Software-Projekt Praktikum
Wintersemester 2011 - 2012

Kurzdokumentation des SWP-Routenplaners

Projektarbeit im Studiengang Informatik

Theoretical Computer Science
Prof. P. Rossmanith
Rheinisch-Westfälische Technische Hochschule Aachen

Betreuer: Alexander Langer, Felix Reidl

Unser Team

Abbildung 1: Das studentische SWP-Team.
Von links nach rechts: Patrick, Arvid, Albert, Jan, Arno, Ioannis, Alex, Olli, Marc

Inhaltsverzeichnis

1 Einleitung

Der SWP-Routenplaner soll es ermöglichen, beliebige Kartendaten aus dem freien Kartenmaterial der OpenStreetMap[3] zu nutzen um darauf die schnellsten Routen zu berechnen. Ein mögliches Szenario wäre ein bevorstehender Urlaub. Der Nutzer - nennen wir ihn Felix - möchte einen Urlaub in Belgien machen. Dort möchte er sowohl in Brüssel, als auch in ein paar anderen Städten Sehenswürdigkeiten bewundern und diese mit dem Auto möglichst schnell erreichen. Nun kann sich Felix die OSM-Daten von Belgien runterladen (z.B. bei der Geofabrik[1]) und diese Daten dem SWP-Routenplaner zum Pre-Processen geben. Dies dauert zwar etwas, aber danach kann er die vorbereiteten Daten mitnehmen und so vor Ort einfach seinen kleinen Laptop aufmachen und diese Daten einladen. Felix kann nun gemütlich durch die gezeichneten Straßen navigieren und mit wenigen Klicks seinen Start- und Zielort angeben. Der Routenplaner berechnet dann die optimale Route und gibt sie sofort aus. Dadurch ist Felix nicht auf das Internet vor Ort angewiesen und kann beliebige Routen in kürzester Zeit berechnen und betrachten. Damit spart Felix viel Zeit und kann Belgiens Sehenswürdigkeiten in Ruhe genießen.

Das folgende Kapitel soll erklären, wie das Programm genutzt werden kann. Die Kapitel danach werden sich mit den Grundlagen und dem Code des Programmes beschäftigen. Im Anhang finden sich darüber hinaus interessante Fragen, Zahlen und Quellen.

2 Benutzung

Wenn das Programm gestartet wird, wird eine bereits pre-processte Karte von Aachen geladen und gezeichnet. Sobald das Laden der vorbereiteten Daten abgeschlossen ist, kann man je nach Wunsch mit der Maus oder mit den Icons am unteren rechten Fensterrand über die Karte navigieren. Die Zoomstufe kann hierbei mittels des Mausrades oder über die Tasten + (Reinzoomen) und - (Rauszoomen) verändert werden.

Die vorhandenen Straßentypen werden in 3 Kategorien eingeteilt. Die oberste Kategorie stellen Autobahnen und autobahnähnliche Straßen (Kraftfahrstraßen und Zubringer) dar. Diese werden in oranger Farbe dargestellt. Die zweite Kategorie umfasst hauptsächlich Bundesstraßen und Landesstraßen. Diese werden in gelber Farbe dargestellt. Alle restlichen Straßen werden mit weißer Farbe dargestellt. Wege die für Autos nicht passierbar sind, werden bereits beim Pre-Proccessing rausgefiltert, daher bietet das Programm auch keine Möglichkeit, diese nachträglich ein oder auszublenden. Auf den Straßen werden weiterhin die Straßennamen angezeigt, welche bei Einbahnstraßen oder getrennten Fahrspuren noch um ein >> erweitert sind, um die korrekte Fahrtrichtung zu kennzeichnen. Die Straßennamen können aufgrund der Kodierung mit Umlauten und Akzenten dargestellt werden, allerdings werden nicht-lateinische Schriftzeichen derzeit nicht dargestellt.

Um die Übersichtlichkeit der Karte auch bei höheren Zoomstufen zu gewährleisten, werden zunächst immer mehr Straßennamen ausgeblendet, bevor im nächsten Schritt die unterste Straßenkategorie nicht mehr gezeichnet werden. An dieser Stelle kann das Programm bei großen Karten einen Moment pausieren, da neue Dateien in den lokalen Speicher geladen werden müssen. Bei großen Zoomstufen wird auch die mittlere Kategorie nicht mehr gezeichnet. Sobald man aber wieder hereinzoomt, werden diese Informationen natürlich wieder hinzugefügt.

Für die Routenplanung selber kann mit den Buttons `Startpunkt auswählen` und `Zielpunkt auswählen` die gewünschte Route anzeigen und berechnen lassen. Start und Zielpunkt werden intern vom Programm gespeichert. Wenn man also nach dem berechnen einer Route einen neuen Startpunkt auswählt, wird eine Route zu dem alten Zielpunkt berechnet.

Die Route startet allerdings nicht an dem Punkt an den geklickt wurde, sondern an der nächsten Kreuzung oder Kurve, um zu gewährleisten, dass der Punkt auch auf einer Straße liegt, falls aus Versehen auf die graue Hintergrundfläche geklickt wurde, statt auf eine Straße. Aus diesem Grund ist es allerdings bei Straßen mit getrennten Fahrbahnen wichtig, auf welche Seite der Fahrbahn man klickt, da die Routenberechnung sonst unter Umständen erst mal bis zur nächsten Wendemöglichkeit vom Zielort wegfährt.

Da die die standardmäßig mitgelieferte Karte aber sehr begrenzt, ist kann die zugrunde liegende Karte durch eine andere Karte ersetzt werden. Dazu kann im Prinzip jede Karte im OSM-Format verwendet werden. Zu diesem Zweck steht der Menüpunkt `oeffnen` → `*.osm preprocessen` zur Verfügung. Praktisch sind durch den eigenen Computer allerdings Grenzen gesetzt, da bereits die Datei Deutschland.osm eine Größe von 20GB hat, lässt sich die gesamte Welt also auch weiterhin nur auf Webbrowsern betrachten. Somit ist klar, dass auch bei kleineren Karten nicht die gesamte Datei in den RAM-Speicher geladen werden kann. Daher wird das Fenster des Programms eingefroren, um alle verfügbaren Ressourcen dem Pre-Processing zur Verfügung zu stellen. Hierbei wird am Dateipfad ein neuer Ordner mit dem Namen `dateiname.osm.mapfiles` erstellt. Die Dateien die in diesem Ordner angelegt werden, sollten auf keinen Fall bearbeitet oder gelöscht werden. Sobald das Pre-Processing abgeschlossen ist, wird im Fenster ein neuer Ausschnitt der neu geladenen Karte gezeigt und es kann wieder über die Karte navigiert werden.

Diese sehr zeitaufwändige Prozedur (je nach Kartengröße mehrere Stunden Laufzeit) muss allerdings für jede osm-Datei nur einmal ausgeführt werden. Falls die selbe Karte erneut geöffnet werden soll, kann über den Menüpunkt `oeffnen` → `gepreprocesste Karte oeffnen`. Hier kann in dem vorher angelegten Ordner die Datei `ProcessedTilesConfig.tiles` geöffnet werden. Diese kümmert sich dann automatisch darum, dass die Karte geladen wird, hat aber eine deutlich kürzere Laufzeit als das Pre-Processing selber.

Auch wenn der Name der osm-Datei im Ordnernamen enthalten ist, wird die Datei nach dem Pre-Processing nicht mehr benötigt. Es ist also möglich vor Beginn eines Urlaubs zu Hause auf einem leistungsfähigen Computer das Pre-Processing zu erledigen, und anschließend den erstellten Ordner auf einen Laptop zu überspielen, um vor Ort seine Tagesrouten zu planen.

3 Grundlagen

Dieses Kapitel befasst sich mit den Grundlagen, die für das Verständnis des Codes nötig sind.

3.1 OpenStreetMap

OpenStreetMap ist ein Open Source Projekt, das gesammelte Geodaten verarbeitet. Die Geodaten werden zuvor von freiwilligen Helfern gesammelt und dann editiert. Die Rohdaten werden hierbei mit Tags und POI (Points of Interest) aufbereitet.

Nodes, Ways und Relations sind die Tags die hier zum Einsatz kommen und für das automatische Generieren der Karten von wichtiger Bedeutung sind. Nodes sind Punkte, die ihre GPS Koordinaten, ihre UniqueID und weitere Attribute enthalten, die aber an dieser Stelle nicht weiter von Bedeutung sind. Ways sind aus mehreren, aber mindestens zwei Punkten, bestehende Linien, die unter anderem Straßen, Wege oder auch Flüsse repräsentieren. Die genau Typisierung ist in den Attributen zu finden.

Relations fassen Objekte (Nodes und Ways) zu Gruppierungen, wie Autobahnen, Kreuzungen oder Kurven zusammen. Die jeweiligen Tags enthalten für ihren Typen spezifische Attribute, wie z.B. eine Node ihre GPS Koordinaten und eine Way ihre Typisierung als Autobahn. Infolgedessen werden, die nun editierten Daten, in OSM Dateien gespeichert. Hierbei ist zu beachten, dass das OSM Dateiformat nie wirklich festgelegt wurde und somit Unterschiede zwischen den einzelnen Dateien festzustellen sind.

3.2 Graphentheorie

Ein zentraler Bestandteil der Informatik ist die Graphentheorie.[7] In einem Routenplaner ist es naheliegend mit Graphen zu arbeiten, da die Kanten intuitiv die Straßen darstellen können und ein Stadtplan so einfach dargestellt werden kann in Form eines Graphen. Wir haben uns dazu entschieden gerichtete Graphen zu nutzen, da wir dadurch Einbahnstraßen einfach darstellen können.

3.3 Algorithmen

Das Berechnen des kürzesten Weges von Ort A nach Ort B ist ein komplexes Problem. Ein schlichtes Ausprobieren oder ein einfacher Dijkstra[4] führen zwar zum Ziel, allerdings wäre die Laufzeit bei dem einfachen Dijkstra-Algorithmus mit etwa $O(n^2 + m)$, wobei n die Anzahl der Knoten und m

die Anzahl der Kanten ist, viel zu hoch. Aus diesem Grund haben wir uns entschieden den „Highway Hierarchies “-Algorithmus zu nutzen. Dieser Algorithmus setzt allerdings ein Pre-Processing der Daten voraus, welches einmalig im Vorfeld gemacht werden muss.[5]

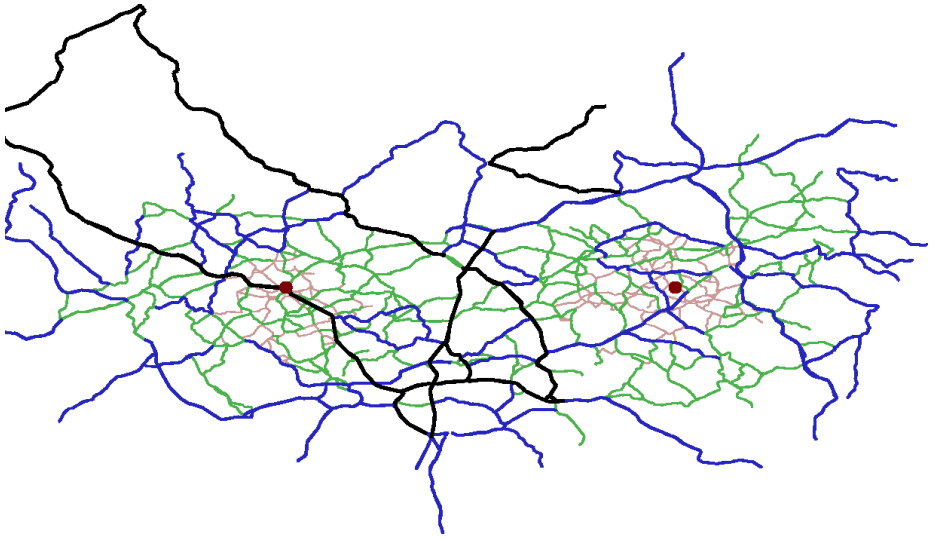

Abbildung 2: exemplarische Darstellung der Hierarchieebenen am Beispiel der Stadt Limburg und eines Punktes in ca. 100km Entfernung.[5]

Die Grundidee des Algorithmus ist recht simpel. Wenn ein Autofahrer einmal von Aachen nach Berlin fahren will, wird er auf dem Weg sicherlich einige Hauptstraßen und Autobahnen nutzen, allerdings keine Straßen in Wohnvierteln von Köln. Also wird eine Filterung vorgenommen. Diese Filterung entscheidet im Pre-Processing, welche Kanten (entsprechend der Straßen) wichtig sind und welche weniger wichtig. Dazu werden mehrere Hierarchieebenen berechnet. Die Routenberechnung selbst kann dann immer höhere Ebenen suchen und dadurch bei jeder neuen Ebene die Menge der betrachteten Kanten weiter auf wichtige Kanten reduzieren. Dies geschieht vom Start- und vom Endknoten aus gleichzeitig, sodass sich die Nachbarschaften treffen. Dadurch wird der Aufwand drastisch reduziert. In Abbildung 2 ist dies exemplarisch dargestellt.

Zunächst werden auf dem ersten Level noch viele kurze Kanten betrachtet (rot). Bei Aufstieg auf die nächste Ebene (grün) werden die Kanten länger und wichtiger. Schließlich sind auf der höchsten Ebene (schwarz) keine kleinen Kanten mehr zu sehen, sondern nur noch die wichtigsten. Da die originale Karte überall eine ungefähr gleiche Kantendichte hat, wie in der untersten Ebene zu sehen ist, ist die Filterwirkung enorm.

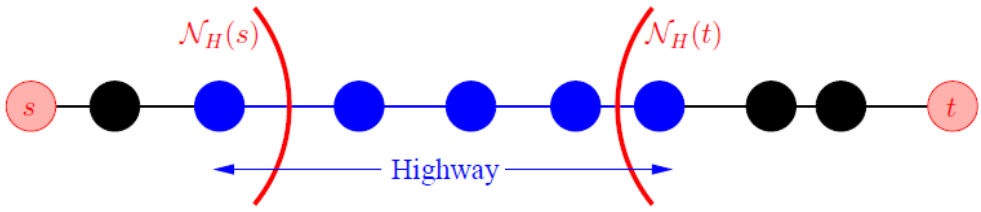

Abbildung 3: exemplarische Darstellung der Ermittlung einer wichtigen Kante mittels Nachbarschaften. N_H entspricht dabei unseren d_H.[5]

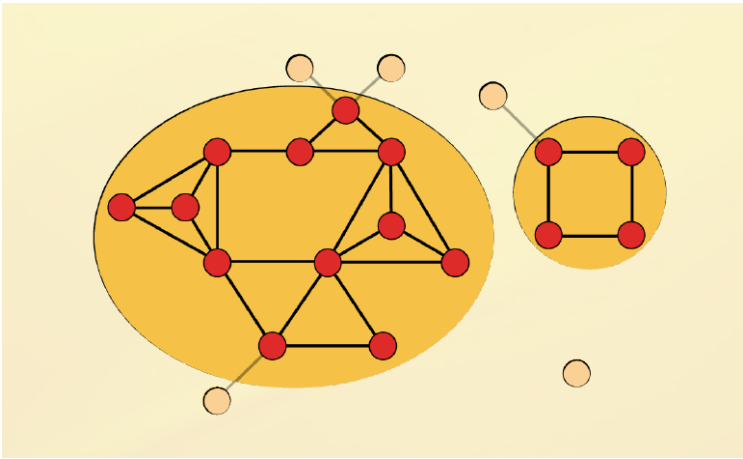

Abbildung 4: Durch entfernen aller Knoten mit einem Grad kleiner als 2 wird der Graph reduziert. Somit bleiben nur die eingekreisten Knoten übrig.[6]

Um diese Filterung vorzunehmen wird zunächst eine Konstante H für die späteren Nachbarschaftsgrößen festgelegt. Bei uns hat sie den Wert 50. Nun wird für jeden Knoten die Nachbarschaft berechnet. Das heißt, für jeden Knoten werden die nächsten H Nachbarn gesucht und die größte Entfernung d_H des Knotens zu einem Nachbarknoten wird dann für diesen Knoten gespeichert. Im nächsten Schritt wird nun für jeden Knoten ein azyklischer Shortest-Path-Baum erstellt. Diese Bäume werden von dem Knoten aus mit der Länge $2d_H + \epsilon$ erstellt. Im letzten Schritt werden nun auf solchen Bäumen die Blätter durchlaufen. Dabei werden Kanten, die die Nachbarschaft von der Wurzel oder des Blattes verlassen oder in keiner der beiden sind, auf die nächste Hierarchieebene gehoben. In Abbildung 3 ist zu erkennen, wie entschieden wird, ob eine Kante eine Ebene erhöht werden muss bei einem ungerichteten Graph.

In den Knoten wird die höchste Ebene gespeichert, welche eine inzidente Kante aufweist. Da auf der neuen Ebene nun durchaus Kanten vorkommen,

die in Zykel oder Sackgassen zeigen wird nun zusätzlich noch der 2-Core bestimmt. Der 2-Core ist eine konkrete Variante des k-Cores.[6] Dabei werden wiederholt alle Knoten entfernt, die auf dieser Ebene einen Grad kleiner als k haben. Dies ist in der Abbildung 4 dargestellt.

In unserer konkreten Implementierung werden einige Hierarchiestufen berechnet. Der Level 0 entspricht dabei einer Kopie der MapGraphen. Der Level 1 ist eine Kontraktion der Kanten des Level 0 und Entfernung von Zykeln und parallelen Kanten. Der folgende Level ist dabei eine Berechnung einer Hierarchiestufe. Der 3. Level entspricht des 2-Cores auf dem 2. Level und einer darauf ausgeführten Kontraktion und Entfernung der Zykel und parallelen Kanten. Die letzten beiden Schritte werden dann abwechselnd wiederholt bis zum höchsten Level.

4 Aufbau und Zusammenspiel

Dieses Kapitel zeigt zunächst den allgemeinen Aufbau und das Zusammenspiel der Module auf. Danach werden die Aufgaben und Aufbauten der einzelnen Module genauer beleuchtet. Für eine genaue Dokumentation der einzelnen Methoden in den Klassen sei an dieser Stelle auf die Javadoc des Projektes verwiesen.

4.1 Der allgemeine Aufbau

Unsere Software ist im Wesentlichen in fünf Paketen organisiert: Import, Graphenbib, Algorithmen, UI und Main. Hinzu kommt ein Paket, dass lediglich eigene Exceptions enthält und daher in der weiteren Beschreibung außer Acht gelassen wird. Die Aufgabe des Importpaket ist es OSM-Daten von der Festplatte zu lesen, für unser Programm aufzubereiten und wieder zu speichern. Das Graphenbibpaket enthält die hierfür benötigten Datenstrukturen, während im Algorithmenpaket die Algorithmen für kürzeste Wege und Hierarchieberechnung zu finden sind. Im UI-Paket ist die graphische Oberfläche implementiert. Das Mainpaket bietet der UI einen Schnittstelle zum Zugriff auf unsere Algorithmen, Datenstrukturen und gespeicherten Daten und regelt diesen.

Unsere Architektur orientiert sich dabei am Model-View-Presenter beziehungsweise Model-View-Controller. Dabei entspricht in unserem Fall der View dem UI-Paket, das Model den Paketen Algorithmen, Graphenbib sowie Import und der Presenter oder Controller dem Mainpaket. Das stark vereinfachte Klassendiagramm (Abbildung 5) mit den wichtigsten Klassen und Funktionen gibt einen ersten Überblick über das Zusammenspiel der Pakete. Die einzelnen Klassen und ihre Funktionen, die momentan vielleicht noch nicht klar sind, werden in den folgenden Abschnitten detaillierter erläutert.

4.2 Import

Der Import gliedert sich in 3 Bausteine. Den OSMImporter (Zentrale), FileTiler und OSMReader, welcher den WayReader nutzt. Als Zentrale, steuert der OSMImporter die Verarbeitung von OSM Daten und beherbergt das FileHandling des gesamten Projekts.

An einem Beispiel wird nun die Funktion des Imports verdeutlicht.

Der OSMImporter wird mit einer `beispiel.osm` Datei initialisiert, dabei wird die Hauptmethode `preProcess()`, aufgerufen welche den weiteren Vorgang steuert.

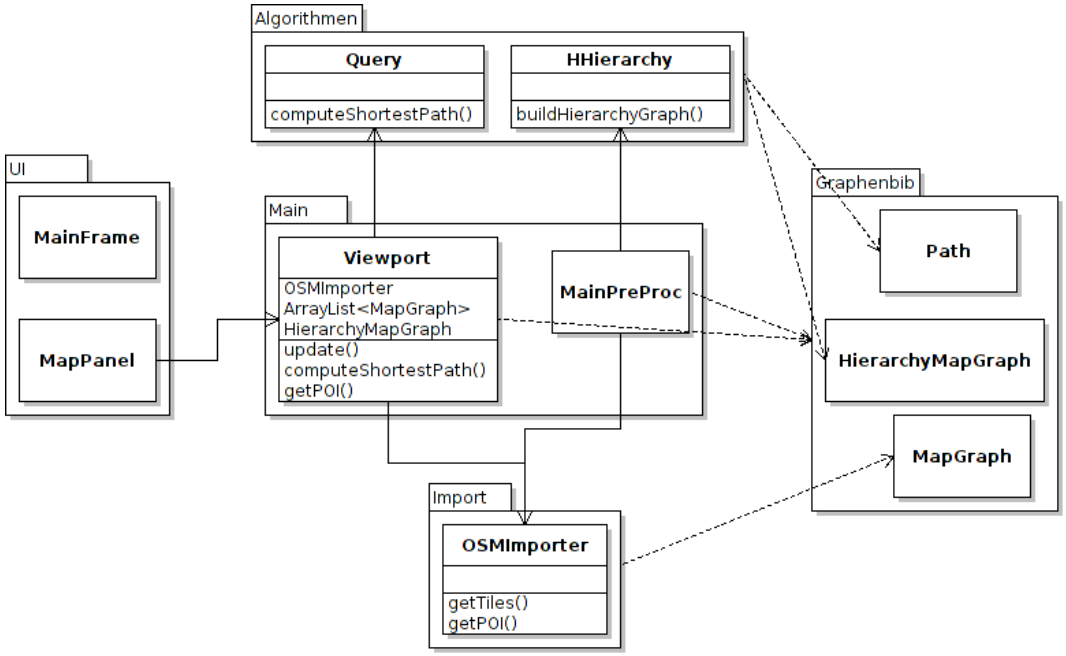

Abbildung 5: Überblick über das Zusammenspiel der Module

Je nach Dateigröße wird entschieden, ob die Datei noch in kleinere Stücke zerteilt werden muss um korrekt angezeigt werden zu können, ohne den Arbeitsspeicher zu sehr zu belasten. Ist dies der Fall, wird die Datei an den FileTiler weitergegeben. Dieser beginnt nun die Grenzen, in welchen sich die Knoten der OSM Datei befinden, zu bestimmen. Darauf folgt anhand der gewonnenen Daten eine Einteilung in so genannte `Tiles`. Ein Tile ist schlicht ein kleiner Ausschnitt aus der großen Fläche, die von der OSM Datei abgedeckt wird. Ist die Einteilung geschehen, werden die Knoten der OSM Datei auf die Tiles aufgeteilt. Hierzu wird ein XML-Parser[8] verwendet, der die OSM Datei Tag für Tag abwandert.

Alle Ways, die die Datei enthält, werden in eine separate Datei (`wayfile.osm`) geschrieben. Nachdem das Tiling abgeschlossen ist, beginnt die eigentliche Erstellung der Graphen unseres entworfenen Dateisystems. Hierfür ist der OSMReader zuständig. Er greift auf die einzelnen Tiles zu und fügt die darin enthaltenen Knoten in den Graphen ein der flächenmäßig ebenso groß ist wie das Tile. Anschließend werden die Ways, die zu den Knoten passen, über den WayReader in ein Hilfskonstrukt eingefügt. Hierbei wird ein Filter aktiv, der unerwünschte Kanten, wie z.B. Gebäude, herausfiltert. Sind alle relevanten Kanten ermittelt, werden die die jeweils zum Graphen gehörigen Kanten in diesen eingefügt. Dabei entstehen Knoten vom Grad null, da sie z.B. zu einem Gebäude gehörten. Diese werden anschließend

entfernt. Da an den Schnittstellen zwischen den Tiles die Kanten nicht korrekt eingefügt werden können, werden in jedem Graph Knoten ohne GPS Koordinate angelegt um die Übergänge zwischen den Tiles unsichtbar zu machen. Da die GPS Koordinaten jedoch zum Anzeigen benötigt werden, werden diese den Knoten nachträglich hinzugefügt. Für jedes Tile wird so ein Graph erstellt, welcher alle notwendigen Informationen enthält. Im Anschluss werden dann `ZoomGraphen` mit weniger Informationen erstellt, die für die großflächigere Anzeige von Straßen gedacht sind. In diesen sind z.B. nur noch Autobahnen enthalten.

Zunächst werden jetzt alle Graphen in den so genannten HierarchyGraph exportiert. Dieser enthält nur die für die Routenberechnung benötigten Informationen.

Bevor jeder Graph als serialisiertes Objekt auf der Festplatte gespeichert wird, um zu einem späteren Zeitpunkt wieder abgerufen werden zu können, wird noch eine ArrayList<Tile> mit den Randdaten wie z.B. Dateiname, Größe,... usw. angelegt.

Das eigentliche Pre-Processing ist hiermit abgeschlossen und der OSMImporter ist bereit für Anfragen anderer Programmteile.

Während des Programmablaufs werden immer wieder Tiles benötigt um andere Bereiche der Karte anzuzeigen. Eine Methode `getTile` lädt die Tiles in den Speicher, die den gewünschten Bereich abdecken. Um einen ersten Punkt auf der Karte zu suchen der angezeigt werden soll, gibt es eine Methode `getPOI`, was für „get Point-Of-Interest“ steht. Diese sucht das Tile mit den meisten Knoten und bestimmt die Koordinate in deren Nähe eine Ballung von Knoten ist. Für gewöhnlich ist dies z.B. die Mitte einer Stadt oder eines Wohnviertels.

4.3 Graphenbib

Wie der Name bereits vermuten lässt, ist dies die Graphenbibliothek. Sie dient dazu Graphen zu erstellen, zu verwalten und Werkzeuge für Graphenoperationen bereitzustellen. Aufgrund der sehr verschiedenen Aufgaben, die im Programm verrichtet werden müssen, gibt es zwei verschiedene Arten von Graphen. Die erste Variante ist der MapGraph und ist sehr nah an der originalen Darstellung der Straßendaten angelehnt. Natürlich besteht ein Graph aus Knoten und Kanten. Im Falle des MapGraphen sind dies die MapNode und die MapEdge. Letztere speichert neben Start- und Zielknoten auch die Länge und ihr Kantengewicht. Dieses wird berechnet aus der Länge der Kante und der durchschnittlichen Geschwindigkeit auf ihrem Straßentyp. Dazu gibt es den enum StreetType, welcher stark auf den Vorgaben aus der OpenStreetMap beruht. Für die Speicherung einer Position sind GPS-Koordinaten

eine sinnvolle und intuitive Wahl. Deshalb wird die Position einer MapNode dieser als GPSCoordinate mitgegeben. Soll hingegen ein Bereich beschrieben werden, wie es in einem MapGraphen interessant sein kann, so ist das GPSRectangle die richtige Wahl.

Soll eine Kante von einem Knoten zu einem anderen führen in dem MapGraphen, so muss diese Kante erstellt werden und bei den entsprechenden MapNodes registriert werden. Da der Graph gerichtet ist, muss dies in jede Richtung erfolgen und die MapNode bietet auch die Möglichkeit explizit nur die ein- oder ausgehenden Kanten auszugeben. Da dieses Verfahren der Registrierung fehleranfällig ist, wird es nach außen gekapselt durch entsprechende Methoden im MapGraphen. Der MapGraph bietet eine Vielzahl Methoden an, um Knoten und Kanten anzulegen ohne selbst ein solches Objekt erstellen zu müssen. Da jeder Knoten eine eindeutige ID hat, werden alle Knoten intern in einer HashMap gespeichert, welche jeder ID den Knoten zuordnen kann. Nach außen wird auch hier eine Methode im MapGraphen bereitgestellt, die es erlaubt, mit Wissen der ID, den entsprechenden Knoten anzufordern. Darüber hinaus bietet der MapGraph auch die Möglichkeit einen Iterator über alle Knoten oder Kanten auszugeben. Aufgrund des Tilings, welches im Bereich Import beschrieben wurde, kann es sein, dass einige Kanten keine korrekten Längeninformationen haben. Da allerdings immer wieder Tiles nachgeladen werden, bietet der MapGraph die Möglichkeit die Kantenlängen zu korrigieren und so eine gewisse Konsistenz zu gewährleisten. Weiterhin bietet der MapGraph noch die Möglichkeit, isolierte Knoten zu löschen, da sie für den Zweck dieses Programmes nicht gebraucht werden. Eine weitere nützliche Methode bietet die Möglichkeit für eine gegebene GPS-Koordinate den räumlich nächsten Knoten auszugeben. Dies ist insbesondere im Zusammenspiel mit der GUI wichtig. Zu guter Letzt bietet der MapGraph auch eine Export-Methode an. Diese extrahiert nur die wichtigsten Informationen und erstellt daraus einen HierarchyMapGraphen.

Der HierarchyMapGaph ist auf den ersten Blick sehr ähnlich zu obigem MapGraphen. Allerdings ist dieser HierarchyMapGraph weniger an die originalen Straßendaten angelegt als an einen kompakten algorithmischen Graphen. Einige Informationen sind daher für ihn uninteressant. Da es sich bei dem HierarchyMapGraphen auch um einen Graphen handelt gibt es auch entsprechende Knoten und Kanten. Dies sind HierarchyMapNode und HierarchyMapEdge. Diese bieten insbesondere die Möglichkeiten ein Level darin zu speichern. Dieser Level entspricht dabei der eigenen Hierarchieebene, wie sie in den Grundlagen erläutert wurden. Weiterhin sind in dem MapGraph eine Vielzahl von Knoten und Kanten enthalten, die Kurven darstellen. Daher bietet der HierarchyMapGraph die Möglichkeit, solche uninteressanten Kanten und Knoten in einzelne Kanten zusammen zu fassen,

die von Beginn der Kurve bis zu ihrem Ende führen. Die kontrahierten Knoten werden dabei anhand ihrer IDs in der HierarchyMapEdge gespeichert. Um den Graphen weiter zu komprimieren, steht eine Methode bereit, die Zykel und parallele Kanten entfernen kann. Da der schnellste Weg gesucht werden soll in dem Programm, ist dadurch kein Informationsverlust gegeben. Zuletzt bietet der HierarchyMapGraph auch noch die Berechnung des eigenen 2-Cores, wie in den Grundlagen beschrieben, an. Dadurch kann der HierarchyMapGraph stark auf die nötigsten Informationen reduziert werden.

Das letzte Werkzeug in der Graphenbibliothek ist die Path-Klasse. Wie der Name vermuten lässt, kann damit ein Pfad beschrieben werden. Damit besteht ein einfach zu handhabendes Objekt, womit komplexe Pfade gespeichert und rekonstruiert werden können.

4.4 Algorithmen

Dieses Modul befasst sich mit den notwendigen Algorithmen zur Routenberechnung und des Pre-Processings. Die Klasse Dijkstra bietet dabei zwei verschiedene Funktionalitäten. Die wichtigste ist dabei die Berechnung der Nachbarschaften. Wie in den Grundlagen erläutert, ist dies ein wichtiges Element bei dem Bestimmen der Hierarchieebenen. Da sich die Nachbarschaftsberechnung sehr stark an dem Dijkstra-Algorithmus orientiert, befindet sich diese Methode in dieser Klasse. Die andere Funktionalität, die angeboten wird, ist ein bidirektionaler Dijkstra-Algorithmus auf wahlweise einem HierarchyMapGraphen oder einem MapGraphen. Diese soll allerdings nicht produktiv eingesetzt werden, sondern dient in diesem Projekt der Verifikation unserer Ergebnisse, da dieser Algorithmus sicher richtige Ergebnisse liefert. Wie in den meisten algorithmischen Klassen in diesem Modul sind die Methoden alle statisch, sodass keine Objekte erstellt werden müssen, sondern alle nötigen Funktionen sofort zur Verfügung stehen.

Innerhalb der Algorithmen gibt es die Hilfsklassen PrQueue und Vertex. Letztere ist im Wesentlichen nichts anderes als ein kleiner Datenspeicher für temporäre Informationen mit Bezug auf einen Knoten. Dies sind beispielsweise aktuelle Entfernungen zu bestimmten Startknoten bei der Nachbarschaftsberechnung. Dementsprechend hat diese Klasse keine wirklichen Methoden, außer einer Reihe von Gettern und Settern. Die PrQueue ist hingegen eine Prioritätswarteschlange. Sie wird ebenfalls bei der Nachbarschaftsberechnung genutzt, um alle erreichbaren Knoten mit ihrer Entfernung zu verwalten und regelmäßig das Element mit der kleinsten Entfernung zu extrahieren. Dementsprechend bietet ein PrQueue-Objekt die Möglichkeit die Größe der Schlange abzufragen, sowie das kleinste Element zu holen oder neue Elemente einzufügen.

Die Query stellt mit ihren statischen Methoden die Möglichkeit zur Verfügung eine Route zu berechnen und dabei die Highway Hierarchies effektiv zu nutzen. Die Query erlaubt es auch, mit Knoten als Start und Ziel zu arbeiten, die selbst nicht im Hierarchigraphen eigenständige Knoten sind, sondern auf kontrahierten Kanten liegen.

Das Herzstück der Algorithmen ist die Klasse HHierarchyMT. Diese Klasse berechnet auf einem übergebenen HierarchyMapGraphen alle Hierarchieebenen. Dies tut sie wie in den Grundlagen beschrieben über alle Ebenen. Bei Aufruf der entsprechenden Methode entscheidet die Klasse selbst, ob Multithreading sinnvoll ist. Ist der Graph großgenug, wird anhand der aktuellen Prozessorkerne die Menge der Knoten aufgeteilt und dann parallel in mehreren Threads die Hierarchien extrahiert. Beim Pre-Processing wird hier die meiste Zeit hier gearbeitet, da mehrfach alle Knoten des Graphen durchlaufen werden und mit diesen gearbeitet wird. Zusätzlich bietet diese Klasse auch Methoden an, um explizit nur eine Ebene zu berechnen oder auf keinen Fall Multithreading zu nutzen.

4.5 GUI

In diesem Modul soll eine grafische Oberfläche zur Verfügung gestellt werden und die Kommunikation mit dem Nutzer ermöglichen. Die zentrale Klasse dieses Moduls ist daher die MainFrame. Sie beschreibt in erster Linie das Fenster was angezeigt werden soll. Beim Konstruieren werden automatisch die Panels MapPanel und SidePanel angelegt und positioniert. Darüber hinaus wird auch der MenuHandler so wie der MouseEventListener angefügt. Das SidePanel verwaltet dabei kleinere Panels. Dies sind das InfoPanel, welches Informationen über die aktuelle Rote anzeigt, das MapControlPanel, das eine Reihe von Buttons anbietet um durch die Anzeige zu navigieren, und ein ConsolePanel, in das bei Bedarf Konsolenausgaben eingeleitet werden können. Der MenuHandler baut in dem Fenster das Menü auf und verweist bei Aktionen auf die entsprechenden Methoden oder öffnet Dialoge. Der MouseEventListener reagiert derweil auf Aktionen mit der Maus zum Zwecke der Navigation.

Das MapPanel stellt eine Vielzahl von Methoden zur Anzeige der Straßen und Routen bereit und verwaltet den Puffer, auf dem das konkrete Zeichnen stattfindet. Dieses konkrete Zeichnen wird dabei vom RepaintWorker übernommen. Dieser wird dabei in einen eigenen Thread ausgelagert um zum Einen sicher zu stellen, dass das Programm auch während des Zeichnens bedienbar bleibt und zum Anderen die Möglichkeit zu bieten bei einer erneuten Eingabe des Nutzers das aktuelle Zeichnen ab zu brechen und den neuesten Auftrag zu bearbeiten. Zunächst wird in einem gepuffertem Bild eine Menge

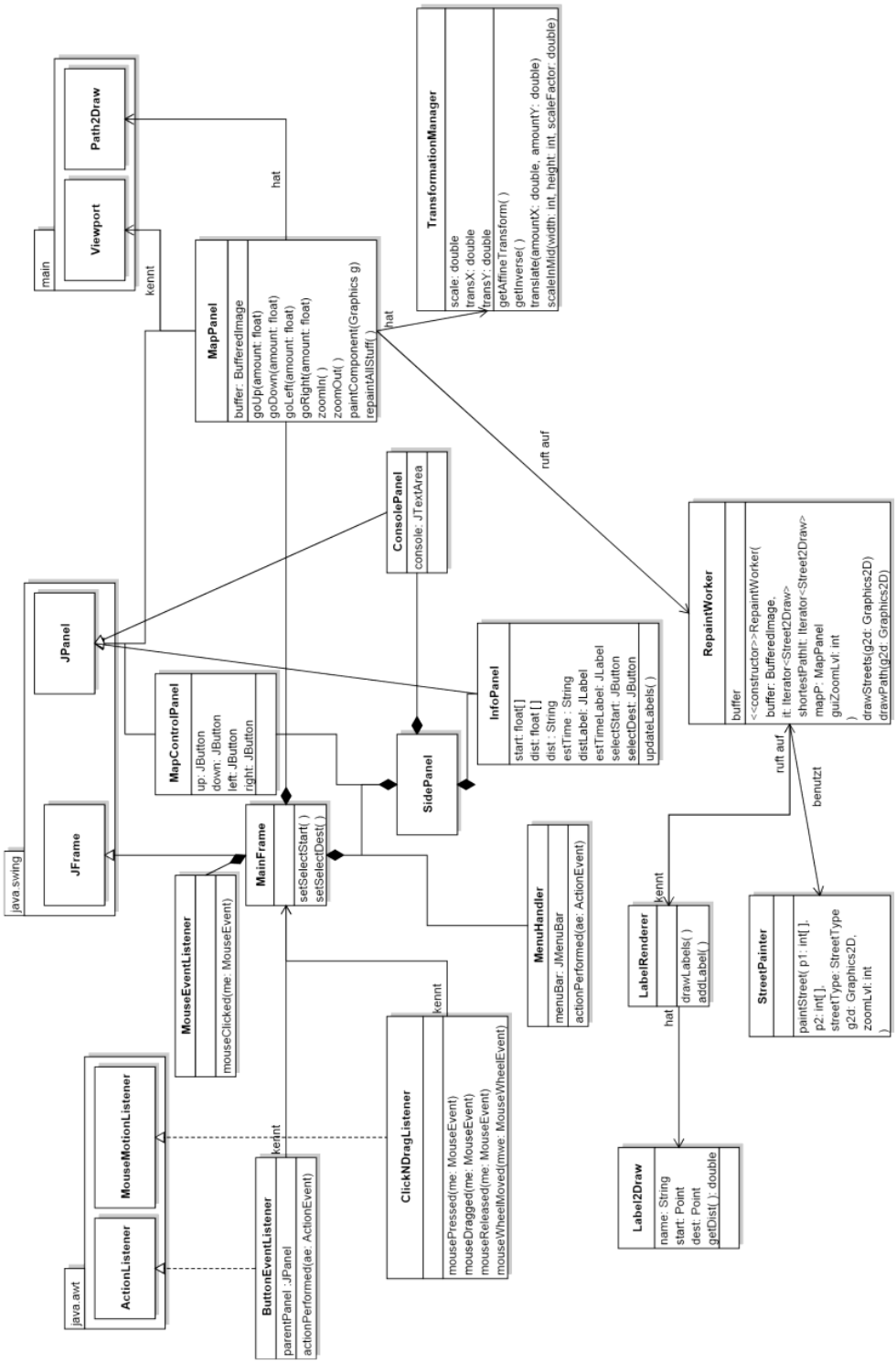

Abbildung 6: Ein Klassendiagramm der GUI

von Straßen gezeichnet, was von dem STREETPAINTER übernommen wird, der insbesondere die Art der Darstellung der Straßen verwaltet. Sind alle Straßen gezeichnet sollen darauf Labels gesetzt werden. Dazu gibt es den LABELRENDERER. Dieser überprüft, ob und wo auf die Straße ein Label passt und zeichnet es dort hin.

Weiterhin gibt es den TRANSFORMATIONMANAGER mit dem das MapPanel Transformationen und Skalierungen vornehmen kann, sowie eine Inverse bereitgestellt bekommt.

4.6 Main

Das Mainpaket besteht im Wesentlichen aus folgenden Klassen: Einer CONSTANTS Klasse, in der alle in unserem Projekt auftretenden Konstanten zentral gesammelt sind, und einer LOGGER Klasse, die einen an unser Projekt angepassten Logger implementiert. Darüber hinaus einer STREET2DRAW Klasse, welche alle Informationen für einen zu zeichnenden Straßenabschnitts kapselt und einer PATH2DRAW Klasse, welche die Daten für einen kürzestens Weg beinhaltet. Zentrale Klassen sind aber die VIEWPORT und MAINPREPROC, welche daher auch im Klassendiagramm am Anfang dieses Kapitels erscheinen. Die MainPreProc besteht letztlich nur aus einer main-Methode, welche das Preprocessing des Imports und der Algorithmen startet und danach über den OSMImporter diese Daten speichert. Sie kann unabhängig von der graphischen Oberfläche per Konsole aufgerufen werden, wird aber auch verwendet, wenn das Preprocessing per GUI gestartet wird.

Die Rolle des Controller in unserer Model-View-Controller artigen Architektur übernimmt dabei hauptsächlich die Viewport Klasse, über welche die graphische Oberfläche die Daten für das Zeichnen von Karten und Berechnen kürzester Wege abfragt. Wie dies genau abläuft, lässt sich an zwei Szenarien erkennen, die im Sequenzdiagramm (Abbildung 7) dargestellt werden. Im ersten Szenario ändert sich der momentane Kartenausschnitt dadurch, dass der User zoomt oder scrollt. Dies teilt die UI der Viewport Klasse mit dem Aufrufe der update-Methode mit und sie erhält dann per Callback die Daten für den neu zu zeichnen Ausschnitt sowie gegebenfalls zu zeichnende kürzeste Pfade. Die Viewport Klasse erhält diese Daten wiederum dadurch, dass sie die entsprechende getTiles Methode des OSMImporters aufruft und kürzeste Pfade intern speichert. Im zweiten Szenario soll ein kürzester Weg berechnet werden, was der User durch Auswählen eines Starts und Ziels veranlasst. Der Viewport verwaltet intern einen HierarchyMapGraph auf dem die Berechnung des kürzesten Weges mittels einer Methode aus dem Algorithmenpaket gestartet wird. Mit dem zurückgegeben Path Objekt wird intern ein Path2Draw Objekt im Viewport gespeichert, sodass beim nächstens Aufruf

der update Methode der neue zu zeichnende Pfad zurückgegeben wird.

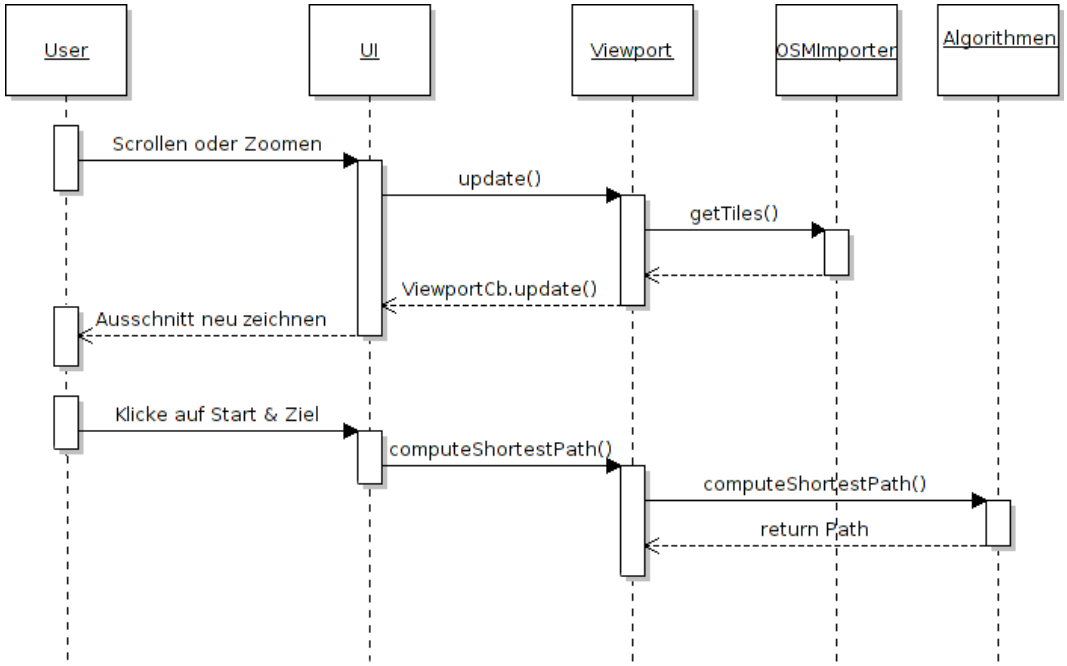

Abbildung 7: Sequenzdiagramm für Abläufe im Programm

A Bekannte Fragen

A.1 Benutzung

- `Welche Mindestanforderungen hat das Programm?`
  Die Berechnung von Routen auf aufbereiteten Daten und das Zeichnen setzen einen Computer mit einer Java-Laufzeitumgebung voraus. Prinzipiell sollte auf all diesen Rechnern dieses Programm laufen können, lediglich die Zeichnungsgeschwinigkeit der Straßen kann auf schwachen Computern langsam sein.

  Wir empfehlen daher einen Computer, dessen Eckdaten einen RAM-Speicher von 512-1024MB und eine Pentium 4-CPU oder vergleichbar nicht unterschreitet.

  Beim Pre-Processen sollte der Rechner hingegen möglichst gut bestückt sein. Wir empfehlen hier mindestens 2GB RAM-Speicher und eine Multi-Core-CPU.

- `Wie viel Zeit muss ich einplanen, um meinen Urlaubsort aufzubereiten?`
  Die benötigte Zeit hängt stark von der gewählten Karte und dem benutzten Computer ab. Im nächsten Anhang befinden sich einige Zahlen und Statistiken, die Helfen dürften, eine grobe Abschätzung der benötigten Zeit vorzunehmen. Da allerdings Faktoren wie Kartenupdates und Prgramme im Hintergrund auf dem Computer die Laufzeit noch weiter erhöhen können, empfehlen wir, diese Aufgabe rechtzeitig zu starten.

- `Auf wie vielen und welchen Gebieten ist das Programm derzeit getestet?`
  Natürlich haben wir das Programm bereits mit verschiedenen Karten und Computern getestet. Die Größe der Karten variierte dabei stark. Zwischen Karten wie der Aachener Innenstadt (weniger als 100 Straßen) und der ca. $35.000 km^2$ großen Karte von NRW haben wir ein gutes Dutzend verschieden dicht besiedelter Gebiete als Karten zum Testen genutzt. Dabei wurden auch andere europäische Länder, sowie einige Orte auf anderen Kontinenten genau betrachtet.

- `Der Routenplaner richtet sich offensichtlich an Autofahrer. Kann ich damit auch Routen für mein Rad oder per Pedes berechnen?`
  Aufgrund der Filterung der Straßen ist derzeit nur eine Berechung für

Autofahrer möglich. Eine Anpassung oder alternative Variante des Programmes wäre denkbar, ist aber derzeit noch nicht in Planung.

- `Kann ich auch eine Route bis zu einer bestimmten Hausnummer berechnen lassen?`
  Durch klicken auf eine Straße wird die Route genau bis zu diesem Punkt berechnet. Eine konkrete Suchfunktion nach speziellen hausnummern existiert allerdings nicht.

- `Brauche ich eine Internetverbindung, wenn ich das Programm benutze?`
  Eine Internetverbndung ist zur Benutzung des Programmes nicht nötig.

A.2 Code und Portierung

- `Ist dieses Projekt auf mobile Plattformen wie Smartphones portierbar?`
  Dieses Projekt wurde in erster Linie auf und für Desktops und Laptops entwickelt. Ein Portierung des Codes ist prinzipiell möglich. Da es sich um ein Java-Prgramm handelt muss das Zielgerät eine solche Laufzeitumgebung anbieten. Eine Anpassung des User Interface an die speziellen Gegebenheiten eines Smartphones ist mit Arbeit verbunden, allerdings möglich. Das Pre-Processing sollte allerdings dennoch auf einem leistungstarken Computer durchgeführt werden. Je nach Leistung des Gerätes kann es allerdings zu verlängerten Laufzeiten beim Zeichnen und Berechnen kommen.

- `Kann es als Internetanwendung laufen?`
  Auch hier müsste eine Portierung stattfinden. Das User Interface ist derzeit an Fenster gebunden. Eine Umstellung auf JavaServer Faces wäre mit etwas Arbeit verbunden, aber durchaus machbar.

A.3 Daten

- `Das Straßennetz meiner Stadt sieht etwas dünn aus. Woran kann das liegen?`
  Unser Programm greift auf die Daten von OpenStreetMap zurück. Jenes Projekt ist allerdings auf Mithelfer angewiesen, die freiwillig Straßen und Orte dokumentieren. Dadurch kann es natürlich vorkommen, dass einige Straßen, die es eigentlich gibt, nicht (korrekt) dokumentiert sind. Weiterhin werden in unserer Ausgabe weder Fusswege

noch zufahrtsbeschränkte Straßen (Anlieger frei, Straßen für Lieferverkehr, etc.) angezeigt.

- `Der Routenplaner sagt mir, ich soll in eine Straße einfahren, die eine Einbahnstraße oder gesperrt ist. Darf ich das dann trotzdem?`
  Natürlich tut es uns Leid, sollte es zu einer solchen Situation kommen, allerdings befreit dieses Programm niemals von den Verkehrsregeln und Gesetzen. Wir kontrollieren nicht die Daten der OpenStreetMap, daher kann es durchaus zu Diskrepanzen zwischen Daten und Realität kommen. Diese können am besten behoben werden, wenn die Leute, denen solche Fehler auffallen sich an OpenStreetMap wenden und dort zur Aufklärung beitragen. Wir übernehmen weder die Garantie für die Korrektheit der Daten, noch die Verantwortung für eventuelles Fehlverhalten der Nutzer unseres Programmes.

- `Was muss ich tun, damit in 2-3 Jahren, der Routenplaner auf dem aktuellsten Stand ist?`
  Um zu gewährleisten, dass die Karten auf dem neuesten Stand sind, sollten vor einer Reise möglichst aktuelle Karten heruntergeladen werden und aufbereitet werden.

- `Wie sieht es mit der Genauigkeit der OSM-Daten aus im Vergleich zu Google Maps und co.?`
  OpenStreetMap lebt von der Mitarbeit vieler Menschen. Derzeit ist die Genauigkeit noch nicht ganz so gut wie bei anderen Anbietern wie TomTom, aber sie wird stetig besser.[2]

A.4 Das Projekt

- `In welcher Zeit und welchem Zusammenhang ist dieses Projekt entstanden?`
  Dieses Projekt ist begleitend zum Studium an der RWTH Aachen University von einigen Studenten in circa 4 Monaten entstanden. Dies geschah in Rahmen eines Praktikums an dem Lehrstuhl für Theoretical Computer Science der RWTH Aachen.

- `Gab es Verluste bei der Arbeit?`
  Leider Bedauern wir den Verlust eines Netzteils im Rahmen von Testläufen, sowie einer o-Taste neben sehr viel Koffein.

- `Wird das Projekt weitergeführt und sind in Zukunft weitere Features zu erwarten?`

Wir haben beschlossen, dieses Projekt als OpenSource zu veröffentlichen, sodass jeder Interessierte mitarbeiten kann. Von uns werden sicherlich auch einige daran weiter mit arbeiten. Einen konkreten Plan mit Meilensteinen und Release-Terminen gibt es allerdings nicht.

B Zahlen und Statistik

Karte (Größe)	CPU	max. Speicher Stack (Xss)/Heap (Xmx)	Zeit Tiling + Hierarchien
Aachen (9 MB)	4x2.5GHz	Default/Default	2s
Münster (460 MB)	2x2.7GHz	100 MB/1 GB	7m
NRW (3.5 GB)	2x2.7GHz	200 MB/5 GB	1:30h
Bremen (80 MB)	4x2.5GHz	20 MB/Default	1m
NRW (3.5 GB)	4x2.5GHz	200 MB/6 GB	1:20h
Köln (800MB)	2x2.7GHz	200MB/4GB	11m
Belgien (1.7GB)	4x2.5GHz	200MB/6GB	1h

C Ausblick

Dies ist eine Sammlung von uns bekannten Wünschen und Möglichkeiten, wie das Programm noch sinnvoll erweitert oder angepasst werden könnte.

- Eine erweiterte/detaillierte Ausgabe der Straßen auf einer Route. Im Optimalfall natürlich mit Hinweisen zum rechts- oder linksabbiegen.

  Eine solche Umsetzung würde bedeuten, dass für alle Übergänge von Straßen entschieden werden muss, in welche Richtung dies geschieht oder ob lediglich die Straße umbenannt wird. Auch könnten mehrspurige Straßen durchaus komplexe Sonderfälle bedeuten.

- Der Import von Bereichsbeschreibungen, sodass beim Rauszoomen über einer Gruppe von Straßen beispielsweise der Name der Stadt geschrieben stünde

  In den OSM-Daten sind einerseits Bereiche definiert, die auch Städte umschließen, als auch andererseits eine Reihe von Positionierungshilfen. Diese konkret einzuladen und bei Bedarf anzuzeigen ist kompliziert in Bezug auf die Heuristik, wann was angezeigt werden sollte.

- Eine Suchfunktion. Eine Eingabemaske könnte ermöglichen gezielt nach Straßen, Orten oder wichtigen Punkten zu suchen.

  Suchfunktionen sind sehr kompliziert. Insbesondere eine solch gute Suche wie bei Google Maps dürfte kaum erreicht werden. Allerdings könnte eine Suche intern alle Kanten auf einen Namen hin überprüfen oder die Knoten nach deren UIDs.

- Auswählbare Filter für zusätzliche Informationen.

  Durch Anpassen der Filter könnten leicht Gebäude, sowie Fusswege angezeigt werden. Es wäre dabei insbesondere wichtig, dafür zu sorgen, dass diese Daten nicht für die Routenberechnung herangezogen würden. Das Fahren auf einem Dachsims sollte nämlich vermieden werden.

- Routenplaner füer Radler und Fussgänger

  Im Zusammenhang mit Filteroptionen könnten Routen für Radler und Fussgänger berechnet werden. Die Algorithmen, Graphendarstellung und visuelle Ausgabe der Straßen und Wege müsste nicht wirklich angepasst werden. Die Arbeit wäre im Wesentlichen in dem Importmodul zu tun. Mit relativ wenig Arbeit ließen sich Programme für Radfahrer und Fussgänger entwickeln. Diese wiederum als ein Programm zusammanzuführen, sodass auf Knopfdruck für eine andere Zielgruppe gerechnet würde wäre etwas aufwändiger.

- Optimierungsarbeiten

  Insbesondere bei großen Karten dauert das Aufbereiten der Daten recht lange. Bereits kleine Optimierungen können sich daher bei großen Karten bemerkbar machen. Vorstellbar wäre beispielsweise eine intelligente Speicherverwaltung, sodass seltener Speicherbereiche aus dem langsamen RAM nachgeladen werden müssen. Weiterhin kann es beim längeren Arbeiten mit sehr großen Karten vorkommen, dass zu viele Dateien von dem Programm geöffnet sind und daher ein Fehler auftritt.

- Kleine Anpassungen der GUI. Beispielsweise ein „clear Route“–Button

- Intelligentes Pre-Processing bei teilweise bekannten Daten

  Hat ein Nutzer bereits aufbereitete Daten von einigen Städten in NRW, könnte die Laufzeit beim Pre-Processing von NRW deutlich reduziert werden, wenn keine redundanten Vorgänge gemacht würden. Dies ist sehr kompliziert und sehr fehleranfällig. Insbesondere bei Karte von verschiedenen Zeiten kann es zu Konflikten kommen. Dies wäre eine große Aufgabe, die allerdings in beschriebenen Fällen durchaus Zeit sparen kann für den Nutzer.

- Favoriten-Speicherung von Adressen

- Optimale Durchschnittsgeschwindigkeiten ermitteln und Strafen für Abbiegen auf andere Straßen.

D Designentscheidungen

Dies ist eine Sammlung von Design- und Codeentscheidungen, die bei uns für Diskussionen gesorgt haben

- Auswahl der Straßen für den Routenplaner

  Straße ist nicht gleich Straße. In den OSM-Daten sind viele verschiedene Straßentypen enthalten. Es fängt bei Autobahnen an, geht über normale Straßen und Wege zu Fuss- und Radwegen, sowie Trampelpfaden und Pferdestrecken. Daher haben wir uns entschieden, alle Straßen zu nutzen, auf denen Autos fahren dürfen, solange dies nicht an weitere Bedingungen geknüpft ist, wie Lieferungsverkehr oder Anliegerstraßen.

- Gerichteter oder ungerichteter Graph

  Graphen sind in der Regel entweder gerichtet oder ungerichtet. Da es eine Vielzahl von Straßen gibt, die nicht in beide Richtungen befahren werden können, haben wir uns dazu entschieden, einen gerichteten Graphen zu nutzen. Die Alternative wäre gewesen, einige Kanten in einem ungerichteten Graphen mit speziellen Flags aus zu statten. Dadurch wäre es ein Pseudo-ungerichteter Graph, bei dem nicht so viele Kanten angelegt werden müssten. Wir waren aber der Meinung, dass ein gerichteter Graph wesentlich effiziernt für unser Vorhaben ist.

- Datentypen für Zeiten und Entfernungen

  Nach vielen Problemen und insbesondere einer Vielzahl von Rundungsfehlern haben wir uns entschieden die Entfernungen in Integer zu speichern, die dann jeweils für einen dm stehen. Bei der Zeit wurde aufgrund der Berechnung ein Long gewählt, wobei dort eine Einheit $1/10000s$ entspricht. Dadurch konnten wir viele Probleme vermeiden.

- Verschiedene Graphentypen

  In unserer Umsetzung gibt es nun einen MapGraphen und einen HierarchyMapGraphen. Diese haben verschiedene Aufgaben. Es wäre möglich, alle Funktionen in einem einzigen Graphen zu vereinen, allerdings hätte das einige Nachteile. Die Trennung erlaubt uns im Rahmen des MapGraphen ein Tiling, sodass die Graphen mit vielen Informationen nur bei Bedarf geladen werden. Gleichzeitig kann der deutlich kleinere HierarchyMapGraph als ganzes im Speicher gehalten werden und bei dem Berechnen der Route muss daher nicht auf die langsame Festplatte zurückgegriffen werden. Dadurch ist unsere Variante eine sinnvolle Aufteilung von Aufgaben.

Literatur

[1] Geofabrik GmbH Karlsruhe. http://www.geofabrik.com, January 2012.

[2] Pascal Neis, Dennis Zielstra, and Alexander Zipf. The street network evolution of crowdsourced maps: Openstreetmap in germany 2007–2011. *Future Internet*, 4(1):1–21, 2011.

[3] OpenStreetMap Foundation. http://www.openstreetmap.org, January 2012.

[4] Prof. Dr. rer. nat. Peter Sanders and Dipl.-Inform. Johannes Singler. Kürzeste Wege. In *Taschenbuch der Algorithmen*, chapter 34, pages 345–352. Springer Verlag, 2008.

[5] Dominik Schultes. Fast and exact shortest path queries using highway hierarchies, 2005.

[6] Thorsten Vogel and Sergej Müller. Core-graphen. 2006.

[7] Wikipedia. http://de.wikipedia.org/wiki/graphentheorie, January 2012.

[8] Wikipedia. http://de.wikipedia.org/wiki/xml-prozessor, January 2012.