

BA ÖV Journey Planning

Implementation und Benchmarking des Connection Scan
Algorithmus

August 2018

Author:

Christian Bühler, Flavio Tobler

Project partner:

Institut für Photonics und ICT (IPI), Hochschule für Technik und Wirtschaft Chur

Examiner:

Prof. Dr. Ulrich Hauser, HTW

Second Examiner:

Lukas Toggenburger, HTW

1 Abstract

Der Connection Scan Algorithmus (CSA) ist ein moderner Algorithmus zur Beantwortung von Anfragen auf zeitplanbasierten Systemen. Diese Arbeit liefert eine Implementation des CSA in Java. Mittels dem Web Frontend des FOSS Reiseplaners OpenTripPlanner (OTP) wird diese Benutzern zugänglich gemacht. OTP verwendet standardmässig den A* Algorithmus.

Im Benchmark konnte die angestrebte Performance-Steigerung des CSA gegenüber A* nicht erreicht werden: Unsere Implementierung ist bei Anfragen auf das gesamtschweizerische Fahrplannetz um den Faktor 10000 langsamer als der zuvor verwendete Algorithmus.

The Connection Scan Algorithm (CSA) is a modern Algorithm for answering inquiries on timetable based systems. This paper provides an implementation of the CSA in Java. User accessibility is granted via the web frontend of the FOSS tripplanner OpenTripPlanner (OTP). OTP uses the A* Algorithm by default.

In the benchmark we couldn't achieve the expected performance increase of the CSA compared to the A* algorithm. Our implementation is slower than the original algorithm by a factor of 10000.

Inhaltsverzeichnis

1	Abstract	I
2	Einleitung	2
3	Aufgabenstellung	3
3.1	Lastenheft	4
4	Ausgangslage	5
4.1	JourneyPlanning	5
4.2	OpenTripPlanner	5
4.3	Verwendung des OpenTripPlanners	5
4.4	Dijkstra Algorithmus	6
4.5	A* Algorithmus	6
4.6	Connection Scan Algorithmus	6
4.7	General Transit Feed Specification	6
5	Methode	8
5.1	Performance Test (Benchmarking)	8
5.2	Verzeichnisstruktur	8
5.3	Zeitplan	9
5.4	Mocking	9
6	CSA für OTP	10
6.1	Datenstruktur	10
6.1.1	Zeittafel (TimeTable)	10
6.1.2	Fusswege (FootpathCSA)	10
6.1.3	Haltestellen (StopCSA)	11
6.1.4	Verbindungen (ConnectionCSA)	11
6.1.5	Fahrten (TripCSA)	12
6.2	Reise (Journey)	12
6.2.1	Reisezeiger (JourneyPointer)	12
6.2.2	Fahrtabschnitte (LegCSA)	13
6.3	Programmablauf	13
6.3.1	Zeittafelerzeuger (TimeTableBuilder)	14
6.3.2	Server	14
6.3.3	Webseitenaufruf	15
6.3.4	Earliest Arrival Connection Scan	15
6.3.5	Profile Connection Scan	16
6.3.6	JourneyToTripPlanConverter	17

7	Produkt	18
7.1	Modellierung der Datenstruktur	18
7.2	Automatische generierte Klassendiagramme	18
7.3	Dummy-GTFS Daten erstellen	19
7.4	Mocking	19
7.4.1	CSAMock	19
7.4.2	TimeTableBuilderMock	20
7.4.3	JourneyToTripPlanConverterMock	20
7.5	TimeTableBuilder	20
7.6	JourneyToTripPlanConverter	21
7.7	ConnectionScanAlgorithm	22
7.7.1	EAS	23
7.7.2	PCS	24
7.8	Performance-Test	25
7.9	OTP mit Schweizer Daten implementieren	26
7.10	Kann OTP ohne .osm file ausgeführt werden?	26
8	Resultate	28
9	Fazit	29

2 Einleitung

Ziel der Facharbeit war es einen OpenSource-JourneyPlanner zu finden, der mit dem grossen Datensatz der Schweizer GTFS-Daten funktioniert, oder gegebenenfalls selbst einen JourneyPlanner zu entwickeln. Der OpenTripPlanner ist ein Programm welches diesen Anforderungen entspricht.

Während des Fachmoduls wurden wir auf den Connection Scan Algorithmus aufmerksam. Dieser ist ein moderner Wegfindungsalgorithmus, welcher einen komplett anderen Ansatz als die bisherigen Wegfindungsalgorithmen wählt. Er sollte eine bessere Performance liefern als die Dijkstra basierten Algorithmen. Er wird jedoch nicht oft verwendet, da er andere Voraussetzungen an die Datenstruktur und den Programmablauf hat als die Dijkstra basierten Algorithmen.

Ziel dieser Arbeit ist es nun den Connection Scan Algorithmus für den OpenTripPlanner zu implementieren und dessen Performance zu testen.

Zuerst werden die für das Projekt wichtigen Grundlagen erläutert. Anschliessend werden die im Projekt verwendeten Methoden erklärt. Dann wird der Connection Scan Algorithmus sowie die von ihm benötigte Datenstruktur genau erläutert. Als nächstes wird unser Programmierprozess genau dargestellt. Zum Schluss werden die Resultate des Performance-Tests präsentiert und analysiert.

3 Aufgabenstellung

- Schreiben sie eine Implementation des CSA für den OpenTripPlanner. Dieser muss OTP-Development-Richtlinien konform sein.
- Implementieren sie die Schweizer GTFS und OpenStreetMap daten für den OTP, so dass Punkt zu Punkt Verbindungen in der Schweiz berechnet werden können.
- Implementieren sie den Schweizer realtime GTFS-Feed, so dass das Programm auf Verspätungen oder Fahrplanänderungen reagieren kann.
- Implementieren sie Skalierungsoptimierungen für das Programm, so dass es mit einem landesweiten System bessere Performanz liefert.
- Führen sie Performance Tests durch und vergleichen sie das Implementierte System mit dem auf Dijkstra basierenden OTP.
- Dokumentieren sie ihre Ergebnisse und schreiben sie einen ausführlichen Bericht.

3.1 Lastenheft

Funktionale Anforderungen		
Anforderung	Anforderungsart	Priorisierung
Der CSA muss als Berechnungsalgorithmus verwendet werden.	Muss	1
Es kann eine Verbindung zwischen 2 Stationen zu einem bestimmten Zeitpunkt errechnet werden.	Muss	2
Fusswege zu Stationen hin können miteinbezogen werden können.	Muss	3
Das Programm kann auf Verspätungen und Fahrplanänderungen reagieren.	Muss	4
Mehrere mögliche Verbindungen können angezeigt werden.	Muss	5
Fusswege zwischen verschiedenen Stationen können miteinbezogen werden.	Soll	6
Die Angezeigte Route soll den Fahrlinien, z.B. Schienen, folgen.	Kann	7

Nicht funktionale Anforderungen

- Der Programmcode muss den OTP-Development-Richtlinien entsprechen.
- Die Query-Zeit muss weniger als 1 Sekunde betragen.
- Die Preprocessing-Zeit muss weniger als 30 Minuten betragen.
- Die Query-Zeit soll schneller als die des originalen OTP sein.

4 Ausgangslage

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

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 Routenplanner bezieht sich ein JourneyPlanner nur auf öffentliche Verkehrsmittel und nicht auf Privatfahrzeuge.

4.2 OpenTripPlanner

Der OpenTripPlanner [1] ¹, kurz OTP, ist eine auf der Maven-Repository aufbauende Trip-Planning Software. Er war Anfangs für Städte ausgelegt, findet nun jedoch auch in ersten landesweiten Verkehrsnetzen Anwendung. 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 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. 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. Dies ist nötig, weil während des Aufrufs Daten im Graphen verändert werden, diese sich jedoch nicht auf andere Aufrufe auswirken dürfen.

OTP steht unter einer GNU Lesser General Public License.

4.3 Verwendung des OpenTrippers

Der OTP wird über das OTPMain-File im Packet org.opentripplanner.otp gestartet. Dabei benötigt er drei Parameter. Mit den Parameter `--build` wird dem OTP ein Pfad mitgegeben, in welchem die GTFS- und OSM-Daten abgelegt sind. Mit dem Parameter `--inMemory` behält der OTP seine Daten ausschliesslich im RAM. Wird der Parameter nicht angegeben so werden die Daten im im Build-Parameter angegebenen Pfad gespeichert.

Mit einem Weiteren Parameter kann festgelegt werden wie viel Arbeitsspeicher das Programm maximal verwenden darf. Der Parameter lautet `-Xmx4g`. Die Zahl steht dabei für die Anzahl zur Verfügung gestellten Gigabyte.

¹<https://opentripplanner.org/>

4.4 Dijkstra Algorithmus

Der Dijkstra Algorithmus [2] ist ein auf einem Graphen basierender Wegfindungsalgorithmus. Er bildet einen Graph, wobei Haltestellen Punkte sind und Verbindungen die Linien zwischen den Punkten. Alle Verbindungen besitzen eine Gewichtung, welche mit der benötigten Zeit korreliert. Der Algorithmus sucht nun den Weg mit der geringsten summierten Gewichtung vom Start- zum Zielpunkt.

4.5 A* Algorithmus

Der A* Algorithmus [3] ist eine Erweiterung des Dijkstra Algorithmus. Er durchsucht den Graphen nicht in alle Richtungen sondern sucht gezielt in die Richtung des Zielpunktes. Dadurch lässt sich die Performance verbessern.

4.6 Connection Scan Algorithmus

Der Connection Scan Algorithm [4] , kurz CSA, ist ein moderner Algorithmus zur Bearbeitung von Anfragen auf zeitplanbasierten Sytemen. 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 Bessten Journey nach mehreren Kriterien sortieren kann. Beide sind darauf ausgelegt genau einen Journey zurückzugeben. Der Algorithmus iteriert über eine nach Abfahrtszeit sortierte Liste aller Verbindungen. Dabei werden vom Start- oder Zielpunkt erreichbare Verbindungen markiert. Dadurch entsteht ein Netz aus Verbindungen welches sich immer weiter aufspannt. Dies wird dann so lange wiederholt bis ein Weg zwischen den beiden Punkten gefunden wurde. Danach gibt der Algorithmus entweder die früheste Ankunftszeit am Zielpunkt, oder den besten Weg vom Startpunkt zum Zielpunkt zurück.

4.7 General Transit Feed Specification

General Transit Feed Specification (GTFS) [5] [6] ist ein von Google entwickeltes Dateiformat zum Austausch von Öffentlichen Verkehrsdaten sprich Fahrpläne. Die Daten werden von der SBB-Plattform² 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

²<https://opentransportdata.swiss/>

enthält. GTFS-RT(RealTime) ist eine Erweiterung der GTFS-Static Daten. Dies ermöglicht es die GTFS-Daten dynamisch zu aktualisieren. [7]

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

5.1 Performance Test (Benchmarking)

Unsere Performance Tests wurden mit der JUnit-Benchmark Bibliothek durchgeführt. Das ganze Programm sowie die grossen Teilschritte wurden geprüft.

Die Tests wurden auf einem «HP Pavilion dv8 Notebook PC» durchgeführt. Das Gerät hat einen Intel i5 Prozessor mit 2.4GHz und vier Kernen. Dem Gerät stehen 8GB RAM zur Verfügung.

5.2 Verzeichnisstruktur

Da der OTP sehr viele verschiedene Pakete besitzt haben wir uns entschieden alles was den CSA betrifft und von uns programmiert wird in einem separaten Paket zu sammeln wie in Abbildung 1 zu sehen. Dadurch behalten wir die Übersicht was wir zusätzlich in das bestehende Programm hinzugefügt haben.

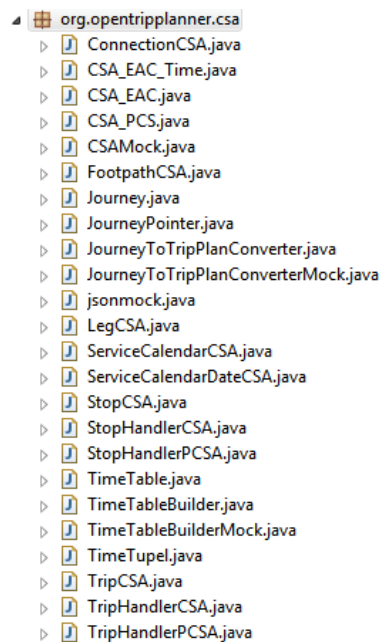


Abbildung 1: Dieses Bild gibt eine Übersicht welche Klassen in unserem Paket vorhanden ist.

Da ein Stop aus der Library gleich bei uns heissen würde. Führt dies zu einem Namenskonflikt mit der Klasse Stop für den CSA und Stop von der onebusaway Library. Ein weiterer Namenskonflikt war Leg diese Klasse ist schon im OTP(Grundversion) vorhanden. Um dieses Problem zu lösen benannten wir die

Klassen, die wir benötigen werden für den CSA um. Indem wir die Klassen mit der CSA Endung erweiterten z.B. Stop in StopCSA.

5.3 Zeitplan

Für das Projekt wurde ein Zeitplan im Excel erstellt. Dann wurde jede Woche der «Ist» mit dem «Soll» zustand verglichen. Nach der Hälfte der Projektzeit stellten wir fest, dass wir für die Aufgaben durchgehend zu wenig Zeit eingeplant haben. Nach einer Absprache mit den Dozenten passten wir den Zeitplan entsprechend an. Das Reagieren auf Verspätungen wurde aus dem Scope entfernt und den anderen Teilaufgaben wurde mehr Zeit zugeteilt. Zu einem späteren Zeitpunkt musste der Scope erneut aufgrund von erhöhtem Zeitaufwand angepasst werden. Die Fusswege zwischen den Stationen sowie die QuadTree-Optimierung wurden aus dem Scope entfernt.

5.4 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 können und das Programm dennoch jederzeit überprüft werden kann entschieden wir uns ein Mockup für die Abschnitte durchzuführen.

6 CSA für OTP

6.1 Datenstruktur

Der CSA benötigt zwei Datenstrukturen. Eine Zeittafel(TimeTable) für die Eingabe von Daten und eine Reise(Journey) für die Rückgabe von Daten.

6.1.1 Zeittafel (TimeTable)

Der TimeTable ist eine Zeittafel die alle benötigten Daten für den CSA zur Verfügung stellt. Die Datenstruktur ist ein Quadrupel aus Sammlungen von Fusswege(FootpathCSA), Haltestellen(StopCSA), Verbindungen(ConnectionCSA) und Fahrten(TripCSA). Neben den .add() und .show() Funktionen für die Sammlungen enthält die TimeTable-Klasse die Methode .getFootpathChange() welche für eine Haltestelle die Umsteigzeit zurückgibt.

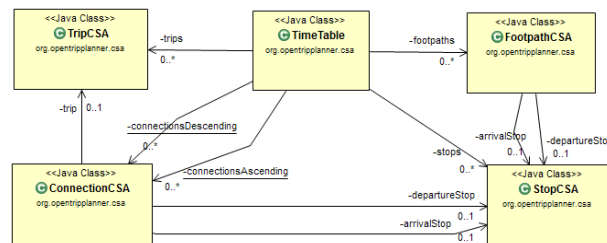


Abbildung 2: Hier ist das UML-Diagramm zur TimeTable Datenstruktur zu sehen.

6.1.2 Fusswege (FootpathCSA)



Abbildung 3: Symbolbild für den Fussweg(FootpathCSA)

Ein Fussweg 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.

6.1.3 Haltestellen (StopCSA)



Abbildung 4: Symbolbild für die Haltestelle(StopCSA) [8]

Ein Stop ist eine Haltestelle für öffentliche Verkehrsmittel. Ein Stop besitzt einen Namen, Längen- und Breitengrad.

6.1.4 Verbindungen (ConnectionCSA)

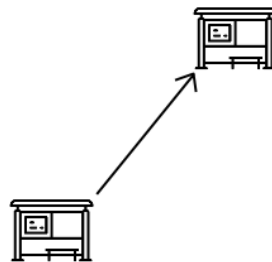


Abbildung 5: Symbolbild für die Verbindung(ConnectionCSA)

Eine Connection ist eine Verbindung zwischen einem DepartureStop und einem ArrivalStop . Die Connection hält Daten wie die Abfahrtszeit vom DepartureStop

und die Ankunftszeit vom ArrivalStop. Zudem weiss die Connection zu welchem Trip sie gehört.

6.1.5 Fahrten (TripCSA)

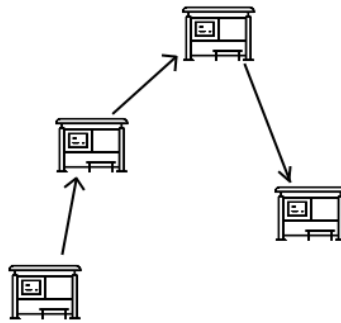


Abbildung 6: Symbolbild für die Fahrt(TripCSA)

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. Ein Trip besteht aus mehreren Connections die aneinandergereiht sind. Des weiteren hält der Trip Informationen über das Verkehrsmittel(Zug,Bus usw.) und welche Geschäftsorganisation dieses Verkehrsmittel zur Verfügung stellt. Ausserdem enthält der Trip Informationen ob der Trip an jenem Tag auch fährt oder nicht.

6.2 Reise (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.

6.2.1 Reisezeiger (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.

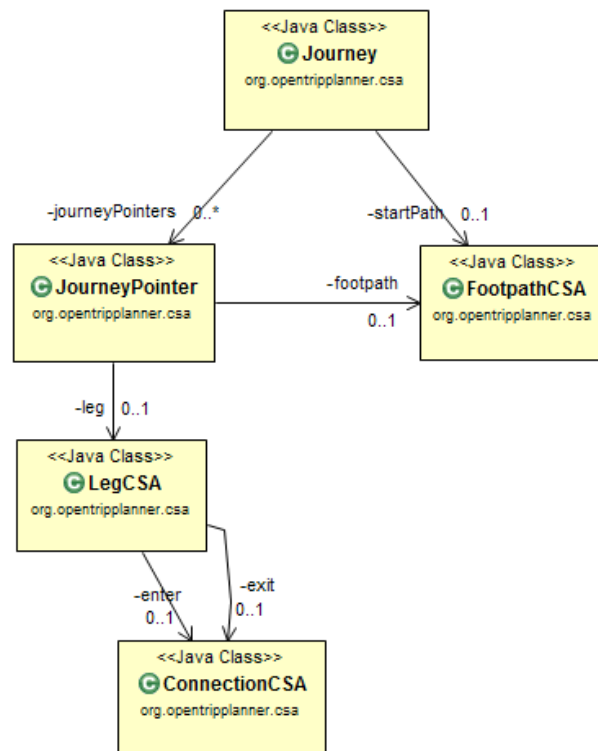


Abbildung 7: Hier ist das UML-Diagramm zur Journey Datenstruktur zu sehen.

6.2.2 Fahrtabschnitte (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.

6.3 Programmablauf

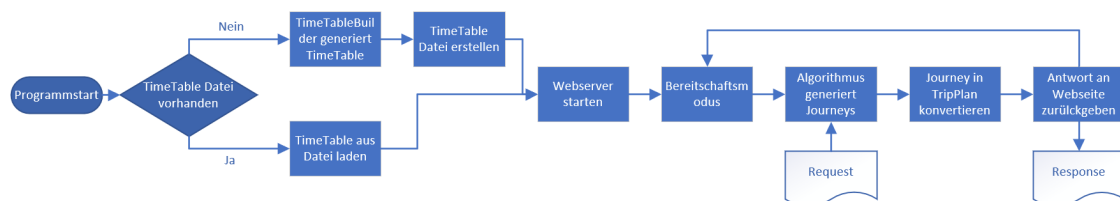


Abbildung 8: Programmablauf des OTP mit dem CSA

6.3.1 Zeittafelerzeuger (TimeTableBuilder)

Da wir für den CSA eine Zeittafel (TimeTable) benötigen, muss diese durch den Zeittafelerzeuger (TimeTableBuilder) erstellt werden. Die zugrundeliegenden GTFS-Daten können jedoch nicht in dieser Form einfach der Zeittafel übergeben werden. Der Zeittafelerzeuger erzeugt die Zeittafel. Durch die Funktion `.loadFromGtfs()` werden die Daten eingelesen und in die benötigte Form gebracht. Dieser Prozess kann durchaus einige Zeit in Anspruch nehmen. Deshalb wird nachdem die Zeittafel fertig befüllt wurde anschliessend serialisiert und als Java-SerializedObjectFile gespeichert. Somit brauchen wir nicht mehr jedesmal die Zeittafel komplett neu einzulesen und zu erstellen. Durch die Funktion `.loadFromSerializedObjectFile()` kann die zwischengespeicherte Zeittafel jederzeit eingelesen werden, was die Zeit um eine schon in Form gebrachte Zeittafel zu erzeugen um einiges verkürzt. Unabhängig von beiden Methoden wird dann der Server gestartet.

6.3.2 Server

Der Server wird über die Main-Funktion gestartet. Bei dem Server handelt es sich um einen GrizzlyServer. Der Server stellt eine Webseite zur Verfügung (Webserver). Über diese Webseite kommuniziert der Server mit allen Clients, indem es alle Requests (Anfragen) entgegen nimmt und alle dazugehörigen Responses (Antworten) zurückschickt. Die Kommunikation läuft hierbei über ein HTTP-Protokoll zwischen Server und Clients (Browser).

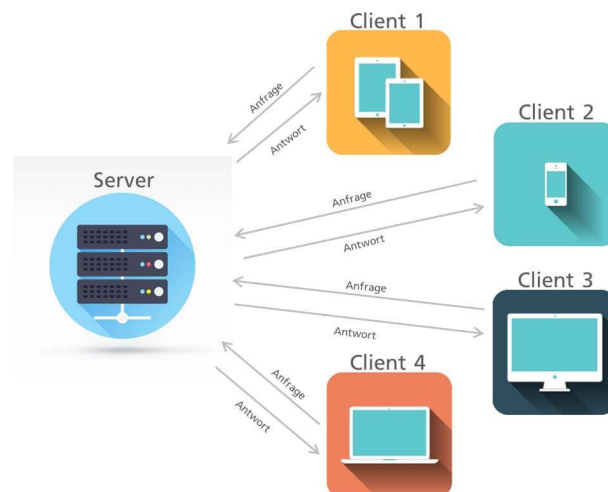


Abbildung 9: Darstellung der Kommunikation zwischen Clients und Server [9]

6.3.3 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.

6.3.4 Earliest Arrival Connection Scan

Der Earliest Arrival Connection Scan Algorithmus [4], kurz EACS, ist ein Wegfindungsalgorithmus welcher MEAT (Minimum Expected Arrival Time) Problem löst. Er sucht sich den schnellsten Weg vom Start zum Zielpunkt und beendet dann seine Suche ohne nach Alternativen zu suchen.

Er 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.

Alternativ kann der EACS auch nur die früheste Ankunftszeit anstelle des kompletten Journeys zurückgeben. In dieser Version müssen keine Zeiger gespeichert werden, was den Algorithmus schneller macht. Es gehen jedoch Informationen über den Reiseweg verloren.

6.3.5 Profile Connection Scan

Der Profile Connection Scan Algorithmus [4], kurz PCS, ist ein Wegfindungsalgorithmus welcher alle möglichen Reisen in einem bestimmten Zeitraum errechnet und sich anschliessend nach den gewünschten Kriterien die beste Reise zurückgibt.

Er 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 des Requests ist. Ist dies der Fall, so wird im Abfahrtsort der Verbindung die Ankunftszeit sowie die dazugehörige Verbindung gespeichert. Zusätzlich wird die Ankunftszeit für das jeweilige öffentlichen Verkehrsmittels 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 im Abfahrtsort und dem öffentlichen Verkehrsmittel gespeichert.

- Kommt man ans Ziel wenn man sitzen bleibt?

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

6.3.6 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 `JourneyPointer` abgearbeitet sind, wird das `Itinerary` dem `TripPlan` hinzugefügt. Nachdem die 3 besten Journeys konvertiert wurden, wird der `TripPlan` an die Webseite zurückgegeben.

7 Produkt

7.1 Modellierung der Datenstruktur

Der `TimeTable` muss alle Daten Fusswege(`FootpathCSA`), Haltestellen(`StopCSA`), Fahrten(`TripCSA`), Verbindungen(`ConnectionCSA`) zur Verfügung stellen. Dies kann sie indem wir Sets(Mengen) verwenden. Der Grund wieso wir Sets verwenden ist das es keine Duplikate geben kann. Den ein Stop darf nur einmal in der Sammlung von `stops` vorkommen.

```
private Set<StopCSA> stops = new HashSet<StopCSA>();
private Set<TripCSA> trips = new HashSet<TripCSA>();
private Set<FootpathCSA> footpaths = new HashSet<FootpathCSA>();
private static Set<ConnectionCSA> connectionsAscending = new LinkedHashSet<ConnectionCSA>();
private static Set<ConnectionCSA> connectionsDescending = new LinkedHashSet<ConnectionCSA>();
```

Abbildung 10: Das Bild zeigt welche Sammlungen wir in der Zeittafel zur Verfügung stehen.

Wie man in der Abbildung 10 erkennen kann verwenden wir HashSets für Haltestellen(`stops`), Fahrten(`trips`) und Fusswege(`footpaths`), weil diese Sammlung nicht sortiert sein müssen. Im `TimeTableBuilder` verwendeten wir noch den `TreeSet` um die Verbindungen(`ConnectionCSA`) zu sortieren bei ihrer Erzeugung. Beim `TimeTable` haben wir die gleiche Sammlung von Verbindungen, jedoch in `LinkedHashSet` erzeugt, weil wir zuvor versuchten die Sammlungen zwischenspeichern mit Textfiles (Jackson 2). Da dies jedoch nicht mit `TreeSets` funktionierte änderten wir diese zu `LinkedHashSet`. Der einzige Unterschied zwischen `LinkedHashSet` und `HashSet` ist das die Daten, die Reihenfolge beibehalten wie man sie auch hinzufügt hat. Da die Sammlungen (`connectionsAscending` und `connectionsDescending`) nachdem erstellen nicht mehr ändern, benötigten wir also nicht zwingend ein `TreeSet` sondern können es auch mit einem `LinkedHashSet` umgesetzt werden.

7.2 Automatische generierte Klassendiagramme

Klassendiagramme können sehr behilflich sein bei der Entwicklung. Sie geben einen Überblick welche Klassen miteinander agieren. Der `Opentripplaner` besteht aus unzähligen verschiedenen Klassen. Um eine bessere Übersicht zu erhalten erzeugten wir mit dem Programm `objectaid`³ (ein Plugin für Eclipse) fast vollautomatisch Klassendiagramme.

³<http://www.objectaid.com/home>

7.3 Dummy-GTFS Daten erstellen

Wir haben selbst GTFS-Daten erstellt, weil der TimeTableBuilder mit dem kompletten Schweizer-GTFS ein paar Stunden braucht um die Daten einzulesen. So brauchen wir nicht bei jedem Ausführen des Programms jedes mal ein paar Stunden zu warten, was uns bei der Entwicklung viel Zeit erspart. Dadurch das 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.

Das Dummy-GTFS wurde anhand der bestehenden Schweizer-GTFS Daten erstellt. Im Schweizer-GTFS sind auch die Daten des öffentlichen Verkehrs von Liechtenstein enthalten. Wir haben uns entschieden ein Teil der Liechtensteinischen Busverbindungen zu übernehmen und selber zu erstellen, um Daten zu verwenden welche im Schweizer-GTFS auch vorhanden sind.

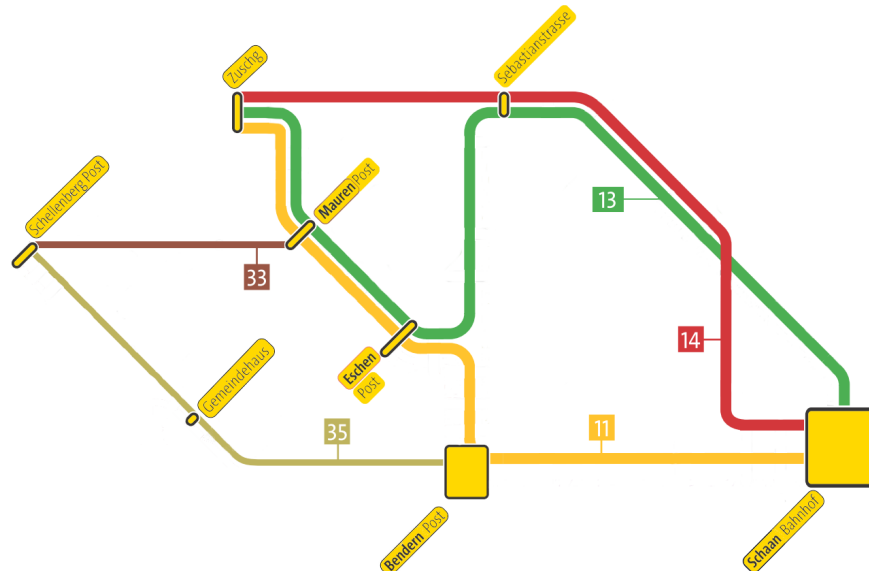


Abbildung 11: Dieses Bild gibt eine Übersicht welche Haltestellen und Routen im Dummy-GTFS vorhanden ist.

7.4 Mocking

7.4.1 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 er-

stellt 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.

7.4.2 TimeTableBuilderMock

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

7.4.3 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.

7.5 TimeTableBuilder

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. Die Objektdaten von der Onebusaway Library sind in Listen abrufbar. z.B. Entspricht im GTFS Stop.txt ein Zeilen-eintrag einem Stop Objekt. Durch diese Library ist es einfacher Daten aus dem GTFS abzufragen. z.B. von einem Stop kann man gezielt nach dem Namen fragen `stop.getName()`.

Durch diese Library lässt es sich nun relativ einfach die Daten aus dem GTFS lesen und anschliessend mit den gewünschten Daten die benötigten Objekte erstellen. Dafür wird die Funktion `.loadFromGtfs()` ausgeführt. Stop, Footpath und Trips lassen sich relativ leicht erstellen weil die GTFS-Daten hierbei schon in sehr ähnlich Form aufgebaut sind. Jedoch ist es bei der Connection etwas schwieriger, da die Daten nicht in dieser Art direkt vorhanden sind. Die Connections müssen beim der Stop_times.txt immer zwei hintereinanderfolgende Einträge (Objekte) vergleichen um zu Erkennen das es sich um eine Connection handelt. Eine Connection wird erkannt wenn sie auf den selben Trip stattfinden und die stop_sequence Zahl muss grösser sein als der vorhergehende Eintrag. Wenn dieser Fall nur Zutrifft wurde eine Connection erkannt. Bevor jetzt aber das Connection Objekt einfach erzeugt werden kann müssen noch die richtigen Referenzen gefunden werden. Den

die Stops und Trips wurden zuvor schon erzeugt. Den eine Connection muss wissen auf welchem Trip sie ist und welche Departure- und ArrivalStops sie enthält. Nachdem also eine Connection gefunden wurde wird er richtige Trip in der bestehenden Liste gesucht, anhand der TripId kann der Trip aus der Liste gefunden werden. Das selbe geschieht auch mit den Departure- und ArrivalStops. Wenn alle Informationen gefunden wurden wird das Connection Objekt erzeugt und in das TreeSet(sortiere Liste) übergeben. Durch die `.compareTo()` Funktion der Connection lässt sie sich an der richtigen Stelle im TreeSet einfügen. Somit bekommen wir alle Connections sortiert nach der Departuretime. Am Ende der Funktion wird ein Java-SerializedObjectFile vom TimeTable erstellt. Somit wird das gewünschte Format zwischengespeichert. Das erstellen dieser Datei benötigt nur 1-2 Minuten und die Datei ist 367MB gross.

Bevor wir uns entschlossen haben das Java-SerializedObjectFile am Ende der Funktion zu erstellen, versuchten wir es stattdessen mit Textfiles die durch Jackson 2.Jackson 2 ist eine library zum konvertieren von JSON(Textfile) zu Java-Objekte und umgekehrt. Doch dies hatte ein grosses Problem. Die Files wurden zwar wieder mit den Angaben richtig eingelesen, jedoch waren die Referenzen auf die richtigen Objekte nicht vorhanden. z.B. ein Stop ist einzigartig und sollte nur einmal vorkommen, jedoch wurden mehrere Stops dann erzeugt mit den selben Namen und Werten. Dies führt wiederum dann zu Probleme beim Algorithmus und konnte deshalb nicht verwendet werden.

Die Funktion `.loadFromGtfs()` benötigt etwa 60h um das Schweizer GTFS einzulesen. Aber durch Optimierung konnte die Zeit verringert werden auf 22h. Indem wir die nicht mehr einfach wie vorher ständig nach dem Trip wieder suchen wenn eine Connection gefunden wurde. Durch zwischenspeichern der Referenz des vorhergehenden Trips kann dieser wiederverwendet werden, wenn die TripId der Einträge übereinstimmen. Somit muss nur ein Trip in der Liste gesucht werden wenn diese TripId nicht übereinstimmen. Zudem kann durch zwischenspeichern der Referenz auf den vorhergehenden Stop eine Suche in der Liste vermieden werden. Da ein Trip und Stop nur einmal vorkommt in der Liste, kann beim durchlaufen mit einer Suchschleife die suche nach dem Trip vorzeitig abgebrochen werden, was verhindert das unnötig weiter nachdem Trip/Stop gesucht wird.

Die Funktion `.loadFromSerializedObjectFile()` kann verwendet werden um zwischengespeicherte TimeTables einzulesen. Das SerializedObjectFile das zuvor aus dem Schweizer GTFS erstellt wurde lässt sich innerhalb von 151s einlesen.

7.6 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 besteht 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 Itinerarys 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. Dabei wird mithilfe der GeometryUtils-Bibliothek und den Start- und Endkoordinaten eine Gerade erstellt. Diese werden dann Zusammengesetzt und bilden den Reiseweg ab.
3. Im TripPlan wird ein Fussweg mithilfe von verschiedenen WalkSteps definiert. Diese benötigen jedoch eine Variable AbsoluteDirection welche 8 Himmelsrichtungen repräsentiert. Dazu werden die Koordinaten mithilfe eine Kompass-Funktion in einen Winkel umgewandelt. 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.

7.7 ConnectionScanAlgorithm

Von den beiden CSA-Versionen [4] kümmerten wir uns zuerst um den Earliest Arrival Scan Algorithmus, da dieser weniger Komplex ist und als Grundlage für den Profile Connection Scan Algorithmus dient.

7.7.1 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 Einstiegs-Connection 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.

7.7.2 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 Zeilstop 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 JourneyPointer wird geprüft ob der Ankuftsstop 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.

7.8 Performance-Test

Um den Performance-Test durchzuführen wird ein JUnit-Test mit der JUnit-Benchmark erweiterung genutzt. Als Grundlage der Test dient der EACS. Mit diesem wurden mehrere Tests durchgeführt durch welche den Flaschenhals des Algorithmus aufzeigen sollten.

Ein Request auf dem gesamtschweizerischen Netzwerk mit dem EACS dauerte ca. 24 Stunden. Um die Dauer einer Anfrage auf eine annehmbare Zeit zu minimieren so dass wir genauere Analysen durchführen konnten, verwendeten wir ein etwas kleineres GTFS, nämlich das von Portland. Dabei stellten wir fest, dass unser Hauptbreakkriterium aufgrund von einigen Methoden, welche wegen dem JUnit-Benchmark zusätzlich als static definiert werden mussten, nicht mehr funktionierte. Das Breakkriterium soll in kraft treten wenn der Algorithmus eine Lösung gefunden hat. Dies wurde aber nicht mehr ausgelöst, somit lief er einfach durch alle Connections ohne einen Treffer zu landen. Somit waren die im Test erzeugten Zeiten nicht repräsentativ für unseren Algorithmus.

Alternativ haben wir die Funktion `System.currentTimeMillis()` verwendet. Die Funktion gibt die aktuelle Zeit in Millisekunden zurück. Durch diese Funktion ist es also möglich die zu Stoppen für beliebige Abschnitte. Indem man am einem gewählten Anfangspunkt/Endpunkt die Zeit speichert. Durch die Differenz der zwei Zeiten vom Anfangs- und Endpunkt lässt sich nun die Zeitspanne berechnen, die das Programm benötigt hat.

7.9 OTP mit Schweizer Daten implementieren

Der originale OTP muss mit den Schweizer GTFS-Daten funktionieren, da er als Grundlage für den Algorithmus dient und die Performance der beiden Algorithmen verglichen wird. 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 von unserem Projekt behandelt werden, können sie ignoriert werden. Dazu fügten wir dem OTP einen Handler für den Routentyp 1700 hinzu welcher ihn als Auto definiert. Da das Auto kein öffentliches Verkehrsmittel ist, ist es für unser Projekt nicht relevant, es wird jedoch vom OTP unterstützt.

Das zweite Problem war der benötigte Arbeitsspeicher. Für die Graphberechnung mit den SBB GTFS-Daten benötigt der OTP ca. 15 GB Arbeitsspeicher. Die von uns benutzten 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.

7.10 Kann OTP ohne .osm file ausgeführt werden?

Das .osm steht für OpenStreetMap und ist eine Plattform⁴ die frei nutzbare Daten(sprich Geodaten) zur Verfügung stellt. Wie z.B. Strassen, Wege, Gebäude, Haltestellen usw. OTP kann ohne .osm file ausgeführt werden. Das Programm erkennt automatisch ob ein OSM-File vorhanden ist. Jedoch werden dann keine Fusswege mit einberechnet um zu einer Haltestelle zu gelangen, wie man in Abbildung 12 erkennen kann. Aber wenn OTP mit dem .osm File ausgeführt wird lässt sich anhand der Informationen einen Gehweg zu der Haltestelle finden (inklusive Laufzeit und Distanz), wie man in Abbildung 13 erkennen kann.

⁴<https://www.openstreetmap.org/>

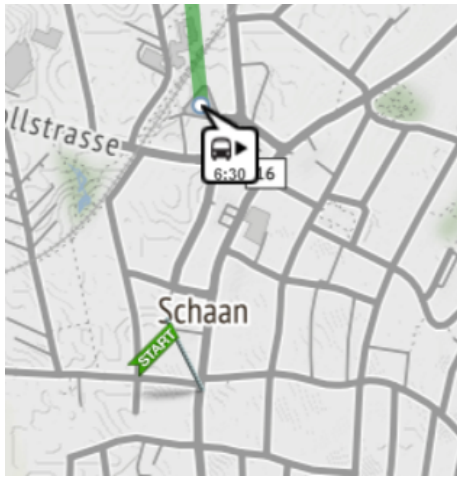


Abbildung 12: ohne .osm File

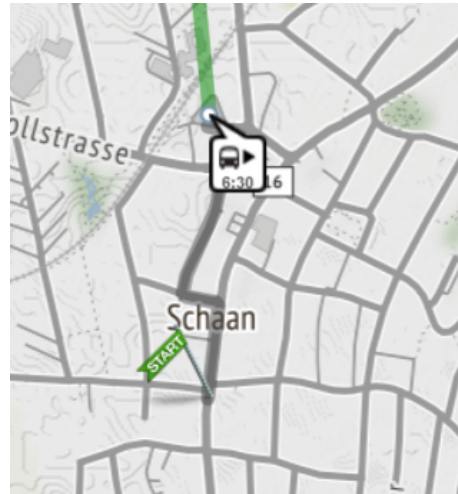


Abbildung 13: mit .osm File

8 Resultate

Die Tests mit den gesamtschweizerischen GTFS-Daten mit dem JUnit-Benchmark führten zu folgenden Ergebnissen. Ein Request benötigt ca. 24h um bearbeitet zu werden. Somit ist der Algorithmus um den Faktor 10000 langsamer als der Originalalgorithmus des OTP, welcher für die selbe Anfrage nur wenige Sekunden benötigt.

Die Tests mit den PortlandGTFS-Daten mit der Funktion `System.currentTimeMillis()` führten zu folgenden Ergebnissen. Ein Request kann in einem Zeitfenster zwischen 1 - 9 Minuten bearbeitet werden. Diese Zeitspanne hängt davon ab, nach wie vielen Schleifendurchläufen eine Lösung gefunden wurde und somit das Break-Kriterium greift. Damit ist der Request um den Faktor 1000 langsamer als der Originalalgorithmus.

Für einen Schleifendurchlauf gibt es zwei mögliche Zeiten. Ist die in der Schleife behandelte Connection nicht relevant, so werden diese nicht bearbeitet. Dabei wird eine Zeit von 1ms benötigt. Wenn die Connection jedoch relevant ist, so wird eine Zeit von 15ms benötigt.

9 Fazit

Unsere Implementation des CSA im OTP ist funktionsfähig. Die von uns angestrebte Performance konnte jedoch nicht erreicht werden. Unsere Implementation ist mit den Schweizer GTFS-Daten um den Faktor 10000 langsamer als der Originalalgorithmus des OTP.

Der grösste Performanceverlust rührt von unserer implementation Der Stop- und Trip-Handler. Die Handler haben zwar eine Referenz auf das ihnen zugehörige Objekt, jedoch hat das Objekt keine Referenz zum Handler. Da in unserem Algorithmus jedoch mit den Objekten gearbeitet wird und dann im zum Objekt gehörigen Handler Informationen abgespeichert werden müssen, muss jedes mal mit einer Suchschleife der richtige Handler gefunden werden. Dies führt dazu, dass in einem Schleifendurchlauf des Algorithmus acht Suchschleifen durchlaufen werden, welche jeweils auch grosse Datensätze durchsuchen müssen.

Um dieses Problem zu lösen müssten die Referenzen gedreht werden, so dass die Objekte auf ihre jeweiligen Handler referenzieren. Somit lassen sich die Suchschleifen vermeiden. Aus zeitlichen Gründen konnten wir diese Optimierung jedoch nicht mehr durchführen.

Abbildungsverzeichnis

1	Dieses Bild gibt eine Übersicht welche Klassen in unserem Paket vorhanden ist.	8
2	Hier ist das UML-Diagramm zur TimeTable Datenstruktur zu sehen.	10
3	Symbolbild für den Fussweg(FootpathCSA)	10
4	Symbolbild für die Haltestelle(StopCSA) [8]	11
5	Symbolbild für die Verbindung(ConnectionCSA)	11
6	Symbolbild für die Fahrt(TripCSA)	12
7	Hier ist das UML-Diagramm zur Journey Datenstruktur zu sehen. .	13
8	Programmablauf des OTP mit dem CSA	13
9	Darstellung der Kommunikation zwischen Clients und Server [9] . .	14
10	Das Bild zeigt welche Sammlungen wir in der Zeittafel zur Verfügung stehen.	18
11	Dieses Bild gibt eine Übersicht welche Haltestellen und Routen im Dummy-GTFS vorhanden ist.	19
12	ohne .osm File	27
13	mit .osm File	27

Literatur

- [1] “Opentripplanner,” Jul. 2018. [Online]. Available: <http://www.opentripplanner.org/>
- [2] “Dijkstra’s algorithm,” Oct. 2013. [Online]. Available: <http://www.cse.unt.edu/~tarau/teaching/AnAlgo>
- [3] R. Geisberger, “Advanced route planning in transportation networks,” Feb. 2011. [Online]. Available: <http://algo2.iti.kit.edu/1814.php>
- [4] J. Dibbelt, T. Pajor, B. Strasser, and D. Wagner, “Connection scan algorithm,” *CoRR*, vol. abs/1703.05997, 2017. [Online]. Available: <http://arxiv.org/abs/1703.05997>
- [5] “Google,” Jul. 2018. [Online]. Available: <https://developers.google.com/transit/gtfs/reference/>
- [6] Gtfs-opentransportdata. [Online]. Available: <https://opentransportdata.swiss/de/cookbook/gtfs/>
- [7] “Google,” Jul. 2018. [Online]. Available: <https://developers.google.com/transit/gtfs-realtime/guides/feed-entities>
- [8] “onlinewebfonts,” Jul. 2018. [Online]. Available: <https://www.onlinewebfonts.com/icon/10491>
- [9] “1und1,” Jul. 2018. [Online]. Available: <https://hosting.1und1.de/digitalguide/server/knowhow/was-ist-ein-server-ein-begriff-zwei-definitionen/>