

Hund ist nicht Katze

Fachmodul ÖV Journey Planning

Christian Bühler Flavio Tobler

August 4, 2018

1 Abstract

Das Ziel dieser Bachelorarbeit war es, zu prüfen ob der «Connection Scan Algorithmus», kurz CSA, für die Wegfindung im Netzwerk der schweizerischen öffentlichen Verkehrsmittel geeignet ist. Zusätzlich soll der CSA für die Verwendung von Matrizen-Abfragen eingeschätzt werden. Dazu wurde das bestehende Programm «OpenTripPlanner» verwendet. Dieses basiert auf dem Dijkstra-Algorithmus. In dieser Arbeit wurde nun der konzeptionell bestehende CSA programmiertechnisch umgesetzt und der bestehende Algorithmus ersetzt. Anschliessend wurde dessen Performance geprüft.

Die Grundversion des CSA ist nicht für diese grossen Datenmengen geeignet, da sie zu Verarbeitungszeiten von mehr als 24h pro Anfrage führte. Um dies zu verbessern wurde eine Quad-Tree-Optimierung hinzugefügt. Diese führte zu einer starken Performance-Verbesserung.

Dies führt uns zu dem Schluss, dass der CSA

Der CSA ist nicht für Matrizenanfragen geeignet, da einer der grossen Geschwindigkeitsvorteile des Algorithmus daher rührt, dass er nur einen kleinen Teil der Verbindungen überhaupt behandeln muss, und die anderen Verbindungen ignoriert werden können. Eine Matrizenanfrage schränkt diesen Vorteil stark ein. Zusätzlich müsste die Struktur des CSA überarbeitet werden, da immer nur Zeiten zu einem bestimmten Zielpunkt gespeichert werden, welche im Falle einer Matrizen-Anfrage nicht differenzierbar sind.

Abstract in Englisch

Inhaltsverzeichnis

1	Abstract	I
2	Einleitung	2
3	Augabenstellung	3
4	Ausgangslage	4
4.1	JourneyPlanning	4
4.2	OpenTripPlanner	4
4.3	Connection Scan Algorithmus	4
4.4	General Transit Feed Specification	4
5	Methode	6
6	CSAfuertOTP	7
6.1	Datenstruktur	7
6.1.1	TimeTable	7
6.1.2	StopCSA	7
6.1.3	TripCSA	7
6.1.4	FootpathCSA	7
6.1.5	ConnectionCSA	8
6.1.6	Journey	8
6.1.7	JourneyPointer	8
6.1.8	LegCSA	8
6.2	Programmablauf	8
6.2.1	TimeTableBuilder	8
6.2.2	Server	8
6.2.3	Webseitenaufruf	8
6.2.4	CSA	9
6.3	Connection Scan Algorithm	9
6.3.1	Earliest Arrival Connection Scan	9
6.3.2	Profile Connection Scan	10
6.3.3	JourneyToTripPlanConverter	10
7	Produkt	12
7.1	OTP mit Schweizer Daten implementieren	12
7.2	Kann OTP ohne .osm file ausgeführt werden?	12
7.3	Modellierung der Datenstruktur	12
7.4	Automatische generierte Klassendiagramme	12
7.5	Dummy-GTFS Daten erstellen	12
7.6	Mocking	13
7.6.1	CSAMock	13
7.6.2	TimeTableBuilderMock	13

7.6.3	JourneyToTripPlanConverterMock	13
7.7	TimeTableBuilder	13
7.8	JourneyToTripPlanConverter	14
7.9	ConnectionScanAlgorithm	14
7.9.1	EAS	14
7.9.2	PCS	15
7.10	Performance-Test	16
8	Resultate	18
9	Fazit	19

2 Einleitung

Einleitung: Nutzen und Sinn des Projektes + Grobe Kapitelübersichtsbeschreibung der Arbeit

3 Aufgabenstellung

Unsere BA Aufgabenstellung

4 Ausgangslage

Um unsere Programm und unser vorgehen zu besser verstehen muessen wir zuerst das von uns verwendete Basisprogramm sowie den von uns verwendeten Basisalgorithms erlaeuern.

4.1 JourneyPlanning

Ein JourneyPlanner ist ein Programm, welches den optimalen Weg für eine Reise zwischen zwei oder mehr Orten findet. Im gegensatz zum RoutePlanning oder Routenplaner bezieht sich ein JourneyPlanner nur auf öffentliche Verkehrsmittel und nicht auf Privatfahrzeuge.

4.2 OpenTripPlanner

Der OpenTripPlanner, kurz OTP, ist eine auf der Maven-Repository aufbauende Multimodale trip planning Software welche anfangs für Städte ausgelegt war, nun aber auch in ersten landesweiten Netzwerken Anwendung findet. Er wurde von einem OpenSource-Kollektiv aus mehr als 100 Personen in acht Jahren entwickelt.

Der OTP basiert auf dem A*-Algorithmus und verwendet GTFS-Daten und OpenStreetMap Daten in Form einer pbf-Datei. In einem preprocessing Schritt wird der Graph für den Algorithmus erstellt. Dieser kann in einer Datei gespeichert werden oder direkt im RAM des Servers gelagert werden. Selbiges wird für die OpenStreetMap-Daten gemacht. Während dem Betrieb kann der Graph angepasst werden, so dass das Programm auf verspätete Züge reagieren kann.

Der OTP erzeugt für jeden Aufruf eine neue Instanz des Graphen. Somit können mehrere Aufrufe parallel behandelt werden.

OTP steht unter einer GNU Lesser General Public License.

!!! Selbstplagiat

4.3 Connection Scan Algorithmus

4.4 General Transit Feed Specification

General Transit Feed Specification (GTFS) ist ein von Google entwickeltes Dateiformat zum Austausch von Öffentlichen Verkehrsdaten sprich Fahrpläne. Die Daten werden von der Plattform¹ zur Verfügung gestellt. GTFS ist ein statisches Dateiformat und beinhaltet keine Echtzeitdaten, wie Verspätungen, Ausfälle etc. und wird deshalb auch GTFS Static genannt. Die Daten werden in verschiedenen Textfiles zur Verfügung gestellt, welche wiederum viele wichtige Informationen enthält.

¹<https://opentransportdata.swiss/>

Dateiname	pflicht?	Definition
agency.txt	ja	Geschäftsstellen die Daten zur Verfügung stellen
stops.txt	ja	Haltestellen mit ihrer Position
routes.txt	ja	Verkehrsverbindungen (Linien) mit den Fahrzeugarten
trips.txt	ja	Fahrten
stop_times.txt	ja	Zeiten in der Fahrzeuge Ankommen/Abfahren an Haltestellen
calendar.txt	ja	Fahrplanveränderungen (Jahreszeiten)
calendar_dates	optional	Ausnahmeplan für bestimmtes Datum
fare_attributes.txt	optional	Fahrpreise und die Art der Bezahlung
fare_rules.txt	optional	Fahrpreisregeln verschiedener Zonen
shapes.txt	optional	Beschreibt den Weg eines Fahrzeuges (Darstellung)
frequencies.txt	optional	Fahrpläne ohne fixe stop Zeiten.
transfers.txt	optional	Umsteigpunkte verschiedener Routen (Linien)
feed_info.txt	optional	Zusätzliche Informationen über den Datensatz

Daten die bisher nicht von der Plattform zur Verfügung gestellt werden: fare_attributes.txt, fare_rules.txt, frequencies.txt.

5 Methode

Evaluationen usw.

6 CSAfuerOTP

Dies ist eine Erläuterung des Endprogramms

6.1 Datenstruktur

Der CSA benötigt zwei Datenstrukturen. Einen TimeTable für die Eingabe von Daten und ein Journey für die Rückgabe von Daten. Da sich einige Klassennamen mit den Bezeichnungen des Dijkstra Algorithmuses überschneiden ist deren Namen mit einem "CSA" erweitert worden. (Sagen dass ersichtlich in Bild)

Datenstrukturen des TimeTables und des Journeys erläutern. Sagen dass CSA hinter namen. Auch sagen was und wieso diese containerklasse

6.1.1 TimeTable

Der TimeTable ist die vom ConnectionScanAlgorithmus als Eingabe benötigte Datenstruktur. Er ist ein Quadrupel aus Sets von StopCSA, TripCSA, FootpathCSA und ConnectionCSA. Das ConnectionCSA-Set ist ein LinkedHashSet, da sie für den Algorithmus anhand der Abfahrtszeit auf- oder absteigend sortiert werden muss. Die anderen Sets sind HashSets. Neben den "add" und "show" Funktionen für die Sets enthält die TimeTable-Klasse die Methode "getFootpathChange" welche für einen Stop die Umsteigzeit zurückgibt.

6.1.2 StopCSA

Ein Stop ist eine Haltestelle für öffentliche Verkehrsmittel. Ein Stop besitzt einen Namen, Längen- und Breitengrad sowie eine AgencyAndID-Nummer. Die Klasse besitzt neben den Gettern, Settern und Konstruktoren eine Methode um den Stop zu klonen.

6.1.3 TripCSA

Als Trip wird die Fahrt eines Öffentlichen-Verkehrsmittels von der Start-Station bis zur End-Station bezeichnet. Es ermöglicht den CSA ohne umsteigen erreichbare Orte zu erkennen. - Weiterschreiben

6.1.4 FootpathCSA

Ein Footpath kann zwei verschiedene Funktionen haben. Er besteht aus einem DepartureStop, einem ArrivalStop sowie einer Dauer. Wenn der DepartureStop und der ArrivalStop gleich sind repräsentiert der Footpath einen Umsteigeprozess. Wenn sie unterschiedlich sind repräsentiert er einen Laufweg zu einem Stop hin oder von einem Stop weg. Neben Gettern, Settern und Konstruktoren hat der Footpath keine weiteren Methoden.

6.1.5 ConnectionCSA

6.1.6 Journey

Ein Journey ist ein vom CSA berechneter Weg vom Start- zum Zielpunkt. Er besteht aus einem StartPath, welcher den Fussweg zur ersten Station hin darstellt, sowie eine Liste aus journeyPointern welche den Weg mit allen Umsteigestationen repräsentiert. Neben den Gettern und Settern gibt es eine Klonfunktion. Für die Liste der journeyPointer gibt es Add-Funktionen zum Einfügen am Anfang, am Ende oder an einem bestimmten Punkt anhand eines Indexes.

6.1.7 JourneyPointer

Ein JourneyPointer ist eine Hilfskonstruktion welche der CSA anlegt um die berechnete Abfolge von Stationen später wieder rekonstruieren zu können. Ein JourneyPointer besteht aus einem Leg sowie einem Footpath. Dabei handelt es sich beim Leg um eine Fahrt in einem ÖV vom einsteigen bis zum Aussteigen und beim Footpath um das darauffolgende Umsteigen oder das Erreichen des Ziels.

6.1.8 LegCSA

Ein Leg ist die Fahrt in einem öffentlichen Verkehrsmittel vom einsteigen bis zum aussteigen. Dies ermöglicht es den Journey nicht von Station zu Station, sondern von Umsteigen zu Umsteigen zu rekonstruieren. Ein Leg besteht aus einer EnterConnection und einer ExitConnection. Er besitzt neben den Gettern, Settern und Konstruktoren keine weiteren Methoden.

6.2 Programmablauf

TimetableBuilder zu Serverstart zu Webseitenaufruf zu CSA zu JourneyToTripPlanConverter

6.2.1 TimeTableBuilder

Erstellt einen Timetable aus GTFS-Daten

6.2.2 Server

Startet server welchen die Webseite für aufrufe zur Verfügung stellt

6.2.3 Webseitenaufruf

Wenn ein Aufruf von der Webseite eingeht so wird die plan Methode der Klasse PlannerResource aufgerufen. Diese erhält die Anfragenparameter in einem RoutingRequest-Objekt. Das im Preprocessing generierte TimeTable-Objekt wird als neue Instanz übergeben. Dies wird gemacht, da der OTP eine separate Instanz des Algorithmus für jeden

Aufruf verwendet. Der TimeTable sowie der Request werden dann der createJourneys-Methode des Algorithmuses übergeben. Dessen Rückgabe wird in einem Set von Journeys gespeichert.

6.2.4 CSA

6.3 Connection Scan Algorithm

Der Connection Scan Algorithm, kurz CSA, ist ein moderner Algorithmus zur Bearbeitung von Anfragen auf zeitplanbasierten Systemen. Er basiert, im Gegensatz zu den gängigen Algorithmen wie z.B. dem Dijkstra-Algorithmus, nicht auf einem gewichteten Graphen. Es gibt zwei Arten des CSA. Zum einen den EACS(EarliestArrivalConnectionScan) welcher die frühestmögliche Ankunftszeit und wenn benötigt auch noch den dazugehörigen Journey zurückliefert. Zum anderen den PCS(ProfileConnectionScan) welcher alle möglichen Journeys berechnet und den Besten Journey nach mehreren Kriterien sortieren kann. Beide sind darauf ausgelegt genau einen Journey zurückzugeben.

6.3.1 Earliest Arrival Connection Scan

Der Earliest Arrival Connection Scan Algorithmus, kurz EACS, arbeitet mit einer nach Abfahrtszeit sortierten Liste von Verbindungen. Über diese iteriert er dann aufsteigend wobei als Startpunkt die erste Verbindung, welche nach der im Request spezifizierten Abfahrtszeit abfährt.

Jede Verbindung wird auf drei Eigenschaften überprüft:

- Ist der Abfahrtsort der Startort?
- Wurde der Abfahrtsort schon von einer früheren Verbindung erreicht?
- Wurde das zur Verbindung gehörende Fahrzeug schon von einer früheren Verbindung benutzt?

Wenn eine dieser drei Bedingungen erfüllt ist so wird dies im Ankunftsort der Verbindung mit einem Zeiger auf den Ort, bei welchem man in das jeweilige ÖV eingestiegen ist, vermerkt. Der Ort bei welchem man eingestiegen wird mithilfe eines Trip-Bits gespeichert. Wenn eine ÖV zum ersten mal verwendet wird so wird der Startort gespeichert und das Trip-Bit wird für das ÖV gesetzt. Wird das ÖV erneut verwendet so ist das Trip-Bit bereits gesetzt und der Startort wird nicht überschreiben.

Sobald der Algorithmus eine Verbindung findet, welche eine der drei Bedingungen erfüllt und gleichzeitig der Ankunftsort dem Zielort entspricht, hat er einen Journey zum Ziel gefunden. Die Schleife wird unterbrochen und der Algorithmus baut sich vom Zielort aus mithilfe der Zeiger den kompletten Journey auf welcher dann als Antwort zurückgegeben wird.

Der ECSA existiert auch in einer schlankeren Version bei welcher keine Zeiger gespeichert werden. Dies führt dazu, dass der ECSA schneller ist, jedoch liefert er nur noch die Ankunftszeit ohne den Reiseweg zurück.

6.3.2 Profile Connection Scan

Der Profile Connection Scan Algorithmus, kurz PCS, arbeitet auch mit einer nach Abfahrtszeiten sortierten Liste von Verbindungen. Im Gegensatz zum EACS iteriert er absteigend über die Verbindungen. Er sucht also die Verbindung vom Zielpunkt aus. Jede Verbindung durchläuft dabei drei Prüfungen:

- Kommt man ans Ziel wenn man aussteigt?

Diese Bedingung überprüft ob der Ankunftsort der Verbindung der Zielort ist. Sollte dies der Fall sein so wird im Abfahrtsort der Verbindung die Ankunftszeit sowie die dazugehörige Verbindung gespeichert. Zusätzlich wird die Ankunftszeit für das jeweilige ÖV gespeichert.

- Kommt man ans Ziel wenn man umsteigt?

Es wird überprüft ob vom Ankunftsort der Verbindung schon ein Weg gefunden wurde, welcher zum Zielort führt. Dazu wird überprüft, ob im Ankunftsort eine Ankunftszeit gespeichert wurde. Ist dies der Fall so wurde schon ein möglicher Weg vom Ankunftsort zum Zielort gefunden. Dann werden die Informationen wie in der ersten Bedingung in Abfahrtsort und dem ÖV gespeichert.

- Kommt man ans Ziel wenn man sitzen bleibt?

Es wird überprüft, ob von diesem ÖV aus schon ein Weg zum Zielort gefunden wurde. Dazu wird überprüft ob für das ÖV schon eine Ankunftszeit gespeichert wurde. Ist dies der Fall so werden die Informationen wie in den ersten beiden Schritten im Abfahrtsort und dem ÖV gespeichert.

Die Suche ist abgeschlossen sobald die Abfahrtszeit der Verbindung früher als die im Request definierte Abfahrtszeit ist.

Nun wird vom Startpunkt aus jede gespeicherte Ankunftszeit überprüft. Dann wird von der zur Ankunftszeit gehörigen Verbindung der Ankunftsort genommen. Von diesem Ort aus werden wieder die gespeicherten Ankunftszeiten überprüft und die gleichen Schritte erneut durchgeführt. Dies wird so lange wiederholt bis der Ankunftsort der Zielort ist. Der gefundene Weg entspricht dann einem Journey zum Ziel. Der beste gefundene Journey kann dann als Response zurückgegeben werden.

6.3.3 JourneyToTripPlanConverter

Die vom Algorithmus gefundenen Journeys werden dann zusammen mit dem Request der generatePlan-Methode des JourneyToTripPlanConverter übergeben. Dieser wandelt die Journeys in ein TripPlan-Objekt um, welches von der Webapplikation als response erwartet wird. Die vom Request benötigten Informationen werden zu Beginn in das TripPlan-Objekt übertragen. Danach wird aus jedem Journey ein Itinerary-Objekt erzeugt. Dieses wird dann mithilfe der dem Journey zugehörigen JourneyPointer befüllt. Für die Legs sowie die Footpaths der JourneyPointer wird ein Leg generiert. Dies ist

jedoch ein Leg-Objekt und kein LegCSA-Objekt. Während des ersten durchlaufs wird zudem aus dem StartPath des Journeys ein Leg generiert. Dazu gibt es die beiden Methoden `legFromLeg()` und `legFromFootpath()`. Diese übertragen die benötigten Parameter und erstellen eine Geometrische-Form welche dem Fahrtweg folgen und für die Anzeige auf der Webseite benötigt werden. Wenn ein Leg aus einem Footpath generiert wird, wird zusätzlich überprüft ob dieser eine Distanz überwindet oder ob der Start- und Zielpunkt gleich sind. Dies dient dazu umsteigewege hinauszufiltern, welche von der Webseite nicht als Leg benötigt werden, jedoch trotzdem in die Zeit mit einfließen. Sobald alle Journey-Pointer abgearbeitet sind wird das Itinerary den TripPlan hinzugefügt. Nachdem die 3 besten Journeys konvertiert wurden wird der TripPlan an die Webseite zurückgegeben.

7 Produkt

7.1 OTP mit Schweizer Daten implementieren

Der originale OTP muss mit den Schweizer GTFS-Daten funktionieren, da er als Grundlage für den Algorithmus dient und einen Performancevergleich zwischen den beiden Algorithmen gemacht werden muss. Es gab zwei Probleme, welche dabei behandelt werden mussten.

Die SBB beschreibt in ihren GTFS-Daten einige Verbindungen mit dem Routentyp 1700 «Miscellaneous». Dieser Typ wird jedoch vom OTP nicht unterstützt da er nicht zugeordnet werden kann. Nach einer Überprüfung der SBB Daten stellte sich heraus, dass der Routentyp 1700 nur für einige Sessellifte sowie Autoverladestationen verwendet wird. Da diese beide nicht in unserem Scope liegen können sie ignoriert werden. Dazu fügten wir dem OTP einen Handler für den Routentyp 1700 hinzu welcher ihn als Auto definiert, da auch das Auto ausserhalb unseres Scopes liegt, jedoch vom OTP unterstützt wird.

Das zweite Problem war der benötigte Arbeitsspeicher. Für die Grafberechnung mit den SBB GTFS-Daten benötigt der OTP ca. 15 GB Arbeitsspeicher. Die von uns benötigten Rechner konnten diese Speichermenge jedoch nicht zur Verfügung stellen. Deshalb wird für die Berechnung mit den grossen Datensätzen ein Laborcomputer der HTW verwendet. Dieser entspricht mit seinen 32GB Arbeitsspeicher den Anforderungen.

7.2 Kann OTP ohne .osm file ausgeführt werden?

7.3 Modellierung der Datenstruktur

Java container klassen + UML Diagramm

7.4 Automatische generierte Klassendiagramme

7.5 Dummy-GTFS Daten erstellen

Das "DummyGTFS" wurde anhand der bestehenden Schweizer-GTFS Daten erstellt. Es wurde nur ein Teil der Liechtensteinischen Busverbindungen übernommen und selber erstellt um den Stil der Schweizer-GTFS Daten zu übernehmen und die gleichen Daten zur Verfügung zu stellen.

Dadurch dass wir dieses GTFS-Daten selber erstellt haben, wissen wir nun was genau vorhanden ist und können dadurch nachvollziehen ob z.B. Die GTFS-Daten richtig eingelesen wurden und daraus auch weitere Methoden auf ihre Richtigkeit überprüfen kann. Des Weiteren hilft uns dieses Dummy-GTFS bei der Zeitersparnis, weil ein komplettes Schweizer-GTFS schon ein paar Minuten braucht um die Daten einzulesen. So

brauchen wir nicht bei jedem Ausführen des Programms jedes mal ein paar Minuten zu warten, was uns bei der Entwicklung viel Zeit erspart.

7.6 Mocking

Das Programm kann grob in drei Abschnitte aufgeteilt werden. Das erstellen des Zeitplans aus den GTFS-Daten, den Wegfindungsalgorithmus und einen Converter welcher das Resultat in die von der Webseite verlangte Form bringt. Damit diese drei Abschnitte separat behandelt werden konnten und das Programm dennoch jederzeit überprüft werden konnte entschieden wir uns ein Mockup für die Abschnitte durchzuführen.

7.6.1 CSAMock

Das Mocking des CSA bekommt ein Zeitplan-Objekt als Eingabe und speichert dieses zur Kontrolle in einem JSON-File. Danach wird manuell ein Journey erstellt welches eine Reise von «Heerbrugg, Dornacherhof» nach «Heerbrugg, Bahnhof» repräsentiert. Diese Verbindung wurde gewählt, da es eine einfache Busfahrt ohne Zwischenstops ist. Dieser Journey wird dann als Antwort zurückgegeben.

7.6.2 TimeTableBuilderMock

Das Mocking des TimeTableBuilders erstellt manuell einen Zeitplan mit zwei Haltestellen und einer Busverbindung. Dieser wird anschliessend als Response zurückzugeben.

7.6.3 JourneyToTripPlanConverterMock

Der als Parameter bekommene Journey wird zur Überprüfung in ein JSON-File gespeichert. Anschliessend wird manuell eine Webseitenantwort aufgebaut. Anfangs wurde eine Reise ohne Zwischenstops, Umsteigen und Fusswege zurückgegeben. Doch in folgenden Schritten wurde das Mocking erweitert um die zuvor erwähnten, komplexeren Fälle zu behandeln.

7.7 TimeTableBuilder

einstiegspunkt in GTFSMODUL + Namenskonflikt mit der onebusaway Bibliothek + Problematik Objektreferenz Die Aufgabe des TimeTableBuilder ist es Daten aus dem GTFS zu lesen und anschliessend eine Datenstruktur daraus zu erzeugen die der Connection Scan Algorithm benötigt. Zum einlesen der GTFS-Daten wird die Onebusaway library verwendet. Diese erzeugt anhand der Daten Objekte. Durch diese Library ist es einfacher Daten aus dem GTFS abzufragen. zb. von einem Stop kann man gezielt nach dem Namen fragen(stop.getName()).

Da ein Stop aus der Library gleich bei uns heissen würde. Führt dies zu einem Namenskonflikt mit der Klasse Stop(für CSA) und Stop von der onebusaway Library. Um dieses Problem zu lösen benannten wir die Klassen die benötigt werden für den CSA um. z.B. Stop -> StopCSA.

7.8 JourneyToTripPlanConverter

Der JourneyToTripPlanConverter bildet einen von der Webseite benötigten TripPlan aus den vom CSA generierten Journeys. Aus dem Mocking sind die benötigten Parameter schon bekannt so dass die Generierung nun nur noch automatisiert werden muss. Dabei gab es jedoch mehrere Punkte welche speziell beachtet werden mussten.

1. Ein Journey besitzt nur eine Dauer für den kompletten Weg. Der TripPlan jedoch speichert sich separate Werte für Fahrzeit, Laufzeit und Wartezeit. Die Fahrzeit und die Laufzeit können dabei einfach aus den Start- und Stop-Zeiten der Legs oder Footpaths übernommen werden. Bei der Wartezeit beseht jedoch das Problem das sie den Zeitunterschied zwischen zwei Abschnitten repräsentiert. Dies bedeutet, dass in der Schleife die beiden zu vergleichenden Zeiten nicht gleichzeitig vorhanden sind. Dieses Problem wird umgangen indem die Endzeit des Itinerarays nach jedem berechneten Leg angepasst wird. Somit stimmt dieser Wert immer mit der Endzeit des Legs des vorhergehenden Schleifendurchgangs überein.
2. Jedes Leg im TripPlan benötigt ein LegGeometry-Object. Dies wird benötigt damit die Webseite eine Linie entlang des Fahrtweges anzeigen kann. Dies wird mithilfe der GeometryUtils-Bibliothek und den Start- und Endkoordinaten eine Gerade erstellt. Diese
3. Im TripPlan wird ein Fussweg mithilfe von verschiedenen WalkSteps definiert. Diese benötigen jedoch eine Variable AbsoluteDirection welche 8 Himmelsrichtungen repräsentiert. Diese musste in einer komplexen Winkelberechnung aus den Koordinaten berechnet werden. Danach wird der Winkel auf 45 Grad Abschnitte gerundet und den jeweiligen Himmelsrichtungen zugeordnet.
4. Der CSA stellt sowohl Fusswege als auch Umsteigeprozesse als Footpaths dar. Umsteigeprozesse werden aber vom TripPlan nicht dargestellt, ausser das sie in die Zeitberechnung miteinfließen müssen. Daher müssen die Fusswege und Umsteigeprozesse unterschieden werden. Nachdem ein Leg generiert wurde, jedoch noch bevor es der Liste von Legs hinzugefügt wurde, wird überprüft, ob die Start- und Zielkoordinaten gleich sind. Ist dies der Fall so handelt es sich um einen Umsteigeprozess und das berechnete Leg wird der Liste nicht hinzugefügt. Die berechneten Zeiten werden jedoch trotzdem für das nächste Leg verwendet.

7.9 ConnectionScanAlgorithm

Von den beiden CSA-Versionen kümmern wir uns zuerst um den Earliest Arrival Scan Algorithmus, da dieser weniger Komplex ist und als Grundlage für den Profile Connection Scan Algorithmus dient.

7.9.1 EAS

Am Anfang werden für alle Stops und Trips Handler angelegt. Handler sind Hilfskonstruktionen welche alle Informationen der Stops und Trips beinhalten, welche im Verlauf

der Berechnung geändert werden müssen wie zum Beispiel die EinstiegsConnection für die Trips. Das ist nötig, da wir mit einem persistenten TimeTable-Objekt arbeiten welches für die nächsten Anfragen nicht verändert werden darf.

Danach werden die Zeitvariablen vorbereitet. Das Jahr, der Monat und der Tag müssen aus dem Request übernommen werden, da die Zeitangaben des Zeitplans nur auf die Uhrzeit und nicht auf das Datum beziehen. Dazu werden die Datumswerte in Variablen gespeichert. Nun wird im Stop-Handler des Startstops die Startzeit des Requests eingetragen. Für alle anderen Stop-Handler wird die Zeit auf den 31.12.20000 gesetzt. Diese Zeit simuliert eine Unendlich grosse Zahl welche trotzdem noch mit den Calendar-Methoden verglichen werden kann.

Dann werden alle aufsteigend nach Abfahrtszeit sortierten Connections durchlaufen. Für jede Connection wird überprüft, ob die im Stop-Handler des Startstops der Connection gespeicherte Zeit vor der abfahrtszeit der Connection ist, oder ob im zur Connection passenden Trip-Handler das Trip-Bit gesetzt ist. Das Trip-Bit ist am Start auf «false» gesetzt. Sobald eine Connection gefunden wird welche eine der zwei vorherigen Bedingungen erfüllt so wird es für den zur Connection passenden Trip auf True gesetzt. Damit werden sich erreichbare Verkehrsmittel gemerkt, so dass weitere Connections im gleichen ÖV auch als erreichbar markiert sind. Die zweite Bedingung prüft ob der Startstop der Connection schon erreicht wurde. Anfangs ist nur die Zeit im Stop-Handler des Startstops gesetzt. Alle anderen Zeiten sind auf Unendlich gesetzt. Somit werden nur Connections behandelt welche vom Startstop ausgehen und später abfahren als die im Request definierte Startzeit. Sobald so eine Connection gefunden wurde wird die Ankunftszeit in den Stop-Handler des Ankunftsstops der Connection geschrieben. Somit ist nun auch dieser als erreichbar markiert und wird bei weiteren Durchläufen beachtet. Zusätzlich wird jedesmal wenn eine Connection gefunden wurde ein JourneyPointer im Stophandler des Ankunftsstops der Connection gespeichert, um den Journey später rekonstruieren zu können.

Das Breakkriterium für die Schleife ist wenn die Abfahrtszeit der Connection später ist als die im Stop-Handler des Zielstops gespeicherte Zeit. Diese Zeit ist auf Unendlich gesetzt und wird erst neu gesetzt wenn ein Journey zum Zielpunkt gefunden wurde. Da die Connections aufsteigend nach Abfahrtszeit durchlaufen werden ist dies auch automatisch die am Frühesten ankommende Reise.

Nun wird vom Zielstop aus der Journey rekonstruiert. Im JourneyPointer des Zielstops steht der zuvor erreichte Stop. Nun wird der JourneyPointer dieses neuen Stops untersucht. Dies wird so lange wiederholt bis der Startstop erreicht ist. Nun werden die gefundenen Stops in umgekehrter Reihenfolge in den Journey eingetragen und der Journey wird zurückgegeben.

7.9.2 PCS

Der Profile Connection Scan Algorithmus erstellt am Anfang auch Stop- und Triphandler sowie die im EAS erwähnten Zeitvariablen. Danach werden die Verbindungen in einer Schleife durchlaufen. Im Gegensatz zum EAS werden sie jedoch absteigend nach Abfahrtszeit durchlaufen. Damit jedoch nur ein Zeitfenster der Verbindungen um die gesuchte Zeit herum behandelt werden muss wird vor der Schleife ein EAS durchgeführt, welcher nur die früheste Ankunftszeit zurückliefert. Dieser Zeit werden dann zwei Stunden hinzugerechnet und sie wird als Einstugspunkt für die Schleife benutzt.

Nun werden drei Zeitvariablen gesetzt. Die erste zeigt wann und ob man ans Ziel kommt, wenn man aus dem ÖV aussteigt. Dies ist nur der Fall, wenn der Ankunftsort der Connection dem Zielort entspricht. In diesem Fall wird die Ankunftszeit der Connection in der Zeitvariable gespeichert. Ist dies nicht der Fall so wird sie auf Unendlich gesetzt.

Die zweite Zeitvariable zeigt wann und ob man ans Ziel kommt, wenn man im ÖV sitzen bleibt. Dabei wird die TripZeit in der Zeitvariablen gespeichert. Ist diese ungleich unendlich so ist das Ziel mit weiterfahren erreichbar.

Die dritte Zeitvariable zeigt wann und ob man ans Ziel kommt, wenn man in ein anderes ÖV umsteigt. Dazu werden alle TimeTupels im Zielstop aufsteigend durchlaufen bis eine TimeTupel gefunden wurde, welches später abfährt als die Ankunftszeit der Connection plus die Umsteigezeit. Da jeder Stop ein default TimeTupel mit der Zeit Unendlich hat wird immer eine Möglichkeit gefunden. Danach werden die drei Zeitvariablen verglichen und der früheste wird behalten. Dies ist nun die schnellste Zeit in welcher man von dieser Connection aus das Ziel erreichen kann. Nun wird von der Abfahrtszeit der Connection die Umsteigezeit abgezogen und es wird mit diesen beiden Zeiten ein neues TimeTupel erstellt. Nun wird überprüft ob die Ankunftszeit nicht unendlich ist und somit der Zielort erreichbar ist. Wenn dies der Fall ist so wird das Tupel zusammen mit einem JourneyPointer in den Stop-Handler geschrieben, für den Trip wird eine TripZeit festgelegt und falls für diesen Trip noch keine ExitConnection festgelegt ist wird die aktuelle Connection eingetragen. Dies wird so lange wiederholt bis die Abfahrtszeit der Connection vor der im Request spezifizierten Startzeit ist.

Nun werden die Journeys vom Startstop aus rekonstruiert. Es wird ein Journey für jedes im Startstop gespeicherte TimeTupel generiert. Dabei wird erste JourneyPointer im Journey eingetragen. Dann werden die im Endstop des Journeys gespeicherten TimeTupels überprüft. Hierbei wird nur das erste Tupel und nicht alle verwendet. Diese Entscheidung wurde aus performance Gründen getroffen, da die Anzahl an schritten ansonsten Exponenziell ansteigen würde. Dies wird so lange wiederholt bis der Zielstop erreicht ist.

Der Vorgang benötigt jedoch noch einen Zusatz, da der Algorithmus auch Reisen findet, welche Kreise fahren und den selben Ort mehrfach anfahren. Um dies zu verhindern

wird eine Liste mit allen schon angefahrenen Stops geführt. Für jeden neuen Journey-Pointer wird geprüft ob der Ankunftsstop schon in der Liste vorhanden ist. Ist dies der Fall so wird ein Bit auf TRUE gesetzt. Am ende werden nur Journey der Rückgabe hinzugefügt bei welchen kein Stop mehrfach angefahren wurde.

7.10 Performance-Test

Um den Performance-Test durchzuführen wird ein JUNIT-Test mit der JUNIT-Benchmark erweiterung genutzt. Dieser führt mehrere Requests durch. Ein Reqeust ohne Umsteigen. Einer mit einmal Umsteigen und einer mit zweimal umsteigen.

Die Tests schlugen jedoch bei dem CSA ohne die QuadTree-Optimierung fehl, da die Anfrage mit den kompletten Schweitzer-Daten länger als 24 Stunden dauerte.

8 Resultate

Unsere Tests und deren Ergebnisse darstellen

9 Fazit

Fazit der BA über den CSA und das Projekt selbst

Abbildungsverzeichnis