BACHELORARBEIT

Maximilian Rieder

Portierung eines Routenoptimierers von R nach Java

9. Oktober 2020

Fakultät:	Informatik und Mathematik
Studiengang:	Bachelor Informatik
Abgabefrist:	10. Oktober 2020
Betreuung:	Prof. Dr. Jan Dünnweber
Zweitbegutachtung:	Prof. Dr. Stefan Körkel

Zusammenfassung

In dieser Arbeit wird ein Routenoptimierer implementiert. Die mathematischen Grundlagen hierzu wurden in der Masterarbeit “Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg” von Helena Huber behandelt und als R Skript umgesetzt. Aufbauend auf den selben mathematischen Grundlagen, wird ein Routenoptimierer in Java umgesetzt. Dieser kann unterschiedliche Datensätze zur Berechnung der Routen verwenden. Zum einen sind dies die Daten für den Bereich Regensburg aus der Schnittstelle des Verbands Deutscher Verkehrsunternehmen, zum anderen interne Daten des Regensburger Verkehrsverbunds. Dafür wird eine Datanstruktur implementiert, die von den Rohdaten entkoppelt ist. Somit ist es möglich, dass unterschiedliche Datensätze vom gleichen Algorithmus verarbeitet werden. Bei der Abfrage des zeitlich kürzesten Wegs werden auch Verbindungen des nächsten Tages miteinbezogen, falls es nicht möglich ist, die gewählte Haltestelle am selben Tag zu erreichen. Zusätzlich wird eine REST-Schnittstelle umgesetzt, um die berechneten Routen auch im Rahmen von verteilen Systemen abfragen zu können.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Auch in der heutigen Zeit greifen immer mehr Menschen auf den Öffentlichen Personen Nahverkehr (ÖPNV) zu. Hierbei sind die Zahlen der Passagiere seit dem Jahr 2004 kontinuierlich gestiegen.[Zeit 2020] Vor allem im Zuge der Digitalisierung ist es für die Nutzer des ÖPNV äußerst wichtig, geeignete Streckenverbindungen nachschlagen zu können, mit welchen man das gewünschte Ziel zum frühest möglichen Ankunftszeitpunkt erreichen kann. Dafür ist eine Routenplanungssoftware zur Routenoptimierung unerlässlich.

1.2 Ziel der Arbeit

Die mathematischen Grundlagen, auf denen dieses Projekt beruht, wurden größtenteils in der Masterarbeit *Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg* betrachtet, wobei auch ein Skript in der Programmiersprache R erstellt wurde, das eine Lösung für die Passagierroutenoptimierung liefert.
Ziel dieser Arbeit ist es, eine solche Routenplanungssoftware zur Abfrage des zeitlich kürzesten Wegs aufbauend auf den selben mathematischen Grundlagen, die in der Masterarbeit dargestellt wurden, nun in Java zu implementieren, da hiermit für eine bessere Interoperabilität mit anderen Modulen gesorgt wird.
Dabei soll die Routenoptimierung für unterschiedliche Datensätze umgesetzt werden, welche jedoch vom gleichen Algorithmus verarbeitet werden sollen. Bei der Abfrage des zeitlich kürzesten Wegs sollen auch Verbindungen des nächsten Tages miteinbezogen werden, falls es nicht möglich ist, die gewählte Haltestelle am selben Tag zu erreichen. Zusätzlich soll eine Programmierschnittstelle umgesetzt werden, über welche die Anfragen einer Berechnung des zeitlich kürzesten Wegs, zwischen zwei Haltestellen, an einem bestimmten Zeitpunkt im Rahmen von verteilten Systemen abgefragt werden kann.

1.3 Aufbau

In Kapitel 2 werden die mathematischen Grundlagen einer geeigneten Datenstruktur, welche die Daten des ÖPNV Regensburgs umsetzen soll, sowie eines Algorithmus, der eine Lösung für den zeitlich kürzesten Weg liefert, beschrieben.

Anschließend wird im Hauptteil (Kapitel 3) auf die Implementierung eingegangen. Kapitel 4 befasst sich mit dem Testen des erstellten Programms und schlussendlich wird in Kapitel 5 ein Fazit über die Arbeit gezogen.

2 Grundlagen

Im Folgenden wird auf die theoretischen Grundlagen eingegangen, auf denen das Projekt basiert.
Dabei wird von einem Haltepunkt gesprochen, wenn es sich um den direkten Ort handelt, an dem eine Linie anhält. Eine Haltestelle hingegen fasst einen oder mehrere Haltepunkte zusammen, da eine Haltestelle unterschiedliche Haltepunkte beinhalten kann, beispielsweise auf beiden Straßenseiten.

2.1 Earliest Arrival Problem

Die grundlegende Problemstellung ist die Suche nach dem zeitlich kürzesten Weg, um im Netz des öffentlichen Personennahverkehrs (ÖPNV) Regensburgs von einer Haltestelle zu einer Weiteren und das zu einer bestimmten Uhrzeit zu gelangen.
Dies entspricht dem Earliest Arrival Problem (EAP), also der Suche nach dem frühesten Ankunftszeitpunkt.

2.2 Modellierung des Graphen

Um die Haltestellen und Linienverbindungen in einem mathematischen Modell umzusetzen, soll ein Graph verwendet werden.
Als Kantengewicht wird die erforderliche Zeit, um von einem Haltepunkt zu einem Anderen zu gelangen, gewählt. [Huber 2019, S. 26]
Die gängigen Modelle, Linienverkehrsnetze mathematisch darzustellen, sind hierbei zum einen der zeitexpandierte Graph und zum anderen der zeitabhängige Graph. [Pajor 2009]
Beim zeitexpandierten Graph werden den Knoten keine Haltestellen zugeordnet, sondern Ereignisse, genauer gesagt Abfahrts- und Ankunftsereignisse. Diese werden so verbunden, dass Kanten zwischen aufeinanderfolgenden Abfahrts- und Ankunftsereignissen einer Linie entstehen. Zusätzlich werden Kanten zwischen Ankunfts- und Abfahrtsereignissen innerhalb einer Haltestelle eingefügt, falls die Ankunftszeit kleiner als die Abfahrtszeit ist. Damit sind alle Linienverläufe und Umstiegsmöglichkeiten abgedeckt. Die Zeit, die für einen Wechsel der Linie benötigt wird, kann durch einen weiteren Knoten modelliert werden. Dieser wird zwischen Ankunftsereignissen und Abfahrtsereignissen einer Station eingefügt,

die nicht der selben Linie angehören, und enthält die Summe aus Ankunftszeit und Umstiegszeit. [Pajor 2009, S. 19-21]
Der Aufbau des Graphen wird exemplarisch in Abbildung 1 dargestellt.

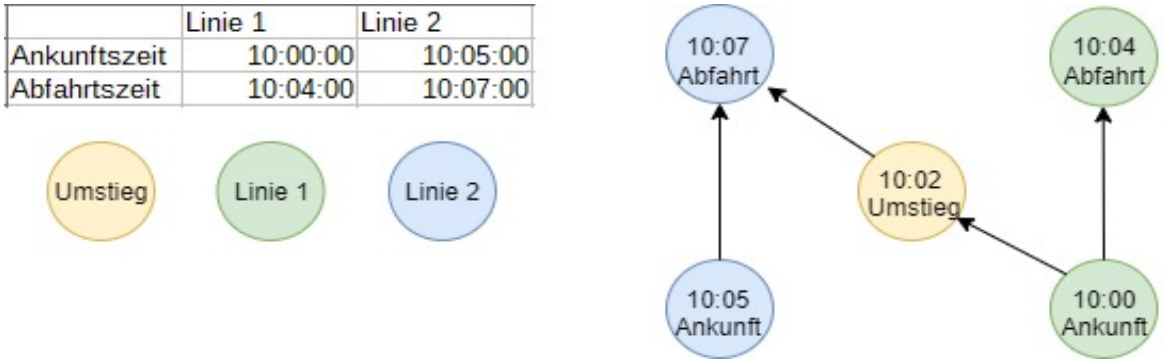


Abbildung 1: Beispielhafte Darstellung eines zeitexpandierten Graphen

Diese Art von Graph wird mit zunehmender Anzahl der Haltestellen und Fahrten jedoch sehr groß und benötigt einen hohen Speicherbedarf.[Schulz 2005] Daraus resultiert auch, dass Algorithmen, die auf diesen angewandt werden, sehr viele Entscheidungsmöglichkeiten haben, was sich negativ auf die Laufzeit der Verarbeitungsvorschriften auswirkt. [Pajor 2009, S.38]
Daher wird das Modell des zeitabhängigen Graphen in dieser Arbeit verwendet, um den ÖPNV zu modellieren. Hierbei wird für jeden Haltepunkt ein Knoten erstellt und Kanten zwischen jenen eingefügt, welche direkt durch eine Linie verbunden sind. Im Gegensatz zu den statischen Kantengewichten des zeitexpandierten Graphen werden hier dynamische Kantengewichte in Abhängigkeit eines beliebigen Zeitpunktes verwendet. Das heißt, die Kanten zwischen den einzelnen Haltepunkten haben unterschiedliche Kantengewichte, je nachdem zu welcher Tageszeit diese betrachtet werden. Dies wird durch eine Gewichtsfunktion realisiert, die die Dauer, eine Kante zu einem bestimmten Zeitpunkt zu durchlaufen, berechnet. Um die Umstiegsmöglichkeiten beziehungsweise -zeiten in dieses Modell einzubinden, wird für jede Haltestelle ein zentraler Knoten eingefügt. Dieser wird durch Transferkanten nur mit den Knoten dieser Haltestelle verbunden. Als Kantengewichte erhalten diejenigen, die von einem Haltepunktknoten zum Haltestellenknoten führen, ein Gewicht von null. Die Kanten in die entgegengesetzte Richtung erhalten als Gewicht die benötigte Umstiegszeit.[PSWZ 2004] Der Aufbau des zeitabhängigen Graphen wird in Abbildung 2 dargestellt.
Da man für einen Wechsel des Haltepunkts an einer Haltestelle den zentralen Haltestellenknoten benutzen muss, kann man zwar ohne Kosten zu diesem wechseln, jedoch wird die Umstiegszeit bei einem weiteren Wechsel zurück zu einem Haltepunkt miteinbezogen. Hierbei ist anzumerken, dass Umstiege, die am gleichen Haltepunkt stattfinden, von diesem mathematischen Modell nicht miteinbe-

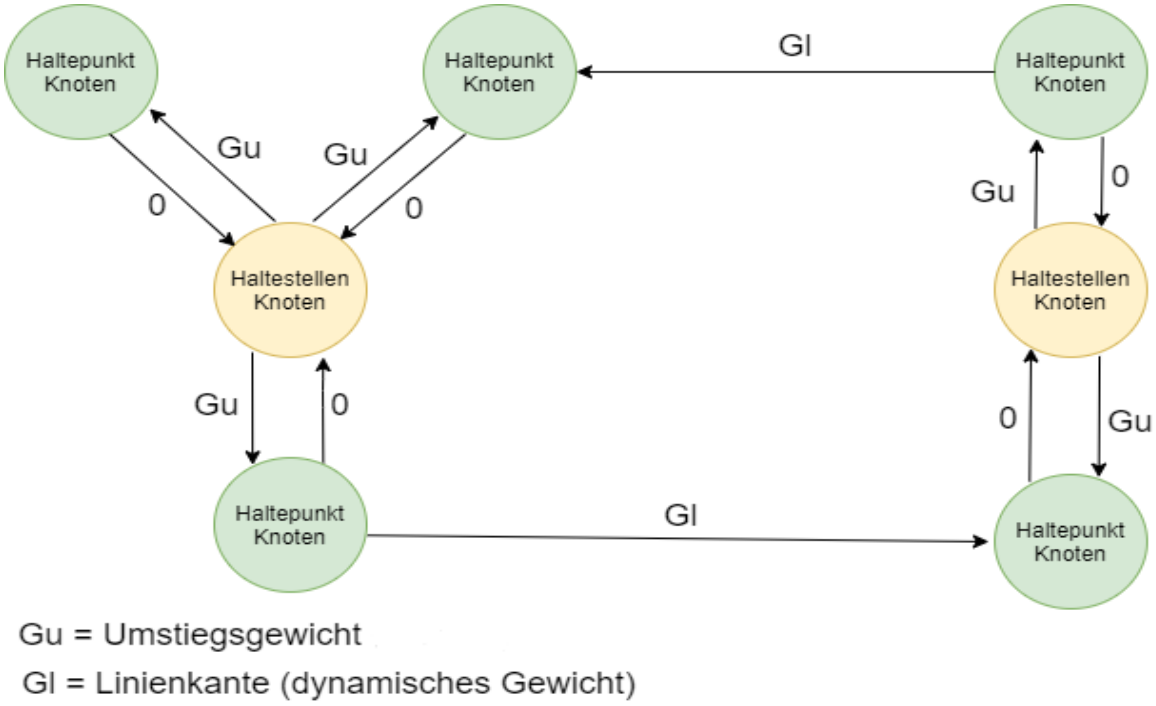


Abbildung 2: Beispielhafte Darstellung eines zeitabhängigen Graphen

zogen werden, da der zentrale Haltestellenknoten dafür nicht durchlaufen werden muss. Weil in der Realität der Wechsel einer Linie am gleichen Haltepunkt nur einige Sekunden dauert und die Fahrzeuge auch für das Ein- und Aussteigen der Passagiere eine kurze Zeit lang halten, können diese Umstiege vernachlässigt werden.
Im Rahmen der Masterarbeit *Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg* ergab sich die Erkenntnis, dass im Normalfall die Umstiegszeit im ÖPNV Regensbugs der Anzahl der Haltepunkte pro Haltestelle in Minuten entspricht. [Huber 2019, S.59]
Im Vergleich zum zeitexpandierten bietet der zeitabhängige Graph Algorithmen in allen Größen einen Vorteil, die das EAP auf Basis dieses Graphen lösen, was in Experimenten durch Pyrga et al. festgestellt wurde.[PSWZ 2007] Darum wird im weiteren Verlauf der Arbeit der zeitabhängige Graph verwendet.
Der zeitabhängige Graph unterliegt im Gegensatz zum zeitexpandierten Graph der First In First Out (FIFO) Beschränkung, welche besagt, dass es auf einer Linie zu keinen Überholvorgängen kommen darf. Damit darf es für keine Kante zwischen zwei Haltestellen dazu kommen, dass eine Verbindung existiert, die einen späteren Abfahrtszeitpunkt als eine andere Verbindung besitzt, jedoch den gleichen oder einen schnelleren Ankunftszeitpunkt. Ist die FIFO Bedingung verletzt ist die Lösung des EAP NP-schwer und damit nicht mehr sinnvoll durch den angepassten Dijkstra-Algorithmus lösbar.[Pajor 2009]

2.3 Verwendung und Anpassung des Dijkstra Algorithmus

Um das EAP auf einem zeitabhängigen Graphen zu lösen, lässt sich eine abgewandelte Form des *Dijkstra-Algorithmus* verwenden, welcher in Algorithmus 1 mithilfe von Pseudocode dargestellt wird.[BJ 2004]
Die Grundidee ist hierbei ähnlich zu der des ursprünglichen Dijkstra-Algorithmus. Als erstes wird die Zeitmarke *a* jedes Knotens *v* der gesamten Knotenmenge *V* mit der Wertigkeit Unendlich versehen (vgl. Zeile 3-5). Daraufhin wird die Zeitmarke des Startknotens *s* auf die Startzeit *t* gesetzt. Weiterhin wird eine *Vorrangwarteschlange Q* verwendet, um die abzuarbeitenden Knoten darin zu speichern. Diese benutzt als Schlüssel die aktuelle Zeitmarke des abgespeicherten Knotens. Der Startknoten wird dabei als erstes eingefügt (vgl. Zeile 8). Anschließend wird, solange *Q* nicht leer ist, immer wieder eine Iteration durchgeführt.
Innerhalb dieser wird ein Knoten *u* mit der aktuell kleinsten Zeitmarke aus der Warteschlange entfernt und einer Liste mit permanent abgearbeiteten Knoten *S* hinzugefügt (vgl. Zeile 10, 11). Falls es sich bei dem Knoten *u* um den Zielknoten *d* handelt wird die Schleife durch dieses Abbruchkriterium frühzeitig beendet, da somit der frühest mögliche Ankunftszeitpunkt gefunden wurde (vgl. Zeile 12-14). Andernfalls wird nun für die Menge an Kanten, die vom aktuellen Knoten *u* ausgehen, eine weitere Iteration durchgeführt (vgl. Zeile 15). Der Hauptunterschied zum originalen Dijkstra-Algorithmus besteht darin, dass statt einer Distanzfunktion hier eine Funktion f_e verwendet wird, die anhand der Kante *uv* und der aktuellen Zeitmarke von *v* einen Zeitpunkt $t\prime$ ermittelt (vgl. Zeile 16). Dieser entspricht dem frühest möglichen Zeitpunkt den Knoten *v* über die Kante *uv* zu erreichen. Hierbei werden nur Ergebnisse betrachtet, deren Ankunftszeitpunkt abzüglich des dazugehörigen Kantengewichts nicht vor dem Abfahrtszeitpunkt liegen. Dies soll in dieser Arbeit wiederum durch entsprechende Selektionen auf eine Fahrzeittabelle mit den Daten für den ÖPNV Regensburg geschehen. Falls die Zeitmarke $t\prime$ kleiner ist, als die aktuelle Zeitmarke von *v*, wurde ein zeitlich kürzerer Weg zu Knoten *v* gefunden. Daher wird *v*, falls nicht vorhanden, in *Q* eingefügt oder der gegenwärtige Schlüssel aktualisiert, da alle Verbindungen, die von *v* ausgehen nochmals überarbeitet werden müssen. Zusätzlich wird die Zeitmarke von Knoten *v* und dessen Vorgängerknoten überarbeitet (vgl. Zeile 17-24).
Schließlich wird vom Zielknoten ausgehend über die zuvor gesetzten Vorgängerknoten der zeitlich festgesetzte Pfad *p* erzeugt und zurückgegeben (vgl. Zeile 28-34).

Algorithmus 1 Angepasster Dijkstra-Algorithmus

1: **statischer Input:** Graph $G = (V, E)$, für jede Kante $e \in E$ eine nicht negative monotone Gewichtsfunktion $f_e : T \to T$, wobei es sich bei T um eine linear geordnete Zeitdomäne handelt (bspw. Integer)
2: **dynamischer Input:** Startknoten $s \in V$, Zielknoten $d \in V$, Startzeit $t \in T$
3: **for** $v \in V$ **do**
4: $a[v] = \infty$
5: **end for**
6: $a[s] = t$
7: $S = empty$
8: $Q = (s, t)$
9: **while** $Q \neq empty$ **do**
10: $u = Q.extractMin()$
11: $S.add(u)$
12: **if** $u == d$ **then**
13: $break$
14: **end if**
15: **for each** $e(u, v) \in E$ mit $e \notin S$ **do**
16: $t\prime = f_{uv}(a[u])$
17: **if** $a[v] > t\prime$ **then**
18: **if** $Q.contains(v)$ **then**
19: $Q.update(v, t\prime)$
20: **else**
21: $Q.add(v, t\prime)$
22: **end if**
23: $a[v] = t\prime$
24: $v.setPredecessor(u)$
25: **end if**
26: **end for**
27: **end while**
28: $v = d$
29: $p = (v, a[v]).p$
30: **while** $v \neq s$ **do**
31: $v = v.getPredecessor()$
32: $p = (v, a[v]).p$
33: **end while**
34: **Output:** p

3 Portierung und Erweiterung der Ergebnisse der Masterarbeit

3.1 Daten- / Schnittstellenbeschreibung

3.1.1 VDV - Schnittstelle

Wie in der zugrunde liegenden Masterarbeit, soll eine einheitliche Schnittstelle die Bedienung und die Vernetzung mit weiteren Projekten gewährleisten.
Als grundlegende Daten, die zur Berechnung des kürzesten Wegs dienen, werden zunächst die selben Daten verwendet, die Helena Huber für ihre Masterarbeit *Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg* vom Regensburger Stadtwerk bereitgestellt bekam (Stand 28.01.2019).
[Huber 2019, S.50] Diese enthalten die grundlegenden Informationen über die Fahrtverläufe und Haltestellen für den öffentlicher Personennahverkehr (ÖPNV) im Raum Regensburg.
Weiterhin sind diese Daten entsprechend der VDV-Schnittstelle formatiert, bei welcher es sich um eine vom Verband Deutscher Verkehrsunternehmen (VDV) herausgegebene Schrift handelt. Diese beschreibt die genaue Anwendung des ÖPNV-Datenmodells, um eine einheitliche Standardisierung zu gewährleisten. [VDV 2013, S. 9] Dies soll unter Anderem zu einer Verringerung von Individualschnittstellen und einer Vereinheitlichung von Datenfeldern und Dokumentationen führen.[VDV 2013, S. 11]
Um die Rohdaten optimal nutzen zu können, wurden diese in CSV Dateien umgewandelt und um überflüssige Einträge, wie beispielsweise Header gekürzt. [Huber 2019, S.56]
Soweit nicht anders erwähnt, gehen die folgenden Informationen über die Daten der VDV Schnittstelle aus der Schnittstellendokumentation zur VDV-Standardschnittstelle Liniennnetz/Fahrplan hervor. [VDV 2013]
Die notwendigen Tabellen sind hierbei *REC_ORT*, *REC_FRT*, *LID_VERLAUF* und *SEL_FZT_FELD*.

REC_ORT enthält Informationen zu den Haltestellen beziehungsweise -punkten und enthält folgende Spalten:

- ONR_TYP_NR (Ortstyp des Haltepunktes)
- ORT_NR (Kennnummer des jeweiligen Haltepunkts)
- ORT_REF_ ORT (Kennummer der Haltestelle)
- ORT_REF_ORT_NAME (Name der Haltestelle)

REC_FRT enthält Daten bezüglich des Fahrplans, von welchen folgende für dieses Projekt notwendig sind:

- FRT_START (Uhrzeit des Linienstarts in Sekunden ab 00:00),
- LI_NR (Linien-Nummer)
- FGR_NR (Fahrzeitgruppe)
- TAGESART_NR (Kennummer für die Tagesart)
- FAHRTART_NR (Kennummer für die Fahrtart)

Weiterhin enthält *SEL_FZT_FELD* Informationen über die Fahrzeitdaten in diesen Spalten:

- FGR_NR (Fahrzeitgruppe)
- ORT_NR (Kennnummer des Start-Haltepunkts)
- SEL_ZIEL (Kennnummer des Ziel-Haltepunkts)
- SEL_FZT (Fahrzeit zwischen Start-Haltepunkt und Ziel-Haltepunkts)
- ONR_TYP_NR (Ortstyp des Start-Haltepunkts)
- SEL_ZIEL_TYP (Ortstyp des Ziel-Haltepunkts)

3.1.2 RVV Testdaten

Im Verlauf des Projekts wurden weitere Daten im Zuge des AVL-Verbund Projekts vom RVV für diese Arbeit zur Verfügung gestellt. Da diese Daten lediglich nur zur internen Verwendung gedacht sind, gibt es keine verfügbare Dokumentation für die Öffentlichkeit. Durch den Mangel an klaren Vorgaben, kann die Datenqualität nicht garantiert werden. Die im Rahmen des Projekts gemachten Annahmen, beziehen sich lediglich auf den Datensatz vom Stand des 22.07.2020 und sollten daher als Provisorium angesehen werden. Im weiteren Verlauf der Arbeit wird dieser aktuelle Datensatz als **Testdaten** referenziert.
Die Daten wurden als mehrere Textdateien (Stand 22.07.2020) übergeben und zur Weiterverarbeitung in CSV-Dateien umgewandelt. Im Folgenden sollen die im weiteren Verlauf verwendeten Daten kurz erläutert werden.
Aus der Tabelle *stops* wurden die Spalte *stop_name*, die Auskunft über den Haltestellennamen gibt, und *stop_id* verwendet. Diese verweist als Kennnummer auf einen Haltepunkt und ist durch Doppelpunkte in mehrere Abschnitte gegliedert. Durch manuelle Analyse der Daten wurden folgende Annahmen getroffen:

- erster Abschnitt: irrelevanter Bezeichner (hier nur: “de”/“gen”)
- zweiter und dritter Abschnitt: einzigartige Kennnummer für Haltestellen
- restliche Abschnitte: Unterscheidungskriterium für die Haltepunkte jeder Haltestelle
- gelegentlich entspricht der letzte Abschnitt “N”, “S” oder “G”: weiteres Unterscheidungskriterium für die Haltepunkte einer Haltestelle
- falls ID auf *_Parent* endet: Theoretischer Haltestellenpunkt (werden nicht angefahren)

Die These alle auf *_Parent* endenden Kennnummern würden keinen realen Haltepunkten entsprechen, ergibt sich daraus, dass diese in den weiteren Tabellen nicht mehr vorkommen und somit auch keine Linienverbindung zu diesen existiert.
Die Inhalte der Datei *stop_times* wurden wie folgt aufgefasst:

- trip_id: Kennnummer der haltenden Fahrt
- stop_id: Kennnummer des Haltepunkts
- arrival_time: Ankunft der Fahrt
- departure_time: Abfahrt der Fahrt

- stop_sequence: Reihenfolge der Haltepunkte einer Fahrt

Weiterhin enthält die Datei *calendar_dates* eine *service_id* und eine Spalte *date*. Da der *exception_type* immer auf eins gesetzt ist, wurde er hier vernachlässigt . Es wird angenommen, dass durch diese Tabelle definiert wird, welche *service_ids* an welchem Tag im Einsatz sind. Das Format, in dem sich *date* befindet, entspricht einer sechsstelligen ganzen Zahl, bei welcher die ersten vier Ziffern das Jahr, die nächsten zwei den Monat und die letzten zwei den Tag angeben.
Eine Tabelle, die im weiteren Verlauf verwendet wird, ist *trips*. Hierbei werden lediglich die Spalten *trip_id* und *route_id* benötigt. Die Tabelle wird dazu genutzt, um von einer *trip_id* auf eine Route zu schließen.
Informationen über Routen befinden sich in der Datei *routes*. Benutzt werden hierbei nur die enthaltenen Spalten *route_id* als Kennnummer für die Routen und *route_short_name*, was den Liniennummern entspricht.

3.2 Datenverarbeitung

Um mit den jeweiligen Daten aus den Comma-separated values (CSV) Dateien in Java arbeiten zu können, bieten sich Dataframes an. Die Tabellen, die für dieses Projekt benötigt werden, können nämlich aufgrund ihrer geringen Größe einfach im Arbeitsspeicher gehalten werden. Beispielsweise besitzen die Daten aus der VDV-Schnittstellenbeschreibung, die im Programmablauf Verwendung finden, zusammen eine Größe von circa eineinhalb Megabyte. Die Testdaten des RVV sind in etwa fünfzehn Megabyte groß. Daher reicht es die Tabellen als CSV-Dateien abzuspeichern und beim Programmstart als Dataframe in den Arbeitsspeicher zu laden. Weiterhin stellen Dataframes einfach zu verwendende Methoden zur Datenmanipulation zur Verfügung. Damit muss kein SQL verwendet werden, was die Lesbarkeit und den Komfort Code zu schreiben oder abzuändern erhöht.
Als geeignete Bibliothek wählte ich *tablesaw*, da diese alle benötigten Methoden bereitstellt und aktiv entwickelt und gewartet wird.[TablesawDoku2020] Tabellen in dieser Bibliothek werden dabei als Dataframe umgesetzt.

3.2.1 Daten Ein- und Ausgabe

Im Rahmen dieses Projekts wurde festgestellt, dass die folgenden Tabellen zur Erstellung des Graphen, beziehungsweise zur Durchführung des angepassten Dijkstra-Algorithmus benötigt werden.
Dabei handelt es sich zum einen um eine Tabelle, die eine genaue Auskunft enthält, zu welcher Uhrzeit man mit welcher Linie von einem Haltepunkt zum nächsten gelangt. Hierbei sollen neben Reisezeit auch Ankunft und Abfahrt enthalten sein. Diese Tabelle wird im weiteren Verlauf der Arbeit als **Fahrzeitplan** (*rideTimetable*/*rideTimetableNextDay*) referenziert. Sie findet hauptsächlich Verwendung in der Funktion zur Berechnung der Ankunftszeitpunkte beim Durchlaufen einer Kante innerhalb des angepassten Dijkstra-Algorithmus (vgl. Kapitel 2.3). Da die Abfragen auf diese Tabelle auch den nächsten Tag betreffen können, wird diese in zweifacher Ausführung, jeweils für den aktuellen und für den folgenden Tag, benötigt. Diese Tabelle ist beispielhaft in Abbildung 3 dargestellt.
Zusätzlich soll eine Tabelle entstehen die die Kennnummer zu den Haltestellen und ihren jeweiligen Haltepunkten, wie auch den jeweiligen Ortsnamen, enthalten. Damit sollen Abfragen ermöglicht werden, wie zum Beispiel welche Haltepunkte zu welcher Haltestelle gehören oder wie der zugehörige Haltestellenname lautet. Diese Tabelle wird zukünftig **Ortsinformationen** (*locationInfo*) genannt. Ihr Aufbau wird exemplarisch in Abbildung 4 skizziert.

Haltepunkt Kennnummer (Von)	Haltepunkt Kennnummer (Nach)	Fahrzeit	Liniennummer	Abfahrt	Ankunft
501500	500500	60	1	18540	18600
500500	500300	60	1	18600	18660
500300	500200	60	1	18660	18720
...	...	...	...	...	...
403002	404103	120	66	73260	73320

Abbildung 3: Beispielhafte Darstellung der Tabelle des Fahrzeitplans

Als letzte Tabelle wird eine Auflistung der Haltestellen-Kennnummern und den

Haltestellen Kennnummer	Haltepunkt Kennnummer	Haltestellenname
1001	100100	Altes Rathaus
1010	101001	Arnulfsplatz
1010	101002	Arnulfsplatz

Abbildung 4: Beispielhafte Darstellung der Tabelle der Ortsinformationen

zugehörigen Umstiegszeiten benötigt, welche weiterhin als **Transferplan** (*transferTimetable*) bezeichnet wird. Mithilfe dieser lassen sich die Umstiegszeiten an den einzelnen Haltestellen herausfinden. Die Struktur dieser Tabelle ist beispielhaft in Abbildung 5 dargestellt.
Zeiten wie beispielsweise Abfahrts- oder Ankunftzeiten werden dabei als Sekun-

Haltepunkt Kennnummer	Umstiegszeit
8008	1
8010	1
1090	3

Abbildung 5: Beispielhafte Darstellung der Tabelle des Transferplans

den nach null Uhr angegeben. Die einzige Ausnahme ist hierbei die Umstiegszeit, welche in Minuten angegeben wird, da über diese keine genaueren Abschätzungen im Sekundenbereich getroffen werden.
Schließlich sollen alle Tabellen in dem Objekt *RoutingDataProcessed* vereint werden.
Durch diese Tabelle ist der angepasste Dijkstra-Algorithmus nicht mehr direkt an

die Rohdaten gekoppelt, was die Integration verschiedener Datensätze ermöglicht. Diese Datensätze müssen lediglich in die eben beschriebene Form gebracht werden.
Um ein RoutingDataProcessed-Objekt zu erstellen, wurde die Klasse *DataInOut* innerhalb dieses Projekts entwickelt. Die Funktion dieser Klasse ist es, aus einem der jeweiligen Datensätze aus Kapitel 3.1.1 und 3.1.2 durch Vorverarbeitung ein *RoutingDataProcessed*-Objekt zu generieren und die dafür erstellten Tabellen als CSV Dateien in einem Unterordner abzuspeichern. Wurden die Daten bereits vorverarbeitet und entsprechend als CSV-Dateien abgespeichert, sollen diese direkt eingelesen und in ein *RoutingDataProcessed*-Objekt umgewandelt werden, um unnötige Verarbeitungszeit einzusparen. Dieser Ablauf wird in Abbildung 6 dargestellt.
Eine Instanz der Klasse *DataInOut* kann unter Angabe des boolschen Werts *use-*

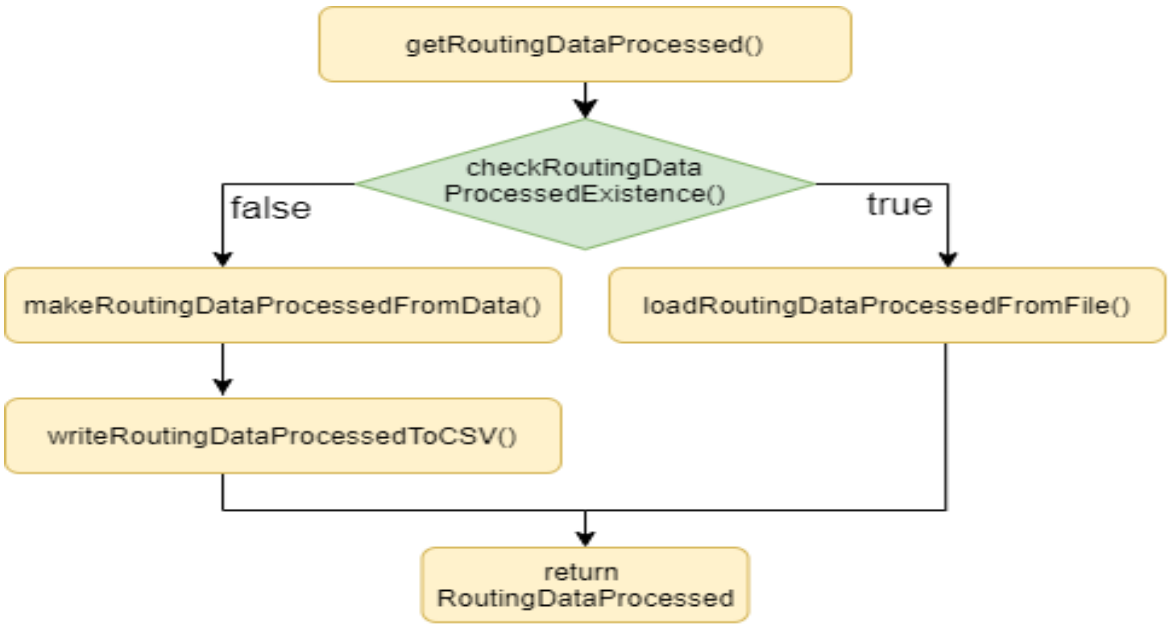


Abbildung 6: Darstellung des Ablaufs der Erstellung von RoutingDataProcessed

TestData erstellt werden, welcher Auskunft darüber gibt, ob die Testdaten oder die Standard-Daten der VDV-Schnittstellenbeschreibung genutzt werden sollen. Weiterhin wird durch diesen Wert entschieden, welchen Ordner beziehungsweise Dateinamen die im weiteren Verlauf beschriebenen Methoden verwenden sollen. Dies geschieht über eine *getPreString* Funktion, welche im Fall, dass *useTestData* true ist, den String “TestData” und im false-Fall den String “StandardData” zur Verfügung stellt.
Um *RoutingDataProcessed* zu erhalten, wurde im Zuge dieses Projekts die Methode *getRoutingDataProcessed* erstellt. Diese erhält als Argument den Tag, für welchen das *RoutingDataProcessed*-Objekt generiert werden soll, und entscheidet

durch die Abfrage der Methode *checkRoutingDataProcessedExistence*, ob nun *makeRoutingDataProcessedFromData* oder *loadRoutingDataProcessedFromFile* aufgerufen wird.
Die Methode *checkRoutingDataProcessedExistence* nutzt dabei die *exists*-Funktion der *File*-Klasse aus dem *Java.io* Paket, um die Existenz des Unterordners beziehungsweise der benötigten CSV-Dateien zu prüfen. Liefert eine dieser Abfragen einen *false*-Wert, gibt die ganze Methode *false* zurück.
Liegen die Tabellen aus *RoutingDataProcessed* bereits als CSV Datei vor, wird die Methode *loadRoutingDataProcesssedFromFile* ausgeführt.
Um eine CSV-Datei in eine *tablesaw* Tabelle umzuwandeln, wurde im Rahmen dieses Projekts zusätzlich die Funktion *getTableFromCSV* erstellt. Diese liest unter Angabe des Dateinamens und Trennzeichens eine CSV-Datei mittels der Methode *read* der *tablesaw*-Tabellenklasse ein und gibt eine Tabelle zurück.
Die Methode *getTableFromCSV* wird nun wiederum in *loadRoutingDataProcessed* verwendet, um die CSV-Dateien einzulesen und daraus ein *RoutingDataProcessed*-Objekt zu erstellen und zurückzugeben. Da *tablesaw* beim Einlesen einer *CSV*-Datei durch Heuristiken den Spalten-Typ festsetzt, kann es passieren, dass dieser nachkorrigiert werden muss, falls dieser nicht wie gewünscht erkannt wird.[TablesawGuide 2020] Beispielsweise muss in *getTableFromCSV* die Spalte für die Liniennummern (*LI_NR*) nach dem Einlesen der Dateien in eine String-basierte Spalte umgewandelt werden, weil eine Tabelle möglicherweise nur ganze Zahlen als Liniennummern enthält und *tablesaw* somit von einer Integer-basierten Spalte ausgeht. Ein anderes Datenset enthält hingegen Linienvarianten wie beispielsweise "10A", welche nur als String abgespeichert werden soll. Daraus resultiert die Verwendung einer String-basierten Spalte für die Liniennummern.
Wurden die Rohdaten noch nicht vorverarbeitet, wird dies durch die Methode *makeRoutingDataProcessedFromData* veranlasst. Dabei wird in Abhängigkeit des *DataInOut* Attributs *useTestData* entschieden, welches Datenset verwendet werden soll. Je nachdem wird nun *getRawRoutingDataStandardData* oder *getRawRoutingDataTestData* aufgerufen. Diese zwei Methoden lesen die jeweiligen CSV-Datensets mit der bereits erwähnten *getTableFromCSV* Funktion ein und geben die dabei erstellten Tabellen entweder als *RoutingDataStandardData* oder *RoutingDataTestData* zurück. Anschließend werden die Tabellen der Rohdaten unter Angabe des zuvor erwähnten Tag-Arguments der *getRoutingDataProcessed*-Methode entweder durch eine Instanz der Klasse *DataPreProcessorStandardData* oder *DataPreProcessorTestData* vorverarbeitet. Diese Klassen wurden im Rahmen des Projekts erstellt und werden in Kapitel 3.2.4 und 3.2.5 genauer erläutert. Als Rückgabe liefern sie ein *RoutingDataProcessed*-Objekt, das von der Methode *makeRoutingDataProcessedFromData* zurückgegeben wird.
Bevor dies passiert, werden die Tabellen des *RoutingDataProcessed*-Objekt mit-

hilfe der Funktion *writeRoutingDataProcessedToCSV* als CSV-Dateien abgespeichert, um die Vorverarbeitung beim nächsten Programmaufruf nicht erneut durchführen zu müssen. *writeRoutingDataProcessedToCSV* verwendet hierbei die Funktion *writeTableToCSV*, welche mithilfe der *tablesaw* Klasse *CsvWriter* eine Tabelle als CSV-Datei abspeichert. Diese zwei Funktionen wurden ebenfalls während der Entwicklung der *DataInOut* Klasse erstellt.
Schlussendlich wurde mit *writeNameIdentFile* für die Klasse *DataInOut* noch eine weitere Methode angefertigt. Diese speichert die Namen aller Haltestellen und deren zugehörige Kennnummern in einer Textdatei und nimmt als Argument die Ortsinformationen aus *RoutingDataProcessed* und den Namen unter dem die Datei gespeichert werden soll. Umgesetzt wird dies durch die Klasse *FileWriter* beziehungsweise *BufferedWriter* aus dem *Java.io* Paket. Die erstellte Datei dient dem Nutzer des Programms dazu, alle Haltestellen mit ihren zugehörigen Kennnummern nachzuschlagen, vor allem da die Ortsnamen gelegentlich nicht erwartungsgemäß geschrieben sind.

3.2.2 Datatransformer

Um mit Eingaben des Nutzers umzugehen oder bestimmte Werte, wie zum Beispiel Abfahrtszeiten, leserlich ausgeben zu können, wird eine Klasse benötigt, die die benötigten Werte in andere Formate umwandeln kann. Daher wurde im Zuge dieser Arbeit die Klasse *DataTransformer* erstellt.
Deren Aufgabe ist, zum einen die Umwandlung von Sekunden nach null Uhr in einen String, der eine Stunden-, Minuten- und Sekunden-Komponente besitzt, was in gleicher weise auch anders herum möglich sein soll. Zum anderen soll eine Methode den Haltestellennamen unter Angabe der zugehörigen Kennnummer beziehungsweise umgekehrt ausgeben können. Damit hierzu Abfragen bezüglich den Haltestellen durchgeführt werden können, muss dem Konstruktor der *DataTransformer*-Klasse die Tabelle mit Ortsinformationen übergeben werden.
Der String, der die Uhrzeit durch Stunden, Minuten und Sekunden beschreibt, kann durch die Funktion *makeTimeFromSeconds* erstellt werden. Dabei werden die Sekunden nach null Uhr modulo sechzig gerechnet, um die Sekunden zu erhalten, die nicht durch Stunden oder Minuten abgedeckt werden können. Anschließend werden die Sekunden, welche als Integer abgespeichert sind, durch sechzig geteilt, um die Anzahl an Minuten ohne die Rest-Sekunden zu bekommen. Die Minuten werden wiederum modulo sechzig gerechnet, um die restlichen Minuten zu erhalten. Daraufhin werden die Minuten durch sechzig geteilt und man erhält somit die Stunden, abzüglich der zuvor errechneten übrigen Minuten und Sekunden. Da es sich bei den beschriebenen Werten um Java-Integer handelt,

wird bei einer Division durch einen weiteren Integer der Rest abgeschnitten, wenn kein explizites *Typecasting* stattfindet. Schlussendlich werden Stunden, Minuten und Sekunden zusammengesetzt und durch Doppelpunkte getrennt zurückgegeben.
Um aus einem solchen String die Sekunden zu errechnen, wurde im Zuge dieser Arbeit die Funktion *makeSekondsFromTime* erstellt. Diese teilt den String mithilfe der String-Methode *split* an allen Doppelpunkten auf und speichert die Substrings in einem Array. Die einzelnen Teile werden als Integer geparst und der erste Eintrag, welcher den Stunden entspricht, mit 3600 multipliziert, was den Sekunden pro Stunde gleichkommt. Der Minutenteil wird respektiv mit 60 multipliziert. Letztendlich werden diese zusammen mit dem Sekundenteil aufaddiert und zurückgegeben.
Die Methode *getCSPIdentifierByName* der Klasse *DataTransformer* liefert unter Angabe des zugehörigen Namens die Kennnummer einer Haltestelle. Dies geschieht durch eine einfache Selektion der Ortsinformationen nach dem Namen der Haltestelle.
Den Namen einer Haltestelle lässt sich durch die Methode *getNameByIdentifier* herausfinden. Dazu wird in den Ortsinformationen zuerst die Spalte der Haltestellenkennnummern nach dem übergebenen Identifikator selektiert und bei einem Fund der Name dieser zurückgegeben. Für den Fall, dass diese Abfrage kein Ergebnis liefert, wird die Selektion auf die Haltepunktkennnummern angewandt und dieses Resultat verwendet.

3.2.3 Ausnahmeregelungen der Daten

Bei einigen Einträgen in den Datenbanken ist es nötig, dass diese während der Vorverarbeitung korrigiert werden. Darunter fallen Ortsnamen, die nicht einzigartig für eine Haltestelle sind und damit durch einen anderen Namen referenziert werden sollen. Weiterhin gibt es einige Umstiegszeiten, bei denen die Berechnung dieser nicht zu einem realitätsnahen Ergebnis führt, was daran liegt, dass die entsprechende Regel nicht für jeden Fall ein optimales Ergebnis liefert. Schließlich gibt es noch einige Haltestellen, die für den Anwendungsfall dieses Projekts nicht relevant sind. Darunter fallen unter anderem Teststationen oder E-Ladestationen.
Für die Korrektur der Werte wurde eine eigens erstellte Klasse *FileExceptionLoader* entworfen, die eine YAML-Datei mithilfe der Bibliothek *SnakeYAML* einliest und in ein *FileExceptions*-Objekt umgewandelt.[SnakeYaml 2020] Hierbei wird bei der Vorverarbeitung der VDV-Dateien die Datei *fileExceptionsStandardData.yaml* und im Falle der Test-Dateien *fileExceptionsTestData.yaml* verwendet.

Das *FileExceptions*-Objekt enthält drei Listen, von denen die erste die Kennnummern der nicht benötigten Haltepunkte enthält. In der zweiten sind die Identifikatoren der Haltestellen und die Umstiegszeiten enthalten, die nicht optimal errechnet werden. Die dritte Liste beinhaltet alle Kennnummern mit Ortsnamen, die nicht einzigartig für jeden Haltepunkt sind.
Die Listen des *FileException*-Objekts werden im Verlauf der Vorverarbeitung dazu verwendet, die angegebenen Spezialfälle zu korrigieren.
In diesem Projekt wurde das YAML-Format gewählt, da es durch seine Listenstruktur einfach für Menschen lesbar ist und da die erwähnten Dateien vom Nutzer verwendet werden sollen, zeichnet sich hier ein Vorteil im Vergleich zu anderen Formaten wie der JavaScript Object Notation (JSON) ab.[yamlDoku2009] Die Struktur einer YAML-Datei wird anhand eines Beispielauszuges der Datei *fileExceptionsStandardData.yaml* in Auflistung 1 dargestellt.

Auflistung 1: Beispielauszug aus fileExceptionsStandardData

```yaml
removeStoppingPointsIdentifiers:
  - 700
  - 800
transferExceptions:
  - identifier: 336
    transferTime: 3
  - identifier: 1010
    transferTime: 3
locationNameOverlaps:
  - identifier: 3060
    newName: "Keplerstra_e_Neutraubling"
```

Für die Vorverarbeitung der Daten ist es notwendig, dass der Benutzer festlegen kann, welcher Datensatz verwendet wird beziehungsweise für welchen Tag die Lösung des EAP stattfinden soll. Hierfür wird die Klasse *ConfigDayDataLoader* genutzt, die auf ähnliche Weise wie *FileExceptionLoader* funktioniert. Hier wird lediglich aus der Datei *configDayData.yaml* das *ConfigDayData*-Objekt erstellt, welches den boolschen Wert *testData* enthält. Dieser gibt Auskunft darüber, ob die Testdateien oder die Standard-Dateien, welche den VDV-Dateien entsprechen, verwendet werden sollen. Zusätzlich lässt sich ein Tag angeben, der im Falle der standard Dateien als ganze Zahl zwischen eins und sieben angegeben wird, was den Wochentagen entspricht. Handelt es sich um die Testdateien, geben die ersten vier Ziffern das Jahr an, die folgenden zwei den Monat und die letzten beiden den Tag.

Der *testData*-Wert aus *ConfigDayData* wird dabei für die Initialisierung der Klasse *DataInOut* genutzt, der Tag wiederum als Argument für die Methode *getRoutingDataProcessed* ebendieser Klasse. Weitere Informationen über die Ausführung des Programms und die Initialisierung der einzelnen Klassen finden sich in Kapitel 3.5.

3.2.4 Vorverarbeitung der Daten aus der VDV-Schnittstelle

Um die Rohdaten, die in Kapitel 3.1 beschrieben wurden, für die Erstellung des Graphen beziehungsweise die Berechnung der Kantengewichte und Umstiegszeiten nutzen zu können, müssen diese vorverarbeitet und in ein *RoutingDataProcessed*-Objekt umgewandelt werden. Dafür wird je nach aktuell verwendetem Datensatz, eine im Zuge dieser Arbeit erstellte Klasse verwendet.
Handelt es sich um die Daten, die den Konventionen der VDV Schnittstelle entsprechen, wird die Methode *preProcess* der Klasse *DataPreProcessorStandardData* verwendet. Der Ablauf der Vorverarbeitung durch diese Klasse wird in Abbildung 7 dargestellt.
Als Argumente erhält die Methode *preProcess* den Tag, für den das *RoutingDataProcessed*-Objekt erstellt werden soll, und ein *RoutingDataStandardData*-Objekt, in dem die im Folgenden beschriebenen Tabellen aus der VDV-Schnittstellenbeschreibung enthalten sind.
REC_ORT ist dabei durch die Tabelle *locationInfo* umgesetzt, welche im Nachfolgenden als **Ortsdaten** referenziert wird. Ebenso werden *REC_FRT* und *SEL_FZT_FELD* den Tabellen *scheduleData* und *drivingTimes* zugewiesen, welche im weiteren Verlauf als **Fahrplandaten** und **Fahrzeitdaten** bezeichnet werden.
Als erstes wird ein *FileException*-Objekt durch eine Instanz der Klasse *FileExceptionLoader* erstellt, in dem die in Kapitel 3.2.3 beschriebenen Listen zur Anpassung der Rohdaten enthalten sind.
Im nächsten Schritt werden irrelevante Daten aus den Ortsdaten gelöscht, wie beispielsweise Betriebshöfe und Teststeige. Dies geschieht unter anderem durch eine Iteration über die Liste *removeStoppingPointsIdentifiers* aus dem *FileException*-Objekt. Durch die Liste *locationNameOverlaps* werden die entsprechenden uneindeutigen Ortsnamen umbenannt. Weiterhin werden alle Anführungs- und Leerzeichen aus den Haltestellennamen (ORT_REF_ORT_NAME) entfernt. Diese Tabelle entspricht den Vorgaben, um als Tabelle mit Ortsinformationen in *RoutingDataProcessed* zurückgegeben zu werden.
Anschließend wird *transferTimetable* mithilfe der Funktion *createTransferTimetable* erstellt. Diese stellt die Tabelle Transferplan mit den benötigten Umstiegszeiten im Rückgabeobjekt dar.

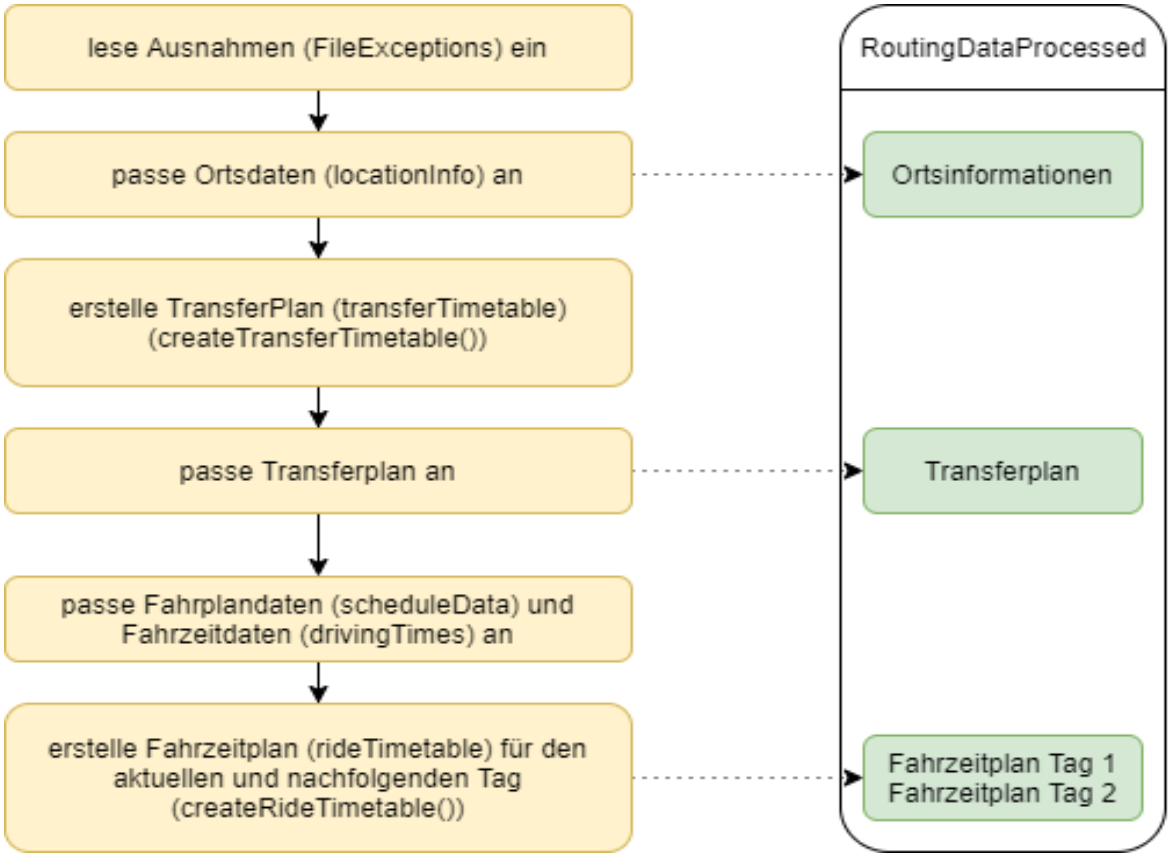


Abbildung 7: Darstellung des Ablaufs der Vorverarbeitung durch *RoutingDataProcessedStandardData*

Innerhalb der Funktion wird pro Haltestelle (ORT_REF_ORT) die Anzahl der Haltepunkte (ORT_NR) in eine Spalte *FREQ* eingetragen, was durch den Codeauszug in Auflistung 2 geschieht. Dabei wird die *count*-Aggregatsfunktion durch einen Aufruf der Methode *summarize* der *tablesaw*-Klasse *Table* verwendet.

Auflistung 2: Auszug aus createTransferTimetable

```
1 Table temp = locationInfo.select("ORT_REF_ORT", "ORT_NR");
2 Table transferTimetable = temp.summarize("ORT_NR",
     count).by("ORT_REF_ORT");
3 transferTimetable =
     transferTimetable.sortAscendingOn("ORT_REF_ORT");
4 transferTimetable.doubleColumn("Count [ORT_NR]").setName("FREQ");
```

Im Rahmen der Masterarbeit *Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg* ergab sich die Erkenntnis, dass im Normalfall die Umstiegszeit der Anzahl der Haltepunkte pro Haltestelle in Minuten minus eins

entspricht. [Huber 2019, S.59]
Deswegen wird nach dieser Regel mit den Einträgen aus *FREQ* die Spalte *TRANSFER_TIME* erstellt, die die Transferzeit für alle Standardfälle enthält. Nach einer weiteren Selektion verbleiben nun noch die Spalten für die Haltestellenkennung und die Transferzeit.
Im Anschluss zur Erstellung des Transferplans werden die Ausnahmen der Regelung mit den Informationen aus dem *FileExceptions*-Objekt angepasst, indem die fehlerhaften Werte durch die angegebenen Werte aus der Liste *transferExceptions* ersetzt werden. Damit ist die Erstellung des Transferplans für das Rückgabeobjekt abgeschlossen.
Als nächstes werden die Tabellen der Fahrplandaten und Fahrzeitdaten angepasst, wobei die Fahrplandaten mithilfe von Selektionen auf die Spalte *TAGESART_NR* in zwei Tabellen aufgeteilt werden. Die erste enthält nur die Informationen für den angegebenen Wochentag und die zweite für den Folgetag. Weiterhin werden irrelevante Daten, wie zum Beispiel alle Einträge mit Haltepunkten an Betriebshöfen, durch Selektion entfernt, da diese keine Rolle für den Nutzer spielen.
Schließlich wurde die Funktion *createRidingTimetable* im Kontext dieses Projekts erstellt, mit welcher die letzte benötigte Tabelle Fahrzeitplan mit den kombinierten Daten aus Fahrplandaten und Fahrzeitdaten entworfen wird. Als Parameter erhält die Funktion die beiden zu kombinierenden Tabellen und das *FileException* Objekt.
Beim Erstellen dieser Funktion ergab sich, dass die Sortierungsmethoden der *tablesaw*-Bibliothek nicht stabil sind. Im weiteren Verlauf der *createRideTimetable*-Funktion wird nämlich eine aus Fahrzeitdaten und Fahrplandaten kombinierte Tabelle nach *FGR_NR* und *FRT_START* sortiert, um die einzelnen Fahrtabläufe nach diesen Werten zu gruppieren. Dies dient als Vorraussetzung für die weitere Verarbeitung. Dargestellt wird dieser Sachverhalt in Abbildung 8, wobei die mit “rot” markierten Bereiche bei gleichem Sortierschlüssel (*FGR_NR* und *FRT_START*) eine andere Reihenfolge aufweisen als vor der Sortierung.
Ursprünglich sind die Fahrzeitdaten so geordnet, dass pro Fahrzeitgruppe (FGR _NR) der Start-Haltepunkt (ORT_NR) des Tabellennachfolgers dem Ziel-Haltepunkt (SEL _ZIEL) des Vorgängers entspricht. Diese Reihenfolge soll bei gleichem Sortierschlüssel beibehalten werden. Daher wird in dieser Funktion in der Tabelle der Fahrzeitdaten die Spalte *INDEX_SORT* eingefügt und nach der anfänglichen Sortierung durchnummeriert, um später durch diese Spalte die ursprüngliche Sortierung beibehalten zu können.
Um die Informationen aus beiden Tabellen zu vereinen, werden diese durch einen *Inner Join* über den Schlüssel Fahrzeitgruppe (FGR_NR) kombiniert. Der zugehörige *tablesaw* Aufruf verhält sich analog zum Structured Query Language (SQL)-*Inner Join* Befehl.

unsortierte Tabelle

FGR_NR	ORT_NR	SEL_ZIEL	SEL_FZT	FRT_START	LI_NR	STR_LI_VAR
138	501500	500500	60	18540	1	"14"
138	500500	500300	60	18540	1	"14"
138	500300	500200	60	18540	1	"14"
138	500200	502300	180	18540	1	"14"
138	501500	500500	60	19740	1	"14"
138	500500	500300	60	19740	1	"14"
138	500300	500200	60	19740	1	"14"

nach FGR_NR und FRT_START sortierte Tabelle

FGR_NR	ORT_NR	SEL_ZIEL	SEL_FZT	FRT_START	LI_NR	STR_LI_VAR
138	500500	500300	60	18540	1	"14"
138	500300	500200	60	18540	1	"14"
138	500200	502300	180	18540	1	"14"
138	501500	500500	60	18540	1	"14"
138	501500	500500	60	19740	1	"14"
138	500500	500300	60	19740	1	"14"
138	500200	502300	180	19740	1	"14"

Abbildung 8: Darstellung der instabilen Sortierung

Daraufhin werden wiederum alle Einträge mithilfe des *FileException* Objekts entfernt, deren Start- oder Ziel-Haltepunkt in der Liste der zu entfernenden Haltepunkte (*removeStoppingPoints*) enthalten ist.
Anschließend wird die kombinierte Tabelle nach Fahrzeitgruppe (FGR_NR), dann nach Fahrtstart (FRT_START) und schließlich nach der Spalte *INDEX_SORT* geordnet, um alle Einträge nach den Fahrten beziehungsweise Startzeiten zu sortieren und die anfängliche Reihenfolge wie in der Fahrzeittabelle bei gleichen Sortierschlüsseln wiederherzustellen.
Um die Ankunft beziehungsweise Abfahrt direkt ablesen zu können, werden die Spalten *ARRIVAL* und *DEPARTURE* erstellt. Mit einer Schleife wird über die zuvor erstellte Tabelle iteriert. Dabei werden bei jedem Eintrag die Werte für die nächste Iteration zwischengespeichert und bei der Verarbeitung zwischen zwei Fällen unterschieden.
Entspricht der gegenwärtige Fahrtstart (FRT_START) oder die Fahrzeitgruppe (FGR_NR) nicht denen des Vorgängers, handelt es sich um einen neuen Fahrtabschnitt. *ARRIVAL* und *DEPARTURE* werden demnach auf der Basis des aktuellen Fahrtstarts (FRT_START) und der Fahrzeit (SEL_FZT) erstellt. Andernfalls wird statt des Fahrtstarts (FRT_START) der *ARRIVAL*-Wert des Vorgängers verwendet.
Schlussendlich wird die Spalte *LI_NR* von einer Integer basierten Spalte in eine String basierte umgewandelt, um Liniennummern wie beispielsweise “10A” zu ermöglichen.

Diese sequenzielle Abarbeitung wird durch die vorherige Sortierung ermöglicht, durch welche, außer zu Beginn einer neuen Fahrzeitgruppe (FGR_NR) oder Fahrtstarts (FRT_START), sich der Vorgänger eines Fahrtverlaufs einen Tabelleneintrag darüber befindet.
Mithilfe dieser *createRidingTimetable* Funktion werden die zwei Fahrzeitpläne erstellt, die für den aktuellen und den nächsten Tag Gültigkeit besitzen. Dies geschieht durch die Übergabe der unterschiedlichen Fahrplandaten für die jeweiligen Tage.
Damit wurden alle Tabellen zur Rückgabe im *RoutingDataProcessed*-Objekt erstellt.

3.2.5 Vorverarbeitung der aktuellen Testdaten des RVV

Für den Fall, dass die Testdaten des RVV verwendet werden sollen, und diese noch nicht vorverarbeitet wurden, kommt die im Zuge dieser Arbeit erstellte Klasse *DataPreProcessorTestData* zum Einsatz.
Die Tabellen der Testdaten, die hier zur Erstellung des *RoutingDataProcessed*-Objekts verwendet werden, wurden in Kapitel 3.1.2 beschrieben. Diese werden im weiteren Verlauf folgendermaßen referenziert:

- stops als **Stopdaten** (*stopData*)
- stop_times als **Stopzeitdaten** (*stopTimesData*)
- calendar_dates als **Kalenderdaten** (*calendarData*)
- trips als **Tripdaten** (*tripData*)
- routes als **Routendaten** (*routeData*)

Die Klasse *DataPreProcessorTestData* hat dabei einen vergleichbaren Aufbau wie *DataPreProcessorStandardData* aus Kapitel 7. Da die Rohdaten jedoch anders aufgebaut sind, werden einige Funktionen anders umgesetzt, beziehungsweise hinzugefügt.
Beispielsweise wird die Methode *createTransferTimetable* in gleicher Ausführung benötigt und da die Methoden oft das gleiche Endergebnis liefern sollen, wurde im Rahmen des Projekts entschieden, dass *DataPreProcessorTestData* von der Klasse *DataPreProcessorStandardData* erben soll. Dabei werden die Methoden *preProcess* und *createTransferTimetable* überschrieben.
Der Ablauf von *preProcess* wird in Abbildung 9 dargestellt. Als erstes wird analog

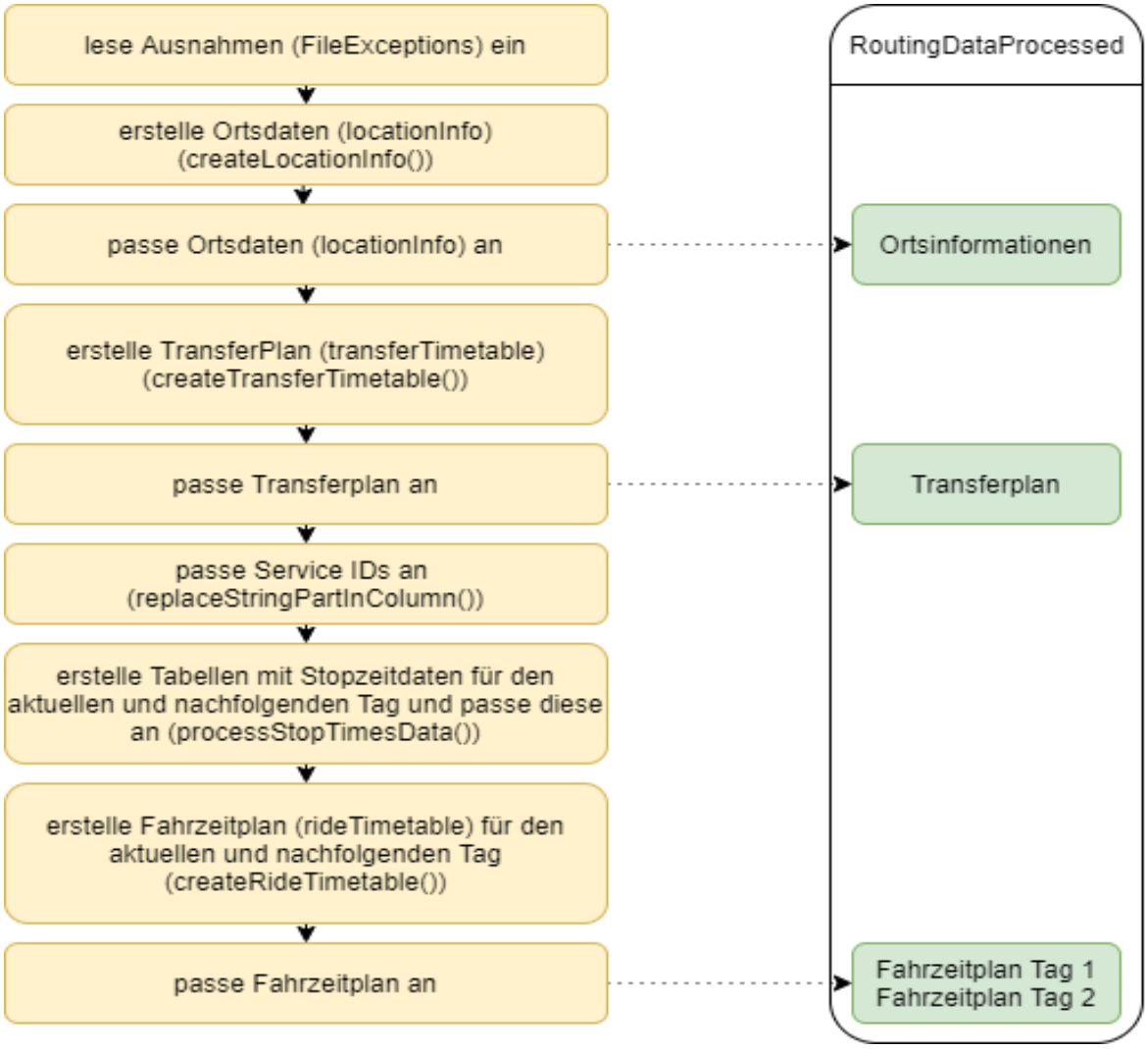


Abbildung 9: Darstellung des Ablaufs der Vorverarbeitung durch *RoutingDataProcessedTestData*

zur *preProcess*-Funktion aus *DataPreProcessorStandardData* eine *FileException*-Objekt erstellt, welches zur Korrektur der Tabellen verwendet wird.
Die Tabelle der Ortsinformationen wird hier von der im Rahmen der Arbeit neu entwickelten Funktion *createLocationInfo* erstellt. Diese bekommt als Parameter die Tabelle mit Stopdaten. Da die Stopdaten lediglich eine Kennnummer für jeden Haltepunkt beitzen und diese als String vorliegt, liegt der Hauptfokus darauf, eine Integer-Kennnummer für die Haltestellen und Haltepunkte zu erstellen.
Als erstes werden *tablesaw*-Spalten für diese Werte, den Ortsnamen, sowie die alte Haltepunktkennnummer erstellt.
Anschließend wird über die Tabelle der Stopdaten zeilenweise iteriert, wobei alle Einträge, deren ursprünglicher Haltepunktidentifikator auf “Parent” endet, übersprungen werden, da diese keinen realen Haltepunkten entsprechen (vgl. Kapi-

tel 3.1.2). Ebenso sollen diese Identifikatoren zur Weiterverarbeitung nur noch aus Ziffern getrennt durch Doppelpunkte bestehen. Dazu werden die Bezeichner “de:” und “gen:” entfernt, da diese nicht relevant sind (vgl. Kapitel 3.1.2). Jetzt werden die weiteren Identifikationsmerkmale “N”, “S” und “G” in unterschiedliche ganze Zahlen umgewandelt.
Nun liegen die Kennnummern so vor, dass die ersten zwei Segmente zwischen den Doppelpunkten eindeutig eine Haltestelle und die gesamten Ziffern einen Haltepunkt referenzieren. Da diese Zahlen sehr lang werden können, werden sie als Double in den entsprechenden Spalten abgespeichert.
Einige Ortsnamen verweisen nicht eindeutig auf eine Haltestelle, da zum Beispiel am Hauptbahnhof die einzelnen Bussteige mit “Bstg” und einer Nummer am Ende referenziert sind. Da der Name aber die gesamte Haltestelle repräsentieren soll, werden diese Endungen entfernt.
Weiterhin wird der alte Identifikator auch in der entsprechenden Spalte gespeichert, da dieser zur Weiterverarbeitung der anderen Tabellen benötigt wird.
Die Iteration endet an diesem Punkt. Zu erwähnen ist, dass die bisher beschriebenen Anpassungen an den einzelnen Werten technisch durch Java String Methoden umgesetzt werden.
In *RoutingDataProcessed* sollen die Kennnummern der Orte als Integer abgespeichert werden. Da diese jedoch größer sind als ein Integer abspeichern kann, müssen diese durch kleinere Kennnummern ersetzt werden. Dazu wird über die Haltestellen- und Haltepunktidentifikatoren iteriert und jeder einzigartige Wert durch einen bei eins beginnenden Zähler ersetzt. Der Zähler wird pro Iterationsschritt um eins erhöht. Anschließend werden die Spalten-Typen mit der entsprechenden *tablesaw*-Methode von Double-basiert auf Integer-basiert geändert und die erstellte Tabelle wird zurückgegeben.
In der Methode *preProcess* werden die zurückerhaltenen Ortsinformationen wie in *DataPreProcessortStandardData* mit dem *FileExceptions*-Objekt angepasst (vgl. Kapitel 7).
Als nächstes wird der Transferplan mit der vererbten Funktion *createTransferTimetable* erstellt und ebenso wie in *DataPreProcessortStandardData* korrigiert (vgl. Kapitel 7).
Für Abfragen auf die *ServiceID* aus den Tripdaten, Stopzeitdaten und Kalenderdaten muss diese angepasst werden. Dies ergibt sich dadurch, dass diese in der Form “Special#” gefolgt von einer Identifikationsnummer vorliegt. Die Kennnummer, die eigentlich durch “Special#1” dargestellt werden sollte, heißt jedoch lediglich “Special”, was bei der Abfrage nach dieser Kennnummer zu Problemen führen kann, da keine eindeutige Nummer die Kennnummer beschreibt. Daher wurde im Rahmen des Projekts zusätzlich die Funktion *replaceStringPartInColumn* erstellt. Diese ersetzt in einer Spalte einer Tabelle alle angegebenen Zei-

chenfolgen durch einen der Funktion übergebenen String. Dies wird durch die *tablesaw*-Methode *replaceAll* der *StringColumn*-Klasse umgesetzt. Mit dieser Funktion werden in den Spalten *trip_id* und *service_id* der bereits erwähnten Tabellen “Special” durch “Special#1” ersetzt.
Um aus den Stopzeitdaten die für den ausgewählten Tag nicht verfügbaren Verbindungen zu entfernen, wurde im Zuge dieser Arbeit die Funktion *processStopTimesData* entworfen. Zusätzlich passt diese die Haltepunktkennnummern an die in *createLocationInfo* erstellten Identifikatoren an und formatiert die Uhrzeiten zu Sekunden nach null Uhr.
Damit in den Stopzeitdaten nur die Verbindungen aufgeführt sind, die an dem gewünschten Tag fahren, müssen die andern Eintäge entfernt werden. Dazu wird eine Liste an ServiceIDs benötigt, welche an diesem Tag nicht fahren. Dafür wurde zusätzlich die Funktion *getDropServiceIDs* erstellt. Diese filtert in den Kalenderdaten nach dem angegebenen Tag und speichert die dort enthaltenen ServiceIDs in einer Liste, welche nun die ServiceIDs enthält, die an diesem Tag fahren. Danach wird diese Liste mit einer Liste aller ServiceIDs abgeglichen und die nicht enthaltenen Kennnummern, die somit nicht für diesen Tag zuständig sind, als Listen-Objekt zurückgegeben.
In *processStopTimesData* wird nun ein solches Listen-Objekt mit der Funktion *getDropServiceIDs* erstellt und durch Iteration über dieses jede Zeile entfernt, deren *trip_id* durch diese ServiceIDs abgedeckt wird.
Weiterhin wird eine Iteration für jede Ortskennnummer in den Stopzeitdaten durchgeführt. Diese Kennnummern werden mithilfe eines Abgleichs der ursprünglichen und der neu erstellten Identifikatoren in den Ortsinformationen durch die neu erstellten Kennnummern ersetzt.
Anschließend wird ein Objekt der Klasse *DataTransformer* instanziiert (vgl. Kapitel 3.2.2). Es werden die Spalten für Ankunfts- und Abfahrtszeiten durchlaufen und mithilfe des *DataTransformer*-Objekts die Zeiten der Form “hh:mm:ss” in Sekunden nach null Uhr umgewandelt, da dieses Format sich deutlich besser für Berechnungen und Vergleiche eignet.
Die Funktion *processStopTimesData* wird zweimal aufgerufen, um die Stopzeitdaten für den aktuellen und den folgenden Tag zu erhalten, wobei sich die Aufrufe durch die Übergabe des Tages unterscheiden.
In der Methode *preProcess* soll nun der Fahrzeitplan durch die im Rahmen des Projektes entwickelte Funktion *createRideTimetable* erstellt werden. Als Parameter erwartet diese eine der angepassten Stopzeitdaten-Sets, die Routendaten und die Tripdaten.
In dieser Funktion werden als erstes die benötigten Spalten für *RideTimetable* erstellt (vgl. Kapitel 3.2.1). Dabei soll auf Basis der Stopzeitdaten, die die Ankunft und Abfahrt an einem bestimmten Ort beschreiben, der Fahrzeitplan erstellt wer-

den, welcher die Fahrten von einem Haltepunkt zum Nächsten darstellt. Als Erstes werden die Stopzeitdaten mithilfe der *tablesaw*-Methode *sortAscendingOn* nach *trip_id* und *stop_sequence* geordnet, um die einzelnen Fahrten zu gruppieren und diese der Fahrtreihenfolge nach zu sortieren. Diese Ordnung wird für die im Weiteren beschriebene Funktion benötigt.
Es wird über die Reihen der Stopzeitdaten iteriert, wobei am ende dieser Iteration die Ortskennnummer, die TripID und die Abfahrtszeit der Vorgängerreihe gespeichert werden. Bei jedem Iterationsschritt findet nun eine Abfrage statt, ob die Vorgängerwerte existieren oder ob die aktuelle TripID mit der des Vorgängers übereinstimmt. Ist dies nicht der Fall, werden diese entsprechend der aktuellen Iteration gesetzt und der nächste Iterationsschritt gestartet. Damit soll der erste Eintrag bezüglich eines Trips übersprungen werden, da hierdurch aus einer Stop-Beschreibung, zusammen mit der des Vorgängers, eine Fahrtbeschreibung zwischen diesen Stops erstellt wird und dies nur für Einträge des gleichen Trips umgesetzt werden soll.
Nun werden die Spalten für den Fahrzeitplan folgendermaßen gesetzt:

- Haltepunktkennnummmer (Von): Identifikator des Vorgängers
- Haltepunktkennnummmer (Nach): aktueller Identifikator
- Fahrzeit: aktuelle Ankunftszeit minus Abfahrtszeit des Vorgängers
- Liniennummer: Liniennummer entsprechend der aktuellen TripID
- Abfahrt: Abfahrtszeit des Vorgängers
- Ankunft: aktuelle Ankunftszeit

Die Liniennummer wird dabei durch einen Abgleich der Schlüssel *TripID* und *RouteID* in den Tabellen der Trip- und Routendaten selektiert.
Wenn alle Iterationsschritte vollzogen wurden, werden die Spalten in einer Tabelle als Fahrzeitplan zurückgegeben.
Damit ist die Erstellung des *RoutingDataProcessed*-Objekts abgeschlossen.

3.3 Graph und Algorithmus

3.3.1 Aufbau des Graphen

Im Rahmen dieses Projekts wurde der zeitabhängige Graph folgendermaßen umgesetzt.
Grundlegend gibt des eine *Graph*-Klasse, die zum einen ein *RoutingDataProcessed*-Objekt besitzt, zum anderen eine Liste an *Nodes*. Weiterhin bietet diese die Methode *shortestWay* an, mit der es möglich ist das EAP zu lösen. Dabei wird das *RoutingDataProcessed*-Objekt benötigt um die Kantengewichte zu einem bestimmten Zeitpunkt zu berechnen.
Die Knoten wurden dabei als Klasse *Node* umgesetzt. Diese enthält das Attribut *identifier*, welches der Kennnummer einer Haltestelle oder eines Haltepunkts entspricht. Um die Art des Knotens festzulegen besitzt dieser eine Referenz auf ein Enum *NodeType*, welches Auskunft darüber gibt, ob es sich um einen Haltepunkt (STOPPING_POINT) oder den Umstiegsknoten für jede Haltestelle (CENTRAL_STOPPING_POINT) handelt. Zusätzlich besitzt ein *Node*-Objekt zur Verbindung der Knoten eine Liste an Kanten, die durch das Interface *Edge* dargestellt werden.
Dieses Interface stellt die Methoden *getToNode*, *getLineNumber* und *isRouteEdge* zur Verfügung, welche dazu dienen, die nötigen Informationen über den Zielknoten, die Liniennummer und die Art der Kante zu erhalten. Zusätzlich wird hier die Methode *isOnlyNextDay* definiert. Diese gibt an, ob eine Kante nur am nächsten Tag zur Verfügung steht.
Implementiert wird das Interface durch die Klassen *RouteEdge* und *TransferEdge*, welche die zwei Kanten-Typen darstellen.
RouteEdge wird verwendet um Routenkanten, also Verbindungen zwischen zwei Haltepunkten, zu realisieren. Dabei besitzt diese Klasse das Attribut *lineNumber*, welches die Linienverbindung spezifiziert, die von der Kante dargestellt wird. Zusätzlich verfügt *RouteEdge* über das Attribut *onlyNextDay*, in welchem die Information gespeichert wird, ob die Kante nur am Folgetag genutzt werden kann.
TransferEdge hingegen stellt die Umstiegskanten dar und soll die Verbindung zwischen dem Knoten eines Haltepunkts und einer Haltestelle abbilden. Die Funktion *getLineNumber* gibt im Falle der Transferkante "transferEdgeßurück, da keine Liniennummern für Umstiege existieren. Weiterhin sind diese immer für beide Tage verfügbar, da Umstiege nicht vom Tag abhängig sind.
Beide Implementierungen des *Edge*-Interface besitzen eine Referenz auf einen Zielknoten (*toNode*).
Der Aufbau des Graphen wird in Abbildung 10 dargestellt.

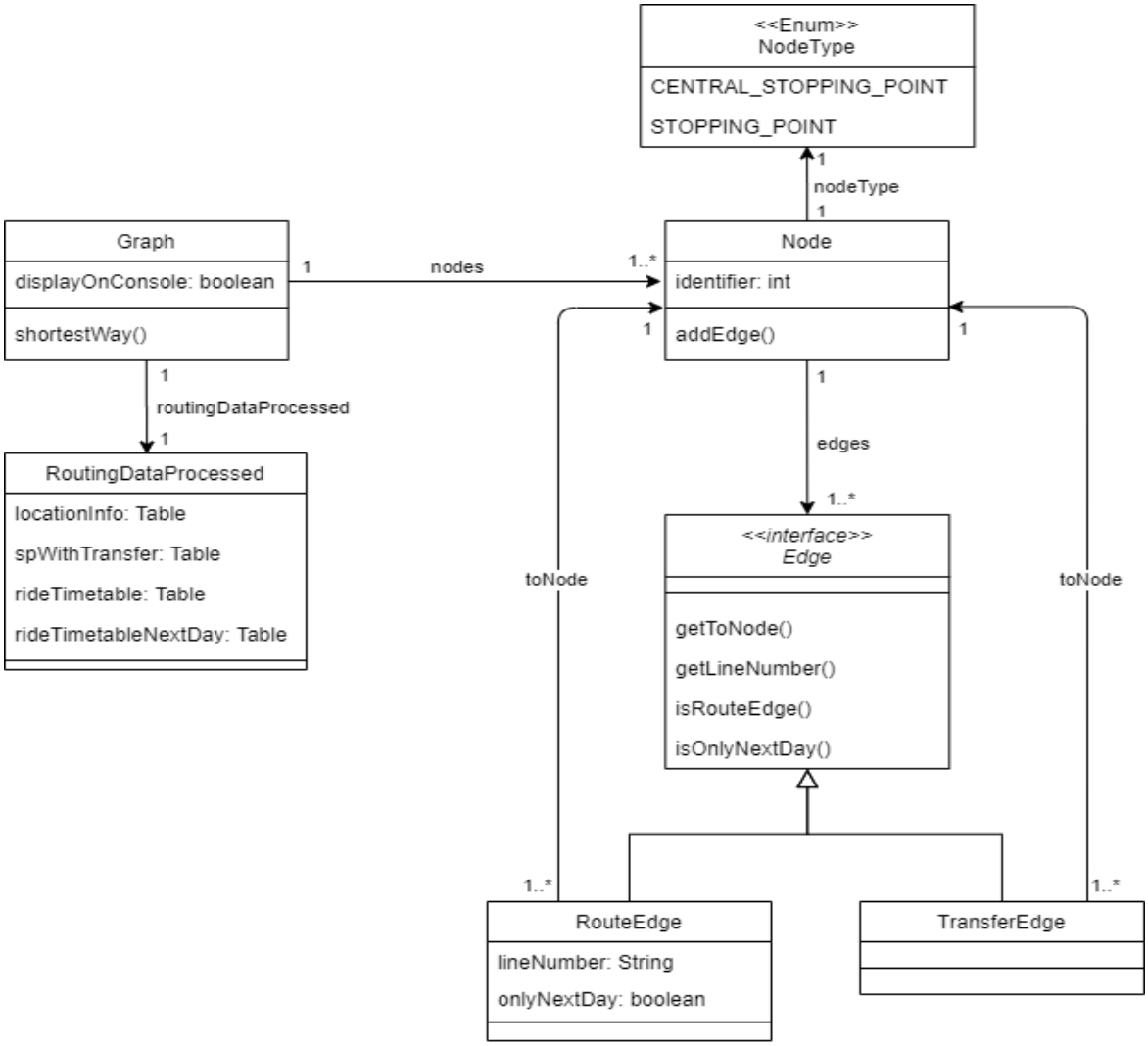


Abbildung 10: UML Diagramm des Graphen

3.3.2 Erstellung des Graphen aus den Daten

Um aus den vorverarbeiteten Daten den Graphen zu erstellen, werden zum einen die Tabellen aus *RoutingDataProcessed*, zum anderen die dem Modell des zeitabhängigen Graphen (vgl. Kapitel 2.1) entsprechenden Knoten benötigt. Diese wiederum sollen durch die jeweiligen Kanten miteinander verbunden sein.
Im Kontext dieser Arbeit, wurde für diesen Anwendungsfall die Klasse *GraphSetup* entwickelt. Zur Instanziierung wird ebenfalls das *RoutingDataProcessed*-Objekt vorausgesetzt, wodurch in dieser Klasse Zugriff auf Ortsinformationen, Fahrzeitplan und Transferplan besteht.
Zur Erstellung eines Graphen steht die *setup*-Methode zur Verfügung. Aus den Ortsinformationen wird jeweils die Spalte für Haltestellen- und Haltepunktkennnummern ausgewählt, wobei unter Verwendung der *unique*-Methode der *table*-

saw-Bibliothek Duplikate entfernt werden.
Für jeden Identifikator der Haltepunkte (*ORT_NR*) wird unter Angabe der Knotenart(*NodeType*) *STOPPING_POINT* ein *Node*-Objekt erstellt und in einer Liste gespeichert. Die Vorgehensweise für die Haltestellen(ORT_REF_ORT) verhält sich gleich. Jedoch wird hier der *NodeType CENTRAL_STOPPING_POINT* verwendet. Beide Listen werden vereint, um die Gasamtheit der Knoten zu erhalten.
Um die Knoten durch geeignete Kanten zu verbinden, wurde die Methode *linkNodesFromColumns* erstellt. Diese verknüpft die Knoten über zwei *tablesaw*-Spalten einer Tabelle, wobei die Namen der Spalten, die Tabelle und eine Liste aller Knoten als Parameter übergeben werden. Dabei muss die erste Spalte die Kennnummern enthalten, deren Knoten mit denen der entsprechenden Kennnummern der zweiten Spalte verknüpft werden sollen. Zusätzlich wird durch den Parameter *onlyNextDay* angegeben, ob die Kanten für den nächsten Tag gelten.
Als erster Schritt wird hierfür eine Untertabelle *edgeTable* erstellt, die nur die zwei als Parameter angegebenen Spalten enthält, von denen die erste die Kennnummern der Knoten enthält, von denen die Kanten ausgehen sollen. An zweiter Position befindet sich diejenigen Kennnummern der Knoten, auf die die Kanten hinweisen.
Anschließend wird über diese iteriert. Für jeden Iterationsschritt wird mit der Funktion *getNodeByIdentifier* der ”Von”- und ”Nach”-Knoten ermittelt.
GetNodeByIdentifier durchsucht mittels einer Schleife alle *Nodes* nach dem Knoten mit dem passenden Identifikator und gibt diesen zurück.
Nachdem die jeweiligen Knoten ermittelt wurden, wird geprüft ob beide *Nodes* dem Knotentyp *STOPPING_POINT* und somit zwei Haltepunkten entsprechen. Ist dies nicht der Fall, wird eine Transferkante mit Verweis auf den Zielknoten erzeugt und dem Ausgangsknoten hinzugefügt. Dabei wird auch der übergebene Parameter *onlyNextDay* für diese Kante gesetzt.
Andernfalls handelt es sich um eine Routenkante, welche unter Angabe der zugehörigen Liniennummer ebenso initialisiert und eingefügt wird. In den bis jetzt erstellten Tabellen existieren jedoch mehrere Einträge für die Verbindung von zwei Haltepunkten mit einer Linie, da die Verbindung mehrmals am Tag abgefahren werden kann. Diese Verbindungen sind in dem gewählten Modell jedoch nicht notwendig, da die Fahrzeit zu unterschiedlichen Zeitpunkten dynamisch berechnet wird. Daher wird vor dem Einfügen von *RouteEdge* geprüft, ob es bereits eine Kante mit gleicher Liniennummer zwischen den beiden Knoten gibt. Ist dies der Fall, wird von einer Erstellung von *RouteEdge* abgesehen.
Die Funktion *linkNodesFromColumn* wird nun auf die Tabelle Ortsinformationen aus *RoutingDataProcessed* angewandt, wobei als Ausgangsspalte die Kennummern der Haltepunkte (ORT_NR) und als Zielspalte die der Haltestellen (ORT_REF_ORT) gewählt wird. Damit werden alle Umstiegskanten in Richtung des Um-

stiegsknotens realisiert. Um die Transferkanten vom Umstiegsknoten zu den Haltepunkten umzusetzen, wird die Funktion erneut aufgerufen, jedoch mit vertauschten Spaltennamen. Als Letztes sollen noch die Haltepunkte untereinander verknüpft werden, was der Erstellung der Routenkanten entspricht. Diese Informationen befinden sich in den Startspalten (ORT_NR) und Zielspalten (SEL_ZIEL) des Fahrzeitplans aus *RoutingDataProcessed*, auf welche die *linkNodesFromColumns*-Methode angewandt wird. Dies wird als Erstes für den Fahrzeitplan des ersten Tages durchgeführt, wobei der Parameter *onlyNextDay* der Funktion *linkNodesFromColumns* auf *false* gesetzt wird. Als Nächstes wird diese Funktion für den Fahrzeitplan des nächsten Tages mit einem *true*-Wert für *onlyNextDay* ausgeführt. Hierbei werden keine Kanten eingefügt, die nur für den nächsten Tag Gültigkeit besitzen, falls diese am ersten Tag auch schon bestehen.
Damit sind alle *Node* Objekte erstellt und mit *Edge*-Objekten verknüpft. Die Knoten werden zusammen mit *RoutingDataProcessed* und dem Attribut *displayOnConsole* dem Konstuktor der Graph-Klasse übergeben. *DisplayOnConsole* ist ein boolscher Wert, der im *true*-Fall dafür sorgt, dass bei der Berechnung des kürzesten Wegs Einzelschritte auf der Konsole ausgegeben werden. Standardmäßig ist dieser auf *false* gesetzt und kann durch die *GraphSetup*-Methode *setDisplayOnConsole* geändert werden.

3.3.3 Prüfung der Bedingungen des angepassten Dijkstra Algorithmus

Um das EAP erfolgreich lösen zu können, muss die FIFO-Bedingung erfüllt sein (vgl. Kapitel 2.2). Daher wurde im Rahmen dieser Arbeit die Klasse *ConditionChecker* zusammen mit der Methode *checkFIFO* erstellt, welche diese Bedingung überprüfen soll. Zusätzlich stellt die Klasse die Methode *checkUniqueLocationNames* zur Verfügung. Mit dieser lassen sich die Haltestellen ausgeben, welche nicht eindeutig durch einen Haltestellennamen referenziert werden.
Um die FIFO-Bedingung zu überprüfen, sollen alle Verbindungen des Fahrzeitplans *rideTimetable* aus dem *RoutingDataProcessed*-Objekt untereinander verglichen werden. Dabei wird geprüft, ob es Verbindungen zweier Orte gibt, bei der die Abfahrt später stattfindet, die Ankunft jedoch gleich oder früher erfolgt. Hierdurch wäre die Bedingung verletzt.
Implementiert wurde diese Überlegung durch zwei geschachtelte Iterationen über die Start-Haltestellen (*ORT_NR*) und Ziel-Haltestellen (*SEL_ZIEL*) des Fahrzeitplans. Hierbei wird in der inneren Schleife eine Vergleichstabelle (*fifoCompare*) durch Filterung des Fahrzeitplans mit den beiden Schleifenelementen erstellt, welche nun alle Verbindungen zwischen den zwei spezifizierten Haltepunkten umfasst. Jede dieser Verbindungen wir nun mit allen anderen Verbindungen in

der Vergleichstabelle verglichen. Umgesetzt wird dies durch zwei weitere zwei Schleifen. Verglichen werden nun immer die Abfahrts- und Ankunftszeiten der durch die Schleifen gewählten Verbindungen. Gibt es eine Verbindung, deren Abfahrtszeitpunkt nach dem einer anderen liegt, jedoch zur gleichen Zeit oder früher am entsprechenden Haltepunkt ankommt, wird noch überprüft, ob diese Verbindung der gleichen Linie angehören, da die FIFO-Bedingung nicht verletzt wird, wenn die Verbindungen durch unterschiedliche Linien stattfinden. Gehören beide Verbindungen jedoch zur selben Linie, wird eine Warnung auf der Kommandozeile ausgegeben, dass der Algorithmus möglicherweise nicht mehr die optimale Lösung findet. Dabei werden auch die Linien mit den entsprechenden Fahrzeiten ausgegeben.
Da es unter anderem für die Eingaben des Nutzers notwendig ist, dass eine Haltestelle nur durch einen Haltestellennamen repräsentiert wird, kann mit der Funktion *checkUniqueLocationNames* geprüft werden, ob dies der Fall ist, um diese anschließend in der entsprechenden *FileException*-Datei anzupassen (vgl. Kapitel 3.2.3). Dafür wird über die Haltestellennamen (*ORT_REF_ORT_NAME*) in *locationInfo* aus *RoutingDataProcessed* iteriert. Die Tabelle der Ortsinformationen wird dabei nach den jeweiligen Haltestellennamen selektiert. Mithilfe der *tablesaw*-Methode *countUnique* wird dabei die Anzahl der einzigartigen Haltestellenkennnummern (*ORT_REF_ORT*) berechnet. Entspricht diese einer Zahl größer als eins, referenziert der Haltestellenname nicht eindeutig eine Haltestelle. Daraufhin werden die Informationen über die Haltestelle mit einer Warnung auf der Kommandozeile ausgegeben, damit diese korrigiert werden können.

3.3.4 Umsetzung des angepassten Dijkstra-Algorithmus

Die im Rahmen dieser Arbeit entwickelte Klasse *Graph* erhält bei der Instanzierung eine Liste aller Knoten, ein *RoutingDataProcessed*-Objekt und den boolschen Wert *displayOnConsole*, welcher bei true dafür sorgt, dass die Zwischenergebnisse des Algorithmus auf der Konsole ausgegeben werden. Diese Klasse stellt die Methode *shortestWay* zur Verfügung und dient der Lösung des EAP. Ihre Umsetzung basiert auf dem in Kapitel 2.3 beschriebenen angepassten Dijkstra-Algorithmus.
Während des Ablauf werden die durchlaufenen Knoten als *WayPoint*-Objekte in einer Instanz der Klasse *Way* abgespeichert. Diese Klassen wurden ebenfalls im Zuge dieses Projekts erstellt. Ein *WayPoint* hat dabei folgende Attribute:

- node: zugehöriger Knoten des WayPoints

- prevNode: Vorgängerknoten über den dieser Knoten erreicht wurde
- isTarget: Angabe, ob es sich bei diesem Knoten um den Ziel-Knoten handelt
- time: Zeitmarke, zu der dieser WayPoint erreicht wurde
- reachedByLineNr: Liniennummer, mit welcher der WayPoint erreicht wurde
- isReachedViaRoute: gibt an, ob dieser WayPoint über eine Routen-Kante erreicht wurde
- departureToThisPoint: Zeitmarke der Abfahrt zu diesem WayPoint

Die Klasse *Way* enthält eine Liste an *WayPoints*, welche für jeden Knoten maximal einen Eintrag enthalten soll, da die Lösung des *EAP* jeden Knoten höchstens einmal erreicht und somit, falls dieser bereits erstellt wurde, der Wegpunkt nur aktualisiert werden soll. Dafür wurde die Funktion *getWayPointFromNode* erstellt, mit welcher sich unter Angabe eines Knotens der entsprechende Wegpunkt ausgeben lässt und somit beim Hinzufügen eines neuen Wegpunktes abgefragt werden kann, ob bereits ein Objekt zu diesem Knoten im Weg existiert. Umgesetzt wird diese Funktion durch Iteration über die Liste der Wegpunkte und entsprechende Abgleichungen der Knoten.
Mit der Funktion *addPoint* kann nun, unter Angabe aller *WayPoint*-Attribute, ein neuer Wegpunkt in die Liste eingefügt, beziehungsweise aktualisiert werden, falls schon ein Wegpunkt für den übergebenen Knoten existiert.
Weiterhin lässt sich der Wegpunkt, der als Ziel (*isTarget*) markiert wurde, mit der Funktion *getTarget* ausgeben, in welcher die Liste an Wegpunkten nach einem true-Wert im Attribut *isTarget* durchsucht wird.
Schließlich stellt die Klasse *Way* noch die Methode *getShortestWayPointsInOrder* zur Verfügung, welche eine Liste an Wegpunkten der Lösung des EAP in richtiger Reihenfolge liefert. Dabei wird als erstes der Ziel-Wegpunkt mit *getTarget* ermittelt und mithilfe des Vorgänger-Attributs der kürzeste Weg durchlaufen. Hierbei werden alle Punkte in einer Liste eingetragen, die schließlich zurückgegeben wird.
Für die Beschreibung der Methode *shortestWay* soll zunächst eine Funktion zur Berechnung der Kantengewichte beziehungsweise der kürzesten Dauer, einen Haltepunkt von einem mit diesem verbundenen Haltepunkt zu einem bestimmten Zeitpunkt zu erreichen, beschrieben werden.
Diese trägt den Namen *getTransportTime*. Vereinfacht wird ihr Ablauf in Abbildung 11 dargestellt.

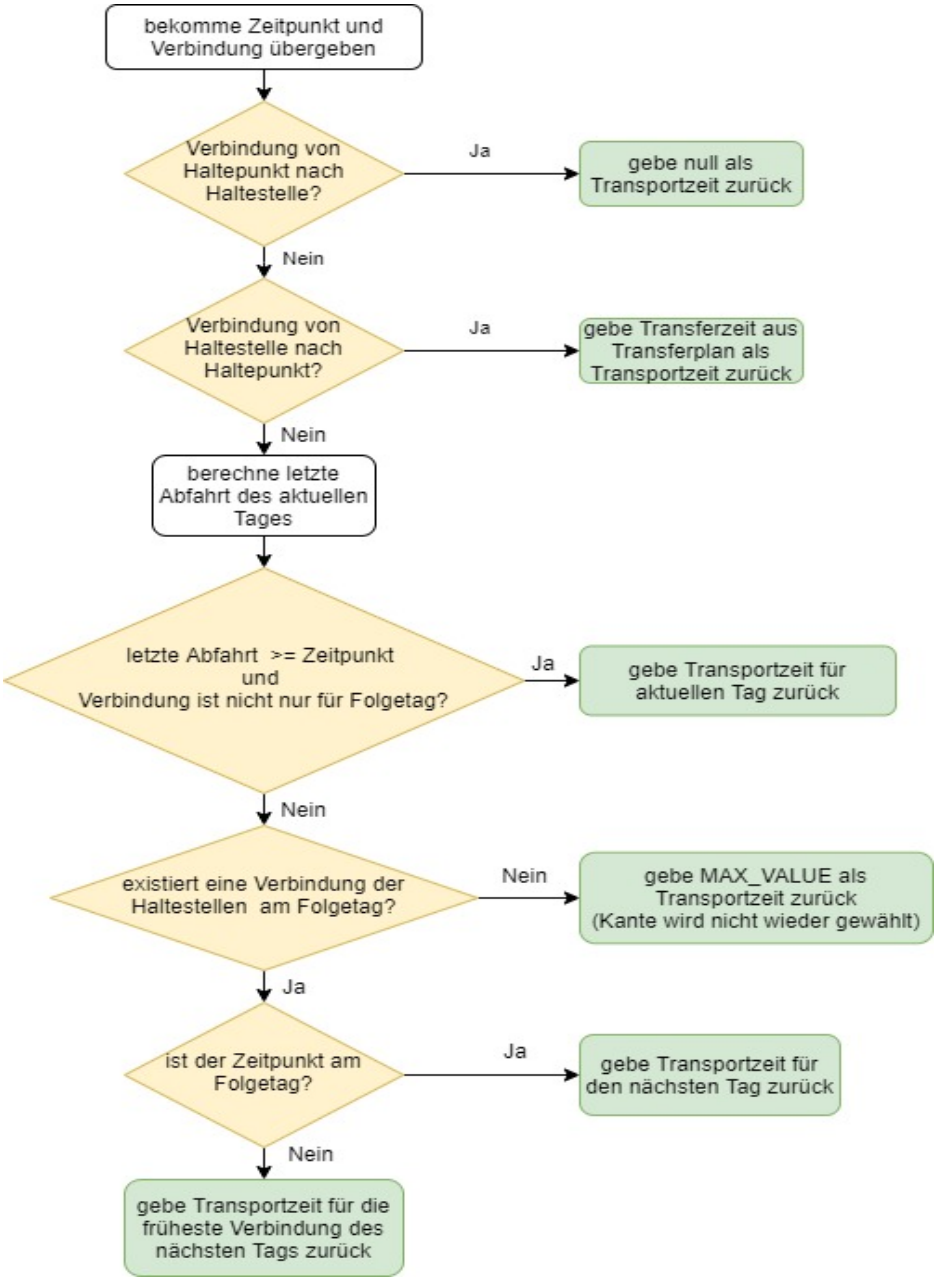


Abbildung 11: Darstellung des Ablaufs von *getTransportTime*

Als Übergabe Parameter erhält diese Funktion den Ausgangsknoten, Endknoten und Zeitpunkt, für welche die Transportzeit der so beschriebenen Verbindung ermittelt werden soll. Weiterhin benötigt *getTransportTime* die Liniennummer dieser Verbindung und das *Way*-Objekt, welches für die Lösung des EAP verwendet wird. Zusätzlich wird über *onlyNextDay* angegeben, ob die Verbindung nur für den nächsten Tag gilt und somit am aktuellen Tag nicht zur Verfügung steht.
Der Rückggabewert ist ein Objekt der Klasse *TransportTime*, welches die Attribute *duration* und *departure* besitzt. Damit lässt sich mit *getTransportTime* sowohl der Abfahrtszeitpunkt, als auch die Zeit, bis der Zielknoten über die gewählte Verbindung erreicht wird, ermitteln.
Als erstes wird nun durch Abfrage der Knoten-Typen ermittelt, um welche Art der Verbindung es sich handelt. Entspricht der Ausgangsknoten einem Haltepunkt und der Endknoten einer Haltestelle, ist dies die erste Variante einer Umstiegskante und die Transportzeit wird mit Dauer null zurückgegeben (vgl. Kapitel 2.2). Der Abfahrtszeitpunkt entspricht dabei dem Ausgangszeitpunkt der Funktion, da diese Kante sofort traversiert werden kann.
Handelt es sich um eine Verbindung von einem Haltestellenknoten zu einem Haltepunktknoten, entspricht dies dem zweiten Fall einer Umstiegskante. Das heißt, hier wird die eigentliche Umstiegszeit berücksichtigt (vgl. Kapitel 2.2). Der Abfahrtszeitpunkt wird ebenso wie zuvor durch den Ausgangszeitpunkt repräsentiert. *Duration* wird hingegen durch Selektion des Transferplans nach der Kennnummer der Ausgangshaltestelle ermittelt und vor der Rückgabe mit sechzig multipliziert, da der Transferplan die Umstiegszeit in Minuten angibt.
Der letzte Verbindungstyp ist eine Routenkante, also eine Verbindung von zwei Haltepunkten. Handelt es sich um eine solche, wird zunächst ein Kanten-Zeitplan (*edgeTimetable*) für den aktuellen Tag erstellt, welcher die Daten des Fahrzeitplans (*rideTimetable*) aus *RoutingDataProcessed* enthält, die ausschließlich die aktuelle Verbindung betreffen. Dazu wird der Fahrzeitplan bezüglich der Ausgangs- und Zielknoten sowie der Liniennummer der betrachteten Verbindung selektiert. Anschließend wird der letzte Abfahrtszeitpunkt des aktuellen Tages aus dem Kanten-Zeitplan ermittelt, was durch Ausführung der *tablesaw*-Aggregatsfunktion *max* auf die Abfahrtsspalte (*DEPARTURE*) umgesetzt wird.
Weiterhin wird die Liniennummer ermittelt, mit welcher der Ausgangsknoten momentan erreicht wird. Dies geschieht durch die Abfrage des Attributs *reachedByLineNr* des zum Ausgangsknoten zugehörigen Wegpunktes, welcher durch die Methode *getWayPointFromNode* des Weg-Objekts zurückgegeben wird.
Bei der Berechnung der Transportzeit einer Routenkante stellt sich als erstes die Frage, ob diese für den aktuellen oder den nächsten Tag berechnet werden soll. Dafür wird der Funktionsparameter *onlyNextDay* und ein Vergleich der letzten Abfahrt des aktuellen Tags mit dem Zeitpunkt der Verbindung verwendet. Ist *on-*

lyNextDay auf wahr gesetzt, oder ist der letzte Abfahrtszeitpunkt der Verbindung größer gleich der Zeitmarke zur Berechnung der Transportzeit, wird die Transportzeit für den nächsten Tag berechnet, andernfalls für den aktuellen Tag.
Für die Berechnung der Transportzeit einer Routenkante wird die im Rahmen des Projekts entwickelte Funktion *getEdgeTimetableWithOnlyEarliestArrival* verwendet. Diese gibt die Einträge des Kanten-Zeitplans, mit der kleinesten Ankunftszeit für die zugehörige Verbindung zurück. Dabei wird der kleinste Ankunftszeitpunkt (*ARRIVAL*) durch die *tablesaw* Aggregatsfunktion *min* ermittelt und *edgeTimetable* anschließend nach diesem gefiltert.
Soll nun die Transportzeit einer Routenkante für den aktuellen Tag ermittelt werden, wird der Kanten-Zeitplan so selektiert, dass nur noch die Einträge enthalten sind, deren Abfahrtszeitpunkt (*DEPARTURE*) später beziehungsweise größer ist, als der Zeitpunkt-Parameter der Funktion. Anschließend wird diese Tabelle mit der eben beschriebenen Funktion *getEdgeTimetableWithOnlyEarliestArrival* weiterverarbeitet und enthält somit nur noch die Werte der Verbindung nach dem angegebenen Zeitpunkt, welche die kürzeste Ankunftszeit besitzen. Mit diesem Ergebnis wird ein *TransportTime*-Objekt erstellt und zurückgegeben, wobei *duration* dem Ankunftszeitpunkt (*ARRIVAL*) abzüglich der Sekunden des angegebenen Zeitpunkts und *deparutre* der Abfahrt (*DEPARTURE*) aus dem zuvor angepassten Kanten-Zeitplan entspricht.
Ergibt sich aus den zuvor genannten Bedingungen, dass die Transportzeit für den Folgetag berechnet werden soll, wird zunächst ein neuer Kanten-Zeitplan für diesen nach dem gleichen Prinzip wie für den aktuellen Tag erstellt. Der einzige Unterschied besteht darin, dass nun der Fahrzeitplan für den Folgetag (*rideTimeTableNextDay*) aus dem Klassenattribut *routingDataProcessed* verwendet wird.
Anschließend wird überpüft, ob die gewählte Verbindung überhaupt für den Folgetag existiert, denn es wäre zum Beispiel möglich, dass eine Linie an einem Tag fährt, am darauf folgenden jedoch nicht. Dazu wird die Reihenanzahl des Kanten-Zeitplans überprüft. Entspricht diese null, gibt es die gewählte Verbindung nicht am Folgetag. Dann wird die Dauer und Abfahrtszeit, welche in Sekunden angegeben werden, im Rückgabe-Objekt auf "*Integer.MAX_VALUE / 2*" gesetzt, was umgerechnet mehr als dreißig Jahre wären. Hierdurch ist die zugehörige Verbindung im angepassten Dijkstra-Algorithmus nicht mehr relevant für einen realistischen Anwendungsfall. Die Halbierung wird durchgeführt, damit es, wenn die Dauer mit einem anderen Wert verrechnet wird, nicht zu einem arithmetischen Überlauf kommt.
Wird der Kanten-Zeitplan für den Folgetag korrekt erstellt, wird noch zwischen den Fällen unterschieden, ob sich der gewählte Zeitpunkt der Verbindung bereits am Folgetag befindet oder nicht, also die ob Sekundenanzahl nach null Uhr größer als 86400 (24 Stunden) ist.

Befindet sich der Zeitpunkt am nächsten Tag, wird die Transportzeit analog zum aktuellen Tag berechnet und zurückgegeben. Die Unterschiede belaufen sich lediglich auf die Verwendung des Kanten-Zeitplans für den folgenden Tag statt des aktuellen und es werden vom übergebenen Zeitpunkt 86400 abgezogen, um die Sekunden nach null Uhr für den Folgetag zu erhalten, da diese für die Berechnung benötigt werden, anstatt den Sekunden nach null Uhr des vorherigen Tages. Diese abgezogenen Sekunden werden vor der Rückgabe wieder auf *departure* addiert.
Falls der gewählte Zeitpunkt noch im Bereich des ersten Tags liegt, aber sich die letzte Abfahrt dieses Tages vor jener Uhrzeit befindet, soll die Transportzeit für die frühestmögliche Ankunft am Folgetag berechnet werden. Dafür ist es nur notwendig, den Kanten-Zeitplan mit der Funktion *getEdgeTimetableWithOnlyEarliestArrival* anzupassen und den frühesten Ankunftszeitpunkt (*ARRIVAL*) abzüglich des angegebenen Zeitpunkts als *duration* und die Abfahrtszeit (*DEPARTURE*) als *departure* in einem *TransportTime*-Objekt zu vereinen. Bevor diese darin zurückgegeben werden, wird auf die beiden Zeiten noch 86400 addiert, da die Verbindung für den nächsten Tag berechnet wurde.
Möchte man nun den angepassten Dijkstra-Algorithmus, welcher durch die Methode *shortestWay* umgesetzt wurde, nutzen, benötigt dieser als Parameter, die Kennnummer der Start- und Ziel-Haltestelle, sowie den Zeitpunkt, ab welchem der zeitlich kürzeste Weg errechnet werden soll. Die Funktionsweise wurde bereits in Algorithmus 1 dargestellt (vgl. Kapitel 2.3).
Als erstes wird in dieser Methode ein neues *Way*-Objekt erzeugt, in welchem die Informationen für den Weg gespeichert werden. Weiterhin wird ein Array für die Zeitmarken der einzelnen Knoten erstellt. Dabei entspricht der Index eines Knotens in der Liste aller Knoten, die im Graph-Attribut *nodes* abgespeichert sind, dem Index des selben Knotens im Zeitmarken-Array. Möchte man also die aktuelle Zeitmarke eines Knotens herausfinden, kann man diese an der Index-Stelle im Zeitmarken-Array nachsehen, an der sich der Knoten in der Liste aller Knoten befindet. Die Designentscheidung, für diese Informationen extra Objekte innerhalb der Funktion zu erstellen und nicht direkt bei den *Node*-Objekten zu speichern, wurde im Rahmen dieser Arbeit getroffen, um ein paralleles Arbeiten mit diesen zu ermöglichen, da der Graph und seine Knoten beziehungsweise Kanten bei der Lösung des *EAP* nicht mehr verändert werden. Damit könnten beispielsweise mehrere Threads die Funktion *shortestWay* ausführen, ohne den kompletten Graph dafür kopieren oder neu erstellen zu müssen.
Die Prioritätswarteschlange wird hierbei durch eine Liste von Knoten und die in Rahmen dieses Projekts entwickelte Funktion *getNodeWithLowestCost* umgesetzt, welche als Parameter die Liste und das Array der Zeitmarken erwartet. Beim Funktionsaufruf wird über die Liste iteriert und dabei die aktuelle Zeitmar-

ke mit der bisher kleinsten gefundenen Zeitmarke verglichen. Ist die aktuelle Zeitmarke kleiner, wird die Variable der kleinsten Zeitmarke aktualisiert, sowie der zugehörige Knoten zwischengespeichert. Nach dem Ablauf der Iteration wird der Knoten mit der kleinsten Zeitmarke zurückgegeben.
Zusätzlich wurde die Funktion *getNodeByIdentifier* erstellt, mit welcher man einen Knoten, unter Angabe seiner Kennnummer und der Liste in welcher dieser enthalten ist, zurückerhalten kann. Dabei wird über die Liste iteriert und abgeglichen ob der Identifikator des Knotens des aktuellen Iterationsschrittes mit der übergebenen Kennnummer übereinstimmmt. Ist dies der Fall, wird der Knoten zurückgegeben.
In *shortestWay* wird nun eine Liste *openNodes* für die Prioritätswarteschlange und *closedNodes* für die Liste an permanent Abgearbeiteten Knoten erstellt.
Danach wird mit *getNodeByIdentifier* der Startknoten aus der Liste aller Knoten extrahiert und in die Prioritätswarteschlange eingefügt. Dabei wird die Zeitmarke des Startknotens im Zeitmarken-Array auf die Startzeit gesetzt. Zusätzlich wird dieser Knoten als Wegpunkt dem Weg-Objekt mithilfe der *addPoint*-Methode hinzugefügt.
Im Anschluss daran wird die Schleife ausgeführt, die erst beendet wird, sobald keine offenen Knoten mehr in *openNodes* enhalten sind. In jedem Iterationsschritt wird dabei ein Knoten mit dem momentan zeitlich kürzesten Weg als aktueller Knoten (*currentNode*) durch *getNodeWithLowestCost* aus der Liste der offenen Knoten extrahiert. Danach wird die Abbruchbedingung, ob die Kennnummer des aktuellen Knotens mit der des Startknotens übereinstimmt, abgefragt. Trifft dies zu wird das Attribut *isTarget* des Wegpunktes des aktuellen Knotens auf wahr gesetzt und die Schleife wird abgebrochen, da somit der zeitlich kürzeste Weg gefunden wurde.
Ist dies nicht der Fall, wird eine weitere Iteration über alle Kanten des aktuellen Knotens gestartet. Dabei wird als erstes der benachbarte Knoten, der durch die Kante des aktuellen Iterationsschrittes erreicht wird, über die *Edge*-Funktion *getToNode* ermittelt. Ist der Nachbarknoten bereits in der Liste der permanent abgearbeiteten Knoten enthalten, wird zum nächsten Iterationsschritt gesprungen. Ansonsten wird mithilfe von *getTransportTime* das *TransportTime*-Objekt erstellt, welches die Dauer, den Nachbarknoten zu erreichen und die zugehörige Abfahrtszeit enthält. Neben den Informationen über die Verbindung wird der Methode auch die Zeitmarke des aktuellen Knotens übergeben. Repräsentiert der aktuelle Knoten die Start-Haltestelle, wird in der Variable *arrivalTime* die Ankunftszeit am Nachbarknoten auf die Zeitmarke des aktuellen Knotens gesetzt. Damit hat diese Verbindung eine Dauer von null, was darstellen soll, dass an der Start-Haltestelle keine Umstiegszeiten herrschen. Dies kann so umgesetzt werden, da die Starthaltestelle, wie jede Haltestelle, nur Verbindungen zu Halte-

punkten und somit auch nur Umstiegskanten besitzt (vgl. Kapitel 2.2). Handelt es sich beim aktuellen Knoten nicht um die Start-Haltestelle, wird die Ankunftszeit (*arrivalTime*) am Nachbarknoten der aktuellen Kante mithilfe des *TransportTime*-Objekts berechnet. Die Ankunftszeit setzt sich hierbei aus der Zeitmarke des aktuellen Knotens, addiert mit *duration* aus dem *TransportTime*-Rückgabeobjekt, zusammen.
Nun wird überprüft, ob die Ankunftszeit am Nachbarknoten geringer ist, als dessen bisherige Zeitmarke. Ist dies der Fall, wurde eine Verbesserung des aktuell kürzesten Wegs gefunden, weshalb dieser aktualisiert werden muss. Dabei wird der Nachbarknoten als erstes in die Liste der offenen Knoten (*openNodes*) eingetragen. Danach wird das Zeitmarken-Array an der Indexstelle des aktuellen Knotens mit der neuen Ankunftszeit aktualisiert. Als letztes wird der dem aktuellen Knoten zugehörige Wegpunkt im Weg-Objekt mit *addPoint* hinzugefügt, beziehungsweise aktualisiert. Dies ist der letzte Schritt in der Iteration über die Kanten des aktuellen Knotens.
Nachdem diese Abgeschlossen wurde, wird der aktuelle Knoten in die Liste der permanent abgearbeiteten Knoten hinzugefügt und falls es noch offene Knoten gibt, also *openNodes* noch Elemente besitzt, wird der nächste Iterationsschritt der äußeren Schleife gestartet.
Anzumerken ist hierbei noch, dass sich einzelne Zwischenschritte des Algorithmus auf der Kommandozeile ausgeben lassen, insofern das *Graph*-Attribut *displayOnConsole* auf wahr gesetzt wurde. Dies geschieht an zwei Stellen. Zum einen zu Beginn jedes Iterationsschrittes über die Liste der noch zu bearbeitended Knoten (*openNodes*). Hierbei wird die Information über den aktuell gewählten Knoten (*currentNode*) ausgegeben, wie beispielsweise den zugehörigen Haltestellennamen und die Zeitmarke an der dieser Knoten momentan erreicht wird. Die Formatierung der Zeitmarke von Sekunden nach null Uhr zu einer leserlichen Uhrzeit und die Umwandlung der Haltestellenkennnummer in den zugehörigen Namen, wird durch eine Instanz der Klasse *DataTransformer* ermöglicht (vgl. Kapitel 3.2.2). Zusätzlich werden bei einer Verbesserung des momentan zeitlich kürzesten Wegs, nachdem errechnet wurde, dass man über eine Kante einen Knoten schneller erreichen kann als zuvor, Informationen über diese Verbindung ausgegeben. Dabei wird unterschieden, welcher Knoten-Typ (*NodeType*) so erreicht wird. Handelt es sich um einen Haltepunkt (*STOPPING_POINT*), wird unterschieden, ob dieser durch eine Routenkante erreicht wurde, oder ob es sich um einen Umstieg handelt. Neben den Namen der zugehörigen Haltestellen wird auch die Zeit ausgegeben, an welcher man den Knoten mit einer verbesserten Zeitmarke erreichen kann. Dafür wird ebenfalls ein *DataTransformer*-Objekt verwendet. Die Ausgabe der Informationen erfolgen dabei durch die *print*-Methode des Java-Pakets *java.lang.System.out*.

3.4 Implementierung einer REST Schnittstelle

Die Lösung des *EAP* im ÖPNV Regensburgs soll auch über eine Representational State Transfer (REST)-Schnittstelle abgefragt werden können. Um diese im Rahmen dieses Projekts zu implementieren, wurde auf das Application Programming Interface (API) *JAX-RS* zurückgegriffen, da dieses die Umsetzung einer REST-Schnittstelle im Rahmen von Webservices in Java ermöglicht.[JAX-RS 2020]
Weiterhin wird auf das *Jersey*-Framework verwendet, da dieses die Möglichkeit einer einfache Implementierung eines Webservices und Unterstützung für *JAX-RS* bietet.[Jersey 2020]
Der kürzeste Weg kann bei einem Aufruf dabei nicht direkt als Weg-Objekt zurückgegeben werden, da die einzelnen Knoten so untereinander verbunden sind, dass rekursive Abhängigkeiten bestehen können.
Daher wird der kürzeste Weg als *ShortestWayDTO* zurückgegeben, welches eine Liste an *RoutingParts* enthält, die die einzelnen Wegstücke darstellen. *RoutingParts* besitzt dabei folgende Attribute:

- fromId: Kennnumer des Start-Haltepunkts oder der Start-Haltestelle des Wegstücks
- toId: Kennnumer des Ziel-Haltepunkts oder der Ziel-Haltestelle des Wegstücks
- fromCSP: Kennnummer der zugehörigen Haltestelle des Startpunktes des Wegstücks
- toCSP: Kennnummer der zugehörigen Haltestelle des Zielpunktes des Wegstücks
- fromName: Name der Starthaltestelle des Wegstücks
- toName: Name der Zielhaltestelle des Wegstücks
- lineNr: Liniennummer der Verbindung
- departureTime: Abfahrtszeit der Verbindung
- arrivalTime: Ankunftszeit der Verbindung
- nextDay: Angabe, ob diese Verbindung am nächsten Tag stattfindet

Im Zuge dieser Arbeit wurde als Webservice die Klasse *RoutingService* zusammen mit dem Interface *RoutingServiceIF* entwickelt. Bei der Instanziierung von *RoutingService* wird durch einen *ConfigDayDataLoader* die Konfigurationsdatei für den Tag und den Datensatz generiert, welche wiederum durch eine Instanz

der Klasse *DataInOut* verwendet wird, um ein Objekt der Klasse *RoutingDataProcessed* zu erstellen. Mit *RoutingDataProcessed* wird nun über eine Instanz der Klasse *GraphSetup* der Graph zur Lösung des *EAP* erzeugt.
Der Service stellt die Methode *getShortestWay* zur Verfügung, welche mithilfe des *Graph*-Objekts ein *Way*-Objekt erstellt. Durch den Aufruf der Methode *getShortestWayPointsInOrder* wird aus dem Weg eine Liste der Wegpunkte des zeitlich kürzesten Wegs erstellt.
Daraufhin wird über diese Liste iteriert, wobei der Vorgänger des aktuellen Iterationsschrittes zwischengespeichert wird. Aus den Informationen der Wegpunkte beziehungsweise deren Vorgänger wird zu jeder Verbindung zweier Wegpunkte ein *RoutingPart*-Objekt erstellt und in einem *ShortestWayDTO* abgespeichert. Dieses wird nach dem Ende der Iteration zurückgegeben.
Zusätzlich wurde, um eingehende REST-Aufrufe verarbeiten zu können, die Klasse *RestResource* erstellt, welche mit der Annotation *@Path(“shortestWay”)* versehen ist. Weiterhin wird der Service in diese Klasse durch ein *RoutingServiceIF*-Interface mit der Annotation *@Inject* injiziert. Zusätzlich wurde die Methode *makeShortestWay* entwickelt, welche die Pfadparameter (*@PathParam*) *startId* (Integer), *targetId* (Integer) und *time* (String in Form: hh:mm:ss) erwartet und mit einer *@GET* Annotation versehen ist. Die IDs entsprechen dabei Haltestellenkennnummern und *time* gibt den Zeitpunkt an, zu welchem der Weg berechnet werden soll. Bei einem GET-Aufruf werden die Parameter der Methode *getShortestWay* des injizierten Services übergeben und durch dieses ein *ShortestWayDTO* erstellt, welches aufgrund einer zusätzlichen *@Produces(“application/json”)*-Annotation als JSON-Objekt zurückgegeben wird.
Um *Jersey* zu konfigurieren wurde in diesem Projekt die Klasse *JerseyConfiguration* erstellt, welche von der Klasse *ResourceConfig* aus dem *Jersey*-Paket erbt. In deren Konstruktor werden durch die *register*-Funktion die für die REST-Aufrufe nötigen Klassen registriert. Das ist zum einen *RestResource*, *RoutingService* und die Klasse *JacksonFeature*, welche dazu verwendet wird das Rückgabeobjekt beim REST-Aufruf automatisch in das JSON-Format umzuwandeln. *RoutingService* wird hier außerdem als *Singleton* deklariert, das heißt es wird nur eine Instanz dieses Services erstellt und für jeden REST-Aufruf verwendet. Möglich wird dies dadurch, dass die Attribute des im Service enthaltenen Graph-Objekt, wie zum Beispiel die Knoten oder die verarbeiteten Routen-Daten (*RoutingDataProcessed*) sich zur Laufzeit nicht mehr ändern (vgl. Kapitel 3.3.4). Würden nämlich beispielsweise Werte in den Knoten des Graphen bei der Abfrage nach dem kürzesten Weg geändert werden und der Graph befände sich in einem Singleton-Service, könnte es hier bei zeitgleichen Aufrufen zu Race-Conditions kommen. Das Entwurfsmuster Singleton wurde hierbei gewählt, damit nicht bei jedem REST-Aufruf ein neuer Graph erstellt werden muss, was bei zu vielen Anfragen zu einer hohen Speicher-

belastung des ausführenden Systems führen würde.

3.5 Ausführung des Programms

Für die Ausführung des Programms wurde im Rahmen dieser Arbeit die Klasse *EntryPoint* erstellt, welche die *main*-Methode der Java-Applikation enthält. Zum einen soll diese Klasse die Lösung des *EAP* auf der Kommandozeile möglich machen, beziehungsweise erleichtern, zum anderen soll, falls gewünscht, der REST-Service gestartet werden können.
Die Nutzereingaben sollen hierbei, neben den beschriebenen Konfigurations-Dateien, über Kommandozeilenparameter erfolgen. Hierfür wurde die Java-Bibliothek *JOpt simple* genutzt, da sich mit dieser auf einfache Art die Eingabe von Kommandozeilenparametern umsetzten lässt.[JOPT 2018]
Die verwendeten Kommandozeilenparameter werden in Tabelle 1 dargestellt, wobei bei manchen eine kurze und lange Schreibweise möglich ist.
Diese Parameter werden in der *main*-Funktion als Strings deklariert und der Funktion *acceptsAll* eines *OptionParsers* der *JOpt simple* Bibliothek übergeben. Dabei wird bei den Optionen wie beispielsweiese *"-start/-s"* durch die Funktion *withRequiredArg* der *JOpt simple* Bibliothek festgelegt, dass ein Argument für diese Option benötigt wird. Das *OptionParser*-Objekt erstellt nun durch die Funktion *parse* ein *OptionSet*, mit welchem man die einzelnen Optionen und deren Arguemente abfragen kann.
Anschließend wird abgefragt, ob die Option *"-restapi/-r"* gesetzt wurde. Falls nicht, soll die im Rahmen der Arbeit erstellte Funktion *localExec* ausgeführt werden. Damit wird das EAP für die angegebenen Optionen gelöst und lokal auf der Kommandozeile ausgegeben. Durch einen *ConfigDayDataLoader* wird hierbei die Konfigurationsdatei für den Tag und den Datensatz generiert, welche wiederum durch eine Instanz der Klasse *DataInOut* verwendet wird, um ein Objekt der Klasse *RoutingDataProcessed* zu erstellen.
Wurde dem Programm die Option *"-createNameIdentFiles"* übergeben, wird die Funktion *makeRoutingDataPreProcessedFromData* des *dataInOut*-Objekts ausgeführt, um *routinDataProcessed* sicher aus den Rohdaten neu zu erstellen und nicht eine zuvor abgespeicherte Version zu laden.
Als nächstes wird mit dem *OptionSet* überprüft, ob die Option *"-createNameIdentFiles"* gewählt wurde. Ist dies der Fall, wird mit *writeNameIdentFiles* des *DataInOut*-Objekts die Nachschlagedatei erstellt.
Ist die Option *"-conditions"* vorhanden, wird mithilfe eines *ConditionChecker*-Objekts die *FIFO*-Bedingung und die Einzigartigkeit der Haltestellennamen über-

Optionen	**zugehöriges Argument**	**Beschreibung**
-s -start	ID (Integer)	Angabe der Kennnummer der Starthaltestelle
-t -target	ID (Integer)	Angabe der Kennnummer der Zielhaltestelle
-d -daytime	“hh:mm:ss” (String)	Angabe des Zeitpunkts zur Berechnung des EAP
-c -conditions		Bedingungen (FIFO und Einzigartigkeit der Haltestellennamen) werden bei Programmstart geprüft
-p -print		Zwischenschritte des Algorithmus werden auf der Kommandozeile ausgegeben
-r -restapi		Programm wird als REST-Service gestartet
-createNameIdentFiles		Dateien (*nameIdent...Data.txt*) mit Namen und Kennnummern der Haltestellen werden für den jeweiligen Datensatz erstellt
-updateRoutingData		*RoutingDataProcessed* wird zwangsweise neu aus den Rohdaten erstellt (nötig bei Änderungen in *configDayData.yaml*)
-h -help		Hilfe über die Funktionsweise des Programms wird auf der Kommandozeile ausgegeben

Tabelle 1: Tabelle der Kommandozeilenparameter

prüft.
Anschließend wird die Existenz der Start-, Stop- und Zeitpunkt-Option geprüft. Falls diese nicht vorhanden sind, wird das Programm beendet, da der Weg ohne Eingabe nicht berechnet werden kann.
Jetzt wird ein *GraphSetup*-Objekt instanziiert, in welchem, für den Fall, dass die *“-print/-p”* Option verwendet wurde, das Attribut *displayOnConsole* auf wahr gesetzt wird, um die einzelnen zwischenschritte des angepassten Dijkstra-Algorithmus auszugeben. Weiterhin wird mit diesem Objekt ein *Graph*-Objekt erzeugt, welches unter Verwendung der Argumente der Optionen Start, Ziel und Zeitpunkt das *Way*-Objekt und damit die Lösung des EAP liefert.
Die Eingaben werden dabei durch eine Instanz der Klasse *DataTransformer* in die benötigte Form gebracht.
Zuletzt soll der zeitlich kürzeste Weg in Kurzform ausgegeben werden. Dafür wird

mit der Methode *getShortestWayPointsInOrder* des *Way*-Objekts die Liste der Wegpunkte des kürzesten Wegs erstellt, über welche Iteriert wird. Dabei werden die Wegpunkte entsprechend formatiert auf der Kommandozeile ausgegeben.
Um den *REST*-Service auszuführen, sollen die erstellten Klassen hierfür als Jersey-Servlet auf einem Webserver ausgeführt werden. Als Server wird dabei *Apache Tomcat* in das Programm eingebettet.[Tomcat 2018] Wurde die Option zum Starten eines Rest-Services gewählt, so wird in der *main*-Methode die hierfür erstellte Funktion *start* ausgeführt. In dieser wird eine Instanz der Klasse *Tomcat* aus dem Paket *org.apache.catalina.startup.Tomcat* erzeugt. Dieser wird ein neu erstellter *ServletContainer*, welchem die im Rahmen dieser Arbeit erstellte *JerseyConfiguration* übergeben wird, hinzugefügt. Außerdem wird hier für das Servlet-Mapping der Pfad auf "/api/*" gesetzt. Anschließend wird der Server über die Funktion *start* des *Tomcat*-Objekts gestartet.
Unter Berücksichtigung der Pfad Annotationen lautet die Struktur eines *GET*-Aufrufs für den zeitlich kürzesten Weg:

http://<Server>:<Port>/api/shortestway/<FromId>/<ToId>/<time>
(Beispiel: http://localhost:8080/api/shortestway/297/128/23:59:00)

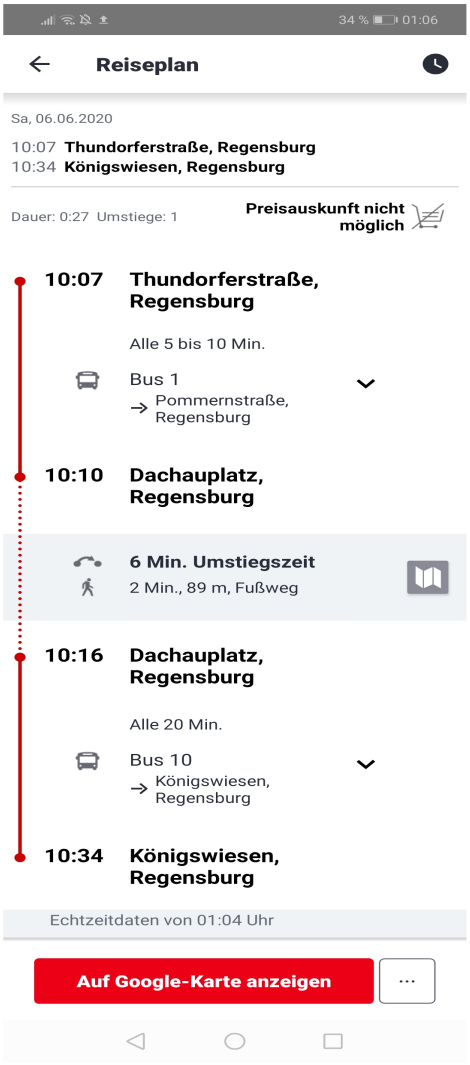
Als Build-Managment-Tool wurde *Maven* verwendet. [Maven 2020] Mit diesem lassen sich alle benötigten Abhängigkeiten (Dependencies) verwendeter Bibliotheken in einer *POM.xml*-Datei angeben und im Projekt verwenden. Damit diese Dependencies in die *Jar*-Datei des kompilierten Programmes inkludiert werden, wurde zusätzlich das *Maven-shade*-Plugin verwendet.[ShadeDoku 2020]
Anzumerken ist noch, dass zur Ausführung des Programms Java 11 oder höher verwendet werden muss, da die Versionen einiger Dependencies nicht mehr mit früheren Versionen von Java kompatibel sind.
Weiterhin müssen für die Ausführung der kompilierten Jar-Datei die Konfigurationsdateien und die Rohdaten im gleichen Verzeichnis wie diese liegen.

4 Test

Um die Funktionsweise des Programms und die jeweiligen Datensätze zu evaluieren, wurden einige Beispielabfragen zur Findung des zeitlich kürzesten Wegs durchgeführt.
Da die Daten aus der VDV-Schnittstellenbeschreibung (Stand 22.07.2020) nicht mehr den aktuellen Fahrplänen entsprechen, ist es schwierig die Lösungen des EAP mit den aktuellen Verbindungen zu vergleichen. Daher werden die Lösungen des EAP mit den Lösungen, die in der Masterarbeit *Optimierung des ÖPNV am Beispiel einer Passagierroutenplanung in Regensburg* durch das R-Skript errechnet wurden, verglichen. [Huber 2019] Da hierbei alle Lösungen übereinstimmen, wird damit davon ausgegangen, dass der angepasste Dijkstra-Algorithmus korrekt funktioniert.
Das bedeutet jedoch auch, dass dieser noch weitere in der Arbeit von Helena Huber beschriebene Verbesserungsmöglichkeiten besitzt.[Huber 2019] Zum einen wird die Möglichkeit, zu Fuß von einer Haltestelle zur nächsten zu gehen, nicht miteinbezogen. Zum anderen werden Routen mit gleicher Ankunftszeit, aber weniger Umstiegen, nicht priorisiert. Die Erweiterung der Fahrplandaten für den Folgetag wurde im Rahmen dieses Projekts miteinbezogen und nicht nur der Fahrplan des selben Tages wiederholt.
Nun soll noch anhand von Beispielen mit den aktuellen Daten des RVV, die Lösungen dieses Projekts mit denen eines laufenden Produktivsystems der Deutschen Bahn verglichen werden. Daher werden die Lösungen des EAP, die mit diesen Daten errechnet wurden, mit denen des *DB-Navigators* verglichen. Dabei wird zum einen die Verbindung zwischen der Haltestelle Thundorferstraße und Königswiesen um 10:00:00 am 6.6.2020 betrachtet (vgl. Abbildung 12). Die Berechnung des frühest möglichen Ankunftszeitpunkt führt bei beiden Systemen zum gleichen Ergebnis, wobei hier anzumerken ist, dass der Algorithmus dieser Arbeit nicht den Komfort des Nutzers miteinbezieht. So liefert die Applikation der Deutschen Bahn eine Strecke mit nur einem Umstieg, der angepasste Dijkstra-Algorithmus liefert eine Lösung mit zwei Umstiegen.
Als nächstes wurden die kürzesten Wege der Strecke Wöhrdstraße nach Regensburg IKEA um 7:30:00 am 4.5.2020 verglichen (vgl. Abbildung 13). Die Ankunftszeitpunkte sind hier wiederum identisch, jedoch beträgt die reine Fahrzeit des Programms dieser Arbeit inklusive Umstiegszeiten eine Stunde und zwanzig Minuten, die des DB-Navigators hingegen nur fünfunddreißig Minuten. Das liegt daran, dass der *DB-Navigator* einen deutlich späteren Abfahrtszeitpunkt an der Starthaltestelle vorschlägt. Allerdings ist die Zeit die man vom angegebenen Zeitpunkt (7:30:00) bis zur Zielhaltestelle benötigt in beiden Fällen gleich, wo-

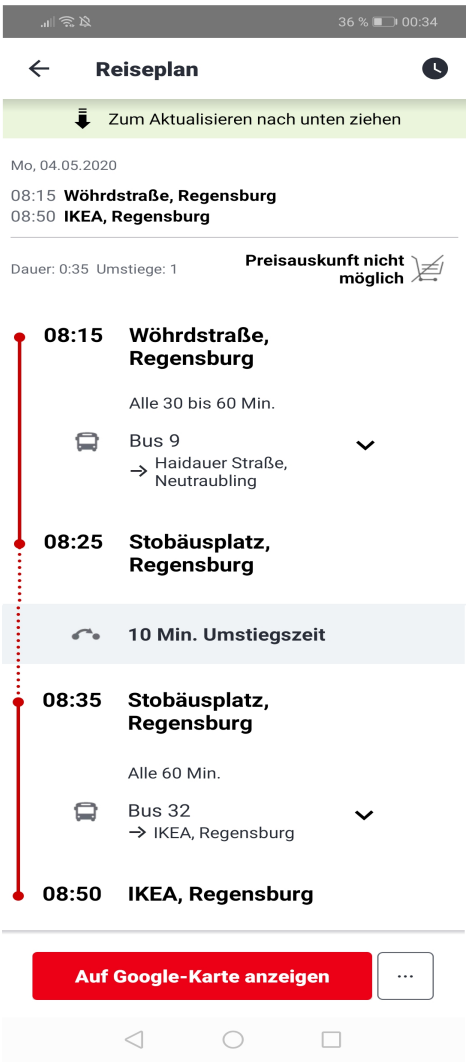
mit der Algorithmus dieser Arbeit durchaus ein korrektes Ergebnis liefert.
Schlussendlich wird noch eine Streckenverbindung betrachtet, deren Ankunftszeitpunkt zwangsläufig am nächsten Tag liegen muss. Dafür wurde wieder die Strecke Wöhrdstraße nach Regensburg IKEA am 4.5.2020 gewählt, jedoch der Startzeitpunkt auf 23:59:00 gesetzt (vgl. Abbildung 14). Sowohl das Programm dieses Projekts, als auch das der Deutschen Bahn, liefern als Ankunftszeitpunkt 6:50:00 am nächsten Tag. Damit wird gezeigt, dass das Verhalten des Programms für Verbindungen die über den aktuellen Tag hinausgehen korrekt funktioniert.

Überprüft man die *FIFO*-Bedingung beim Programmstart durch den entsprechenden Parameter, ergibt sich, dass diese für die Daten der VDV-Schnittstelle erfüllt ist. Die aktuellen Test-Daten des RVV hingegen weisen zum Beispiel am 04.05.2020 einige Verbindungen auf, die diese Bedingung verletzten. Damit kann an diesem Tag nicht mehr die Korrektheit des Ergebnisses garantiert werden. Am 06.06.2020 andererseits ist die FIFO-Bedingung wiederum für die erfüllt.



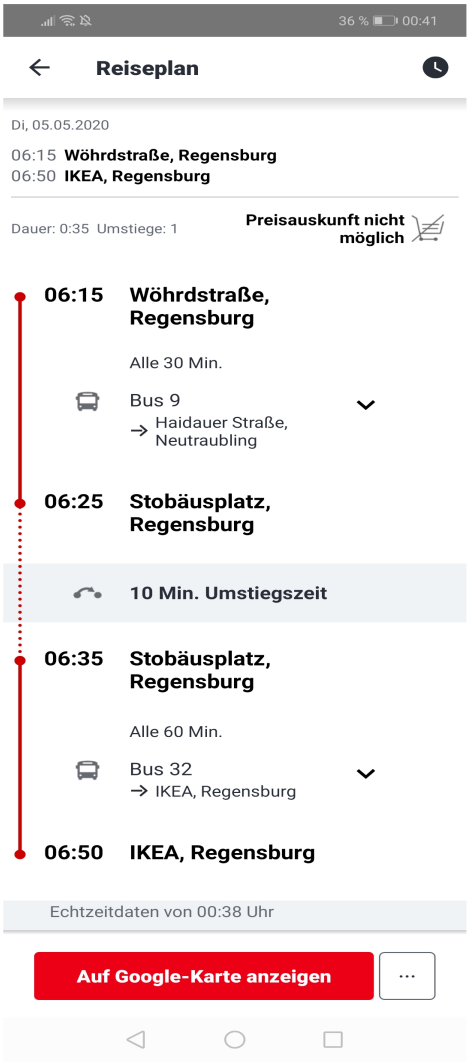

```
You are at Regensburg Thundorferstraße at 10:0:0
Reach Regensburg Fischmarkt at 10:6:0 with line 2 (drive at: 10:5:0)
Reach Regensburg Keplerstraße at 10:7:0 with line 2 (drive at: 10:6:0)
Reach Arnulfsplatz at 10:9:0 with line 2 (drive at: 10:7:0)
Reach Regensburg Bismarckplatz at 10:10:0 with line 2 (drive at: 10:9:0)
Reach Regensburg Justizgebäude at 10:12:0 with line 2 (drive at: 10:10:0)
Transfer
Reach Regensburg Gutenbergstraße at 10:14:0 with line 7 (drive at: 10:12:0)
Reach Regensburg Asamstraße at 10:15:0 with line 7 (drive at: 10:14:0)
Reach Regensburg Simmernstraße at 10:16:0 with line 7 (drive at: 10:15:0)
Reach Regensburg Klenzestraße at 10:18:0 with line 7 (drive at: 10:16:0)
Transfer
You are at Regensburg Klenzestraße at 10:21:0
Reach Regensburg Kirche St.Paul at 10:32:0 with line 10 (drive at: 10:31:0)
Reach Regensburg Königswiesen at 10:34:0 with line 10 (drive at: 10:32:0)
```

Abbildung 12: Vergleich der zeitlich kürzesten Wege von der Haltestelle Thundorferstraße nach Königswiesen um 10:00:00




```
You are at Regensburg Wöhrdstraße at 7:30:0
Reach Regensburg Weichs-DEZ at 7:32:0 with line 3 (drive at: 7:30:0)
Transfer
You are at Regensburg Weichs-DEZ at 7:36:0
Reach Regensburg Weißenburgstraße at 7:39:0 with line 5 (drive at: 7:37:0)
Transfer
You are at Regensburg Weißenburgstraße at 7:42:0
Reach Regensburg Prinz-Ludwig-Straße at 7:45:0 with line 10 (drive at: 7:43:0)
Reach Regensburg Zuckerfabrikstraße at 7:46:0 with line 10 (drive at: 7:45:0)
Reach Regensburg Liebigstraße at 7:48:0 with line 10 (drive at: 7:46:0)
Transfer
You are at Regensburg Liebigstraße at 7:50:0
Reach Regensburg Bukarester Straße at 7:57:0 with line 30 (drive at: 7:55:0)
Transfer
You are at Regensburg Bukarester Straße at 7:59:0
Reach Rgbg. Peter-Henlein-Straße at 8:43:0 with line 32 (drive at: 8:42:0)
Reach Regensburg Kremser Straße at 8:45:0 with line 32 (drive at: 8:43:0)
Reach Regensburg, Irl at 8:47:0 with line 32 (drive at: 8:45:0)
Reach Regensburg Sulzfeldstraße at 8:49:0 with line 32 (drive at: 8:47:0)
Reach Regensburg IKEA at 8:50:0 with line 32 (drive at: 8:49:0)
```

Abbildung 13: Vergleich der zeitlich kürzesten Wege von der Haltestelle Wöhrdstraße nach Regensburg IKEA um 7:30:00




```
You are at Regensburg Wöhrdstraße at 23:59:0
Reach Regensburg Weichs-DEZ at 5:4:0 with line 13 (drive at: 5:1:0)
Transfer
You are at Regensburg Weichs-DEZ at 5:8:0
Reach Regensburg Weißenburgstraße at 5:29:0 with line 5 (drive at: 5:27:0)
Transfer
You are at Regensburg Weißenburgstraße at 5:32:0
Reach Regensburg Prinz-Ludwig-Straße at 5:45:0 with line 10 (drive at: 5:43:0)
Reach Regensburg Zuckerfabrikstraße at 5:46:0 with line 10 (drive at: 5:45:0)
Reach Regensburg Liebigstraße at 5:48:0 with line 10 (drive at: 5:46:0)
Transfer
You are at Regensburg Liebigstraße at 5:50:0
Reach Regensburg Bukarester Straße at 6:9:0 with line 33 (drive at: 6:8:0)
Transfer
You are at Regensburg Bukarester Straße at 6:11:0
Reach Rgbg. Peter-Henlein-Straße at 6:43:0 with line 32 (drive at: 6:42:0)
Reach Regensburg Kremser Straße at 6:45:0 with line 32 (drive at: 6:43:0)
Reach Regensburg, Irl at 6:47:0 with line 32 (drive at: 6:45:0)
Reach Regensburg Sulzfeldstraße at 6:49:0 with line 32 (drive at: 6:47:0)
Reach Regensburg IKEA at 6:50:0 with line 32 (drive at: 6:49:0)
```

Abbildung 14: Vergleich der zeitlich kürzesten Wege von der Haltestelle Wöhrdstraße nach Regensburg IKEA um 23:59:00

5 Fazit

Im Rahmen dieser Arbeit wurde eine Software zur Lösung des Problems erstellt, den zeitlich kürzesten Weg zu einem gewählten Zeitpunkt von einer Haltestelle zu einer anderen zu finden. Zusätzlich wurde eine *REST*-Schnittstelle implementiert, um Abfragen dieses Wegs von anderen Systemen aus nutzen zu können.
Ein wichtiger Teil hierbei ist die Verarbeitung der Daten in eine Datenstruktur, die nicht mehr von den Rohdaten abhängig ist. Damit können die unterschiedlichen Datensätze vom selben Algorithmus bearbeitet werden.
Weiterhin wurde der angepasste Dijkstra-Algorithmus so umgesetzt, dass auch Verbindungen, die den nächsten Tag betreffen, korrekt integriert sind.
Das die Tests betreffende Kapitel zeigt hierbei, dass die berechneten Lösungen für die Daten aus der VDV-Schnittstellenbeschreibung zu einem korrekten Ergebnis führen. Wird der interne Datensatz des RVV verwendet, führt dies, verglichen mit etablierter Software, zu richtigen Ergebnissen. Diese Daten verletzten jedoch die *FIFO*-Bedingung, weshalb für deren Korrektheit nicht garantiert werden kann. Das liegt vermutlich daran, dass die Datensätze entweder nicht richtig gepflegt wurden oder bestimmte Bedingungen in den Datensätzen nicht berücksichtigt wurden. Da jedoch keine Dokumentation vorliegt, sollte man falls möglich einen korrigierten Datensatz einpflegen. Die Daten aus der VDV-Schnittstelenbeschreibung erfüllen nämlich die mathematischen Bedingungen, um die Korrektheit dieses Datensatztes zu belegen. Zusätzlich kann der jetzige angepasste Algorithmus noch weiter Verbessert werden. Beispielsweise finden Fußwege zwischen unterschiedlichen Haltestellen keine Beachtung und dieser bezieht den Komfort des Nutzers nicht mit ein.
Insgesamt wurde im Zuge dieses Projekts eine mögliche Implementierung einer Routenplanungssoftware in Java dargestellt.

Abbildungsverzeichnis

Algorithmenverzeichnis

Listings

Tabellenverzeichnis

Literatur

[BJ 2004] Brodal, Gerth S.;Jacob, Riko: *Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries*, Aarhus-Dänemark, Zürich-Schweiz: ELSEVIER, 2004

[Huber 2019] Huber, Helena: *Optimierung des OPNV am Beispiel einer Passagierroutenplanung in Regensburg*, (Masterarbeit, Mathematik), Regensburg: Ostbayerische Technische Hochschule, 2019

[JAX-RS 2020] JAX-RS, https://github.com/jax-rs, zuletzt aufgerufen: 01.10.2020

[Jersey 2020] Jersey, https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/index.html, zuletzt aufgerufen: 05.10.2020

[JOPT 2018] Jopt Simple, http://jopt-simple.github.io/jopt-simple/, zuletzt aufgerufen: 08.10.2020

[Maven 2020] Maven, http://maven.apache.org/guides/index.html, zuletzt aufgerufen: 07.10.2020

[Pajor 2009] Pajor, Thomas: *Multi-Modal Route Planning*, (Diplomarbeit am Institut für Theoretische Informatik), Karlsruhe: Universität, 2009

[PSWZ 2004] Pyrga, Evangelia; Schulz, Frank; Wagner, Dorothea; Zarolia-gis, Christos: *Towards Realistic Modeling of Time-Table Informationthrough the Time-Dependent Approach*, Patras-Griechenland, Karlsruhe-Deutschland: ELSEVIER, 2004

[PSWZ 2007] Pyrga, Evangelia; Schulz, Frank; Wagner, Dorothea; Zaroliagis, Christos: *Efficient Models for Timetable Information in Public Transpor-tation Systems*, New York: ACM Journal of Experimental Algorithmics, 2007

[ShadeDoku 2020] Shade-Plugin, http://maven.apache.org/plugins/maven-shade-plugin/, zuletzt aufgerufen: 23.09.2020

[Schulz 2005] Frank Schulz: *Timetable Information and Shortest Paths*, (PhD thesis, Informatik), Karlsruhe: Universität (TH), 2005

[SnakeYaml 2020] SnakeYaml, https://bitbucket.org/asomov/snakeyaml/wiki/Documentation, zuletzt aufgerufen: 23.09.2020

[TablesawDoku2020] Tablesaw, https://jtablesaw.github.io/tablesaw/, zuletzt aufgerufen: 23.09.2020

[TablesawGuide 2020] Tablesaw Guide, https://jtablesaw.github.io/tablesaw/userguide/toc, zuletzt aufgerufen: 23.09.2020

[Tomcat 2018] Tomcat, http://tomcat.apache.org/tomcat-8.0-doc/, zuletzt aufgerufen: 07.10.2020

[VDV 2013] VDV, Verband-Deutscher-Verkehrsunternehmen: *VDV-Standardschnittstelle Lininennetz/Fahrplan*, Köln: Arbeitsgruppe OPNV-Datenmodell, 2013

[yamlDoku2009] YAML, https://yaml.org/spec/1.2/spec.html, zuletzt aufgerufen: 05.10.2020

[Zeit 2020] Zeit, https://www.zeit.de/mobilitaet/2020-04/oeffentliche-verkehrsmittel-bahnverkehr-busse-fahrgastzahlen, zuletzt aufgerufen: 07.10.2020

Abkürzungsverzeichnis

API Application Programming Interface

CSV Comma-separated values

EAP Earliest Arrival Problem

FIFO First In First Out

JSON JavaScript Object Notation

ÖPNV öffentlicher Personennahverkehr

REST Representational State Transfer

RVV Regensburger Verkehrsverbund

SQL Structured Query Language

VDV Verband Deutscher Verkehrsunternehmen

YAML Yet Another Markup Language

Erklärung

1. Mir ist bekannt, dass dieses Exemplar der Bachelorarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.
2. Ich erkläre hiermit, dass ich diese Bachelorarbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ort, Datum und Unterschrift

Vorgelegt durch:	Maximilian Rieder
Matrikelnummer:	3096141
Studiengang:	Bachelor Informatik
Bearbeitungszeitraum:	10. Mai 2020 – 10. Oktober 2020
Betreuung:	Prof. Dr. Jan Dünnweber
Zweitbegutachtung:	Prof. Dr. Stefan Körkel