Entwicklung eines Multimodalen Reiseplaner mit Graphhopper für AGADE Traffic

1st Dennis Zimmer

Technische Hochschule Mittelhessen
Friedberg, Germany
dennis.zimmer@mnd.thm.de

2nd Johannes Nguyen

Technische Hochschule Mittelhessen
Friedberg, Germany
johannes.nguyen@mnd.thm.de

***Abstract*—Diese Arbeit beschäftigt sich mit der Entwicklung eines Moduls, um einen multimodalen und einfachen multikriteriellen Routenplaner zu erstellen, welcher die Open Source Bibliothek von Graphhopper nutzt. Dazu wurde sich mit dem privaten und öffentlichen Verkehr auseinandergesetzt, und Modellierungsmethoden der Verkehrsnetzwerke diskutiert. Anhand von Verkehrssimulationen und aktuellen Reiseplaner wurde sich weiterhin mit dem Routing des öffentlichen Verkehrs und dessen Verkehrsmitteln beschäftigt. Mithilfe der Modellierungsmethoden und des Routings wurde dann eine mögliche Lösung zur Erstellung eines Routenplaners mit multimodalem und multikriteriellem Routing für öffentliche Verkehrsmittel erarbeitet. Das entwickelte Modul nutzt geografische Daten, um Straßennetzwerke abzubilden, und Fahrplandaten, um diese mit dem öffentlichen Verkehr zu erweitern, sowie Graphhopper, um das eigentliche Routing durchzuführen. Auch ein erstelltes Routenmodell, welches nach eigenen Anforderungen entwickelt wurde, um die Ergebnisse des Routings darzustellen, wird genutzt. Dieses Modul ist dabei so entwickelt worden, dass es eigenständig genutzt werden kann. Auch einfache Manipulationen des Netzwerkes sind in anfänglicher Ausführung möglich. Für dies werden einige Grundlagen, welche zum Verständnis der Arbeit benötigt werden, noch einmal aufgefasst und erklärt, sowie zukünftige Verbesserungen und mögliche Erweiterungen diskutiert.**

I. Einleitung

Durch limitierte Anzahl an möglichen Parkplätzen, teuren Mietplätzen und Parkhäusern, ist die Nutzung von öffentlichen Verkehrsmitteln, sowie Planen einer Reiseroute besonders in großen Städten eine wichtige tägliche Entscheidung. Auch morgens und abends während des Berufsverkehrs muss man entscheiden, ob man vielleicht lieber eine andere Route als sonst nimmt. Um solche Entscheidungen erleichtern zu können sind Verkehrssimulationen eine nützliche Methode. Sie ermöglichen den Verkehr zu simulieren und vorherzusagen, um Planungen zu erleichtern. [1] Auch beim Planen von Baustellen und für Katastrophen- und Evakuierungsereignisse können Verkehrssimulationen genutzt werden. [2] Engpässe wie stark befahrene Straßen können identifiziert und durch Anpassungen der Infrastruktur behoben werden. [1], [3] Verkehrssimulationen liefern Beobachtungen, wie sich diese Ereignisse und Entscheidungen auf den Verkehr auswirken können, um mögliche Maßnahmen zu planen und auftretenden Problemen entgegenzuwirken. [1], [3]

Damit dies ermöglicht werden kann, besitzen Verkehrssimulationsprogramme eine Routing Engine [1]–[3] oder ähnliches, um optimale Routen für einzelne Individuen zu erstellen, anhand der Verkehrslage oder andere Faktoren. Auch Reiseplaner wie Google Maps, Rhein-Main-Verkehrsverbund (RMV), oder auch Deutsche Bahn (DB) haben eine ähnliche Funktion, indem sie optimale Routen für Reisende berechnen, unabhängig davon, ob diese mit dem privaten Auto oder mit öffentlichen Verkehrsmitteln unterwegs sind. Um verlässliche Routen zu finden werden multikriterielle, d.h. mehrere Kriterien für eine optimale Route, und multimodale Ansätze, d.h. Nutzen mehrerer Verkehrsmittel, genutzt. [4] [1] [1], [5]

AGADE Traffic ist ein auf den Verkehr angepasstes Framework und stellt eine Verkehrssimulation dar. Dabei liegt der Fokus des Systems auf einem dynamischen Entscheidungsprozess für individuelle Verkehrsteilnehmer. [6]

Die Simulation besitzt, wie auch andere Verkehrssimulationen, ein Multi-Agenten-System. [3], [6] Durch dieses System werden die Verkehrsteilnehmer als einzelne, eigene Individuen dargestellt. Weiterhin wird das System für die Simulation von allen verkehrsbezogenen Fragestellungen genutzt. Die einzelnen Verkehrsteilnehmer werden durch sogenannte Software-Agenten repräsentiert, welche für sich selbst in einer gegebenen Umgebung autonom agieren können. [7]

Dabei treffen die Agenten nach dem Belief-Desire-Intention (BDI) Konzept Entscheidungen. [7] Beliefs stellen dabei das Wissen der Agenten über deren Umwelt dar, die Desires sind die Ziele der jeweiligen Agenten, und die Intentions sind mögliche Pläne zum Erreichen des Ziels.

Allerdings wird bisher in dieser Simulation nicht die Benutzung von öffentlichen Verkehrsmitteln wie Bus und Bahn miteinbezogen, und keine multimodale Routenplanung genutzt.

[1] Abschnitt: 1 Introduction, Seite 1-7

Problemstellung. Für eine realistische Simulation ist jedoch sowohl die Einbindung der öffentlichen Verkehrsmittel notwendig, da diese einen nicht unbedeutenden Anteil am täglichen Verkehr darstellen. [1] Auch ist eine multimodale Verkehrsdarstellung wichtig, um die echte Welt möglichst gut repräsentieren zu können. [4] [1]

Ziel dieser Arbeit ist es eine Mini-API zu entwickeln, welche es AGADE Traffic ermöglichen soll alle wichtigen Aufgaben hinsichtlich des öffentlichen Nahverkehrs zu verantworten. Im Rahmen dieser Arbeit soll gezeigt werden, wie Verkehrsnetzwerke und der öffentliche Verkehr modelliert werden können, um somit multimodales und multikriterielles Routing zu betreiben.

Dabei gibt es einen großen Unterschied zwischen privatem und öffentlichem Verkehr. Bei privatem Verkehr handelt es sich zum Großteil um private Verkehrsmittel wie Autos, Motorräder, LKWs usw. Diese operieren auf den jeweiligen Verkehrsnetzwerken. Es gibt dort keine Einschränkung, was Abfahrtszeiten oder Haltestellen angeht, da selbst und nach eigenen Bedürfnissen entschieden werden kann, und sie somit zeitunabhängig sind. [4] [2] [1]

Währenddessen sind die Mittel des öffentlichen Verkehrs Bus, Bahn, usw. Dabei haben private Personen keinen Einfluss auf Abfahrtszeiten und Haltestellen. Der öffentliche Verkehr wird durch Fahr- und Zeitpläne geregelt und hat feste Haltestellen, die zu einer bestimmten Zeit in bestimmter Reihenfolge befahren werden. [8] [3] [4] [2] Auch sind die Fahrzeiten vorgegeben, die von einer Haltestelle zur nächsten benötigt werden, welche aufgrund von Verspätungen und sonstigem nicht immer der Realität entsprechen.

Um nun das öffentliche Verkehrsnetzwerk modellieren zu können, benötigt man, Fahrplandaten, welche diese Elemente des öffentlichen Verkehrs darstellen, sowie eine Methode, welche es ermöglicht aus einem Fahrplan des öffentlichen Personen Nahverkehr (ÖPNV) ein Modell zu entwickeln. Um dann noch multimodales Routing betreiben zu können, müssen des Weiteren noch Straßennetzwerke in das Modell mit eingebunden werden.

Mithilfe dieses Modells erfordert es schließlich noch einer Routing Engine, mit dessen Hilfe man auf dem Modell Routing betreiben kann, um Verbindungsabfragen bearbeiten zu können.

Gliederung der Arbeit. Abschnitt zwei vermittelt Grundlagenwissen zu den beiden verwendeten Dateiformaten, welche für die Verkehrsnetzwerke und Fahrpläne gewählt worden sind, sowie zu Graphentheorie und nützlichen Modellen zur Erstellung von Graphen, welche den ÖPNV darstellen können. Des Weiteren werden Grundlagen zu den beiden verwendeten Bibliotheken Graphhopper und OneBusAway vermittelt. Hierzu wurde jeweils Primär- und Sekundärliteratur geprüft und diskutiert.

Abschnitt drei definiert die Anforderungen an das Programm und stellt das geplante Softwarekonzept und die Struktur dar und erklärt die einzelnen Klassen des Programms, anhand von UML Diagrammen.

Abschnitt vier der Arbeit beschäftigt sich mit der tatsächlichen Implementation und zeigt Teile der Umsetzung anhand von Code, und erläutert diesen. Des Weiteren wird die Funktion des Programms mithilfe eines Anwendungsbeispiels und Screenshots demonstriert. Auch wie sich die Implementation im Gegensatz zur ursprünglichen Planung geändert, hat wird diskutiert.

Anschließend wird noch eine Zusammenfassung geliefert und ein Ausblick auf zukünftige Arbeiten, entstandene Probleme, sowie mögliche Verbesserungen des Programms werden erläutert.

II. Related Work

In diesem Abschnitt werden Grundkonzepte und Formate vorgestellt, die im weiteren Verlauf der Arbeit benutzt und vorausgesetzt werden, sowie zugrundeliegende Literatur.

Viele Verkehrssimulationsprogramme bestehen aus mehreren Modulen, um eine realistische Simulation gewährleisten zu können. [2], [3] Darunter sind Module, welche für die Routen- und Reiseplanung zuständig sind und einem Reiseplaner gleich sind oder sehr ähneln, weshalb nun hauptsächlich von Reiseplaner die Rede ist, wenn es um das Planen und Finden von geeigneten Routen geht.

Alle Reiseplaner benötigen Daten, auf denen sie operieren können, um Verbindungen von Punkt zu Punkt zu finden. Dazu gehören zu einem Kartendaten für Straßennetzwerke und Fahrplandaten für den öffentlichen Verkehr. Verwendet wurden zu einem OpenStreetMap für Kartendaten und General Transit Feed Specification für Fahrplandaten, auf welche gleich genauer eingegangen wird.

Eine bewährte Methode, um Kartendaten von Reiseplanern darzustellen, ist mit einem Graph, welche auch sehr oft verwendet wird. [4] [4] [5], [9] Einige wenige Ausnahmen wie z.B. OpenTripPlanner, nutzen andere Methode wie RAPTOR [8] [5], und z.B. auch Google Maps nutzt eine bestimmte Speed-Up Methode, den Transfer Pattern Algorithmus, der jedoch mit hoher Vorbereitungszeit und Ressourcenintensivität verbunden ist. [4] [2] Auch Fahrplandaten lassen sich in besonderen Arten von Graphen gut darstellen. [5], [8] [3] Deshalb wurde für die Umsetzung auch die Graphhopper Bibliothek ausgewählt, da diese mit Graphen arbeitet und auch öffentlichen Verkehr unterstützt.

A. OpenStreetMap

Das OpenStreetMap-Projekt (OSM) wurde 2004 in Großbritannien gegründet. OSM ist ein freier und nicht kommerzieller Community Kartendienst mit dem Ziel, weltweite, freie Kartendaten zu erheben und diese gesammelten geografischen Daten dann der Entwickler Community frei zur Verfügung zu stellen. [10] Somit wird auch eine freie Weltkarte erstellt. [10]

Dabei wird die Verkehrsinfrastruktur wie Straßen, Wege, sowie Points of Interest abgebildet. Im Vergleich zu anderen

[2] Abschnitt: 2 Literature Overview Seite 9-38

[3] Abschnitt: 4 Journey Planning in Public Transit Networks

[4] Abschnitt: 4 Seite Public Transit Journey Planning 47-67

[5] Abschnitt: 6 Final Remarks

Kartendiensten ist OSM ein freier, von jedem nutzbarer Service. [10] Es werden nutzbare Geodaten gesammelt, welche von freiwilligen Personen per GPS-Daten bereitgestellt werden. Diese Daten können für alle Arten von Karten oder auch Navigationssoftware wie Routenplanern genutzt werden.

Die OSM Karte bzw. Datenbank besteht aus geometrischen Grundelementen und Relationen. Dabei sind die Daten-Grundelemente Punkte, Linien und Relationen. Diesen können Attribute, welche jeweils aus einem Schlüssel-Werte Paar bestehen, zugeordnet werden, um weitere Eigenschaften zu beschreiben.

Punkte in OSM entsprechen dabei einzelnen geografischen Punkten mit einem Längen- und Breitengrad. Diese können Orte, Sehenswürdigkeiten, Städte usw. beschreiben, indem man ihnen, Attribute zuweist. [11]

Linien in OSM entsprechen mehreren aufeinanderfolgenden Geradenabschnitten. Diese bestehen aus mehreren Punkten, und man kann ihnen ebenfalls Attribute zuweisen. Dadurch lassen sich Geschwindigkeit oder Typ der Straße definieren. [11] Diese Linien haben eine Richtung und je nach Attribut lässt sich somit die Richtung von z.B. Einbahnstraßen definieren. Des Weiteren können Punkte einer Linie, auch Punkte einer anderen sein, um Kreuzungen oder ähnliches darzustellen.

Eine Fläche ist eine Form einer geschlossenen Linie mit besonderen Attributen, um diese als solche zu identifizieren, da eine Fläche kein eigenes Datenformat besitzt. Bei einer geschlossenen Linie sind Anfang und Endpunkt gleich.

Als letztes Grundelement gibt es noch Relationen, welche aus ein oder mehr Attributen bestehen. Sie gruppieren Punkte, Linien oder andere Relationen. Dazu beschreiben diese logische oder geografische Beziehungen zwischen den Elementen. Die Elemente einer Relation besitzen des Weiteren auch eine Rolle, welche die Bedeutung des jeweiligen Datenelements in der Relation beschreibt.

Per Render-Software können aus diesen Daten dann bildliche Darstellungen erzeugt werden. Die Kartendaten können im OSM Dateiformat oder anderen heruntergeladen und genutzt werden.

B. General Transit Feed Specification

Das General Transit Feed Specification Format (GTFS) wurde früher als Google Transit Feed Specification von Google in Zusammenarbeit mit einigen Verkehrsunternehmen entwickelt. Es ist das mittlerweile am meisten verbreitete und bekannteste Format zum Darstellen von Daten des ÖPNV [9], [12], und wird von vielen Reiseplanern genutzt. [12], [13]

Das GTFS Format ist ein standardisiertes Format. Zudem ist das Format äußert ausführlich dokumentiert und stellt den Aufbau gut dar. Es liefert geografische Daten von Haltestationen und Fahrplandaten, sowie einen Zeitplan für Informationen der öffentlichen Verkehrsmittel.

Diese Daten werden meistens von Unternehmen oder Verkehrsverbunden auf verschiedenen Portalen und eigenen Webseiten, als Daten angeboten. [14]–[17] Des Weiteren ist das GTFS Format statisch, was bedeutet, dass sich die Informationen, welche die GTFS Daten liefern, nicht ständig ändern. Wie oft diese Daten aktualisiert werden hängt von den Anbietern ab, dabei ist eine Zeitspanne von mehreren Monaten bis hin zu einem Jahr je nach Bedarf üblich. Auch der Umfang der Daten kann stark variieren und ist von den Anbietern abhängig. Angeboten werden die Daten in GTFS Feeds.

Ein Feed besteht wie in Fig. 1 zu sehen aus mindestens 5 und maximal 17 verschiedenen Text Dateien, welche in einer Zip Datei zusammengefasst sind. Jede dieser Text Dateien stellt dabei einen Aspekt der Verkehrsinformationen dar. Davon müssen manche angegeben werden und mache sind nur als optional zu betrachten, um einen validierten Feed erstellen zu können. [18]

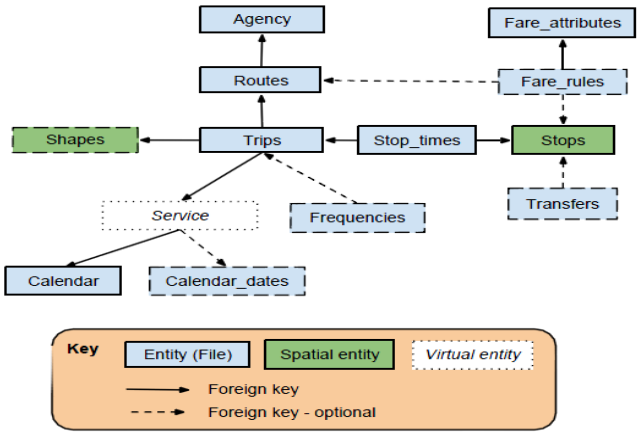

Fig. 1. Abbildung welche den Aufbau eines GTFS Feeds und die wichtigsten Text Dateien darstellt. [12]

Zu den Pflicht-Textdateien gehören die agency, stops, routes, trips und stop-times. Typisch zu diesen sind jedoch noch transfers, calendar dates, calendar und eine feed-info Datei. Der Aufbau des GTFS Formates und der Text Dateien ähneln dem, relationaler Datenbank Tabellen. [19] Die Text Datei agency stellt die Verkehrsagenturen dar, welche eine ID, Namen, URL der Agentur, etc. besitzt. Die stops Datei beinhaltet alle Haltestellen mit dazugehörigen GPS-Daten. Weiterhin beinhaltet diese pro Haltestelle unter anderem eine ID, Namen, Code und den Breitengrad und Längengrad der Haltestelle. In der Routendatei werden Verbindungen bzw. ganze Routen von einem Startpunkt zu einem Zielpunkt dargestellt. Diese besitzen eine eigene ID, eine zugeordnete Agentur durch eine agency ID, einen Namen, etc. Die Text Datei der Trips stellt wiederum elementare Verbindungen dar, d.h. eine Verbindung von einer Haltestelle auf eine direkt darauffolgende. Ein Trip besitzt somit eine eigene ID, eine route ID, um mehrere Trips einer Route zuordnen zu können und ggf. eine Service ID, einen Namen, etc. Zuletzt unter den Pflicht-Dateien ist noch die stop-times Text Datei. Hier werden nun die verschiedenen Haltezeiten eines Stopps dargestellt. Diese werden per Trip ID einem Trip zugeordnet, besitzen eine Ankunfts- und Abfahrt-

szeit, eine stop ID für die Zuordnung eines Stopps, etc. Zuletzt noch eine wichtige Text Datei welche Umsteigemöglichkeiten an einer Haltestelle beschreiben. Hier gegeben durch die Text Datei der transfers.txt, welche jedoch fast nie angegeben ist.

Eine Erweiterung des GTFS Formates, ist GTFS-Realtime. Dadurch lässt sich das GTFS Format um einen dynamischen Teil, durch die Integration von Echtzeitinformationen über z.B. Verspätungen und Baustellen der öffentlichen Verkehrsmittel, erweitern. [19], [20]

C. Graphen

Graphen sind für die Modellierung von Verkehrsnetzwerken, als auch für Modellierung des ÖPNV beliebt. [4] [2] [21] Besonders für den Bereich des öffentlichen Nahverkehrs gibt es Modelle um diesen darzustellen, auf die jedoch später nochmal eingegangen wird. Reiseplaner oder Methoden, um multimodale Netzwerke darzustellen, können mehrere Schichten von Graphen nutzen, um einzelne Modi im Verkehr darzustellen, welche miteinander verknüpft werden. [1], [8], [9] [6] Hier kommen dann diese Modelle von Graphen zur Verwendung. Graphen sind in diesem Zusammenhang des Routings schon gut in der Literatur erforscht. [9]

1) ***Graph Theorie****:* Ein Graph $G = (V, A)$ besteht aus einem Set von Eckpunkten V (Vertices engl.) und einem Set von Kanten A (Arcs engl.). Bei einem Graph sind Vertices über Arcs miteinander verbunden. Über solche Graphen lässt sich auch schon operieren, jedoch lassen sich daraus wenige Informationen gewinnen, aufgrund weniger Kriterien und Attributen. [4] [7] Dies ist das Basis Konzept von Graphen. Sie lassen sich jedoch auch noch in mehrere Kategorien unterteilen, wodurch man Graphen weiter definieren kann. Weitere Arten von Graphen sind dabei ein gerichteter, ungerichteter, gewichteter und ungewichteter Graph. Hierbei ist der Basis Graph $G = (V, A)$ sowohl ungerichtet als auch ungewichtet.

Ein gerichteter Graph besteht aus einem Set von Vertices und einem Set von Arcs, wobei $A \subseteq V \times V$ aus geordneten Paaren $(u, v) \in V$ besteht. Des Weiteren muss $u \in G$ und $v \in G$ gelten, wenn $(u, v) \in A$ gilt. Zu beachten allerdings ist, dass (u, v) und (v, u) nicht den gleichen Arcs entsprechen. Bei einem gerichteten Graph kann ein Arc nur einseitig überquert werden. Damit lassen sich z.B. Einbahnstraßen in einem Straßennetzwerk darstellen und bei öffentlichen Verkehrsmitteln die Fahrtrichtung des Fahrzeugs. [22] Bei einem ungerichteten Graph bestehen die Arcs aus ungeordneten Paaren und es gilt $A \in \{\{u, v\} \,|u, v \in V\}$. Die Paare $\{u, v\}$ und $\{v, u\}$ sind hierbei gleich. Arcs bei einem ungerichteten Graphen können somit von beiden Seiten aus überquert werden.

Bei einem gewichteten Graphen werden einem Arc bestimmte Kosten zugewiesen, welche "bezahlt" werden müssen, um diesen zu überqueren. Diese Kosten sind sehr flexibel und können einfache Zahlen, als auch bestimmte Attribute und Funktionen sein. [4] [7] Somit können dann Entfernungen, Zeiten, etc. im Straßennetz dargestellt werden. Im Sinne des ÖPNV können die Fahrzeiten von einer Haltestelle zur nächsten so dargestellt werden. Bei einem ungewichteten Graph gibt es keine Kosten, um einen Arc zu überqueren. Graphen lassen sich beliebig weiter augmentieren, indem man weitere Attribute den Vertices oder Arcs zuordnet.

Um diese jetzt für den ÖPNV zu verwenden, muss aus einem Fahrplan ein gerichteter und gewichteter Graph erstellt werden, sodass das Shortest-Path Problem gelöst werden kann, und dass dieser die Zeitabhängigkeit des Fahrplans repräsentiert. [4] [4] Dazu gibt es mehrere mögliche Ansätze. Zwei der Ansätze sind zum einen, der Time-expanded Graph und zum anderen der Time-dependent Graph. Beide sind beliebte Methoden, um den öffentlichen Verkehr als Graph darzustellen. [8] [3] Ziel beider Ansätze ist es, den Graph, um den Aspekt der Zeitabhängigkeit zu erweitern. [4] [4] Beide Modelle besitzen gleiche Funktionen und Möglichkeiten, unterscheiden sich jedoch in der Umsetzung und Suchgeschwindigkeit. [4] [4] [8] [3] Hierbei gibt es jeweils ein simples und realistisches Modell.

2) ***Time-expanded Graph****:* Das simple Time-expanded Modell nutzt die Idee von Events. [8] [3] Zur Erstellung werden ein Fahrplan und ein Set von elementaren Verbindungen benötigt.

Für jede elementare Verbindung C in dem Fahrplan werden zwei Vertices erstellt, ein Abfahrt-Vertex und ein Ankunft-Vertex, welche dann mit einem Arc verbunden werden. Die Vertices besitzen immer einen indirekten Zeitstempel. Der Zeitstempel der Vertices ist dann entweder durch die Abfahrtszeit oder die Ankunftszeit gegeben. Analog dazu besitzt jeder Vertex einen Stopp-Wert.

Die Arcs werden von dem Modell dann so erstellt, dass für jede Verbindung ein Arc zwischen den Abfahrt- und Ankunft-Vertices erstellt wird. Um einen Transfer zwischen verschiedenen Trips zu ermöglichen, erstellt das Modell wie in Fig. 2 zu sehen, Transfer Arcs zwischen anliegenden Vertices desselben Stopps in Reihenfolge der fortlaufenden Zeit.

Problem des simplen Modells ist, dass es jedoch nicht eine minimale Umsteigezeit berücksichtigt. Das reale Modell behandelt genau dieses Problem, indem es minimale Umsteigezeiten innerhalb eines Stopps hinzufügt. [4] [4]

Vorteil des Time-expanded Modells ist, dass es simpel ist

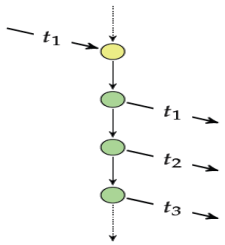

Fig. 2. Ein simpler Time-expanded Graph mit Ankunft (gelb) und Abfahr (grün) Vertices, sowie Transfer Arcs. [4]

[6] Abschnitt: 5 Multimodal Journey Planning
[7] Abschnitt: 3 Fundamentals, Seite 39-45

und Shortest Path Algorithmen wie Dijkstra einfach angewendet werden können. [4] [4] Der große Nachteil des Graphen ist, dass er ziemlich groß ist. [4] [4] [8] [3]

*3) **Time-dependent-Graph**:* Das simple Time-dependent Modell zielt auf einen kleineren Graphen ab als das expanded Modell. Anstatt das Vertices durch Events erstellt werden und somit die Zeitabhängigkeit mit einbindet, wird bei diesem Modell eine spezielle Travel-Time Funktion [4] [4] den Arcs zugeordnet.

Die Idee der Funktion ist, mehrere elementare Verbindungen innerhalb eines Arc darzustellen und dann je nach Zeit, entsprechende Kosten auf den Arc zu mappen. [4], [23] [4] Die Abfahrts- und Ankunftszeiten werden dann hier durch die Travel-Time Funktion gegeben.

Wie auch im expanded Modell benötigt man einen Fahrplan und ein Set an Verbindungen. Das Modell erstellt dann einen gerichteten Graphen. Wie in Fig. 3 zu sehen, wird für jeden Stopp S, ein Vertex $p \in S$ erstellt und ein Arc p_1p_2 zwischen p_1 und p_2 ,wenn es eine elementare Verbindung von p_1 zu p_2 gibt. Um die Zeitabhängigkeit zu erhalten, werden die Kosten von der Travel-Time Funktion erstellt. Auch bei dem Time-dependent Graph Modell werden minimale Umsteigezeiten erst mit dem realistischen Modell berücksichtigt. [4] [4] Dafür muss der Graph etwas erweitert werden.

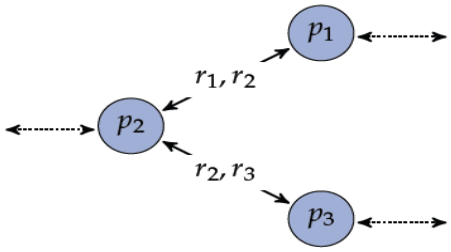

Fig. 3. Ein simpler Time-dependend Graph [4]

Vorteil dieses Modells ist die Größe des Graphen, welche deutlich kleiner ist als die des expanded Modells, und dadurch schnellere Abfragezeiten ermöglicht. [4] [4] Der Nachteil ist jedoch die Anwendung der Algorithmen, welche komplexer sind, [4] [4] da während der Abfragen, die Travel-Time Funktion, welche den Arcs zugeordnet ist, erst ausgewertet werden muss, bevor der Algorithmus fortfahren kann.

Auch gibt es einige Speed-Up Techniken für die Algorithmen, sowie Verbesserungen, um wiederum schnellere Abfragezeiten zu ermöglichen. [8] [3] Um ein Graph für ÖPNV zu modellieren wird bei beiden Ansätzen ein Fahrplan benötigt, welcher durch die GTFS Daten gegeben werden kann.

D. Graphhopper

Graphhopper ist eine in Java geschriebene Routing Engine und Reiseplaner, welcher multikriteriellen und multimodalen Verkehr berücksichtigt. Es werden sowohl eine Open Source Routing Bibliothek auf GitHub angeboten [24] als auch mehrere APIs, zum Lösen von Routing Problemen und Optimierungen. Beide besitzen ähnlich viele Funktionen. Sie können sowohl auf Servern, Desktops als auch auf Mobilgeräten (Android und IOS) genutzt werden. Unternehmen wie die Deutsche Bahn und OSM nutzen Graphhopper bereits für ihre Reiseplaner und ihr Routing, [25] wodurch eine gewisse Zuverlässigkeit gegeben ist.

Graphhopper ist zudem auch sehr schnell und speichereffizient [24] und kann verschiedene Algorithmen wie Dijkstra und Verbesserungen wie A* und Contraction Hierarchien nutzen. Die Lizenz erlaubt es jedem Graphhopper anzupassen und in kommerziellen oder freien Produkten zu integrieren. [24] Es bietet eine Alternative zu existierenden Routing Angeboten durch optimierte Abfragegeschwindigkeit und Nutzen von OSM Daten als Kartenmaterial. Graphhopper kann für die Planung des städtischen Verkehrs, Verkehrssimulationen oder auch für Reiseplaner genutzt werden.

Die Graphhopper API besteht aus mehreren kleinen APIs, welche jeweils für andere Funktionen genutzt werden können. Die API kann genutzt werden, um einfaches Punkt zu Punkt Routing mit verschiedenen Modi, wie Fahrzeugauswahl, zu lösen und daraus dann Reisezeiten, Distanz, Abbiegevorschriften etc. zu erhalten. Des Weiteren lassen sich Tourenprobleme mit mehreren Fahrzeugen unter Berücksichtigung mehrerer Bedingungen lösen. Optimierungen von Routen und Analysetools stehen ebenfalls zur Verfügung. Die Graphhopper API ist ein RESTful Webservice mit einfacher JSON Schnittstelle.

Die Open Source Bibliothek besteht aus zwei Hauptprojekten. Zum einen jsprit, welche eine Optimierungs-Engine zum Lösen von Tourenplanungsproblemen ist. Der andere Teil ist die Graphhopper Routing Engine, welche ein schneller und speichereffizienter Java Router ist, der sowohl als Server als auch offline zur Verfügung steht. [25] Dieser arbeitet mit Graphen als Grunddatenstruktur. Wie auch vorher, lassen sich hier auch verschiedene Algorithmen verwenden, welche sich in drei Modi aufteilen lassen.

Der Speed Modus, welcher auf Geschwindigkeit ausgelegt ist, ist schnell und hat eine leichte Arbeitsspeicherbelastung. [13] Jedoch ist nur wenig Anpassung möglich und man benötigt eine lange und ressourcenintensive Vorbereitungszeit. Der flexible Modus ist um einiges weniger schnell, benötigt aber keine so aufwendige Vorbereitungszeit und ermöglicht mehr Anpassungen. [13] Beim Routing mit öffentlichen Verkehrsmitteln steht jedoch nur der flexible Modus zur Verfügung. Zuletzt gibt es noch den hybriden Modus, welcher eine Mischung der beiden Modi darstellt.

Die Graphhopper Routing Engine lässt sich recht einfach per Maven Abhängigkeiten integrieren. [13] Sie nutzt OSM Daten, um Straßennetzwerke darzustellen. Es werden auch mehrere Gewichtungen und Fahrzeugprofile bereitgestellt, welche auch angepasst werden können, [13] um somit multikriterielles und multimodales Routing zu ermöglichen. Auch wird das Modellieren des öffentlichen Verkehrs unterstützt, wozu GTFS Daten verwendet werden. Des Weiteren kann Graphhopper auch alternative Routen liefern. Um die GTFS Daten darzustellen verwendet Graphhopper, wie in Fig. 4 zu

sehen, eine Version eines time-expanded Graphen mit einigen Anpassungen um schnelleres Routing zu garantieren. Eine davon wäre z.B., dass nur der Graph für den aktuellen Tag angezeigt wird, was die Größe drastisch verringert. Diese Funktion kann jedoch auch ausgestellt werden. [13]

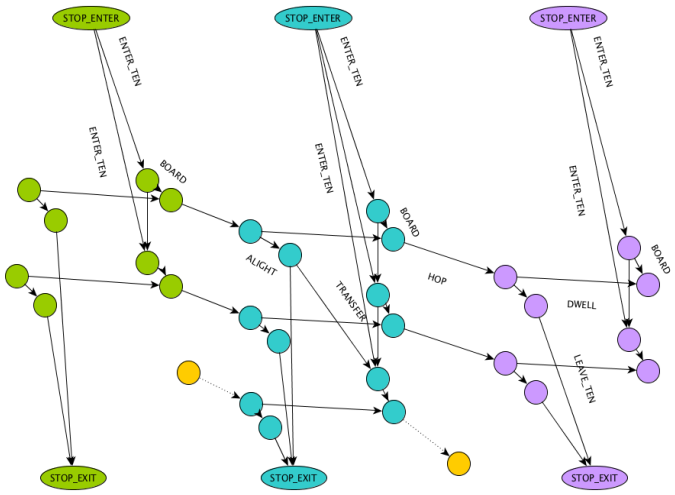

Fig. 4. Ausschnitt eines von Graphhopper erstellten Time-expanded Graphen. [13]

E. *OneBusAway*

Um individueller mit GTFS Daten arbeiten zu können und diese gegebenenfalls auch zu verändern, neue hinzuzufügen, um dadurch Veränderungen im ÖPNV zu testen, benötigt man ein Programm, welches diese Daten lesen und in einer sinnvollen Struktur speichern kann.

OneBusAway (OBA) ist ein in Java geschriebenes Programm zur Livedarstellung von öffentlichen Verkehrsmitteln. Es zeigt live Fahrpläne für Linien und Haltestellen sowie Positionen von Fahrzeugen und Verspätungen in Echtzeit. OBA ist Open Source [26] und bietet verschiedene Module zum Arbeiten und Anpassen von GTFS Daten an.

Davon ist one-bus-away-gtfs das Hauptmodul und zuständig für das Lesen und Schreiben von GTFS Daten. Das Modul speichert die gelesenen Daten intern in einer angepassten Hashmap. Auf diese Struktur lässt sich zugreifen, und die darin liegenden Daten bearbeiten. Des Weiteren können neue GTFS Daten und Feeds erstellt, oder alte überschrieben werden. Dabei ist die Struktur übersichtlich aufgebaut und ermöglicht einfache Abfragen auf dieser.

III. Programm Aufbau

In diesem Abschnitt wird der Aufbau des Public-Transit-for-AGADE Programms vorgestellt. Dabei wird auf beide Module publicTransportRouting und gtfsEditing eingegangen. Es werden verwendete Technologien und Konzepte, sowie Bibliotheken für die jeweiligen Module dargestellt. Hierbei wird auch auf die einzelnen Klassen der Module näher eingegangen und deren Funktionen erläutert.

Die Aufgabe ist es eine Art Mini-API zu entwickeln, welche es ermöglicht, alle wichtigen Aufgaben hinsichtlich des öffentlichen Verkehrs zu verantworten. Aufgrund der Anforderungen sollte es möglich sein, eine Vielzahl an Verbindungsabfragen innerhalb einer kurzen Zeit zu tätigen und als Rückgabe eine Liste an möglichen Verbindungsauswahlen zu bekommen. Deshalb war ein wichtiges Kriterium der Arbeit, die Optimierung hinsichtlich der Laufzeit. Weiterhin soll das Modul, Routen von einem Ausgangspunkt zu einem Zielpunkt berechnen und wiedergeben, welche öffentliche Verkehrsmittel nutzen. Aufgrund der Anforderung ist es dabei wichtig, sowohl Zwischenhaltestellen anzugeben als auch Abfahrts- und Ankunftszeiten in Ticks zu einer bestimmten Zeit zu berechnen und darzustellen usw. Des Weiteren wurde das Modul so entwickelt, dass es eine möglichst losgelöste Architektur besitzt, um es unabhängig und eigenständig nutzen zu können. Deshalb wurde auch auf permanente Abhängigkeiten zu andern APIs und Systemen so gut wie möglich verzichtet.

Für das Programm wird das SDK von Java 8 verwendet, da dieses durch die Anforderungen vorausgesetzt wurde. Als Build-Management-Tool ist hier Maven zum Einsatz gekommen, mit welchem man Java Programme standardisiert erstellen und verwalten kann, sowie Integration von Bibliotheken durch Abhängigkeiten ermöglicht werden. Für die Module des Public-Transit-for-AGADE Programms wurde das Model-View-Controller Schema verwendet. Als Programm Schema wurde des Weiteren auch das Facade Pattern gewählt. Es bietet eine einheitliche, einfache Schnittstelle, und ermöglicht später eine simple Integration und vereinfacht die Bedienung der Module, da in der jeweiligen Facade Klasse alle wichtigen Funktionen zu finden sind, und von da aus verwendet werden können.

Als Bibliotheken wurden die Graphhopper Open Source [24] Bibliothek für das Routing und Erstellen des Graphen, und die OneBusAway [26] Bibliothek für die Arbeit mit GTFS Daten verwendet. Als Input Datenformat wurden OSM Daten und GTFS Daten genutzt.

Public-Transit-for-AGADE besteht aus zwei verschiedenen Modulen, welche jeweils andere Funktionen erfüllen. Das publicTransportRouting ist das Hauptmodul und besitzt die Hauptfunktion des Programms. Dabei handelt es sich um das Routing und Verwalten von Verbindungsabfragen. Das zweite ist das gtfsEditing Modul und besitzt die Funktion GTFS Daten erstellen und bearbeiten zu können. Beide Module besitzen eine eigene Facade Klasse und können so unabhängig voneinander genutzt werden. Dies wurde so umgesetzt, da beide zu unterschiedliche Funktionen besitzen, um diese in einer Facade Klasse abzubilden und so eine logische Trennung der Funktionen zu erreichen.

A. *Modul: publicTransportRouting*

Das publicTransportRouting Modul umfasst die Hauptfunktion des Programms. Es Verantwortet alle Angelegenheiten, welche etwas mit Routing und dem ÖPNV zu tun haben. Des Weiteren können mit dem Modul Graphen für den öffentlichen

Verkehr, aus OSM und GTFS Daten, erstellt und vorhandene Graphen geladen werden. Als Ergebnis einer Routenberechnung werden dann JSON oder YAML Dateien wiedergegeben, welche die berechneten Routen enthalten.

Das Modul erlaubt multimodales Routing des öffentlichen Verkehrs. Hierbei werden für den multimodalen Aspekt alle in den GTFS Daten vorhandene öffentliche Verkehrsmittel verwendet, sowie das Reisen zu Fuß berücksichtigt. Für den Teil des öffentlichen Verkehrs werden die GTFS Daten verwendet, und OSM Daten für das Straßennetzwerk und Gehwege.

Graphhopper wird in diesem Modul für die Integration der Fahrplandaten durch Erstellen eines Graphen, sowie für Verbindungsabfragen durch multikriterielles und multimodales Routing verwendet. Wie in Fig.5 zu sehen ist, sind die Klassen in Controller, Model und Service Pakete unterteilt, welche beschrieben und deren Funktionen erklärt werden.

Innerhalb des Controller Pakets befinden sich alle Klassen, welche gewisse Funktionen repräsentieren. Dazu gehören der GtfsGraphController, GraphhopperController, GraphhopperResponseHandler, FileDownloader, TransitionConfigHandler, RouteLoader, RouteIterator und TimeController. In dem Model Paket befinden sich alle Klassen, welche das Routenmodell definieren, in welchem die fertigen Routen dargestellt werden. Das Service Paket beinhaltet nur die PT_Facade_Class, welche das Nutzen der Funktionen des Moduls erlaubt.

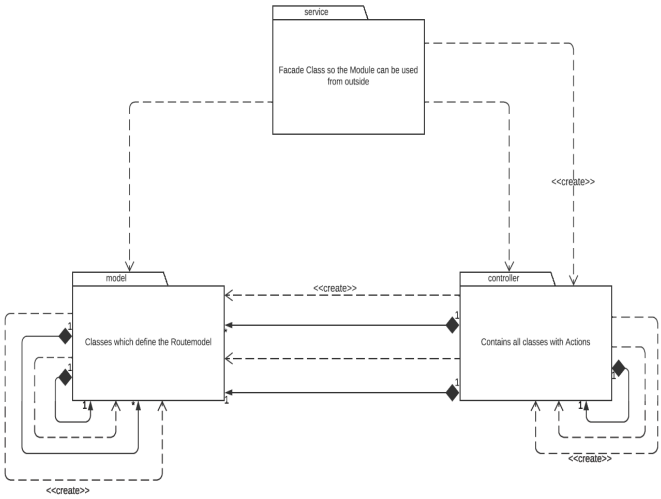

Fig. 5. UML-Diagramm zum darstellen der Zusammenhänge der jeweiligen Pakete in dem publicTranportRouting Modul.

1) ***GtfsGraphController****:* Die GtfsGraphController Klasse ist zum Erstellen und Laden, sowie Speichern eines time-expanded Graphen von Graphhopper verantwortlich. Des Weiteren werden Konfigurationsdateien, welche Eigenschaften und Informationen über den Graphen enthalten, erstellt und gespeichert, um einen bereits erstellten Graphen wieder laden zu können.

2) ***GraphhopperController****:* Der Graphhopper Controller ist zum Erstellen von Pfaden zu gegebenen Verbindungsanfragen verantwortlich. Ein Pfad ist die Rückgabe von Graphhopper und stellt gefundene Routen zu den Anfragen dar. Die Klasse wird zum Erstellen von Anfragen an Graphhopper genutzt, welche erst definiert werden, um diese anschließend zum Routing weiterzugeben. Weiterhin speichert die Klasse einen Pfad, welcher in die Routenstruktur übertragen worden ist, als JSON oder YAML Datei.

3) ***GraphhopperResponseHandler****:* Der GraphhopperResponseHandler ist für die Umwandlung der von Graphhopper gegebenen Pfade in die erstellte Routen-Datenstruktur verantwortlich. Es werden aus einem gegebenen Pfad alle nötigen Informationen, die den Anforderungen des Programms entsprechen, entnommen und mit diesen dann eine Route mit dem zugrunde liegenden Routenmodell erstellt.

4) ***FileDownloader****:* Die FileDownloader Klasse wird genutzt um OSM und GTFS Daten anhand von URLs herunterzuladen und an vorgegebene Orte zu speichern. Eine Anwendung ist das Herunterladen von Testdateien.

5) ***TransitionConfigHandler****:* Der TransitionConfigHandler ist dazu da, um die von Graphhopper erstellte Konfigurationsdatei, welche zum Erstellen und Laden des Graphen genutzt wird, zu verwalten. Dabei wird die GraphhopperConfig, welche Konfigurationen und Einstellungen des Graphen beinhaltet, in eine TransitionConfig umgewandelt, welche alle nötigen Infos enthält. Außerdem funktioniert dies auch in die entgegengesetzte Richtung. Diese Umwandlung macht es möglich die Datei zu speichern und wieder abzurufen.

6) ***RouteLoader****:* Der RouteLoader, wird zum Laden einer existierenden Route, welche als JSON oder YAML Datei gespeichert ist, benötigt. Diese wird dann intern in dem Programm zwischengespeichert, sodass darauf zugegriffen werden kann.

7) ***RouteIterator****:* Die RouteIterator Klasse enthält mehrere Methoden, um einfache Abfragen auf einer Route zu ermöglichen, wie z.B. den nächsten oder vorherigen Stopp einer Route etc. Die Klasse wird dazu benutzt, um über Routen zu iterieren und ermöglicht gegebenenfalls einfache Abfragen auf dieser.

8) ***TimeController****:* Der TimeController ist für alle Angelegenheiten, welche sich mit Zeit und Zeitumrechnungen befassen verantwortlich. Es werden viele Methoden angeboten, um Zeiten in andere Formate oder Formen umzuwandeln. Dabei sind besondere Formen vorhanden, welche sich an den vorgegebenen Anforderungen orientieren. Als Beispiel dient hierzu ein Tick, welcher eine Minute entspricht.

9) ***Route****:* Die Routen Klasse repräsentiert die oberste Instanz des Routenmodells. Sie beinhaltet eine Liste von Legs und eine Liste von Stopps. Des Weiteren werden Reisezeit, Ankunftszeit, Anzahl der Umstiege, Laufdistanz, Kosten, Anzahl der befahrenen Stopps und die Anzahl der Legs repräsentiert, wovon einige als Entscheidungskriterium genutzt werden können. Die Klasse enthält auch jeweils eine Methode, um Stopps und Legs der Route hinzuzufügen. Dabei kann eine Route beliebig viele Stopps und Legs beinhalten.

10) ***Leg****:* Ein Leg [9] repräsentiert die zweite Instanz des Routenmodells und stellt dabei einen Teil der Route dar, welche mit demselben Fahrzeug zurückgelegt wird. Steigt man z.B. innerhalb einer Route um, so beginnt ein neues Leg. Dabei

wird, an einem Stück zurückgelegter Fußweg, ebenfalls als eigenes Leg bezeichnet.

Diese Klasse besitzt eine Liste von Stopps, welche innerhalb des Legs befahren werden. Weiterhin besitzt ein Leg eine ID, einen Typ, welcher die Verkehrsmodi Fuß oder PT (öffentliche Verkehrsmittel) darstellen, sowie einen Start- und Zielort, eine Start- und Zielzeit in einem Zeit- und Tick Format, ein Fahrzeug wie Bus, Zug, etc. und die Anzahl der befahrenen Stopps.

Dabei kann ein Leg beliebig viele Stopps besitzen und einer Route zugeteilt sein. Die Klasse besitzt des Weiteren auch eine Methode, um Haltestellen dem Leg hinzuzufügen.

11) ***Stop****:* Die Stop Klasse repräsentiert die Haltestellen des Routenmodells. Dabei haben diese eine Ankunfts- und eine Abfahrtszeit, zu denen dieser Stopp innerhalb der Route befahren wird. Diese Zeiten sind dort auch in Ticks dargestellt. Neben den Zeiten besitzt eine Haltestelle auch eine Location, um dieser einen Ort zuzuordnen.

Eine Stop Klasse ist einer Route sowie einem Leg zugeordnet und besitzt selbst eine Location.

12) ***Location****:* Die Location Klasse wurde von AGADE Traffic implementiert und stellt einen Ort mit Breiten- und Längengrad dar, welcher zudem einen Namen besitzen kann.

13) ***PT_Facade_Class****:* Diese Klasse ist die Facade Klasse des Moduls. Hier werden alle nötigen Funktionen durch Methoden repräsentiert, um die Hauptfunktionen des Moduls richtig nutzen zu können und auf die anderen Klassen zugreifen zu können. Hierbei wird das Erstellen des Graphen, sowie Laden eines existierenden, Erstellen von Verbindungsabfragen und das eigentliche Routing ermöglicht. Weiterhin wird das Laden einer vorhandenen Route ebenfalls unterstützt.

B. gtfsEditing

Neben der Hauptfunktion kümmert sich dieses Modul nicht direkt um das Routing des öffentlichen Verkehrs, sondern beeinflusst dieses indirekt, durch Hinzufügen und Ändern, der dem Graph zugrundeliegenden Daten. Durch diese Änderungen lassen sich im Vorhinein Graphen bearbeiten, bevor diese erstellt werden. Die Daten, welche dafür verwendet werden, sind die GTFS Daten, welche den öffentlichen Verkehr darstellen. Diese Funktion beschränkt sich jedoch nur auf den Teil des öffentlichen Verkehrs und nicht auf das Straßennetzwerk, da dafür Änderungen an den OSM Daten vorgenommen werden müssten.

Aktuell ist dies noch nicht völlig implementiert. Jedoch ist es für die Dateien, welche sich um die Kosten des öffentlichen Verkehrs kümmern, möglich. Bei diesen handelt es sich um die fare_attribute.txt und um die fare_rules.txt Datei. Dabei ist es möglich diese zu ändern, falls sie vorhanden sind und zu erstellen, falls das nicht der Fall ist. Die Feeds können dann nach der Änderung mit den neuen Informationen überschrieben oder separat gespeichert werden.

Um diese Funktionen umzusetzen, wird für dieses Modul das Open Source Programm OneBusAway genutzt. Allerdings wird aktuell nur das one-bus-away-gtfs Modul genutzt, um mit GTFS Daten arbeiten zu können. [27]

Wie in Fig. 6 dargestellt, ist das Modul auch nach dem MVC Schema aufgebaut. In dem Controller Paket befinden sich zum einen eine FareController, ReaderHandler und TransformerHandler Klasse. Im Model Paket sind die Klassen Fare_Attribute, Fare_Rule, sowie die Enumerationen PaymentMethod und Transfers. Die EditorFacade Klasse ist die einzige Klasse des Service Pakets.

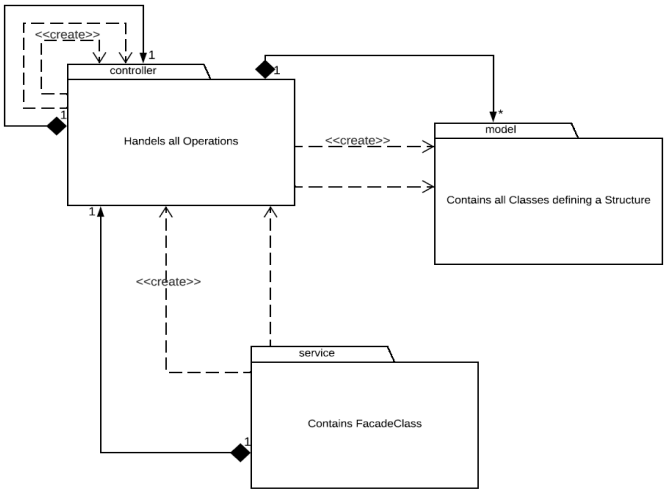

Fig. 6. UML-Diagramm zum Darstellen der Zusammenhänge der jeweiligen Pakete in dem gtfsEditing Modul.

1) ***FareController****:* Die FareController Klasse beinhaltet Methoden, um Kostenattribute und Kostenregeln zu definieren und einander zuzuordnen. Dabei werden jeweils verschiedene Optionen von Attributen und Regeln angeboten, zwischen denen bei der Erstellung entschieden werden kann. Die Klasse verwaltet die Logik der Transformation bzw. Änderungen. Die eigentlichen Änderungen werden hier jedoch nicht durchgeführt, aber sie werden zu einem Transformer gesendet, welcher alle Änderungen sammelt, um diese dann auf einmal umzusetzen.

2) ***ReaderHandler****:* Die ReaderHandler Klasse kümmert sich um das Einlesen der Daten durch eine GTFS-Reader Klasse von OneBusAway. Auch das interne Speichern der Daten in einer von OneBusAway zur Verfügung gestellten Datenstruktur, welche auf einer Hashmap basiert, wird ermöglicht. Auf diese Struktur lässt sich dann auch zugreifen.

3) ***TransformerHandler****:* Der TransformerHandler ist für die Verwaltung des Transformer, sowie aller Sachen, welche direkt mit der Transformation der GTFS Daten zu tun haben, verantwortlich. Es werden die definierten Änderungen zwischengespeichert und anschließend umgesetzt. Nach der Umsetzung werden die Daten dann abgespeichert.

4) ***Fare_Attribute****:* Die Fare_Attribute Klasse repräsentiert die Kostenattribute. Darunter ist ein Preis, eine Gültigkeit, eine ID, Bezahlmethode, sowie Transfer Optionen und Weiteres, welche definiert werden müssen, um ein Attribut erstellen zu können.

5) ***Fare_Rule****:* Die Fare_Rule Klasse repräsentiert dann die Kosten Regeln. Dabei werden die definierten Attribute, Routen

oder auch Zonen etc. zugeteilt. Die Kosten Regel ist somit eigentlich nur eine Zuteilung der Attribute.

6) ***PaymentMethod****:* Die PaymentMethod Klasse ist eine Enumeration welche gültige Bezahlmethoden [18] darstellt. Darunter zählt, ob man beim Betreten der Route bezahlt oder davor.

7) ***Transfers****:* Die Transfer Klasse ist ebenfalls eine Enumeration und enthält alle gültigen Transfer Optionen. [18] Dabei handelt es sich hauptsächlich darum, ob diese beschränkt oder unbeschränkt sind.

8) ***Editor Facade****:* Die EditorFacade ist die Facade Klasse des Moduls. Hier werden alle nötigen Funktionen durch Methoden repräsentiert, um die Aufgabe des Moduls darzustellen. Beim Erstellen werden GTFS Daten eingelesen. Die Klasse enthält einen FareController, über den auf die Methoden zum Ändern der Kostendateien zugegriffen werden kann. Auch das Ausführen der Änderungen ist als Methode vorhanden.

IV. Implementation

In diesem Abschnitt des Dokuments werden Teile des Codes vorgestellt. Dabei wird sich jedoch nur auf das Modul publicTranportRouting beschränkt, da dieses die Hauptfunktion des Programms repräsentiert. Zu den Hauptfunktionen gehören das Routing und die Verwaltung des Graphen, d.h. Erstellung, Speicherung und Laden.

Dieser Abschnitt behandelt, wie Graphhopper in Public-Transit-for-AGADE implementiert wurde. Des Weiteren wird beschrieben, inwiefern die Bibliothek von Graphhopper wo eingesetzt und genutzt wird, und wie Graphhopper innerhalb des Programms angesprochen wird. Dazu werden diese Sachen jeweils mit dem entsprechenden Code erläutert. In dem darauffolgenden Abschnitt wird auf Aspekte eingegangen, die während der Implementation, anders als in der Planung angedacht, umgesetzt werden mussten. Schließlich wird ein Anwendungsbeispiel mit zugehörigem Code dargestellt und kurz erklärt.

Hierbei wird Graphhopper von drei Klassen direkt für eine Funktion benutzt, sowie von einigen weiteren, welche jedoch nur ein paar Model Klassen nutzen. Weiterhin ist Graphhopper durch einfache Maven Abhängigkeiten implementiert worden und somit nutzbar.

A. Implementation von Graphhopper

1) ***Graph erstellen****:* Die ersten beiden Stellen an denen Graphhopper genutzt wird, ist zum einen die Erstellung des Graphen, und zum späteren Laden eines erstellten. Dazu werden zwei verschiedene Methoden in der Klasse GtfsGraphController verwendet.

Um ein Graph erstellen zu können benötigt Graphhopper eine GraphhopperConfig Klasse, welche Einstellungen bzw. eine Art Anleitung für die Erstellung eines Graphen von Graphhopper darstellt. Wie in Fig. 7 zu sehen, benötigt diese Konfigurationsklasse einige Einstellungen und Informationen, um damit einen Graphen erstellen zu können. Zum einen müssen die Dateipfade der OSM und der GTFS Dateien als String angegeben werden, welche zum Erstellen des Graphen verwendet werden sollen. Dabei wurden unter Ressourcen extra Ordner für OSM, GTFS, Graph, und Routen Dateien angelegt. Hier wird dann automatisch auf diese Ordner, je nach geforderter Datei, verwiesen, sodass man nicht immer die ganzen Pfade, sondern nur die Namen der Dateien angeben muss. Des Weiteren benötigt die Konfigurationsdatei eine Location und einen Namen, wie der fertige Graph gespeichert werden soll. Hierzu wurde der Name der OSM Datei verwendet, sodass der Speicherort und Name des Graphen nicht manuell erstellt werden müssen. Zuletzt wird noch ein flagEncoder benötigt, welcher ein Routing Profil darstellt, dass dann verwendet wird, um den Graphen zu erstellen. Dafür wurde das vordefinierte Profil “foot“ gewählt, welches normalerweise nur Routing zu Fuß ermöglicht. Beim Vorhandensein von GTFS Daten wird jedoch für das Routing der öffentliche Verkehr in Verbindung mit Fußwegen genutzt.

Nun wird ein GraphopperGtfs-Objekt mithilfe der Konfigurationsdatei erstellt und die Methode “init“ des Hoppers aufgerufen, wodurch der Graph erstellt und gespeichert wird. Anschließend wird das GraphhopperGtfs-Objekt noch geschlossen, wie in Fig. 7 zu sehen, mit der Methode “importAndClose“.

```
public static void createGraph(String OsmFile, String GtfsFile){
    //Sets the information for the config File which is needed to create the Graph

    //In graphopper their is no extra Encoder for public transit instead it uses
    // "foot" which includes foot an public transit if a gtfs File is given
    String flagEncoder = "foot";
    String osmFile = "src\\main\\resources\\OSM_Files\\"+OsmFile;
    String graphLocation = "src\\main\\resources\\graph\\"+OsmFile+"_with_Transit";
    String gtfsFile = "src\\main\\resources\\GTFS_Files\\"+GtfsFile;

    //Creating a new config File with the set variables
    GraphHopperConfig config = createConfig(flagEncoder,osmFile,graphLocation,gtfsFile);

    System.out.println("Create Graph .........");
    //Creating a hopper explicit for gtfs which then creates and saves the time expanded graph
    GraphHopperGtfs hopper = new GraphHopperGtfs(config);
    hopper.init(config);
    //After creating the hopper an graph is closed properly
    hopper.importAndClose();
```

Fig. 7. Code der createGraph-Methode des Public-Transit-for-AGADE Programms, um einen Graph mithilfe von Graphopper zu erstellen.

2) ***Graph laden****:* Die zweite Anbindung zu Graphhopper ist das Laden eines existierenden Graphen, in der Methode “loadGraph“ aus der GtfsGraphConroller Klasse.

Um einen Graphen laden zu können, wird ebenfalls zuerst eine Konfigurationsdatei benötigt. Dazu wird eine von einem existierenden Graphen geladen, welche vorher gespeichert sein muss. Diese muss nun wieder einem GraphHopperGtfs-Objekt übergeben werden und die Methode “init“ mit dieser ausgeführt werden. Damit der Graph nun geladen werden kann, muss von dem Hopper die “load“ Methode mithilfe der Location des Graphen aufgerufen werden. Diese Location wird aus der geladenen Konfigurationsdatei entnommen.

Als nächstes wird die Klasse PtRouteResource von Graphhopper genutzt, welche das Routing auf dem time-expanded Graph ermöglicht. Um dieses Objekt zu erstellen werden, wie in Fig. 8 gezeigt, einige andere Objekte benötigt, welche von dem GraphHopperGtfs übergeben oder erstellt werden können. Mit “createFactory“ und den nötigen Parametern wird dann die

PtRouteResource erstellt und zurückgegeben. Mithilfe dieser, kann dann Routing auf dem geladenen Graphen durchgeführt werden.

```
public PtRouteResource loadGraph(String graphFolderName){
    System.out.println("Loading Graph .........");

    GraphHopperConfig config;
    //Load the config of an existing graph to load the graph again
    config = loadConfig(graphFolderName);

    GraphHopperGtfs hopper = new GraphHopperGtfs(config);

    hopper.init(config);
    //loading the graph from the location given in the config file
    hopper.load(config.getString( key: "graph.location",config.toString()));

    //Creating some Objects which are needed to then create the PtRouteResource
    LocationIndex index = hopper.getLocationIndex();
    GtfsStorage storage = hopper.getGtfsStorage();
    PtRouteResource PT = PtRouteResource.createFactory(new TranslationMap().doImport(),hopper,index,storage)
                                        .createWithoutRealtimeFeed();

    return PT;
}
```

Fig. 8. Code der loadGraph-Methode des Public-Transit-for-AGADE Programms, um einen Graph mithilfe von Graphopper zu laden.

3) ***Verbindungsanfrage erstellen****:* Die nächste Anwendung ist die Erstellung einer Verbindungsanfrage, welche dann mithilfe des Routings in eine fertige Route umgewandelt wird. Hier wird Graphhopper jedoch nicht direkt angesprochen, sondern nur eine Klasse für spätere Verwendung erstellt und definiert. Hierbei ist die Methode, Teil der Klasse GraphhopperController.

Dabei benötigt das Erstellen eines Graphhopper-Request jeweils nur einen Start- und einen Zielort, diese sind wiederum mit den jeweiligen Breiten- und Längengraden anzugeben. Nun müssen wie in Fig. 9 zu sehen, noch ein paar Attribute angegeben werden, bevor die Anfrage zum Routing bereit ist. Dazu gehören zum einen die frühestmögliche Abfahrtszeit der Anfrage, welche durch die, an die Methode übergebene Zeit angegeben ist, sowie das Setzen der Profil Abfrage auf "true", damit dann die gesetzten Routing Profile beachtet werden. Weiterhin wird festgelegt, dass Transfers nicht ignoriert werden, und somit bei der Routenfindung auch ein Kriterium darstellen. Dadurch wird eine Art des multikriteriellen Routings ermöglicht. Zuletzt wurde noch eine Anzahl an möglichen Lösungen bzw. Routen festgesetzt welche, Graphhopper dann als Antwort liefern soll.

```
public void createRequest(Location from, Location to, LocalDateTime dateTime, ZoneId zoneId) {
    Request request = new Request(from.getLat(), from.getLon(), to.getLat(), to.getLon());
    request.setEarliestDepartureTime(dateTime.atZone(zoneId).toInstant());
    request.setProfileQuery(true);
    request.setIgnoreTransfers(false);        //Transfers are taken into consideration
    request.setLimitSolutions(5);             //setting a maximum of resulting Routes

    this.request = request;
}
```

Fig. 9. Code der createRequest-Methode des Public-Transit-for-AGADE Programms, um eine Anfrage an Graphhopper zu erstellen.

4) ***Routing****:* Die letzte direkte Verbindung bzw. Anfrage an Graphhopper ist das eigentliche Routing bzw. die Verbindungsanfrage in eine Antwort von Graphhopper umzuwandeln. Hierbei wird wie in der vorherigen Funktion die Klasse GraphhopperController genutzt.

Dazu wird in erster Linie nur die Anfrage und die PtRouteResource benötigt. Indem, wie in Fig.10 dargestellt, die Methode "route" von der PtRouteResource mit der Anfrage aufgerufen wird, führt Graphhopper das Routing für diese Verbindungsanfrage aus, und liefert ein Antwort Objekt. Aus diesem Objekt können dann entweder alle Pfade oder der beste Pfad hinsichtlich der Zeit extrahiert werden. Ein Pfad ist dabei eine Routen Klasse von Graphhopper, welche aus einer Reihe von Legs besteht, und diese dann jeweils wiederum aus mehreren Stopps bestehen. Jede dieser Klassen besitzen viele weitere Attribute, welche für mehr Informationen abgerufen werden können. Wurden dann alle erwünschten Pfade aus der Response (Antwort von Graphhopper) extrahiert und in eine Array-List hinzugefügt, können mit der Methode "createRoute" für jede dieser Pfade eine Route erstellt werden. Dabei ist "createRoute" keine Methode von Graphhopper, sondern eine eigene.

```
public void findRoute(Request request, PtRouteResource PT, String routeSelection, String saveFileFormat) {
    System.out.println("Searching Routes .........");
    response = PT.route(request);             //routing form graphhopper and setting the response
    path = new ArrayList<>();

    //selects the how much an which routes should be created "all" and "best" --> default = "best"
    switch (routeSelection){
        case "all":
            path.addAll(response.getAll());
            break;
        case "best":
        default:
            path.add(response.getBest());
            break;
    }
    createRoute(saveFileFormat);
}
```

Fig. 10. Code der ptRouteQuery-Methode des Public-Transit-for-AGADE Programms, um eine Anfrage in eine Route umzuwandeln.

5) ***Routenerstellung****:* In der Klasse GraphhopperResponse-Handler wird Graphhopper nicht direkt benutzt, jedoch wird auch, wie in der Funktion zum Erstellen einer Anfrage, ein Objekt welches Graphhopper generiert genutzt. Hier werden dann aus den Pfaden von Graphhopper die Routen gebildet, und zum Speichern zurückgegeben. Hierbei wird nur sehr grob auf die einzelnen Methoden textlich eingegangen, da diese nur indirekt mit Graphhopper verbunden sind.

In der Methode "buildRoute" wird die Route kreiert und initialisiert. Dazu gehört eine Liste von Legs und Stopps, anzulegen, welche im weiteren Verlauf gefüllt werden. Des Weiteren werden aus dem Pfad nötige Entscheidungskriterien und Informationen entnommen und der Route hinzugefügt.

```
public Route build() {
    buildRoute();           //creates the Route
    buildLegs();            //creates and adds all legs to the Route as well as all stops
    deleteDoubleStops();    //checks the stops list and delete doubles

    return route;
}
```

Fig. 11. Code der build-Methode der GraphhopperResponseHandler Klasse, welche die Zwischenschritte, der Routenerstellung darstellt.

In der "buildLegs" Methode, werden die jeweiligen Legs erstellt und einer Route zugeordnet. Dazu wird eine Liste von Stopps erstellt welche dann gefüllt wird. Aus dem Pfad

werden die Legs entnommen und auf ihren Typ überprüft. Je nach Typ, d.h. öffentliche Verkehrsmittel “pt“ oder per Fuß “foot“, werden dann unterschiedliche Attribute überprüft, extrahiert und in das neu erstellte Leg eingefügt. Des Weiteren werden Start- und Zielort abgerufen und eingefügt, sowie Start- und Endzeit eines Legs in Ticks als auch in ein normales Zeitformat umgerechnet und angegeben. Dazu wird der TimeController verwendet. Nachdem die Grundattribute des Legs erstellt worden sind, werden die Stopps des jeweiligen Legs hinzugefügt.

Die Funktion ist ähnlich der, wie Legs erstellt werden nur in leicht veränderter und in für Stopps angepasster Form. Hierbei werden dann aus dem Graphhopper Leg die Stopps angeschaut und nützliche Informationen in das neue Format überführt, sowie neue Informationen wie Ankunfts- und Abfahrtsticks berechnet und hinzugefügt. Nachdem alle Informationen extrahiert wurden und ein Stopp erstellt wurde, so wird dieser dann der Route und dem Leg hinzugefügt, und dieses Leg dann auch der Route.

Nachdem alle Legs und Stopps der Route erstellt wurden, werden alle Legs und Stopps dieser noch einmal durchlaufen und fehlende Informationen hinzugefügt.

Die ”deleteDoubleStops” Methode überprüft die Stopp Liste der Route, um doppelte aufeinanderfolgende Stopps zu löschen, da diese Redundant sind und gleiche Informationen enthalten. Dies geschieht, da jedes Leg als Startort den Zielort des letzten Legs besitzt, wodurch beim Erstellen der Stopp Liste der Route diese sich doppeln.

B. Anwendungsbeispiel

Als nächstes werden noch Anwendungsbeispiele bzw. eine Demo dargestellt und erklärt.

Um das Programm bzw. die Funktionen testen zu können, werden zwei Dateien benötigt, um den Graphen zu erstellen. Dabei handelt es sich um OSM und GTFS Dateien. Diese können, wie in Fig. 12 zu sehen ist, über die statische Methode “downloadFiles“ der Facade Klasse, unter Angabe der URLs der jeweiligen Dateien heruntergeladen werden. Dabei werden diese unter Ressourcen in den jeweiligen Ordnern gespeichert. Es ist allerdings darauf zu achten, dass die jeweiligen OSM und GTFS Daten zueinander passen und denselben Raum abdecken, um keine Fehler zu verursachen.

```
public void downloadFiles() {
    String osmFile_URL = "https://download.geofabrik.de/europe/germany/berlin-latest.osm.pbf";
    String gtfsFile_URL = "https://www.vbb.de/media/download/2029/GTFS.zip";

    PT_Facade_Class.downloadFiles(osmFile_URL, gtfsFile_URL);

}
```

Fig. 12. Code um OSM und GTFS Daten herunterzuladen.

Nachdem die Daten vorhanden sind kann, ebenfalls wieder über eine statische Methode “createGraph“ der Facade Klasse, der Graph erstellt werden. Wie in Fig. 13 zu sehen ist, werden als Übergabeparameter zum einen der Name der OSM Datei und der Name der GTFS Datei benötigt. Der Graph wird hierbei dann wieder unter Ressourcen im Graph-Ordner mit dem Namen der OSM Datei gespeichert.

```
public void createGraph() {
    String osmFile = "berlin-latest.osm.pbf";
    String gtfsFile = "gtfs.zip";

    PT_Facade_Class.createGraph(osmFile, gtfsFile);
}
```

Fig. 13. Code zum Anwenden der createGraph Metode der Facade Klasse um einen Graph zu erstellen.

Wenn der Graph erstellt wurde oder schon ein Graph vorhanden ist, muss dieser geladen werden, um Routen erstellen zu können. Dazu wird wie in Fig. 14 gezeigt, ein Objekt der Facade Klasse erstellt und eine Zeitzone des Gebietes, auf welchem Routing betrieben werden soll, sowie das volle Datum mit Jahr, Monat, Tag, die Startzeit ab wann das Routing starten soll und ein Datei Format (JSON oder YAML), als welche die fertigen Routen gespeichert werden sollen, benötigt. Danach wird die ”loadGraph” Methode der Facade Klasse aufgerufen, welche nun den Ordnernamen des Graphen als Parameter benötigt, um diesen zu laden. Zu finden ist dieser im Graph-Ordner unter Ressourcen.

```
public void loadGraph() {
    String graphFolderName = "berlin-latest.osm.pbf_with_Transit";

    PT_Facade_Class pt = new PT_Facade_Class( zoneId: "Europe/Paris", year: 2020,
                                              month: 8, day: 6, hour: 10, min: 0, fileFormat: "JSON");
    pt.loadGraph(graphFolderName);
}
```

Fig. 14. Code zum Anwenden der loadGraph Methode zum Laden eines Graphen.

Nachdem die Route geladen wurde, können nun beliebig viele Verbindungsabfragen erstellt werden. Um eine Abfrage zu erstellen werden zwei Locations benötigt, welche als Start- und Zielort dienen. Weiterhin wird eine Abfragezeit benötigt, welche dann die frühestmögliche Abfahrtszeit darstellt. Wie in Fig. 15 zu sehen ist, ist dabei das volle Datum und die Uhrzeit nötig. Zuletzt wird als Parameter nur noch die Routenauswahl gefordert, welche entweder “all“ oder “best“ ist. All sorgt dafür, dass alle Ergebnisse (max. 5) als Route zurückgegeben werden, während Best nur die eine beste Route hinsichtlich der Zeit zurückgibt. Mit all diesen Parametern ist dann nur noch die Methode ”ptRouteQuery” der Facade Klasse aufzurufen. Diese Methode leitet die Anfrage dann an Graphhopper weiter und erstellt die fertige Route. Danach wird die Route gespeichert.

Wenn die Route erstellt und gespeichert wurde, kann diese mit der statischen Methode ”loadRoute” von der Facade Klasse mithilfe des Dateipfads der Route wieder geladen werden. Hierbei ist jedoch noch der komplette Dateipfad anzugeben.

C. Änderungen zur ursprünglichen Planung

Während der Implementation bzw. Entwicklung gab es jedoch auch Abweichungen zur eigentlichen Planung des Pro-

```
public void ptRouteQuery() {
    Location from = new Location( lat: 52.456428,  lon: 13.139477);
    Location to = new Location( lat: 52.503893,  lon: 13.502197);
    LocalDateTime queryTime = LocalDateTime.of( year: 2020,  month: 8,  dayOfMonth: 6,
                                                hour: 11,  minute: 34);
    String routeSelection = "all";
    pt.ptRouteQuery(from, to, queryTime, routeSelection);
}
```

Fig. 15. Code zum Anwenden der ptRouteQueryMethode, zum erstellen einer Verbindungsanfrage und Route.

```
public void loadRoute() {
    PT_Facade_Class.loadRoute( filePath: "src\\main\\resources\\Routes\\" +
            "Route_52.456672,13.140033 -- 52.503737,13.502879.json");
}
```

Fig. 16. Code zum Anwenden der Methode loadROute, um eine vorhandene Route wieder einzulesen.

gramms, welche betrachtet werden mussten. Auf die größten wird hier nun eingegangen.

1) ***Routenmodell****:* Zu einem waren die von Graphhopper zurückgegebenen Pfade nicht in der Form, welche den Anforderungen an das Programm entsprechen, und repräsentieren zum Teil zu viele Informationen, die nicht gebraucht worden sind. So wurde deshalb ein extra Routenmodell erstellt, welches nur die nötigen Informationen und die richtige Form besitzt. Es wurde so entwickelt, dass es den Anforderungen angepasst ist und viele unnötige Informationen nicht miteinbezieht. Weiterhin ist dieses Modell deutlich übersichtlicher, enthält angepasste Entscheidungskriterien und macht es möglich, sowohl Zeitangaben in Ticks darzustellen als auch die Routen als Datei speichern zu können.

2) ***Zeitumrechnungen****:* Übergebene und geforderte Zeit der beiden Bibliotheken und zurückgegebene Zeit, unterscheiden sich aufgrund von verschiedenen Formaten voneinander. Deshalb wurde der TimeController, welcher nur für die Umwandlung in Ticks gedacht war, erweitert. Er wurde so erweitert, dass er die Zeit in die gegebenen und geforderten Formate jeweils umwandeln kann. Dies wird auch für Zeiten genutzt, welche dann in der fertigen Route dargestellt werden.

3) ***Iterieren über die Route****:* Da beim Füllen der Routenstruktur, diese nicht lückenlos gefüllt werden kann, muss direkt nach dem Erstellen diese noch einmal durchlaufen werden, um diese Lücken zu füllen und ggf. doppelte Informationen zu entfernen. Dazu wurde ein extra Iterator für das Routenmodell implementiert, um das Durchlaufen der Route zu ermöglichen. Beim späteren Laden der Route kann dieser dann auch genutzt werden, um über die Route zu iterieren.

4) ***Speichern der Konfigurationsdatei****:* Anfänglich wurde nicht beachtet, dass die Konfigurationsdatei, welche zum Erstellen des Graphen gebraucht wird, auch für das Laden dieses Graphen wieder benötigt wird. Nun ließ sich diese jedoch nicht einfach abspeichern und laden. Dazu wurde dann die Klasse TransitionConfigHandler implementiert, welche die Informationen der Konfigurationsdatei in eine eigen erstellte Klasse überführt, um diese dann zu speichern. Diese Funktion ist auch in die entgegengesetzte Richtung möglich, wenn diese wieder geladen werden muss.

5) ***Verbindungsabfragen****:* Entgegengesetzt zu der eigentlichen Planung eines multikriteriellen und multimodalen Routings, ist dies mit der momentanen Version des Programms nur in begrenztem Maße möglich. So wird multimodales Routing aufgrund der begrenzten vordefinierten Profile bisher nur für Fußwege und öffentliche Verkehrsmittel verwendet, und berücksichtigt andere Verkehrsmittel nicht. Diese müssten per eigens definierten Profilen angepasst und hinzugefügt werden.

Des Weiteren wird multikriterielles Routing auch nur begrenzt unterstützt, da man nicht das Hauptkriterium, nachdem Routing betrieben werden soll, aussuchen kann. Dabei gibt es mehrere bestimmter Kriterien, welche in einer Reihenfolge berücksichtigt werden. Um dieser Tatsache indirekt etwas entgegenwirken zu können, wurden einige Kriterien in dem Routenmodell, nach welchen man die Routen bewerten kann, hinzugefügt.

ZUSAMMENFASSUNG

Allgemein zu sagen ist, das Routing von öffentlichen Verkehrsmitteln für Routenplaner und auch Simulationsprogramme in vielerlei Hinsicht wichtig und hilfreich ist. Dabei kann durch Simulationen analysiert werden, wie sich mögliche Veränderungen von Infrastruktur oder Fahrplänen auf den Verkehr auswirken würden, und mithilfe von Planern sinnvolle Reiserouten für Individuen erstellen, welche den öffentlichen Verkehr und andere Verkehrsmittel miteinbeziehen.

In der Arbeit wurde Grundwissen vermittelt, für die jeweiligen Technologien, Formate und genutzten Bibliotheken, welche im Verlaufe der Arbeit benötigt wurden. Das GTFS Format wurde als Format für die Fahrplandaten der öffentlichen Verkehrsmittel genutzt, und Aufbau sowie Struktur der Daten und Text Dateien erklärt. Das OSM Format wurde hingegen für die Darstellung des Straßennetzwerks und der Karten genutzt, um mit diesen den multimodalen Aspekt zu unterstützen. Zum Modellieren der Netzwerke wurde auf Graphen eingegangen. Dazu wurden zwei besondere Graph Modelle, den time-expanded und den time-dependent Graph, vorgestellt und deren Besonderheiten, Vor- und Nachteile erklärt. Beide Graphen sind dafür gedacht, den öffentlichen Verkehr darstellen zu können, indem dem Graphen der Aspekt der Zeitabhängigkeit hinzugefügt wird. Bei dem Time-expanded Graph werden dazu feste Zeitwerte den Arcs zugeordnet, während bei dem time-dependent Graph jeweils eine Funktion auf den Arcs diese Aufgabe übernimmt.

Graphhopper ist eine Routing Engine, und wurde als solche für das Programm verwendet. Dabei wird öffentliches multimodales Routing mithilfe von GTFS und OSM Daten unterstützt. Als Grundlage für das Routing nutzt Graphhopper einen leicht angepassten time-expanded Graphen, um dieses zu beschleunigen. Zum Editieren der GTFS Daten wurde das OneBusAway Programm genutzt. Dieses ist ein Programm zur Livedarstellung von Fahrplänen für den ÖPNV. Es bietet ein Modul zum Editieren der GTFS Feeds.

Das entwickelte Programm nutzt Graphhopper sowie GTFS und OSM Daten, um Routing des öffentlichen Verkehrs zu ermöglichen. Mit dem Programm lassen sich eine Vielzahl an Verbindungsabfragen berechnen, um eine Auswahl an möglichen Routen mit Zwischenhaltestellen zu erhalten. Die Rückgabe der Route erfolgt per JSON oder YAML Datei, welche gespeichert wird, und bei Verwendungen wieder abgerufen werden kann. Durch die Facade Struktur wird eine einfache Schnittstelle zur Integration und Bedienung angeboten. Dabei wird Graphhopper für die Erstellung des time-expanded Graphen genutzt, sowie für das multimodale Routing auf diesem.

Das GtfsEditing Modul hingegen, zum Lesen, Bearbeiten sowie Speichern der GTFS Feeds, um so Änderungen am Graph bzw. Verkehrsinfrastruktur im Voraus bewirken zu können. Hierbei trifft dies aber bisher nur für die Dateien, welche die Kosten darstellen, zu. Dieses Modul besitzt auch eine extra Facade Klasse, um die Funktionen der Module möglichst zu trennen.

Auch Anforderungen und der Umfang des Programms wurde dargestellt. Dabei ist das Programm eigenständig und unabhängig anderer Programme nutzbar. Weiterhin wurde die eigentliche Implementation der Graphhopper Bibliothek in dem publicTransportRouting Modul dargestellt und anhand von Code gezeigt. Graphhopper wird für die Verwaltung des Graphen, d.h. Erstellung, laden und speichern, sowie Routing und beim Erstellen der Route, verwendet. Direkt wird Graphhopper dafür genutzt, wenn aus den GTFS Daten, OSM Daten und einer GraphhopperConfig, der time-expanded Graph erstellt wird oder ein vorhandener Graph geladen wird. Weiterhin wird Graphhopper auch beim eigentlichen Routing verwendet, indem aus einer Verbindungsabfrage ein Pfad erstellt wird. Des Weiteren wird bei weiteren Funktionen wie Erstellen der Verbindungsanfrage und erstellen der Route, Graphhopper indirekt durch Verwenden von Klassen genutzt.

Zum Benutzen des publicTransportRouting Moduls wurde ein Anwendungsbeispiel vorgestellt. Dieses Beispiel wurde dann mithilfe von vorhandenen Screenshots des Codes einiger Methoden, erläutert und deren Funktionen bei dem Benutzen des Moduls erklärt.

Anders als zur eigentlichen Planung, mussten einige Sachen angepasst oder anders gelöst werden. Da die Rückgabedatei von Graphhopper nicht alle wichtigen Infos oder zu viel darstellt, wurde ein extra Routenmodell anhand der Anforderungen erstellt. Des Weiteren mussten Zeitumrechnungen hinzugefügt werden, um mehrere Zeitdarstellungen zu bedienen. Auch wurde eine extra Konfigurationsdatei erstellt, da sich die GraphhopperConfig nicht einfach speichern ließ. Für das eigentliche Routing sind multimodale und multikriterielle Aspekte im Vergleich zu geplantem bisher auch nur in begrenzten Maßen verfügbar.

Future Work

Zuletzt wird in dieser Arbeit noch ein Ausblick auf zukünftige Erweiterungen, sowie auf mögliche Verbesserungen des Programms eingegangen. Des Weiteren wird auf Probleme hingewiesen, die aktuell keine Lösung besitzen bzw. in der Zukunft angegangen und möglicherweise behoben werden können. Dazu wird auf beide der entwickelten Module einzeln eingegangen. Hierbei zuerst auf das Hauptmodul und danach auf das Bearbeitungsmodul.

Wie es im Laufe der Arbeit schon dargestellt wurde, ist in der aktuellen Version des Programms nur bedingt multimodales und multikriterielles Routing möglich. Bei dem multimodalen Aspekt ist es bisher nur möglich öffentliche Verkehrsmittel inklusive Fußwege zu benutzen. Zukünftig könnte man diese Funktion um weitere Verkehrsmittel wie Auto, Fahrrad, Leihfahrzeug etc. erweitern, um vollständigen oder zumindest erweiterten multimodalen Verkehr zu ermöglichen. Dazu können Graphhopper-Profile angepasst oder eigene erstellt werden, die diese Funktionen dann darstellen sollten. Weiterhin sollte ein ähnliches Vorgehen auch multikriterielles Routing auf Abfrage ermöglichen, um somit ein Hauptkriterium, nach dem Routing betrieben werden soll, zu setzen. Dabei sollte es möglich sein, eigene Gewichtungen für Graphen zu erstellen und definieren, um diese dem Graphen hinzuzufügen, damit mehrere Kriterien zum Routing zur Auswahl stehen.

Um möglichst realistisches Routing zu ermöglichen, könnte zukünftig auch die Erweiterung GTFS-Realtime des GTFS-Formates, in das System integriert werden, um auch Livedaten für das Routing berücksichtigen zu können. Hierzu benötigt es einen Anbieter der Feeds zur Verfügung stellt, damit Graphhopper diese nutzen kann.

Des Weiteren kann zum besseren Arbeiten mit den fertigen Routen, die Klasse des Route Iterator durch weitere Methoden erweitert werden um bessere und mehr Abfragen auf einer Route, für mögliche Weiterverarbeitungen, tätigen zu können.

Im Falle des Bearbeitung Moduls der GTFS Daten ist dieses im Rahmen der Arbeit nur in anfänglichen Schritten behandelt und entwickelt worden. Zukünftig könnte es sein, neben den GTFS Daten, welche die Kosten darstellen, auch die anderen Daten bearbeiten und erstellen zu können, um den Graphen, welcher für die Routenberechnung bzw. Simulation genutzt wird zu manipulieren. Somit könnte man analysieren, wie sich diese Veränderungen auswirken. Weiterhin kann die bereits vorhandene Funktion der Kostenerstellung noch verbessert werden, indem diese weiter automatisiert und benutzerfreundlicher gestaltet wird. Somit könnte man die vielen erforderlichen Benutzereingaben reduzieren und die Validierung der Daten automatisieren, um diese nicht manuell überprüfen zu müssen.

Ein Problem, welches sich jedoch erst zukünftig lösen lassen kann, ist die Verfügbarkeit der GTFS Daten für den ÖPNV. Denn nicht alle Verkehrsunternehmen bieten GTFS Daten an, wodurch das Routing mit öffentlichen Verkehrsmitteln für diese Bereiche nicht möglich ist. Zukünftig könnte es jedoch durch die weitere Verbreitung des GTFS Formates und des immer häufiger zur Verfügung stellen von Open Data für Entwickler möglich sein, dass auch diese Unternehmen diese Daten veröffentlichen.

References

[1] K. Nagel, "Multi-modal traffic in transims," Zürich, Switzerland, 2001, (Stand 14.10.2020). [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.7186&rep=rep1&type=pdf

[2] K. S. Lee, J. K. Eom, and D. seop Moon, "Applications of transims in transportation: A literature review," in *Procedia Computer Science*, E. Shakshuki and A. Yasar, Eds. Elsevier, 2014, vol. 32, pp. 769–773. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050914006899

[3] M. Rieser, "Modeling public transport with matsim," in *The Multi-Agent Transport Simulation MATSim*, K. W. Axhausen, A. Horni, and K. Nagel, Eds. London: Ubiquity Press, 2016, pp. 105–110. [Online]. Available: https://library.oapen.org/handle/20.500.12657/32162

[4] T. Pajor, "Algorithm engineering for realistic journey planning in transportation networks," Ph.D. dissertation, 2013. [Online]. Available: https://publikationen.bibliothek.kit.edu/1000042955

[5] K. Giannakopoulou, A. Paraskevopoulos, and C. Zaroliagis, "Multimodal dynamic journey-planning," 2019, (Stand 12.10.2020). [Online]. Available: https://www.mdpi.com/1999-4893/12/10/213/html

[6] J. Geyer, J. Nguyen, T. Farrenkopf, and M. Guckert, (Stand 14.10.2020) von GitHub. [Online]. Available: https://github.com/KITECloud/AGADE-Traffic

[7] ——, "Agade traffic," 2019.

[8] H. Bast, D. Delling, A. Goldberg, and et al., "Route planning in transportation networks," 2015, (Stand 14.10.2020). [Online]. Available: https://arxiv.org/pdf/1504.05140.pdf

[9] J. Hrnčíř and M. Jakob, "Generalised time-dependent graphs for fully multimodal journey planning," in *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, 2013, pp. 2138–2145, (Stand 14.10.2020). [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6728545

[10] OpenStreetMap, "Openstreetmap homepage," (Stand 14.10.2020). [Online]. Available: https://www.openstreetmap.de/index.html

[11] OpenSteetMap, "Openstreetmap wiki," (Stand 14.10.2020). [Online]. Available: https://wiki.openstreetmap.org/wiki/Main_Page

[12] D. Comeaux, Marcy, HeidiG, Y. Yuan, and et al., "General transit feed specification," 2019, (Stand 20.09.2020). [Online]. Available: https://www.transitwiki.org/TransitWiki/index.php/General_Transit_Feed_Specification

[13] Graphhopper GmbH, "Graphhopper Documentation GitHub," (Stand 8.8.2020). [Online]. Available: https://github.com/graphhopper/graphhopper/blob/1.0/docs/index.md

[14] MobilityData IO, "Open mobility data - feeds," (Stand 20.09.2020). [Online]. Available: http://transitfeeds.com/feeds

[15] P. Brosi, "Gtfs.de-feeds-deutschlandweite gtfs feeds," (Stand 20.09.2020). [Online]. Available: https://gtfs.de/de/feeds/

[16] Deutsche Bahn AG, "Deutsche Bahn Datenportal Datensätze," 2016, (Stand 14.10.2020). [Online]. Available: https://data.deutschebahn.com/dataset?groups=datasets

[17] Verkehrsverbund Rhein-Ruhr AöR, "Open data öpnv," (Stand 20.09.2020. [Online]. Available: https://www.opendata-oepnv.de/ht/de/willkommen

[18] Google LLC, "Transit APIs-Static Transit Overview," 2019, (Stand 20.09.2020). [Online]. Available: https://developers.google.com/transit/gtfs

[19] GTFS.org, "Gtfs," (Stand 14.10.2020). [Online]. Available: https://gtfs.org/

[20] Google LLC, "Transit APIs-Realtime Transit," (Stand 14.10.2020). [Online]. Available: https://developers.google.com/transit/gtfs-realtime

[21] Y. Wang, Y. Yuan, Y. Ma, and G. Wang, "Time-dependent graphs: Definitions, applications, and algorithms," in *Data Science and Engineering*. Springer, 2019, vol. 4, p. 352–366. [Online]. Available: https://doi.org/10.1007/s41019-019-00105-0

[22] T. Pajor, *Multi-modal route planning*, 2009, ch. 3. [Online]. Available: https://i11www.iti.kit.edu/extra/publications/p-mmrp-09.pdf

[23] G. Stølting Brodal and R. Jacob, "Time-dependent networks as models to achieve fast exact time-table queries," *Electronic Notes in Theoretical Computer Science*, vol. 92, pp. 3 – 15, 2004, proceedings of ATMOS Workshop 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104000040

[24] Graphhopper GmbH, "Open Source Code - Graphhopper Routing Engine / GitHub," (Stand 8.8.2020). [Online]. Available: https://github.com/graphhopper/graphhopper

[25] ——, "Graphhopper - Homepage," (Stand 8.8.2020). [Online]. Available: https://www.graphhopper.com/de/

[26] OneBusAway, "onebusaway-gtfs-modules," (Stand 14.10.2020). [Online]. Available: https://github.com/OneBusAway/onebusaway-gtfs-modules

[27] ——, "onebusaway - tools for improving the usability of public transit," 2015, (Stand 14.10.2020). [Online]. Available: http://developer.onebusaway.org/modules/onebusaway-gtfs-modules/current/