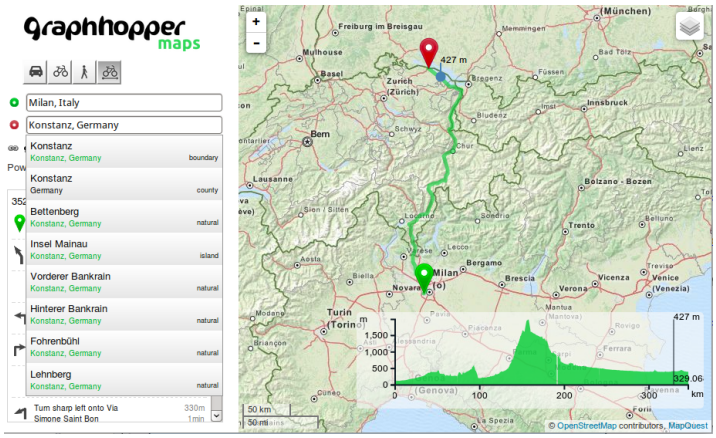


Abbildung 7.1: Die Desktop-Anwendung von Graphhopper (Abbildung von karussell.wordpress.com)

8 Zusammenfassung und Ausblick

In dieser Arbeit wurde eine App implementiert und vorgestellt, die eine offline Reiseplanung ermöglicht. Die Berechnung der besten Reise wird dabei direkt auf dem Android Gerät durchgeführt. Es wurden der RAPTOR-Algorithmus und das Routing mithilfe von Transfer-Patterns vorgestellt, implementiert, getestet und evaluiert. Dabei ist festzustellen, dass Routing in öffentlichen Verkehrsnetzen mit einer, für den Benutzer tolerierbaren, Rechenzeit auf einem Android Gerät grundsätzlich möglich ist. Bei der Evaluation wurde ermittelt, dass das Routing mit Transfer-Patterns in dieser Implementierung um einiges schneller ist als das Routing mit der Implementierung des RAPTOR-Algorithmus. Der RAPTOR-Algorithmus hat jedoch in anderen Bereichen Vorteile gegenüber der Transfer-Patterns-Variante. Wenn es zukünftigen Entwicklern gelingt, die Fahrplandaten komplett in den Arbeitsspeicher zu laden, kann die benötigte Rechenzeit des RAPTOR-Algorithmus wahrscheinlich um einiges verringert werden. Ein anderer Ansatz um die Rechenzeit zu verbessern, wäre eine Optimierung der benutzten Datentypen, um den benutzten Arbeitsspeicher der App zu minimieren. Solange sich die deutschen Verkehrsbetriebe nicht dazu entschließen ihre Fahrplandaten öffentlich zu machen oder selbst eine Offline-App entwickeln, werden wohl vorerst keine anderweitigen Offline Reiseplaner-Apps für Deutschland erscheinen. Für andere Länder wäre solch eine App allerdings durchaus denkbar. Im Allgemeinen ist es jedoch sinnvoll, dass die Anfragen online über den Server der Verkehrsbetriebe bearbeitet werden, da die Reisevorschlägen des Servers den Informationen, die eine Offline-App geben kann, durch ihre Aktualität überlegen sind.

Literaturverzeichnis

[BCE+10] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, F. Viger. Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns. In *Proceedings of the 18th Annual European Conference on Algorithms: Part I*, ESA'10, S. 290–301. Springer-Verlag, Berlin, Heidelberg, 2010. (Zitiert auf Seite 21)

[Bei] S. Beiersmann. Strategy Analytics: Android steigert Marktanteil auf fast 85 Prozent. http://www.zdnet.de/88200592/strategy-analytics-android-steigert-marktanteil-auf-fast-85-prozent/. (Zitiert auf Seite 7)

[DPW12] D. Delling, T. Pajor, R. F. Werneck. Round-Based Public Transit Routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. Society for Industrial and Applied Mathematics, 2012. (Zitiert auf Seite 21)

[Jod] JodaTime. JodaTime Project. http://www.joda.org/joda-time/. (Zitiert auf Seite 35)

[Kau14] S. Kaufmann. *Opening Public Transit Data in Germany*. Dissertation, University of Ulm, 2014. (Zitiert auf Seite 11)

[Map] Mapsforge. Mapsforge Project. http://mapsforge.org/. (Zitiert auf Seite 34)

[Pro] G. Project. GraphHopper. https://graphhopper.com/. (Zitiert auf Seite 41)

[Ste13] J. Sternisko. *On Compact Representation and Robustness of Transfer Patterns in Public Transportation Routing*. Master's thesis, University of Freiburg, Germany, 2013. (Zitiert auf Seite 26)

Alle URLs wurden zuletzt am 05. 12. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift