HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

PlazaRoute

Fussgänger-Routing über offene Flächen im urbanen Raum

Studienarbeit

HSR Hochschule für Technik Rapperswil
Institut für Software

Herbstsemester 2017

Autoren: Jonas Matter, Robin Suter
Betreuer: Prof. Stefan Keller

Abstract

Die heute gängigen Routing-Engines sind für den motorisierten Individualverkehr optimiert. Bei diesem halten sich die Verkehrsteilnehmer an vorgegebene Regeln und Strecken. Fussgänger allerdings optimieren intuitiv ihre Route, so wählen sie zum Beispiel über eine öffentliche Fläche den möglichst kürzesten Weg zum Ziel. Bestehende Routing-Engines navigieren entlang der Kante des Platzes, anstatt ihn direkt zu überqueren.

Es werden bestehende Algorithmen, Visibility-Graph und SpiderWeb-Graph, zur Traversierung von offenen Fussgänger-Flächen evaluiert, analysiert und optimiert. Mit einer Vorverarbeitung von OpenStreetMap-Daten wird gezeigt, wie eine Routing-Engine ein natürliches Fussgänger-Routing über offene Flächen unterstützen kann.

Als praktische Umsetzung wird in Python mit Hilfe des Fahrplan-Services von search.ch ein Service für ein multimodales Routing mit öffentlichen Verkehrsmitteln erarbeitet. Mit einem eigens entwickelten Plugin für das Geoinformationssystem QGIS können die optimierten Routen visualisiert werden.

Durch die Vorverarbeitung von OpenStreetMap-Daten ist die Grundlage für ein natürliches Fussgänger-Routing geschaffen, auf welcher gängige Routing-Engines aufsetzen können. Das hat die Konsequenz, dass mehr Daten zu verarbeiten sind, jedoch müssen die bestehenden Routing-Engines dadurch nicht erweitert werden. In einem visuellen Vergleich zeigen die eingesetzen Algorithmen deutlich bessere Ergebnisse als die bestehenden Implementationen in den Routing-Engines, während die Datenmenge für die Schweiz um weniger als 0.5% steigt.

Mit dem entwickelten Backend und einem zugehörigen QGIS-Plugin können Benutzer mit einem beliebigen Start- und Endpunkt ein ÖV-Routing mit optimiertem Verhalten bei Fussgänger-Routen durchführen.

Abstract

Today's most notable open source routing engines are optimized for motorized traffic but show deficiencies in pedestrian routing. With open spaces, routers usually navigate along the edges of the available area, whereas pedestrians would naturally take shortcuts through the open space while avoiding obstacles. Past research has shown multiple approaches to this problem. This project compares a few of the approaches used to address this problem.

Utilizing publicly available geographic data from OpenStreetMap and the help of existing algorithms, an implementation is proposed to optimize geographic data for existing routing engines that enhances the capacity to produce pedestrian routing approximating natural behavior. The optimization is refined using shortest-path algorithms to minimize additional data volume.

Furthermore, the optimized pedestrian routing is used in combination with existing services for public transport routing in Switzerland, providing a practical application addressing multimodal transportation. With a newly-developed plugin for QGIS, users are able to visualize the optimized routes.

The implementation of two of the different approaches toward data processing, visibility graph and SpiderWeb graph, demonstrates a clear improvement of routes for pedestrian navigation compared to existing methodology utilized in current routing engines. In the future, this implementation could serve as a reference to integrate these approaches directly into routing engines to enhance the usability of these programs for pedestrians and provide an optimized user experience regardless of the method of transit.

Management Summary

Ausgangslage

Die heute gängigen Routing-Engines können effizient über Knoten und Kanten routen und wurden konkret für den motorisierten Individualverkehr optimiert. Für den nicht-motorisierten Individualverkehr (Fussgänger, Rollstuhlfahrer, Radfahrer, etc.) sieht die Lage drastisch anders aus. Der motorisierte Individualverkehr hält sich durchwegs an vorgegebene Regeln und Strecken. Ein Fussgänger hingegen verhält sich grundlegend auf eine andere Art und Weise. Wo ein Autofahrer an die Streckenführung gebunden ist, optimiert ein Fussgänger intuitiv. So überquert er einen Platz auf direktem Weg und läuft nicht an der Strasse um den Platz herum, wie es ein Autofahrer tun muss. Dies ist in den Routing-Engines jedoch Status quo. In Abbildung 1 ist sichtbar, wie über den Fischmarktplatz in Rapperswil-Jona, Schweiz geroutet wird. Es liegt in der Natur des Menschen, den Platz ressourcenschonend zu überqueren, was momentan nicht der Fall ist.

Die Problematik beschränkt sich nicht nur auf Flächen wie Plätze und Parks, sondern umfasst auch Strassen und weitere Arten von offenen Flächen wie Berge und Strände.

Durch den Einzug der mobilen Geräte ist es üblich, dass von einem beliebigen Startpunkt aus das Routing gestartet wird. Befindet man sich in diesem Fall gerade auf einer Fläche, wird aktuell eine Mittelsenkrechte zur Kante gezogen, auf welcher das Routing fortgesetzt wird.

Ein Fussgänger ist in vielen Fällen auf die öffentlichen Verkehrsmittel angewiesen, was in einem multimodalen Routing resultiert. Das genaue Ansteuern einer bestimmten ÖV-Haltestelle in die richtige Fahrtrichtung gestaltet sich schwierig, da für eine ÖV-Haltestelle die gleiche Koordinate für beide Fahrtrichtungen von verschiedenen Services retourniert wird. Für ein Fussgänger-Routing ist dies suboptimal, da sich diese Koordinate in manchen Fällen direkt auf einer Haupstrasse befinden kann und ÖV-Haltestellen nicht immer auf der gegenüberliegenden Strassenseite liegen müssen.

Ziele, Vorgehen und Technologien

Im Kontext der Arbeit *PlazaRoute* wird die Problematik des Fussgänger-Routings über offene Flächen im urbanen Raum aufgegriffen. Kurz zusammengefasst heisst das, dass eine Routing-Engine ein natürliches Fussgänger-Routing generieren kann.

In Kombination mit dem ÖV-Routing soll ein multimodales Routing möglich sein, welches von einem beliebigen Startpunkt aus an eine Zieldestination routet und dabei ÖV-Haltestellen genau ansteuert.

Um diese Anforderungen erfüllen zu können, werden Algorithmen zum Traversieren von offenen Fussgänger-Flächen evaluiert, analysiert, getestet und optimiert. Nach Abschluss dieser Tätigkeit wird es möglich sein, eine OpenStreetMap (OSM) Datei so aufbereiten zu können, dass Routing-Engines dem

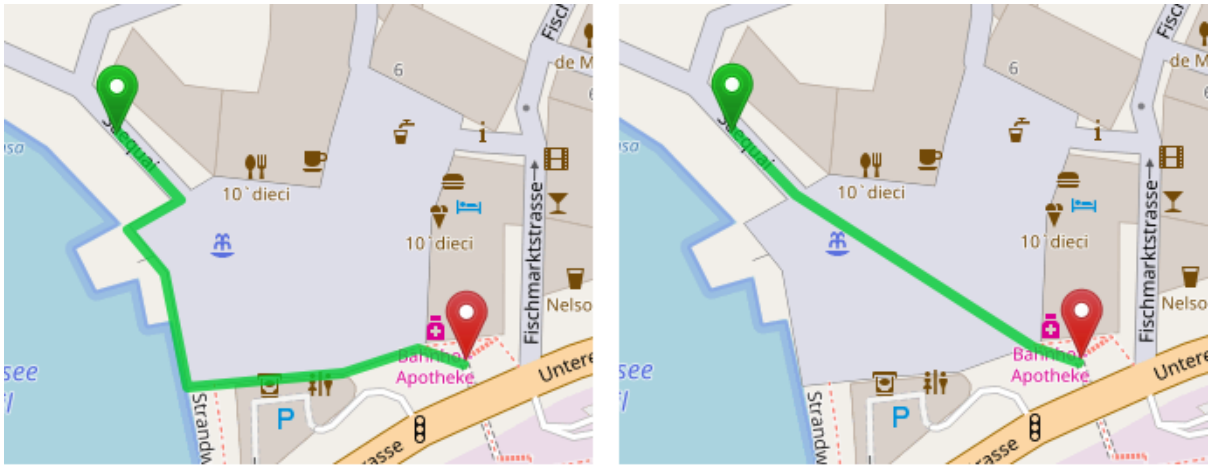

Abbildung 1: Vergleich aktueller Stand einer Routing-Engine (links) mit dem Ergebnis der Vorverarbeitung (rechts); Fischmarktplatz in Rapperswil-Jona, Schweiz; Screenshot aufgenommen am 17.12.17

Fussgänger ein natürliches Routing bieten können. Die durch die Algorithmen erzeugten Graphen werden mit Shortest-Path-Algorithmen vereinfacht, um die zusätzliche Datenmenge zu minimieren.

Um die Optimierung an einem praktischen Beispiel zeigen zu können, wird parallel in Kombination mit dem Service von search.ch ein Backend für ein multimodales Routing entwickelt. Dies wird durch eine visuelle Darstellung in einem QGIS-Plugin abgerundet, welche den Mehrwert für einen Fussgänger konkret aufzeigen wird.

Die zugrundeliegende Technologie ist Python.

Ergebnisse

Durch die Umsetzung einer Vorverarbeitung mit einem Visibility-Graph oder SpiderWeb-Graph in Kombination mit einem Shortest-Path-Algorithmus (Dijkstra oder A*) ist die Grundlage für ein natürliches Fussgänger-Routing geschaffen, auf welcher gängige Routing-Engines und andere Interessenten operieren oder auf der Implementation aufsetzen können.

Fussgänger können zu jedem Zeitpunkt und von jeder Position aus ein Routing durchführen, welches ihrem Verhalten entspricht.

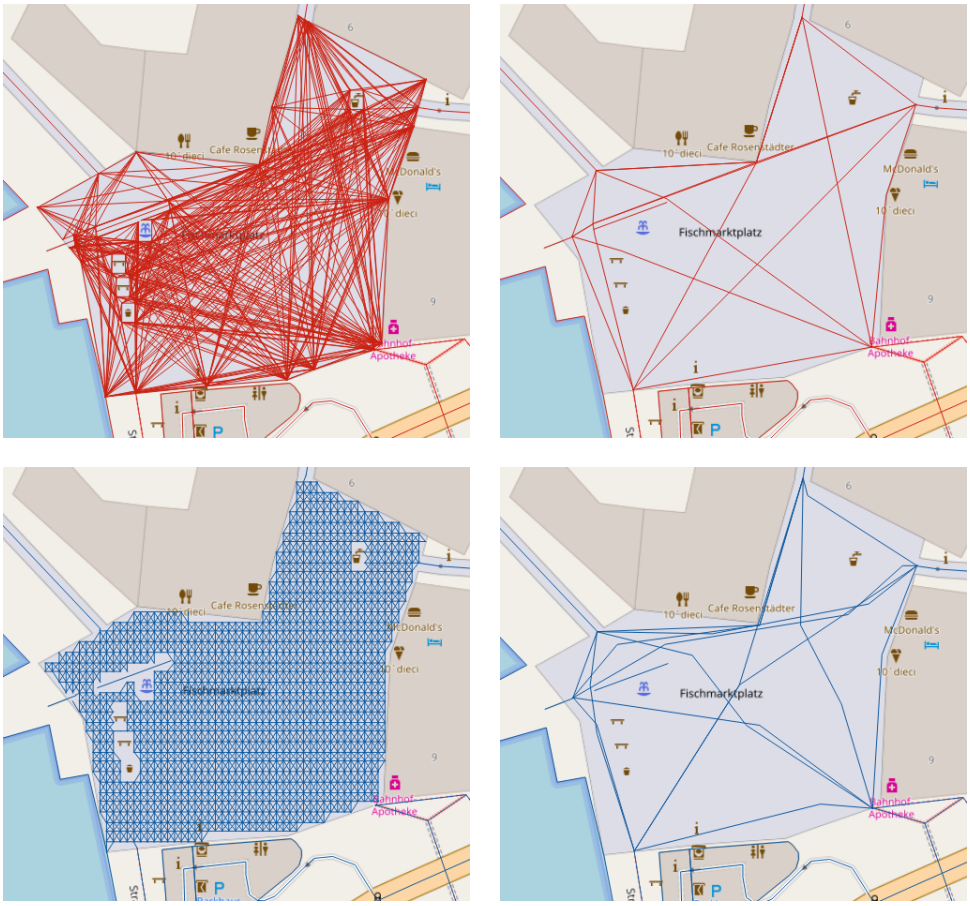

Abbildung 2: Vergleich der Algorithmen Visibility-Graph (oben) und SpiderWeb-Graph (unten), vor und nach der Optimierung durch Shortest-Path-Algorithmen

Um bequem und auf direktem Weg nach Hause zu gelangen, ist das transportmittelübergreifende Routing als Service verfügbar und kann in einem QGIS-Plugin visualisiert werden.

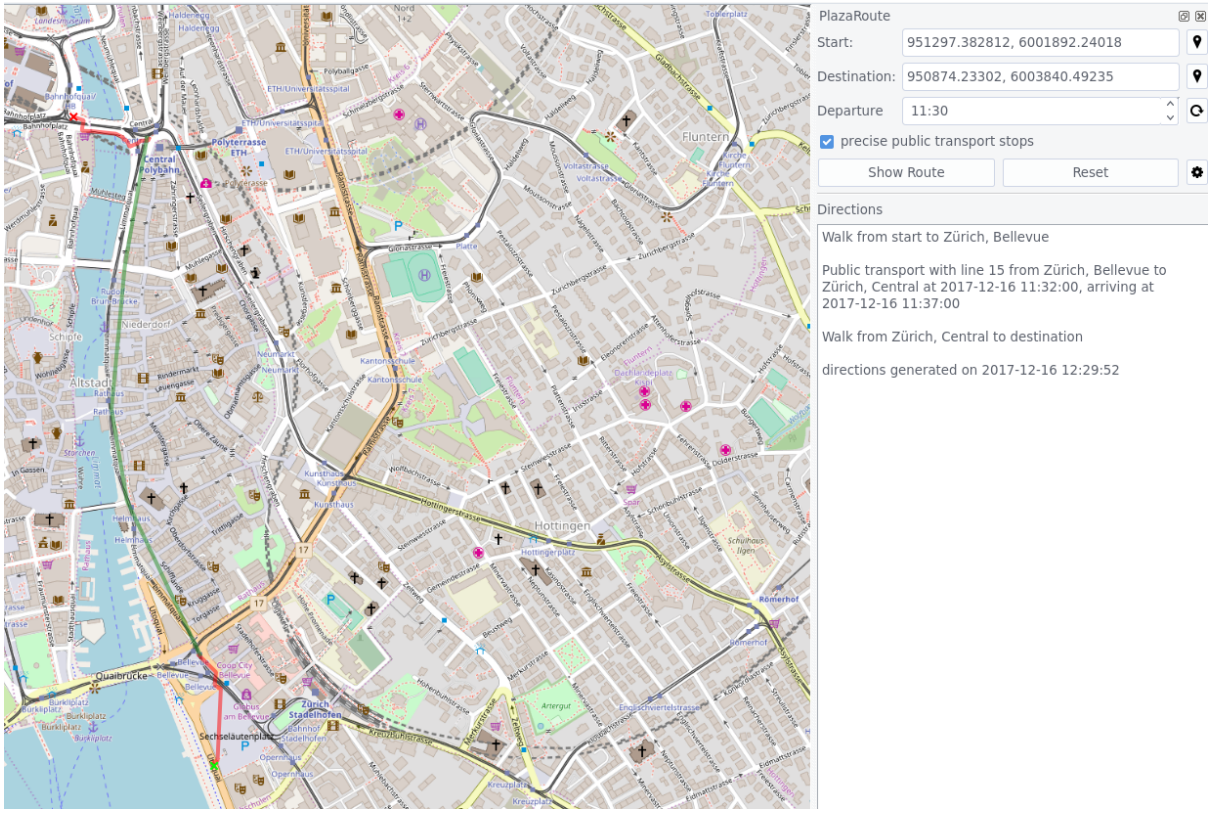

Abbildung 3: Berechnete Route in QGIS-Plugin PlazaRoute

Ausblick

Die bestehende Lösung bietet weiteren Raum für Optimierungen. So können einige Flächen, welche für Fussgänger begehbar sind, durch fehlende Einstiegspunkte momentan nicht verarbeitet werden.

Die Arbeit hat sich bewusst auf den urbanen Raum beschränkt, da die Mehrheit der Routings in diesem Umfeld durchgeführt werden. Diese Grenze soll erweitert werden und Flächen ohne konkrete Begrenzungen wie Berge und Strände einbeziehen.

Die Vorverarbeitungen der Flächen stösst im Bezug auf die Performanz an Grenzen. Ein Vergleich mit einer Lösung in PostGIS oder C++ ist denkbar.

Inhaltsverzeichnis

I Technischer Bericht

1 Einführung

Diese Arbeit befasst sich grundlegend mit dem Fussgänger-Routing über offene Flächen im urbanen Raum und das Durchführen eines multimodalen Routings, welches das Fussgänger- und ÖV-Routing kombiniert. Im folgenden ist die Problemstellung erläutert und es wird eine Abgrenzung gemacht, was Teil dieser Arbeit ist.

1.1 Grundlagen und Begriffe

Im Weiteren werden ausgewählte theoretische Grundlagen und Begriffe eingeführt, welche für das Verständnis der Arbeit von Relevanz sind. Für die nicht eingeführten Begriffe kann das Glossar und Abkürzungsverzeichnis zur Hand gezogen werden.

1.1.1 Fläche

Als Fläche werden für diesen Zweck Plätze verstanden, die für Fussgänger frei begehbar sind. Flächen sind in OSM keine eigenständige Datenelemente. Es handelt sich dabei um Polygone (sprich geschlossene Linien) oder Multipolygone (siehe 1.1.3), welche in OSM mit den Attributen *highway=pedestrian* oder *highway=footway* versehen sind. Fehlt in letzterem *area=yes*, wird die geschlossene Linie als Rundweg interpretiert, über welche nicht geroutet werden soll. [1]

1.1.2 nicht-motorisierter Individualverkehr

Zum nicht-motorisierten Individualverkehr gehören unter anderem Fussgänger, Rollstuhlfahrer und Radfahrer.

Zur besseren Lesbarkeit wird in der Arbeit nur noch von Fussgänger gesprochen. Der Fokus dieser Arbeit beschränkt sich dabei auf Fussgänger, da sich etwa für Rollstuhlfahrer und Radfahrer andere Problemstellungen auftun, welchen den Rahmen der Arbeit sprengen.

1.1.3 Multipolygon

Multipolygone sind in OSM ein oder mehrere Polygone, die jeweils einen geschlossenen Pfad beschreiben. Jedes Polygon kann entweder als einen äusseren oder inneren Ring dienen. So kann ein innerer Ring eine Teilfläche eines äusseren Rings ausschneiden. Polygone in einem Multipolygon können disjunkt sein. [2]

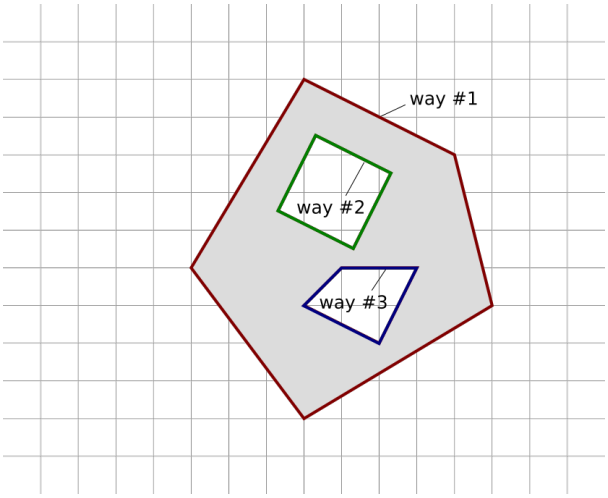

Abbildung 4: Beispiel eines Multipolygons in OSM mit einem äusseren und zwei inneren Ringen. [2]

Im Geographic Information System (GIS) hingegen ist ein Multipolygon als mehrere sich nicht überschneidende Polygone definiert. Jedes Polygon wird durch ein oder mehrere geschlossene Pfade beschrieben. Dabei ist der erste Pfad der äussere Rand, alle weiteren Pfade beschreiben innere Ringe. [3]

1.1.4 Hindernisse auf Flächen

Auf Flächen können Hindernisse existieren, durch welche nicht geroutet werden darf. Dazu gehören selbstredend Gebäude und dergleichen, aber auch Barrieren, die mit mit dem Attribut *barrier=** gekennzeichnet sind [4], was beispielsweise eine Mauer oder Hecke beschreiben kann.

1.1.5 ÖV-Haltestelle und Kante

Eine ÖV-Haltestelle kann mehrere Plattformen umfassen. Normalerweise gehören zu einer ÖV-Haltestelle zwei Plattformen, je eine in jede Fahrtrichtung. Eine dieser Plattform wird im folgenden allgemein als Kante bezeichnet. Der Unterschied ist in Abbildung 5 ersichtlich. Eine ÖV-Haltestelle muss sich dabei nicht auf zwei Kanten beschränken. Betrachtet man den Bahnhof Stadelhofen in Zürich, Schweiz, sieht man, dass weit mehr als zwei Kanten zu einer ÖV-Haltestelle gehören können.

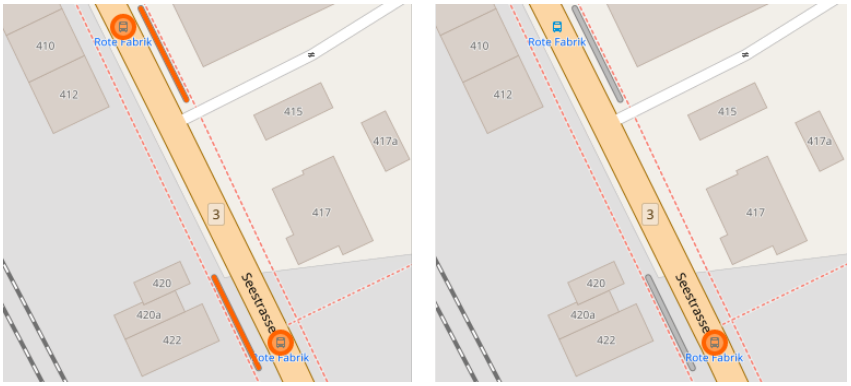

Abbildung 5: links: ÖV-Haltestelle; rechts: eine spezifische Kante

1.2 Problemstellung und Vision

Die heute gängigen Routing-Engines können effizient über Graph-Kanten und Knoten routen, um so den schnellstmöglichen Weg finden zu können. Diese wurden stetig für den motorisierten Individualverkehr optimiert, da sich diese unter anderem an vorgegebene Regeln halten und bisher einen grösseren Markt geboten haben. Bei Fussgänger ist das nicht immer der Fall. In den folgenden Unterkapiteln werden einige Probleme erläutert, welche immer noch Praxis für Fussgänger sind.

1.2.1 Routing über offene Flächen

Wie in der Abbildung 6 erkennbar is, routet die Routing-Engine GraphHopper [5] den Graph-Kanten nach um den Platz herum. Dies ist ein natürliches Verhalten für den motorisierten Individualverkehr. Ein Fussgänger hingegen nimmt den direkten Weg über Flächen. Oft handelt es sich dabei nicht nur um eine leere Fläche, sondern es sind Hindernisse wie Brunnen, Kunstwerke, WCs, etc. darauf stationiert, um welche auf eine natürliche Weise geroutet werden muss. Eine Route, welche direkt auf das Hindernis zusteuert, um dieses dann zu umlaufen, ist zwar ein Fortschritt zur aktuellen Lösung, entspricht aber kaum einem normalen Fussgänger-Verhalten.

1.2.2 eingezeichnete Fussgängerrouten über offene Flächen

Wenn man die gleiche Abbildung 6 nochmals betrachtet, sieht man, dass Mapper bereits einige Fusswege auf dem Platz eingezeichnet haben, um dem Routing-Problem über offene Flächen entgegen zu steuern. Dies kann in einigen Situationen wie dem Helvetiaplatz kontraproduktiv, aber in anderen wieder von Vorteil sein. Betrachtet man beispielsweise den Central Park in New York in Abbildung 7, macht es Sinn, dass das Routing den vorgegebenen Wegen folgt. Eine Wiese gilt als offene Fläche, eignet sich aber nicht immer als vorteilhafter Bewegungsuntergrund.

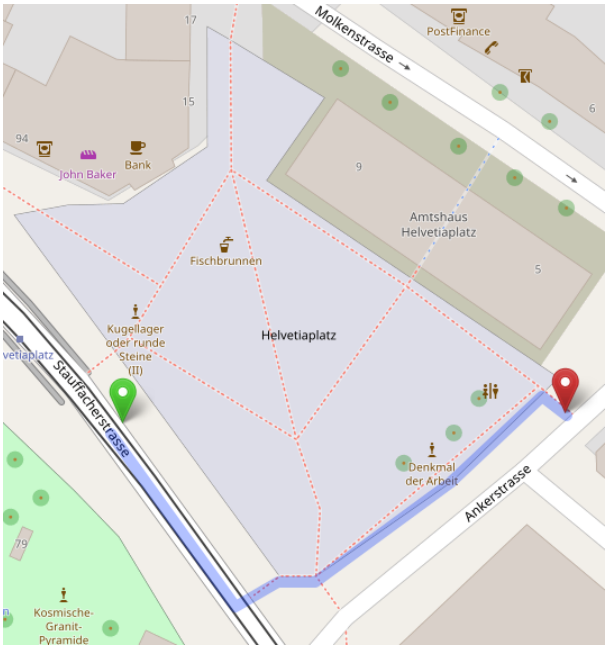

Abbildung 6: Fussgänger-Routing mit GraphHopper [5] über den Helvetiaplatz, Zürich, Schweiz; Screenshot von openstreetmap.org aufgenommen am 08.10.2017

1.2.3 topologisch nicht verbundene Wege

In Abbildung 8 ist ein topologisch nicht verbundener Graph zu sehen. In OSM kann es vorkommen, dass solche Wege auf eine offene Fläche treffen, mit dieser aber nicht topologisch verbunden sind. Eine Routing-Engine muss in der Lage sein, dies zu erkennen und aufzuräumen, so dass vom auftreffenden Weg über die offene Fläche geroutet werden kann.

1.2.4 Routing über über weitere Arten von offenen Flächen (Berge, Strände)

Berge und Strände sind bekanntermassen schwieriger zu überqueren als normale Plätze. Sei dies aufgrund der zurückzulegenden Höhendifferenz oder der Unterlage, welche das Fortbewegen einschränkt. Zusätzlich zum Problem 1.2.1 kommt hier dazu, dass das Umlaufen dieser offenen Flächen (Berge, Strände) in manchen Situationen effizienter sein kann als das Überqueren.

1.2.5 Datenaufbereitung für das Fussgänger-Routing über Strassen

Beim Fussgänger-Routing über Strassen ergeben sich andere Anforderungen als beim Routing für den motorisierten Individualverkehr. So müssen etwa Bürgersteige und Fussgänger-Streifen beachtet werden, um zu entscheiden, ob eine Strasse für Fussgänger begehbar ist.

Abbildung 7: Eingezeichnete Fussgänger-Routen auf dem Central Park, New York City, USA; Screenshot von openstreetmap.org aufgenommen am 08.10.2017

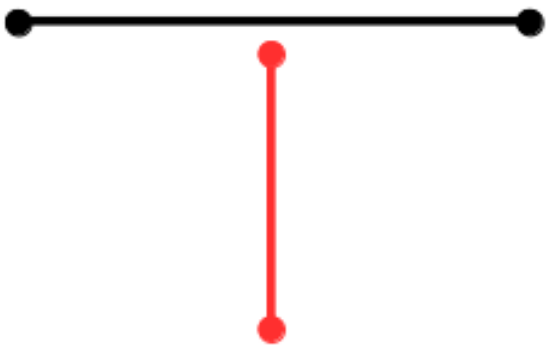

Abbildung 8: topologisch nicht verbundener Graph

1.2.6 Start-/Endpunkt auf der Fläche

Beginnt das Routing oder ist der Zielort auf einer Fläche, so ziehen die aktuellen Routing-Engines, falls keine Wege wie in 1.2.2 beschrieben eingezeichnet sind, eine Mittelsenkrechte zur nächsten Polygonkante der Fläche und routen von dort weiter. Dies ist gut auf dem Sechseläutenplatz in Abbildung 9 sichtbar. Falls bereits Wege auf der Fläche eingezeichnet sind, wird über diese geroutet.

1.2.7 Routing bei zwei benachbarten Flächen

Es kann vorkommen, dass zwei Flächen unmittelbar nebeneinander liegen. Dies ist beim Bundesplatz in Bern, Schweiz der Fall, welcher direkt an den Bärenplatz grenzt. Dies ist in Abbildung 10 sichtbar. Das erschwert das Routing über Flächen, da nicht direkt Einstiegspunkte eines Polygons genutzt werden können.

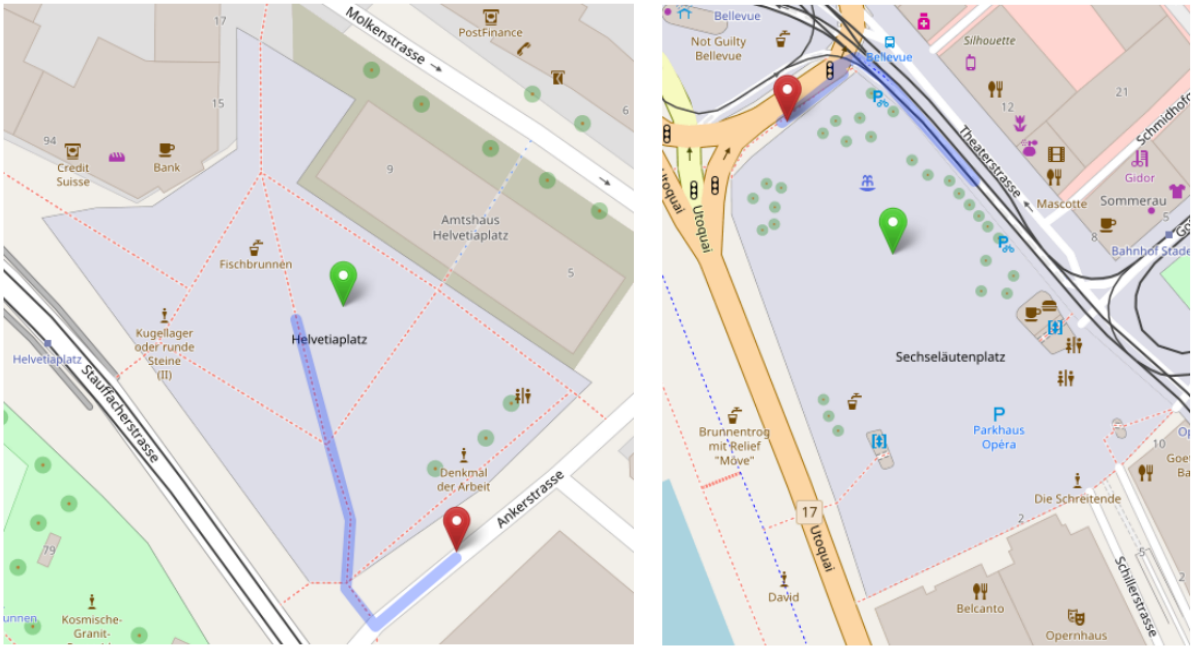

Abbildung 9: Fussgänger-Routing mit GraphHopper über den Helvetiaplatz (links) und Sechseläutenplatz (rechts), Zürich, Schweiz mit Startpunkt auf der Fläche; Screenshot von openstreetmap.org aufgenommen am 14.10.2017

1.2.8 mehrere Kanten mit dem gleichen Namen

In den meisten Fällen gibt es an einem Standort mit einer ÖV-Haltestelle zwei Kanten mit dem gleichen Namen, je eine für die jeweilige Fahrtrichtung. In manchen Fällen ist es so, dass diese Kanten ein Stück voneinander entfernt liegen. Es muss entschieden werden, zu welcher Kante geroutet werden soll.

1.3 Ziele und Unterziele

1.3.1 Routing über offene Flächen

Das Problem wie in 1.2.1 beschrieben wurde bereits in einigen Arbeiten aufgegriffen [6] [7]. Diese beiden Ansätze (Visibility-Graph und SpiderWeb-Graph) werden analysiert, dazu Tests in QGIS implementiert und mit Probanden getestet. Ziel ist es, die optimale Variante zu eruieren, um über offene Flächen routen zu können. Diese Variante soll so aufbereitet werden, dass eine OSM-Datei so verarbeitet werden kann, dass danach eine Routing-Engine auf diesen Daten operieren kann.

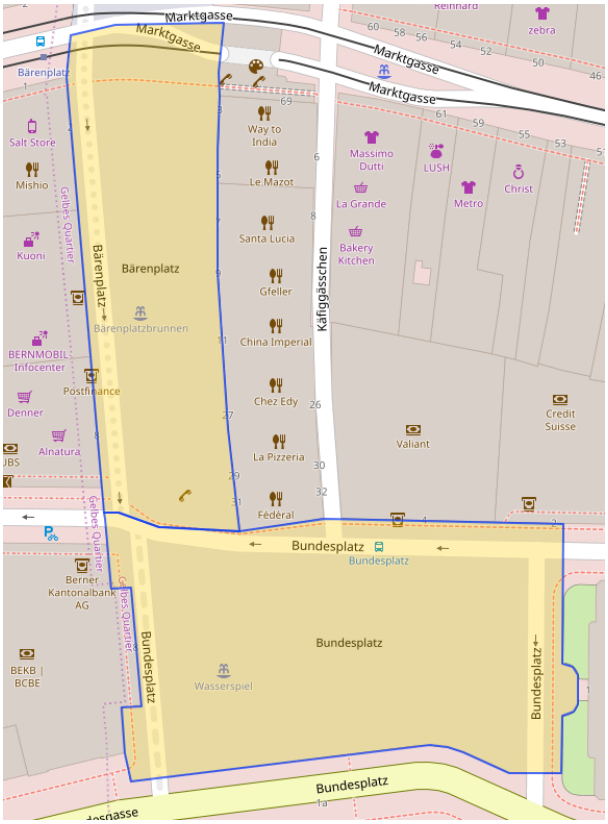

Abbildung 10: Zwei benachbarte Flächen (Bundesplatz, Bärenplatz) in Bern, Schweiz; Screenshot von overpass-turbo.eu aufgenommen am 18.10.2017

1.3.2 Routing-Engine evaluieren

Die ”grösseren”Routing-Engines sollen analysiert und es soll festgehalten werden, welche sich für den Hauptzweck dieser Arbeit am besten eignet. Zusätzlich wird geprüft, wie die verarbeitenden Daten aus 1.3.1 in die bestehenden Routing-Engines eingehängt werden können.

1.3.3 eingezeichnete Fussgängerrouten über offene Flächen

Im Kontext dieser Arbeit werden die Pfade beibehalten und keine weiteren Abklärungen bezüglich dieser Fragestellung unternommen. Im Hinblick auf eine Folgearbeit wäre es von Interesse zu wissen, in welchen Situation man sich auf vorhandene Wege über offene Flächen verlassen kann (wie in Abbildung 7) und wann sie zu ignorieren sind. Es gilt zu prüfen, ob diese Abgrenzung mit den gegebenen Informationen überhaupt möglich ist.

1.3.4 topologisch nicht verbundene Wege

Dieses Problem ist der Vollständigkeit halber aufgeführt, wird in dieser Arbeit aber nicht weiter verfolgt.

1.3.5 Routing über über weitere Arten von offenen Flächen (Berge, Strände)

Im Kontext dieser Arbeit wird kurz ein theoretischer Ansatz aufgezeigt, wie man mit einem kostenbasierten Graphen den Eigenschaften diesen Unterflächen gerecht werden könnte.

1.3.6 Datenaufbereitung für das Fussgänger-Routing über Strassen

In dieser Arbeit wird der aktuelle Stand und die Herausforderungen für die Datenaufbereitung für allgemeines Fussgänger-Routings aufgezeigt. Für das Routing über offene Flächen sind diese Probleme allerdings nicht weiter relevant. Wir gehen davon aus, dass die verwendeten Routing-Engines ein genügend gutes Verhalten für Fussgänger-Routing über Strassen aufzeigen.

1.3.7 Start-/Endpunkt auf der Fläche

Der Fussgänger-Routing kann von einem beliebigen Startpunkt aus gestartet werden. Somit soll es möglich sein, dass das Routing auf einer Fläche beginnen kann und man nicht wie in 1.2.6 beschrieben zur nächstgelegenen Polygonkante geroutet wird. Es soll geklärt werden, ob die Vorverarbeitung der Flächen, wie es in 1.3.1 vorgesehen ist, ausreicht, oder ob zu einem späteren Zeitpunkt ein Echtzeit-Verarbeitung der Fläche vom aktuellen Standort aus in Betracht gezogen werden muss.

1.3.8 Routing bei zwei benachbarten Flächen

Es soll geprüft werden, wie Fussgänger-Flächen bearbeitet werden können, die direkt benachbart sind. Wenn die beiden Flächen unabhängig voneinander verarbeitet werden, gibt es unter Umständen eine Polygonkante, die von einem Fussgänger problemlos übergangen werden könnte, logisch aber als eine Abgrenzung interpretiert wird, die in der Route nicht überquert wird.

1.3.9 nächste ÖV-Haltestellen finden

Ein weiterer Fokus der Arbeit liegt auf dem Finden und Erreichen der nächsten ÖV-Haltestelle, von welcher ein ÖV-Routing durchgeführt werden kann. So soll eruiert werden, wie von einem bestehenden Startpunkt aus die nächsten ÖV-Haltestellen identifiziert werden können, um diese an search.ch für das weitere ÖV-Routing zu übergeben.

1.3.10 mehrere Kanten mit dem gleichen Namen

Wurde die nächste ÖV-Haltestelle wie in 1.3.9 beschrieben eruiert, so gilt es, zur Kante in die richtige Fahrtrichtung zu routen.

1.3.11 Prototyp und Deliverables

Das Resultat besteht aus zwei Teilen. Einerseits wird es möglich sein, OSM-Daten vorzuverarbeiten, welche an eine Routing-Engine übergeben werden können, um das Fussgänger-Routing im urbanen Raum sicherzustellen. Andererseits wird ein Backend entwickelt, welches ein multimodales Routing, sprich Fussgänger- in Kombination mit einem ÖV-Routing anbietet. In einem weiteren Schritt wird die Vorverarbeitung und das multimodale Routing in einem QGIS-Plugin sichtbar gemacht. Die detaillierten Anforderungen sind im Teil II Kapitel 2 beschrieben.

1.4 Rahmenbedingungen, Umfeld, Definitionen und Abgrenzungen

Die Arbeit befasst sich mit Flächen im urbanen Raum. Berge, Strände, Parks, etc. werden bewusst ausgeklammert, um nicht den Rahmen der Arbeit zu sprengen. Voraussetzung ist, dass Python als Programmiersprache verwendet wird und OSM-Daten [8] verarbeitet werden.

Als Drittsysteme sind das Application Programming Interface (API) von search.ch [9] für das ÖV-Routing, Overpass [10] für das Extrahieren von spezifischen OSM-Daten und Nominatim [11] für das Geocoding zu gebrauchen.

Die Backend-Komponente sollen als Docker-Container für eine einfache Portierung und Deployment verfügbar sein. Die Funktionalität der Komponenten wird kontinuierlich mit Continuous Integration sichergestellt.

1.5 Vorgehen und Aufbau der Arbeit

Die Studienarbeit nimmt sich zu erst dem Problem des Fussgänger-Routing über Flächen im urbanen Raum an. Es wird geklärt, wie der Stand der Technik ist und welche Vorarbeiten und Lösungen in diesem Bereich existieren. Bestehende Lösungsvorschläge werden in Python implementiert und in QGIS getestet, um so die bestmögliche Variante zu eruieren. Sobald die optimale Lösungsvarianten identifiziert sind, muss geklärt werden, wie die Vorverarbeitung der Daten (beispielsweise das Einzeichnen von möglichen Routen über Flächen) an eine bestehende Routing-Engine übergeben werden kann. Damit dies möglich ist, ist zu prüfen, welche Routing-Engine die Anforderungen bestmöglich abdeckt.

In einem weiteren Schritt soll ermittelt werden, wie die nächsten ÖV-Haltestellen eruiert werden können, um diese Punkte an das API von search.ch [9] übergeben zu können. Aufgabe von search.ch ist es, für einen Start und eine Destination die ÖV-Route zu genieren. So kann der schnellstmögliche Weg von einem beliebigen Punkt an ein beliebiges Ziel in Kombination mit Fussweg und öffentlichem Verkehr ermittelt werden.

Die vorhin aufgeführten Aspekte werden im Teil I behandelt.

In einem zweiten Teil werden die im Teil I erarbeiteten Artefakte in einem Prototyp als QGIS-Plugin zusammengeführt. Im QGIS-Plugin wird ein multimodales Routing visuell dargestellt. So ist ein optimiertes Fussgänger-Routing in Kombination mit dem öffentlichen Verkehr möglich. Dazu müssen OSM-Daten für das Fussgänger-Routing über offene Flächen in urbanen Raum aufbereitet werden können und das multimodale Routing als Backend verfügbar sein.

Die Implementation wird im Teil II beschrieben.

2 Stand der Technik

2.1 Aktuelle Situation

In OSM gibt es mehrere Arten, wie Mapper mit Flächen umgehen. So wirken einige Mapper dem Fussgängerrouting-Problem über Flächen mit zusätzlichen eingezeichneten Wegen entgegen, oder aber die Fläche ist ganz von Wegen befreit.

In Abbildung 11 und 12 ist ein Vergleich gängiger Routing-Engines abgebildet, welche ein Fussgänger-Profil anbieten. In Abbildung 11 sieht man schön, wie auf openstreetmap.org die eingezeichneten Fusswege über den Platz genutzt werden. Diese Route entspricht offensichtlich nicht einem natürlichen Fussgänger-Verhalten, da normalerweise der direkte Weg über den Platz gewählt wird, sofern keine Hindernisse im Weg sind. Betrachtet man in Abbildung 12 hingegen den Sechseläutenplatz, sieht man,

dass ohne eingezeichnete Wege alle getesteten Anbieter um den Platz herum führen.

Abbildung 11: Routing-Vergleich von verschiedenen Anbietern mit Fussgänger-Profil über den Helvetiaplatz, Zürich, Schweiz; Links: Google Maps, Mitte-Oben: openstreetmap.org mit GraphHopper, Mitte-Unten: openstreetmap.org mit Mapzen, Rechts: openrouteservice.org; Screenshots aufgenommen am 13.10.2017

Abschliessend kann man sagen, dass alle getesteten Routing-Engines mit den Fussgänger-Profilen mit Stand 13.10.2017 scheitern, wenn keine Wege über die Fläche eingezeichnet sind, und so die Dauer und Streckenlänge verfälscht wird.

2.2 Lösungsansätze

2.2.1 Routing über offene Flächen

In den folgenden Unterkapitel werden bestehende Lösungsansätze aus der Literatur diskutiert, die das Problem des Routings über offene Flächen, wie in Kapitel 2.2.5 beschrieben, lösen sollen. Die beiden vielversprechendsten Ansätze Visibility-Graph und SpiderWeb-Graph werden in Kapitel 3 tiefergehend miteinander verglichen.

Visibility-Graph

Ein Ansatz zur Verhinderung von Kollisionen auf Flächen wurde in [12] als Visibility-Graph beschrieben. Dabei wird in einem ersten Schritt für jeden Knotenpunkt einer Fläche eine Verbindung zu jedem anderen Knotenpunkt gezeichnet. Für unsere Zwecke werden dabei alle Knotenpunkte eines Polygons, Hindernisse, sowie Schnittpunkte mit Strassen und Wegen (Einstiegspunkte) beachtet.

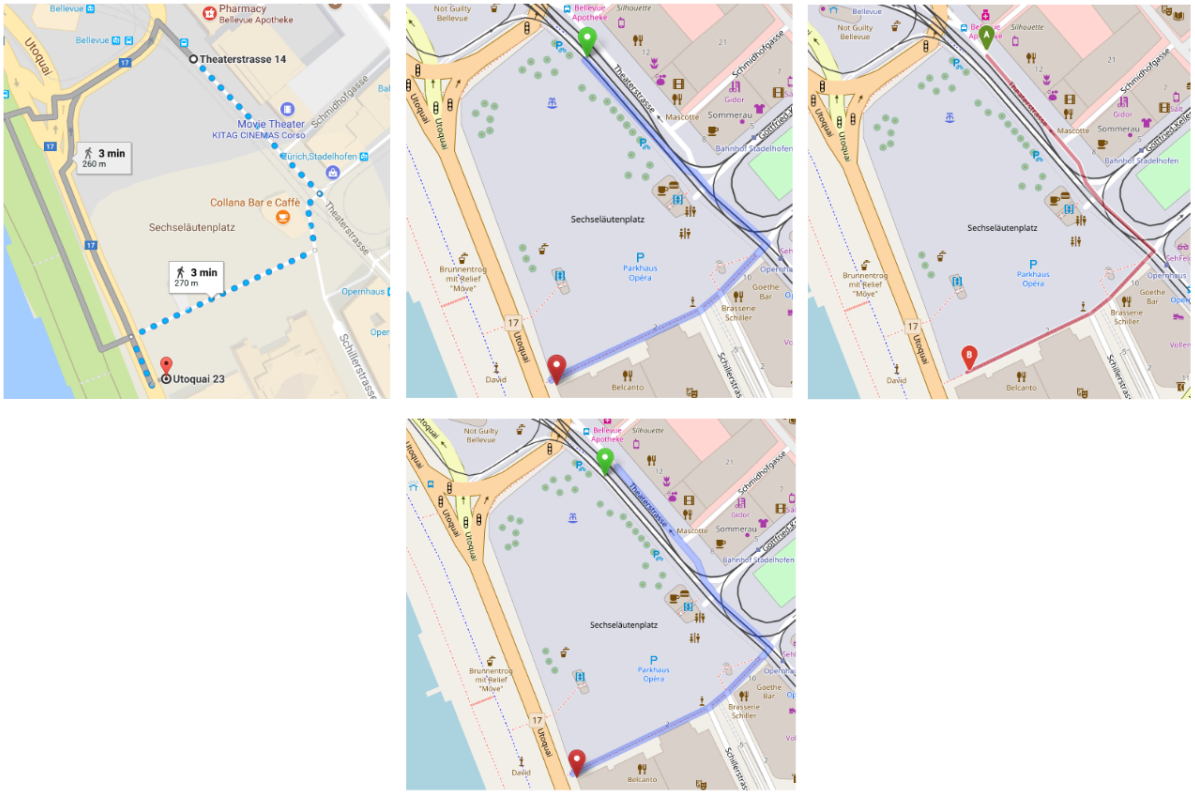

Abbildung 12: Routing-Vergleich von verschiedenen Anbietern mit Fussgänger-Profil über den Sechseläutenplatz, Zürich, Schweiz; Links: Google Maps, Mitte-Oben: openstreetmap.org mit GraphHopper, Mitte-Unten: openstreetmap.org mit Mapzen, Rechts: openrouteservice.org; Screenshots aufgenommen am 13.10.2017

In einem zweiten Schritt werden alle Verbindungen verworfen, die nicht komplett innerhalb der Fläche liegen oder mit einem Hindernis darauf kollidieren.

```
def optimized_visibility_graph():
    visibility_graph = []
    for edge in all_edges:
        if edge_does_not_collide(edge):
            visibility_graph += edge
    for point in entry_points:
        paths = shortest_path_to_all(visibility_graph, entry_points)
        add_to_routing_graph(paths)
```

Listing 1: Konstruktion eines optimierten Visibility-Graphen

Für die Verwendung im Routing beschreibt [6] einen Ansatz, um die Kanten des Visibility-Graphen zu reduzieren (siehe Listing 1). Mit dem Visibility-Graphen allein gäbe es mit n Knotenpunkten bis zu $n(n-1)/2$ Kanten. Für die Reduktion wird für jeden Einstiegspunkt (wie aus Schritt 1 bekannt) jeweils der kürzeste Pfad auf dem Visibility-Graphen zu allen anderen Einstiegspunkten berechnet. Alle Kanten, die nicht zu einem kürzesten Pfad gehören, werden verworfen. Abbildung 13 zeigt einen Vergleich zwischen eines Visibility-Graphen vor und nach dem Reduktionsverfahren.

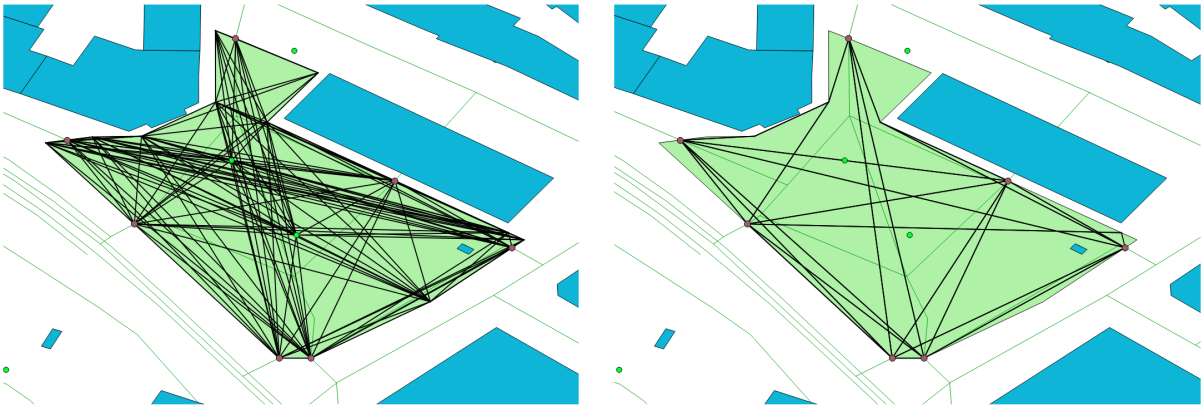

Abbildung 13: Visibility-Graph über den Helvetiaplatz, Zürich, Schweiz; Links: Unveränderter Visibility-Graph; Rechts: Reduziert auf kürzeste Pfade

SpiderWeb-Graph

Die Arbeit [7] befasst sich mit dem Flächenrouting für Nutzern von Elektrorollstühlen, indem ein Spinnennetz (siehe Abbildung 14) über das Polygon gelegt wird. Dies hat den Vorteil, dass auf der Fläche zusätzliche Linien und Kanten vorhanden sind (wobei statische Hindernisse berücksichtigt werden), welche für das Routing verwendet werden können. Diese Idee und das Grundprinzip wurde im Folgenden übernommen.

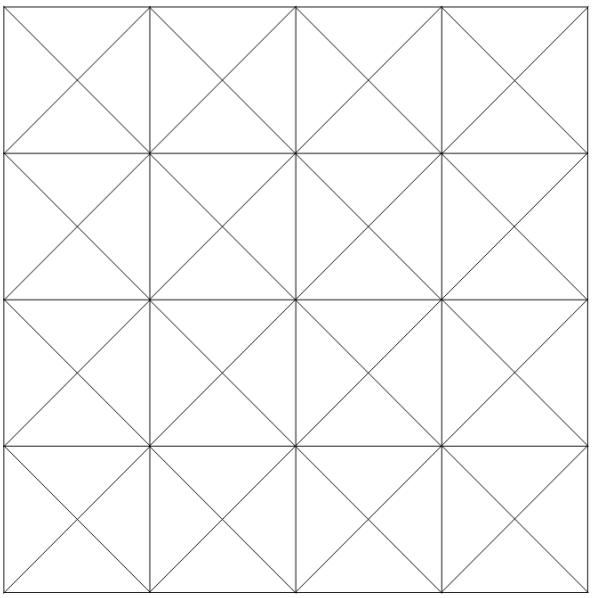

Abbildung 14: Spinnennetz

Damit ein Spinnennetz über eine Fläche gezeichnet werden kann, muss eine Bounding-Box berechnet werden. Es handelt sich dabei um ein Rechteck, welches die ganze Fläche überdeckt.

Vor dem Zeichnen des Spinnennetz ist es möglich, den Abstand zwischen Spalten und den Zeilen (Spacing) zu definieren. Dies hat hohe Auswirkung auf die Anzahl Zeilen und Spalten und somit auf die gezeichneten Pfade über die Fläche, da mit einem kleineren Spacing eine ”Glättung”der Pfade möglich ist. Die gezeichneten Pfade widerspiegeln so eher das Verhalten eines Fussgängers. Ein Abstrich

des kleiner Spacings ist sofort ersichtlich, wenn man das Spinnennetz in der Abbildung 14 betrachtet. Man kann annehmen, dass dieses Spinnennetz über eine Fläche von 4*4 cm mit einem Spacing von einem 1 cm gezeichnet wird. Verkleinert man nun das Spacing auf 0.5 cm, steigt die Anzahl der zu zeichnenden Linien von 72 auf 272. Halbiert man ein weiteres Mal, ist man bereits bei 1056 Linien. Damit ein Routing über das Spinnennetz und somit Abzweigungen möglich sind, kann nicht eine Linie pro Zeile oder Spalte genutzt werden.

Auf dem Spinnennetz wird ähnlich wie beim Visibility-Graph zwischen allen Einstiegspunkten der Shortest-Path berechnet. Für die weitere Verarbeitung werden nur noch diese optimierten Pfade verwendet. Zur Verdeutlichung der Idee ist der Pseudocode in Listing 2 zu betrachten.

```
1 def create_spiderweb_graph(plaza_list):
2     for plaza in plaza_list:
3         spiderweb = draw_spiderweb(plaza)
4         entry_points = get_entry_points(plaza, road_layer)
5         connect_entry_points_with_spiderweb(entry_points, spiderweb)
6         calculate_shortest_path(spiderweb)
```

Listing 2: SpiderWeb Pseudocode

Das Zeichnen des Spinnennetzes hat eine Effizienz von $O(N \times M)$, wobei N und M die Anzahl Spalten und Zeilen sind. Die Anzahl der Zeilen und Spalten erfahren beim Halbieren des Spacings jeweils ein quadratisches Wachstum. Durch ein kleineres Spacing steigen auch die Anzahl möglicher Pfade über die Fläche an, was Auswirkung auf die Dauer der Generierung des Shortest Paths hat.

Betrachtet man das Resultat in Abbildung 15, sieht man in Schwarz den generierten Shortest Path über die Fläche von allen Einstiegspunkten aus.

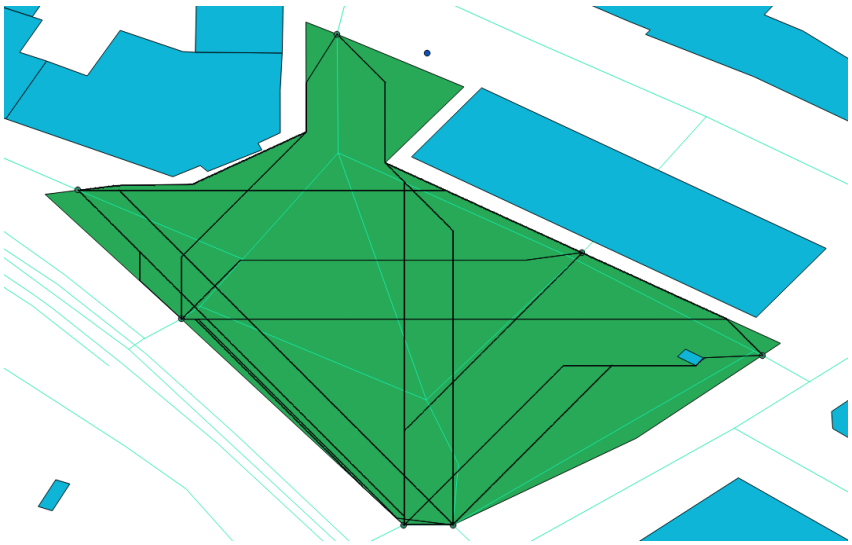

Abbildung 15: Resultat SpiderWeb-Graph über Helveltiaplatz, Zürich, Schweiz

Es wurde ebenfalls geprüft, ob die Rotation des Spinnennetz auf die Gegebenheit der Fläche eine Auswirkung auf die Shortest Path-Generierung hat. Bei einem Test, welcher in Abbildung 16 ersichtlich

ist, wurden keine entscheidenden Vorteile gefunden, warum sich der zusätzliche Aufwand rechtfertigen würde. Das Spinnennetz wurde um 45 beziehungsweise um 90 Grad gedreht. Die berechneten kürzesten Pfade sind fast identisch.

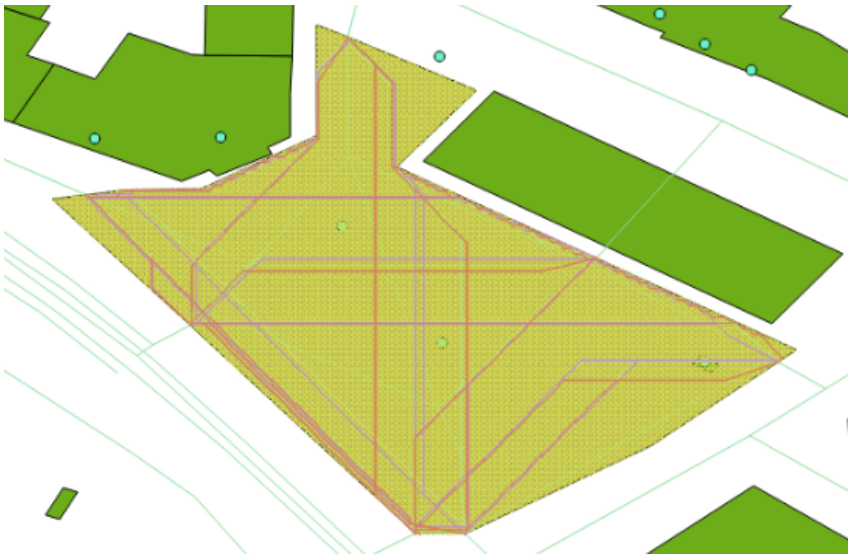

Abbildung 16: Rotation des SpiderWebs um 45 und 90 Grad im Vergleich mit keiner Rotation

Straight Skeleton

Die Methode des *Straight Skeleton* wurde erstmals in [13] eingeführt. Dabei werden Schritt für Schritt die Kanten des Polygons zur Mitte zusammen geführt, es entstehen Verbindungen mit scharfen Ecken zu den Eckpunkten des Polygons. In Abbildung 17 ist beispielhaft ein Graph aufgezeichnet.

Die Andwendung dieser Methode für Flächenrouting wurde bereits in [6] diskutiert. Es stellt sich heraus, dass die *Straight Skeleton* Methode keine natürlichen Routen für Fussgänger ergibt, da sie tendenziell zur Mitte der Fläche gehen und sich dabei scharfe Kurven ergeben. Aus diesen Gründen wird dieser Ansatz nicht weiter behandelt.

2.2.2 Routing über über weitere Arten von offenen Flächen (Berge, Strände)

Wie in 1.2.4 beschrieben, ergeben sich beim Überqueren von offenen Landflächen wie Berge oder Strände andere Fragestellungen. Hier spielt die Unterlage eine grössere Rolle. Faktoren wie Höhe oder Beschaffenheit des Untergrunds sind nun relevant. Für diese Problematik wird ein technischer Lösungsansatz vorgestellt und bewertet.

Technischer Ansatz mit Cost Path Analyse

Ein Ansatz für die Berechnung von Routen über offene Flächen ist die Cost Path Analyse [14]. Dieses Verfahren wird oft angewendet, um in einer Fläche Korridore zu finden, um darin beispielsweise neue Strassen zu erschliessen [15].

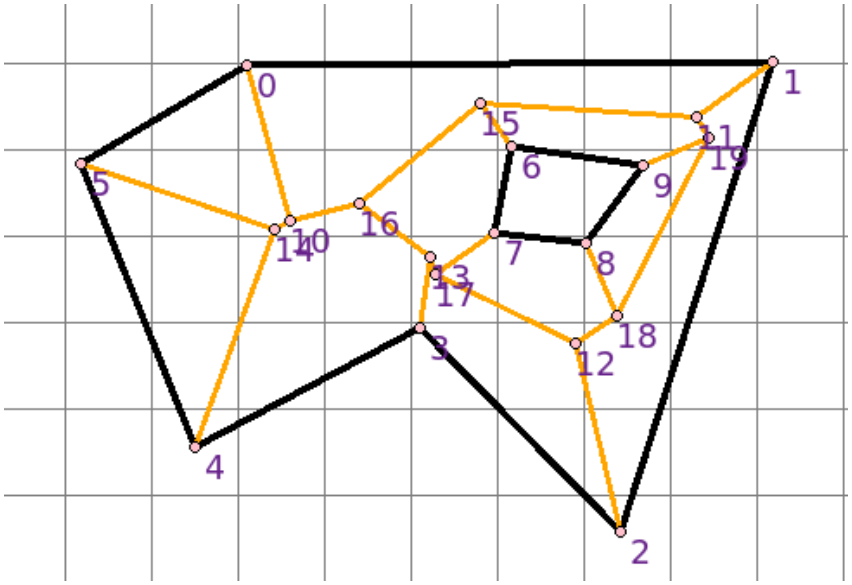

Abbildung 17: Straight Skeleton eines Multipolygons; Erstellt mit der Software pySkeleton von Ecole Centrale Paris

In einem ersten Schritt wird ein Raster über die Fläche gelegt. Jeder Kachel wird ein Wert zugewiesen, der den Kosten entspricht, um diese Fläche zu traversieren. Die Kosten können zusammen gesetzt sein aus mehreren Faktoren wie die Steigung oder Beschaffenheit des Untergrunds. So bekäme etwa ein Sandstrand höhere Kosten als eine Asphalt-Fläche. Dieses Raster wird auch Cost Surface genannt. [16]

Sobald ein Startpunkt der Route bekannt ist, kann als nächstes ein Cost Distance Raster erstellt werden. Mit Hilfe der Cost Surface wird im Raster vom Startpunkt aus zu jedem erreichbaren Punkt die minimalen Kosten akkumuliert [17]. Ist nun der Zielpunkt der Route ebenfalls bekannt, kann von diesem Zielpunkt aus jeweils zur Nachbar-Kachel mit den geringsten Kosten navigiert werden, bis der Startpunkt erreicht wurde [14]. In Abbildung 18 ist eine Cost Path Analyse beispielhaft dargestellt.

Bewertung

Im Kontext des Fussgänger-Routings gibt es mit dem Ansatz der Cost Path Analyse einige weitere Fragestellungen. Für jede Fläche müsste zuerst eine Cost Surface erstellt werden. Das Cost Distance Raster kann dann erst zur Laufzeit berechnet werden, da Start- und Zielpunkt vorher nicht bekannt sind.

Bei offenen Landflächen in Bergen ist es ausserdem schwierig zu entscheiden, ob ein Fussgänger eine Fläche passieren kann. Solche Flächen sind in OSM oft nicht gemappt und es existieren nicht immer Informationen über den Untergrund. Es kommen auch andere Faktoren wie z.B. die Jahreszeit hinzu. So könnte das Begehen einer Wiese im Sommer problemlos, im Winter aber durch eine Schneedecke zeitweise unmöglich sein. Bei Stränden sieht es wiederum anders aus, diese sind in OSM meist als Flächen gemappt. Dort spielt aber auch das Höhenprofil normalerweise keine Rolle, womit ein Ansatz wie unter 2.2.1 wieder effizienter sein könnte.

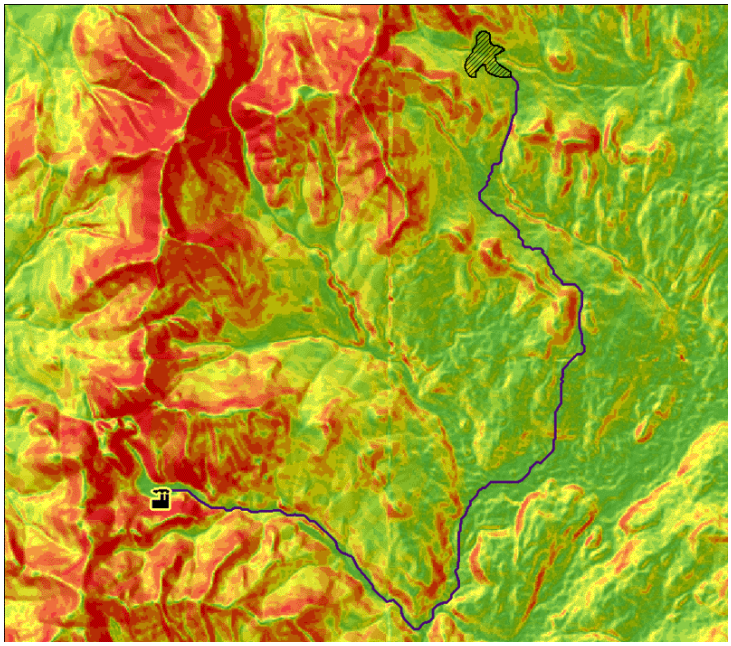

Abbildung 18: Resultat einer Cost Path Analyse; Grafik von [17]

2.2.3 Datenaufbereitung für das Fussgänger-Routing über Strassen

Damit eine Routing-Engine die Kartendaten für das Routing nutzen kann, müssen diese zuerst zu einem Graphen aufbereitet werden. Für das Fussgänger-Routing ergeben sich dabei andere Herausforderungen als für das Routing des motorisierten Individualverkehrs. So muss etwa beim Fussgänger-Routing beachtet werden, wann ein Fussgänger eine Strasse überqueren kann. Dafür müssen gewisse Annahmen getroffen werden. Eine kaum befahrene Strasse in einem ländlichen Gebiet kann höchstwahrscheinlich jederzeit überquert werden, während in einem städtischen Gebiet eher Fussgängerstreifen verwendet werden sollten.

Ein wichtiger Bestandteil der Datenaufbereitung ist der Einbezug von Bürgersteige. Diese werde in OpenStreetMap auf unterschiedliche Weise gemappt [18]. Sie können als eigene Wege oder als Eigenschaft der Strasse definiert werden. In [19] wurde ein Datenmodell für Fussgänger-Routing entwickelt. Dabei ist die Datenqualität der OSM-Daten bei Bürgersteigen weiterhin ein Problem. In den letzten Jahren gab es Bemühungen, die Datenqualität von Bürgersteige [20] und Fussgängerstreifen [21] durch die Analyse von Sattelitenbilder zu verbessern.

Abschliessend kann man sagen, dass die Datenaufbereitung für Fussgänger-Routing weiterhin eine Herausforderung ist. Auch wenn diese für die Hauptproblemstellung von Routing über offene Flächen nicht direkt relevant sind, ist dies ein wichtiger Bestandteil für das Fussgänger-Routing als Ganzes.

2.2.4 Start-/Endpunkt auf der Fläche

Ein Spezialfall des Fussgänger-Routings entsteht, wenn der Start- oder Endpunkt auf einer Fussgänger-Fläche liegt. Wie in 1.2.6 beschrieben ist es üblich, dass Routing-Engines eine Mittelsenkrechte zum nächten Punkt auf dem Strassennetz ziehen.

In unserem Prototyp erfolgt die komplette Berechnung der kürzesten Wege über Fussgänger-Flächen im Voraus. Zur Laufzeit (bei einer Anfrage des Benutzers) werden die neu berechneten Fusswege lediglich von der Routing-Engine verwendet. Für unsere Zwecke ergeben sich so zwei mögliche Ansätze:

1. Zur Laufzeit wird durch die Routing-Engine vom Start- bzw. Endpunkt auf der Fläche der kürzeste Weg zum Strassennetz in Richtung der Route berechnet, dabei werden Hindernisse umgangen.

2. Die von uns berechneten Wege über Flächen reichen aus, da die Routing-Engine eine Mittelsenkrechte auf den nächstgelegenen Fussweg auf der Fläche zieht. Zur Laufzeit müssen keine zusätzlichen Berechnungen durchgeführt werden.

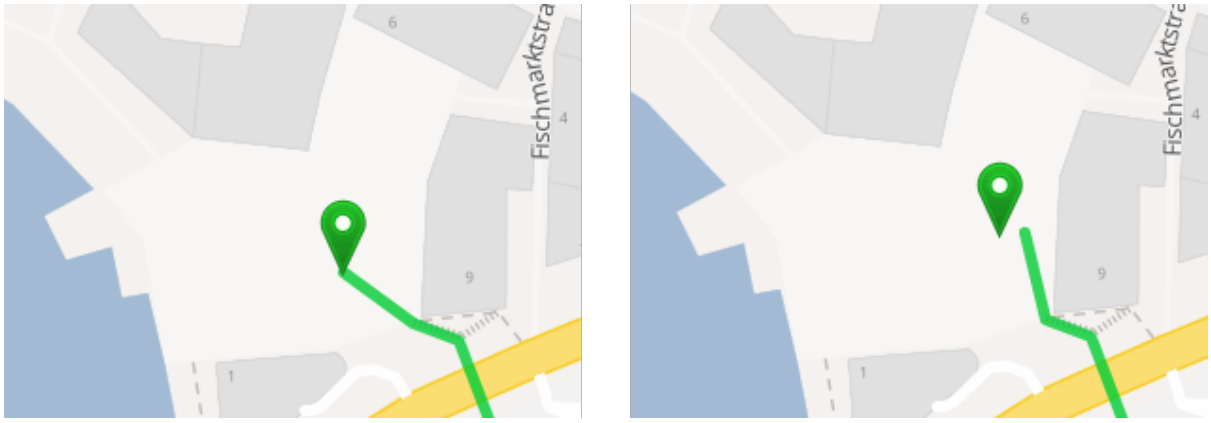

Abbildung 19: Vergleich einer beispielhaften Route mit dem Ansatz 1 (links) und Ansatz 2 (rechts); Gezeigt auf dem Fischmarktplatz, St. Gallen, Rapperswil; Screenshot von GraphHopper [5]

Wie in Abbildung 19 zu sehen, ergibt der zweite Ansatz ohne zusätzlichen Berechnungen genügend gute Resultate, da es zwischen jedem Ein- und Ausgangspunkt ein Fussweg hat, der bereits den kürzesten Weg beschreibt. Eine zusätzliche Berechnung des kürzesten Pfades wie im ersten Ansatz ergibt keine signifikante Verbesserung der Route.

2.2.5 Routing bei zwei benachbarten Flächen

Wie in 1.3.8 beschrieben, gibt es bei zwei benachbarten Flächen einen Spezialfall für das Routing. Bei unserer Implementation wissen die beiden Flächen nichts voneinander und es werden keine Einstiegspunkte erkannt, wo der Fussgänger die Abgrenzung der beiden Flächen überqueren könnte.

Im Rahmen unserer Arbeit haben wir in der Literatur keine entsprechende Diskussion dazu gefunden. Wir haben für die Lösung des Problems zwei Ansätze überlegt:

1. Die beiden Flächen miteinander kombinieren: Wenn sich zwei Fussgänger-Flächen berühren, können sie geometrisch vereint und als eine einzige Fläche angesehen werden.

2. Gemeinsame Einstiegspunkte zwischen den Flächen definieren: An der Kante, wo sich zwei Flächen berühren, werden künstlich in einem regelmässigen Abstand Einstiegspunkte eingefügt, wo für das Routing der Fussgänger die Kante überqueren kann.

2.3 Verbesserungsmöglichkeiten

Für die in Kapitel 2.2 diskutierten Ansätze für Routing über offene Flächen werden Defizite und mögliche Verbesserungsmöglichkeiten aufgezeigt.

2.3.1 Einstiegspunkte

Sowohl der SpiderWeb-Graph wie auch der Visibility-Graph benutzen Einstiegspunkte, um die kürztesten Pfade zu finden und den Graphen mit dem bestehenden Strassennetz zu verbinden. Es kommt allerdings oft vor, dass es auf einer Fussgängerfläche keinen oder nur einen einzigen Einstiegspunkt gibt, wodurch auch kein Graph berechnet werden kann. In der Realität könnte es aber trotzdem möglich sein, eine solche Fläche zu Fuss zu überqueren, auch wenn keine Strasse daran angrenzt.

Eine mögliche Lösung dieses Problems könnte sein, die Umgebung der Fussgänger-Fläche zu analysieren und künstliche Einstiegspunkte dort einzufügen, wo ein Fussgänger die Fläche betreten kann. Dies könnte von einer angrenzenden Fussgänger-Fläche aus sein (siehe Kap. 2.2.5) oder von einer anderen Fläche, wobei der Zutritt zur Fussgänger-Fläche nicht durch ein Hinderniss verunmöglicht werden soll.

2.3.2 SpiderWeb-Graph

Ein Problem des SpiderWeb-Graphen ist die Auflösung des Gitters (SpiderWebs), die gewählt werden muss, um optimale Routen zu erhalten. Bei einer zu kleinen Auflösung macht die Route viele Kurven, da sie ständig dem Gitter entlang fahren muss. Eine sehr grosse Auflösung des SpiderWebs ist aber für die Praxis sehr rechenaufwändig, da die benötigte Rechenzeit quadratisch mit einer höheren Auflösung steigt.

Eine mögliche Verbesserung für dieses Problem ist die Anwendung einer Glättung auf die erzeugte Route. Damit wird die Anzahl der Kurven verringert und so einen natürlicheren Pfad erzeugt. Ein Ansatz dafür bietet der Douglas-Peucker Algorithmus [22], der eine Linien-Geometrie vereinfachen kann. Dabei muss allerdings sichergestellt werden, dass die Glättung der Route nicht mit einem Hinderniss kollidiert oder ausserhalb der Fläche fällt. Mit Hilfe der Glättung kann so eine kleinere Auflösung für das SpiderWeb gewählt werden, was zusätzlich Rechenzeit spart.

3 Bewertung Routing über offene Flächen

Das Kapitel 2.2.1 hat zwei Algorithmen hervorgebracht, namentlich Visibility-Graph und SpiderWeb-Graph, welche im Folgenden bewertet werden. Dazu werden messbare Kriterien definiert, gewichtet und die beiden Varianten daran gemessen. Am Ende wird den Varianten eine Punktzahl auf einer linearen Skala von 1 bis 5 zugewiesen und ein Fazit gezogen.

3.1 Kriterien

3.1.1 Verarbeitungszeit

Der massgebende Maximalwert von *2 Stunden* ist in der nicht-funktionalen Anforderung 2.2.4 im Teil II definiert. Die Verarbeitungszeit wird bewertet auf einer Skala von 1 (2 Stunden) bis 5 (0 Minuten).

3.1.2 zusätzliche Datenmenge

Die Algorithmen generieren zusätzliche Geometrien. Da die öffentlichen Plätze nur einen kleinen Teil der Kartendaten bilden, soll die zusätzliche Datenmenge keinen signifikaten Teil ausmachen. Der massgebende Maximalwert *1%* ist in der nicht-funktionalen Anforderung 2.2.5 im Teil II definiert. Die zusätzliche Datenmenge im Verhältnis zur originalen Datei wird bewertet auf einer Skala von 1 (1%) bis 5 (0%).

3.1.3 Natürlichkeit

Für die Natürlichkeit einer Route ist es schwer, messbare Kriterien zu definieren. Als eine natürliche Route versteht man, wenn man auf direktem Weg über den Platz geht und keine Umwege macht. Treten Hindernisse auf dem Platz auf, widerspricht es einem natürlichen Fussgänger-Verhalten, wenn man direkt aufs Hindernis zu läuft und dieses am Rande umgeht. Um trotzdem ein Fazit ziehen zu

können, werden 3 Plätze (Helvetiaplatz in Zürich, Fischmarktplatz in Rapperswil-Jona und der Bahnhofplatz in Bern) untersucht. Dabei wird für beide Varianten eine interaktive Routing-Oberfläche Probanden zu Verfügung gestellt. Die Probanden haben die Möglichkeit, selbst über den Platz zu routen. In Abbildung 20 sind einige Routen beispielhaft dargestellt. Danach bewerten diese die Natürlichkeit mit Punkten von 1 bis 5. Als massgeblicher Vergleichswert wird der Durchschnitt der Bewertungen aller Probanden genommen.

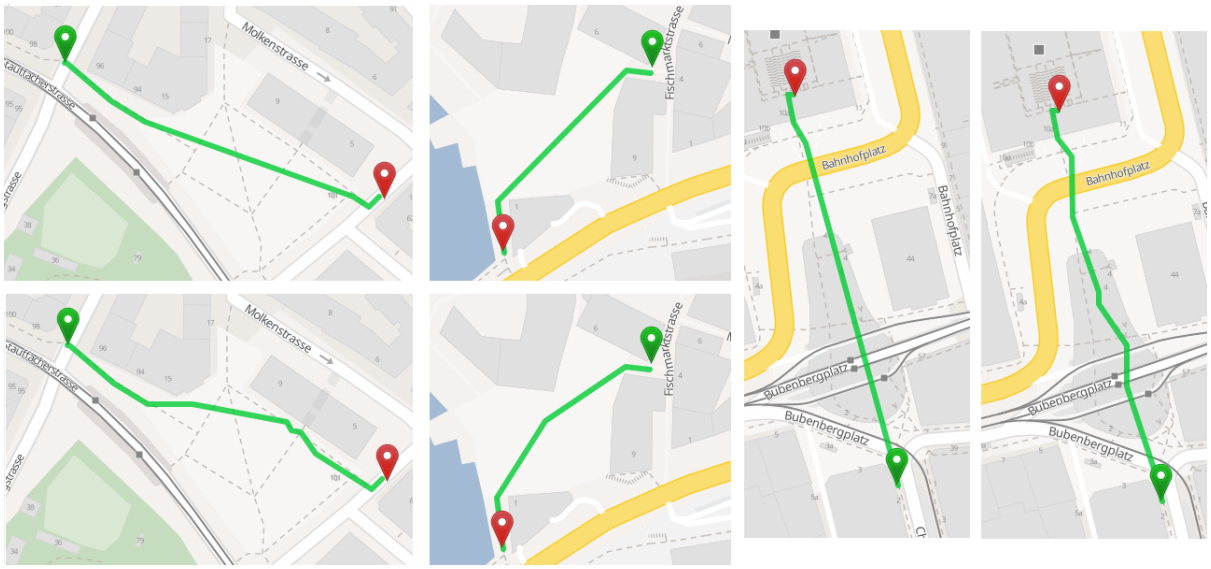

Abbildung 20: Beispielhafte Routen über die drei getesteten Flächen; Jeweils oben bzw. links Visibility-Graph, unten bzw. rechts SpiderWeb-Graph

3.2 Resultate

Die Vorverarbeitung wurde auf einem Rechner mit einem Intel i7 2600k Prozessor und Fedora 26 als Betriebssystem ausgeführt.

Für die Vergleichswerte der Verarbeitungszeit und der zusätzlichen Datenmenge wird die Datei *switzerland-exact.osm* [8], Stand 12.11.17, verwendet. Im OSM-Format ist die Datei 6.3 GB gross.

3.2.1 Verarbeitungszeit

Die Vorverarbeitung wurde mit beiden Algorithmen auf der Test-Datei ausgeführt und dabei die benötigte Zeit gemessen. Die Zeit für den ersten Schritt, das Importieren der OSM-Daten, wird dabei abgezogen, da dies unabhängig vom jeweiligen Algorithmus ist. Die Resultate sind in Tabelle 1 dargestellt.

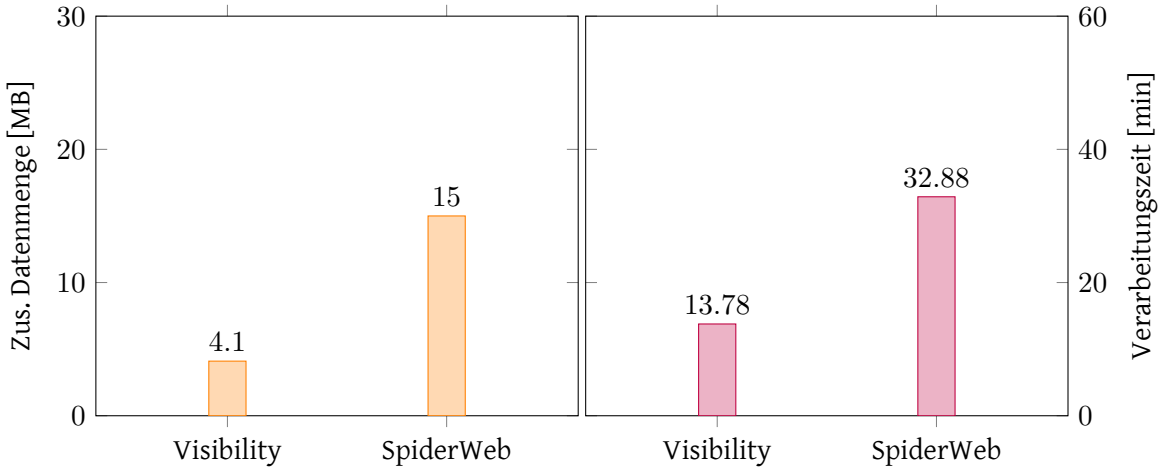

Abbildung 21: Vergleich der zusätzlichen Datenmenge und der Verarbeitungszeit von Visibility-Graph und SpiderWeb-Graph

Tabelle 1: Resultat: Verarbeitungszeit (ohne Initialer Import der Daten)

	Visibility-Graph	**SpiderWeb-Graph**
Verarbeitungszeit	13min 47s	32min 53s
Bewertung 1-5	4.54	3.90

3.2.2 zusätzliche Datenmenge

Tabelle 2 zeigt die zusätzliche Datenmenge im OSM-Format, die durch die Vorverarbeitung mit dem jeweiligen Algorithmus entstanden ist, verglichen mit der Dateigrösse der OSM-Datei vor der Vorverarbeitung.

Tabelle 2: Resultat: zusätzliche Datenmenge

	Visibility-Graph	**SpiderWeb-Graph**
zusätzliche Datenmenge	4.13 MB	15.1 MB
in Prozent der Originalgrösse	0.06%	0.24%
Bewertung 1-5	4.76	4.04

3.2.3 Natürlichkeit

Für die Evaluation wurde die Bewertung der Natürlichkeit mit fünf Probanden durchgeführt. Die einzelnen Bewertungen sind in Tabelle 3 abgebildet. In Abbildung 22 sind die durchschnittlichen Bewertungen pro Platz und Algorithmus dargestellt.

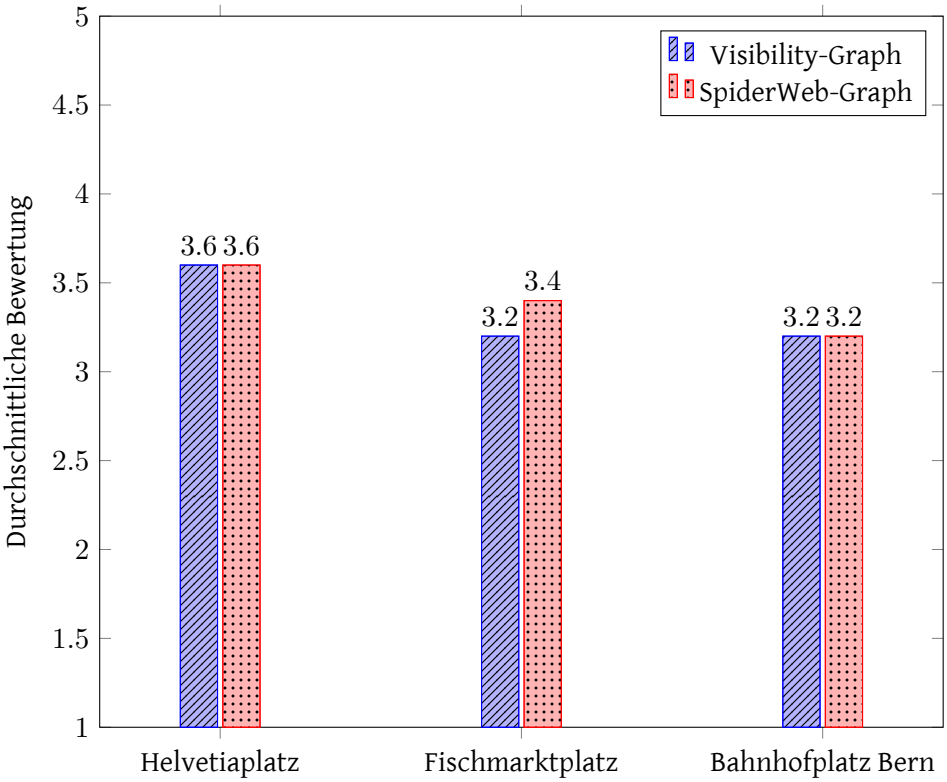

Abbildung 22: Durchschnittliche Bewertungen der Natürlichkeit auf den verschiedenen Plätzen

Tabelle 3: Resultat: Natürlichkeit

	Visibility-Graph	**SpiderWeb-Graph**
Helvetiaplatz		
Proband 1	4	3
Proband 2	3	4
Proband 3	4	3
Proband 4	4	4
Proband 5	3	4
Fischmarktplatz		
Proband 1	4	3
Proband 2	4	3
Proband 3	3	4
Proband 4	2	3
Proband 5	3	4
Bahnhofplatz Bern		
Proband 1	4	3
Proband 2	3	4
Proband 3	4	2
Proband 4	3	4
Proband 5	2	3
Resultat	**3.33**	**3.4**

3.3 Auswertung und Schlussfolgerung

Die einzelnen Werte sind in Tabelle 4 zusammen gefasst und gewichtet. Es zeigt sich, dass der Visibility-Graph knapp besser abschneidet, der Unterschied ist allerdings sehr gering und nicht eindeutig.

Bei der Bewertung der Natürlichkeit durch Probanden ergaben sich einige Erkenntnisse. So bewerteten einige Probanden eine direkte Route mit etwas “scharfen” Ecken — wie sie der Visibility-Graph erzeugt — natürlicher, während andere die etwas kurvigeren Routen des SpiderWeb-Graphs bevorzugten. Die Bewertungen der Natürlichkeit ist allerdings zu relativieren, da fünf Probanden in diesem Kontext nicht ausreichen, um eine abschliessende Erkenntnis zu gewinnen. Für einen ersten Eindruck genügt dies aber.

Abschliessend kann man sagen, dass beide Algorithmen gute Resultate bringen. Wenn die Verarbeitungszeit wichtig ist, bringt der Visibility-Graph Vorteile, der SpiderWeb-Graph erzeugt dafür tendenziell die etwas natürlicheren Routen.

Tabelle 4: Resultat der Evaluation Flächen-Algorithmus

ID	Titel	Relatives Gewicht (0-1)	Visibility-Graph	SpiderWeb-Graph
1	Verarbeitungszeit	0.3	4.54 / 1.36	3.9 / 1.17
2	zusätzliche Datenmenge	0.2	4.76 / 0.95	4.04 / 0.81
3	Natürlichkeit	0.5	3.33 / 1.67	3.4 / 1.7
Total			**12.6 / 3.98**	**11.36 / 3.68**

4 Umsetzungskonzept

Die Arbeit besticht durch einen grossen theoretischen Teil, welchem zu Beginn grosser Bedeutung beigemessen wird, da der Rest der Arbeit massgeblich von den Ergebnissen dieses Teils abhängt. So werden zuerst Algorithmen evaluiert und mit Tests in QGIS auf ihre Machbarkeit geprüft, um eine erste Bewertung der Algorithmen durchführen zu können. Dadurch ist klar, welche sich für die Implementierung eignen und welche verworfen werden können.

Parallel werden Abklärungen bezüglich der Fremdsysteme durchgeführt. Nach der Elaboration-Phase ist klar, auf welche Algorithmen gesetzt werden kann und ob die Fremdsysteme die gewünschten Anforderungen erfüllen und verwendet werden können.

Mit diesem Wissen lässt sich die Vorverarbeitung der OSM-Daten und das Backend, welches ein multimodales Routing anbietet, umsetzen.

Steht das Backend und kann eine Routing-Engine auf die von uns vorverabeitenden OSM-Daten operieren, wird die Visualisierung der Ergebnisse in einem QGIS-Plugin sichtbar gemacht.

5 Resultate

In diesem Kapitel werden die Resultate der Arbeit präsentiert. Die technische Beschreibung zur Implementation befindet sich in Teil II Kapitel 5.

5.1 Zielerreichung

In der Evaluations-Phase wurde der Stand der Technik analysiert (Kap. 2). Mit ersten Tests und Proof-of-Concept Implementationen in QGIS konnten dabei die Stärken und Schwächen der jeweiligen Algorithmen heraus gearbeitet werden. Es stellte sich schnell heraus, dass die beiden Ansätze Visibility-Graph und SpiderWeb-Graph sich am besten für unser Problem eignen. Parallel dazu wurden die Umsysteme analysiert, die für unsere Problemstellung eines multimodalen Routings benötigt werden (siehe Teil II Kap. 3). Dazu gehörte eine Analyse bestehender Routing-Engines und das Finden der ÖV-Haltestellen im näheren Umkreis.

Im nächsten Schritt wurden die Vorverarbeitung von OSM-Daten mit der Flächenoptimierung implementiert. Die beiden Algorithmen — Visibility-Graph und SpiderWeb-Graph — wurden dabei parallel implementiert. Dies ermöglichte es uns, beide Ansätze optimal miteinander zu vergleichen (siehe Kapitel 3).

Zusammen mit der Vorverarbeitung haben wir einen Service [23] implementiert, der mit Hilfe der Routing-API von search.ch [9] ein Routing mit öffentlichen Verkehrsmitteln ermöglicht, wobei für das Fussgänger-Routing unsere optimierten Daten der Vorverarbeitung verwendet werden, um ein natürliches Fussgänger-Routing über offene Flächen zu erreichen. Dabei werden von einem beliebigen Startpunkt aus mehrere Haltestellen in der Umgebung gesucht und mit der Kombination von Fussgänger- und ÖV-Routing die optimale Route ermittelt.

Die Koordinaten der Kanten werden dabei so optimiert, dass die Kante in die richtige Fahrtrichtung angesteuert werden kann.

Damit unser Service getestet und visualisiert werden kann, wurde ein Plugin für QGIS entwickelt [24]. Dieses ermöglicht es, mit unserem Service Routen für den öffentlichen Verkehr interaktiv zu berechnen und visualisieren.

5.2 Ausblick: Weiterentwicklung

Die von uns implementierte Vorverarbeitung für OSM-Daten kann als Referenz dienen, um in Zukunft die Optimierung für Fussgänger-Flächen in bestehende Routing-Engines einzubauen. Es wäre sinnvoll,

die Algorithmen direkt in Routing-Engines zu integrieren, statt in einem sepparaten Schritt zuerst OSM-Daten aufbereiten zu müssen.

Die jetzige Lösung bietet noch weiteren Raum für Optimierung. So können im Moment einige Plätze nicht verarbeitet werden, weil zu wenig Einstiegspunkte existieren, obwohl in der Realität der Platz problemlos begehbar wäre. Ein ähnliches Problem besteht auch, wenn mehrere Fussgänger-Flächen direkt aneinander liegen. Ansätze für Lösungen dazu werden in den Kapiteln 2.3.1 respektive 2.2.5 diskutiert.

Die jetzige Lösung mit Python stösst mit der Performanz an ihre Grenzen. Es ist denkbar, eine Lösung mit PostGIS oder C++ zu realisieren.

5.3 Dank

Wir möchten folgenden Personen für ihre Unterstützung und Mitwirkung bei dieser Arbeit danken:

Prof. Stefan Keller, IFS Institut für Software, für die Zeit, Ressourcen, Kontakte, Know-How und Unterstützung, von welcher wir jederzeit profitieren konnten.

Christian Helbling, localsearch, für den Erfahrungsaustausch im Bereich der Fahrplandaten und search.ch.

Prof. Dr. Olaf Zimmermann, IFS Institut für Software, für die wertvolle Expertise in Sachen Software-Architektur.

Mitarbeiter, IFS Institut für Software, für den regen Know-How-Austausch und die Unterstützung bei der Produktivsetzung.

II SW-Projektdokumentation

1 Überblick

Der Teil I Technischer Bericht verfolgt das Ziel, dem Leser einen Überblick über die Problemstellung und über den Inhalt der Arbeit zu geben. Dabei wurde dem Stand der Technik besondere Beachtung geschenkt. Der Teil SW-Projektdokumentation legt den Fokus auf die konkrete Umsetzung von Plaza-Route.

2 Anforderungsspezifikation

2.1 Use Cases

Im folgenden sind die funktionalen Anforderungen an PlazaRoute mit all seinen Komponenten, welche im Kapitel 4 aufgeführt sind, als Use Cases im Brief-Format beschrieben. Zur Übersicht ist das Use Case Diagramm in Abbildung 23 zu betrachten.

2.1.1 Aktoren

Tabelle 5: Aktoren

Aktor	Beschreibung und Interessen
User	Ein User ist ein Fussgänger, welcher schnellstmöglich von Punkt A zu Punkt B kommen möchte. Er fungiert in der Komponente QGIS-Plugin.
Admin	Ein Admin ist daran interessiert, dass aktuelle Daten dem Aktor User zur Verfügung stehen. So möchte er Daten vorverarbeiten und dem Aktor User diese zur Verfügung stellen können.

2.1.2 UC01: schnellstmöglicher Weg finden

Aktoren: *User*

Include: UC02: Flächen im urbanen Raum natürlich begehen, UC03: ÖV-Verbindung finden

Nachdem der User einen Start- und Endpunkt (geografische Standorte) angegeben hat, erhält er den schnellstmöglichen Weg, welcher mit öffentlichen Verkehrsmitteln und zu Fuss machbar ist.

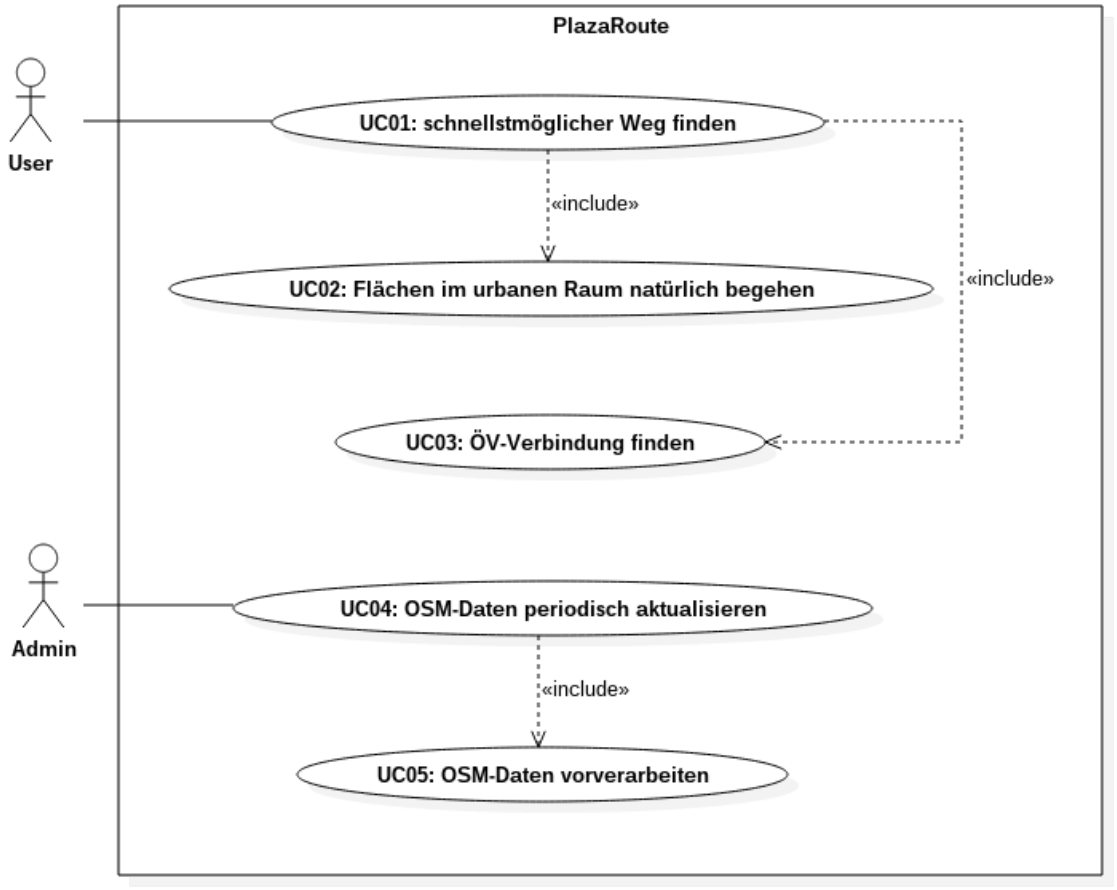

Abbildung 23: Use Case Diagramm

2.1.3 UC02: Flächen im urbanen Raum natürlich begehen

Aktoren: *User*

Der User wird beim Fussgänger-Routing über Flächen im urbanen Raum auf direktem Weg natürlich (Definition im Kapitel 3.1.3 im Teil I) geroutet. Dies betrifft insbesondere das Umgehen von Hindernissen.

2.1.4 UC03: ÖV-Verbindung finden

Aktoren: *User*

Der User wird zu Fuss zu einer ÖV-Haltestelle geroutet, welche sich in einem Umkreis von 1 km befindet. Falls mehrere ÖV-Haltestellen verfügbar sind, wird die ÖV-Haltestelle gewählt, welche die kürzeste Reisezeit (Fussweg + ÖV-Weg) hat.

2.1.5 UC04: OSM-Daten periodisch aktualisieren

Aktoren: *Admin*

Include: UC05: OSM-Daten vorverarbeiten

Der Admin aktualisiert die OSM-Daten, auf welche in UC01: schnellstmöglicher Weg finden geroutet wird, periodisch und integriert sie in die Routing-Engine.

2.1.6 UC05: OSM-Daten vorverarbeiten

Aktoren: *Admin*

Der Admin führt die Vorverarbeitung der OSM-Daten durch. Unter Vorverarbeitung versteht man das Einzeichnen von Fusswegen auf Flächen im urbanen Raum. Diese Daten werden der Routing-Engine übergeben und ermöglicht, dass die Routing-Engine den schnellstmöglichen Weg über Flächen im urbanen Raum finden kann.

2.2 Nicht-funktionale Anforderungen

2.2.1 NFA01: Docker-Images

PlazaRoute wird in Docker-Images ausgeliefert, um eine Verfügbarkeit auf unterschiedlichen Systemen sicherstellen und um das Deployment vereinfachen zu können.

2.2.2 NFA02: austauschbare Routing-Engine

Die Routing-Engine, welche von PlazaRoute verwendet wird, soll austauschbar sein. Dabei soll nur die konkrete Implementation des Services, welche fürs Routing genutzt wird, angepasst werden müssen.

2.2.3 NFA03: periodische Aktualisierung der OSM-Daten

Die OSM-Daten werden wöchentlich für ein optimiertes Fussgänger-Routing über Flächen im urbanen Raum vorverarbeitet und der Routing-Engine übergeben.

2.2.4 NFA04: Dauer der Vorverarbeitung

Die Vorverarbeitung der OSM-Daten (Optimierung auf Flächen im urbanen Raum) darf maximal 2 Stunden dauern.

2.2.5 NFA05: zusätzliche Datenmenge

Die zusätzlichen Daten für die Optimierung (zusätzlich eingezeichnete Wege über Flächen im urbanen Raum) dürfen nicht mehr als 1% des Kartenmaterials betragen.

2.2.6 NFA06: Dauer des Unterbruchs bei OSM-Daten-Integration

PlazaRoute ist während dem Integrieren der neuesten OSM-Daten in die Routing-Engine maximal 2 Stunden nicht verfügbar. Der User wird nicht über den Unterbruch informiert.

3 Analyse

In diesem Kapitel werden Abhängigkeiten und Umsysteme analysiert, die potentiell benötigt werden. Neben der Evaluation der Routing-Engine (3.1) gehören Overpass (3.2) und search.ch (3.3) zu den wichtigsten Systemen in unserer Applikation.

3.1 Evaluation Routing-Engine

Eine Evaluation der gängigen OSM-Routing-Engines wurde in [25] ausgiebig durchgeführt. Bei unserer Evaluation beschränken wir uns auf die drei "grössten"Routing-Engines *OSRM*[26], *Valhalla*[27] und *GraphHopper*[5]. Die Auswahl fiel auf diese drei, weil sie von den in [25] evaluierten Routing-Engines anhand der Github-Metriken mit Abstand die grösste Verbreitung haben.

Bei einem Test mit einer von uns vorverarbeiteten OSM-Datei gab es bei *Valhalla* sofort Probleme, weil wir negative OSM-IDs verwenden, um Konflikte mit bestehenden und zukünftigen Daten zu vermeiden. Für die weitere Evaluation beschränken wir uns daher auf die beiden Routing-Engines *GraphHopper* und *OSRM*.

Die Engines werden an folgenden Kriterien gemessen:

3.1.1 Infrastruktur-Integration

Es wird analysiert, wie einfach die Routing-Engine aufgesetzt werden kann. Ziel ist es, dass die Routing-Engine in einem eigenen Docker-Image (siehe NFA01: Docker-Images) auf dem Server läuft.

GraphHopper und *OSRM* sind bereits als Docker-Image verfügbar und können somit einfach auf einem beliebigen Server gestartet und verwendet werden.

3.1.2 Applikation-Anbindung

Da mit der Routing-Engine kommuniziert werden muss, wird geprüft, wie einfach die Routing-Engines an eine Applikation angebunden werden können.

Beide Routing-Engines bieten zusätzlich ein fertiges HTTP-Backend an, welches für das Routing genutzt werden kann. Somit können beide Routing-Engines auf die gleiche Art und Weise eingebunden werden.

3.1.3 OSM-Daten-Integration

Für die Flächentraversierung müssen eigene OSM-Daten generiert werden. Diese vorverarbeiteten OSM-Daten müssen der Routing-Engine übergeben werden.

Sowohl *OSRM* als auch *GraphHopper* können beim Starten der Graphen-Aufbereitung eine OSM-Datei übergeben werden.

3.1.4 Geschwindigkeit der Vorverarbeitung

Die von uns erzeugte OSM-Datei wird der Routing-Engine zur Verarbeitung übergeben, um daraus einen Routing-Graphen zu generieren. Da dieser Prozess für jeden Durchlauf unserer Plaza-Vorverarbeitung aufs Neue geschehen muss, soll dies möglichst schnell sein.

Bei einem Test mit einer von uns vorverarbeiteten OSM-Datei brauchte *GraphHopper* nur wenige Minuten, während *OSRM* über 30 Minuten für die Verarbeitung beanspruchte.

3.1.5 Konfigurationsmöglichkeiten

Für unsere Zwecke ist es sinnvoll, wenn wir die Routing-Engines auf das Fussgänger-Routing konfigurieren können. So sind z.B. Contraction Hierarchies nicht notwendig, da wir keine grossen Routen berechnen.

GraphHopper bietet die meisten Konfigurationsmöglichkeiten. Contraction Hierarchies können deaktiviert werden und für die Shortest-Path-Berechnung werden die A*- und Dijkstra-Algorithmen [28] [29] unterstützt. *OSRM* bietet ebenfalls einen Modus ohne Contraction Hierarchies, allerdings nur mit dem Dijkstra-Algorithmus.

3.1.6 Fussgänger-Profil

Für das Routing werden bestimmte Profile verwendet, um das Routing für verschiedene Anwender wie Autofahrer oder Fussgänger zu optimieren. Für uns ist es von Vorteil, wenn eine Routing-Engine bereits ein vorkonfiguriertes Profil für Fussgänger mitbringt.

GraphHopper und *OSRM* bringen beide bereits ein vorkonfiguriertes Fussgänger-Profil mit.

3.1.7 Resultat

Bezüglich der Integration unterscheiden sich *GraphHopper* und *OSRM* kaum. Der Entscheid fiel auf *GraphHopper*, da die Vorverarbeitung deutlich schneller ist und die Konfigurationsmöglichkeiten uns mehr Freiheiten bieten. Mit der Auswahl zwischen Dijkstra [29] und A* [28] können wir auch vergleichen, welcher Algorithmus besser für Flächen-Traversierungen geeignet ist.

3.2 nächste ÖV-Haltestellen finden

Das grundlegende Ziel ist es, von einem Startpunkt aus eine Destination zu Fuss und mit dem öffentlichen Verkehr zu erreichen. Dabei sollen die ÖV-Haltestellen in einem zu Fuss machbaren Umkreis berücksichtigt werden. Von diesen ÖV-Haltestellen ausgehend wird das ÖV-Routing an die Zieldestination durchgeführt.

Für die Anforderung 1.3.9 bietet sich Overpass an. Overpass ermöglicht es, über eine umfassende API selektiv Daten von OSM zu beziehen. Dabei besteht die Option, die Overpass Query Language (QL) oder XML-Abfragen zu verwenden. Die Suche lässt sich nach allem einschränken, was der Mapper in den OSM-Daten spezifizieren kann. So ist das Filtern nach Objekttyp, Keys, Tags, etc. unbeschränkt möglich und bietet so eine hohe Flexibilität. Overpass liefert die Resultate im JSON- oder XML-Format.

Für eine einfache Intergration in Python gibt es Overpass Wrapper. Dabei wurden zwei Libraries berücksichtigt, namentlich *overpass-api-python-wrapper* und *OverPy*. Die Libraries haben einen ähnlich häufigen Updatezyklus. Für beide wurde ein Proof of Concept implementiert, welcher die ÖV-Haltestellen im einem kleinen Umkreis vom Stadelhofen, Zürich, Schweiz abfragt.

Die Entscheidung fiel dabei auf OverPy. Ausschlaggebend war die ausführlichere Dokumentation und dass OverPy Klassen für Nodes, Ways, etc. und Hilfsfunktionen, welche das Ganze übersichtlich halten, anbietet. Bei *overpass-api-python-wrapper* besteht der Nachteil, dass das JSON-Resultat der Abfrage selber geparsed und verarbeitet werden muss.

```
1  def get_public_transport_stops(start_position: tuple) -> dict:
2
3      bbox = _parse_bounding_box(*start_position, BUFFER)
4
5      query_str = f"""
6          [bbox:{bbox}];
7          node["public_transport"="stop_position"];node["highway"="bus_stop"];
8          out body;
9          rel["type"="public_transport"];
10         out center;
11         """
12
13     public_transport_stops = _query(query_str)
14     # ...
```

Listing 3: ÖV-Haltestellen von OSM mit Overpass beziehen

In Listing 3 ist zu sehen, wie für eine Bouding Box, welche im Süden durch den minimalen Breitengrad, im Westen durch den minimalen Längengrad, im Norden durch den maximalen Breitengrad und im Osten durch den maximalen Längengrad begrenzt ist, die ÖV-Haltestellen abgefragt werden. Die Werte der Bounding Box werden dabei aufgrund eines Ausgangspunkt und einem konfigurierbaren Buffer berechnet. In diesem Beispiel wird für die Abfrage die erwähnte Overpass QL verwendet.

Es ist ebenfalls möglich, eine Umkreissuche mit `around` durchzuführen. Dies hat Performance-Nachteile und es gibt keine entscheidenden Gründe, warum es vorteilhafter sein sollte, wenn man einen Kreis statt ein Rechteck um einen Ausgangspunkt zieht.

3.3 Search.ch Anbindung

Search.ch bietet für Fahrplan-Abfragen eine API [9] an. So ist auch eine ÖV-Routensuche möglich. Diese ist zum aktuellen Zeitpunkt unter [9] verfügbar und nimmt die in Tabelle 6 spezifizierten und für uns relevanten Parameter entgegen. In einer ersten Version wird nur `from`, `to` und `date` verwendet.

Also Antwort erhält man ein JSON-Objekt, welches eine Liste an möglichen Verbindungen beinhaltet.

Tabelle 6: Search.ch Route API Spezifikation; Teilauszug von [9]

Parameter	Zwingend	Beschreibung	Beispiel	Default
`from`	`true`	Abfahrtsstation	Zürich, Sternen Oerlikon	
`to`	`true`	Abfahrtsstation	Zürich, Hallenbad Oerlikon	
`date`	`false`	Datum	29.10.2017	today
`time`	`false`	Zeit	15:00	now
`time_type`	`false`	date/time als Abfahrts- oder Ankunftszeit	arrival	depart

In einer zurückgelieferten Verbindung sind alle Teilstrecken definiert. Die erste Teilstrecke bestimmt den Ausgangspunkt der ÖV-Verbindung. Diese ist von besonderer Relevanz. In 1.2.8 wurde aufgezeigt, dass es normalerweise mehrere Kanten mit dem gleichen Namen geben kann und nun zur richtigen geroutet werden muss. So kann mithilfe der `stopid` der Start- und Ausstiegshaltestelle, welche dem OSM-Tag `uic_ref` entspricht, und Overpass die Kante in die richte Fahrtrichtung ermittelt werden.

3.3.1 Einschränkungen

Die Anzahl täglicher Anfragen ist für den freien Gebrauch auf 1000 beschränkt.

4 Architektur

In diesem Kapitel wird die Architektur unserer Applikation und die Schnittstellen zu den Umsystemen besprochen. Als Anhaltspunkt wird das C4 Modell [30] von Simon Brown verwendet. In einem ersten Schritt wird unsere Applikation in den Kontext des grösseren Systems gesetzt. Anschliessend teilen wir das System PlazaRoute in einzelne Container und den zentralen Container Plaza Routing in einzelne Komponenten auf.

4.1 Systemkontext

Abbildung 24 zeigt das System PlazaRoute mit den Umsystemen auf. Beim Betrachten der C4-Diagramme ist zu beachten, dass diese nicht der UML-Spezifikation folgen. Gestrichelte Pfeile bedeuten, dass eine Anfrage in Richtung der Pfeilspitze an ein System geht. Der Datenfluss läuft in die entgegengesetzte Richtung der Pfeile.

Der User bedient die QGIS Desktop Applikation mit dem von uns entwickelten Plugin. Dieses leitet die die Eingabe der Start- und Zielkoordinaten an das System PlazaRoute weiter. Als Antwort sendet PlazaRoute eine Routenbeschreibung an das QGIS-Plugin zurück, welches diese im QGIS darstellt.

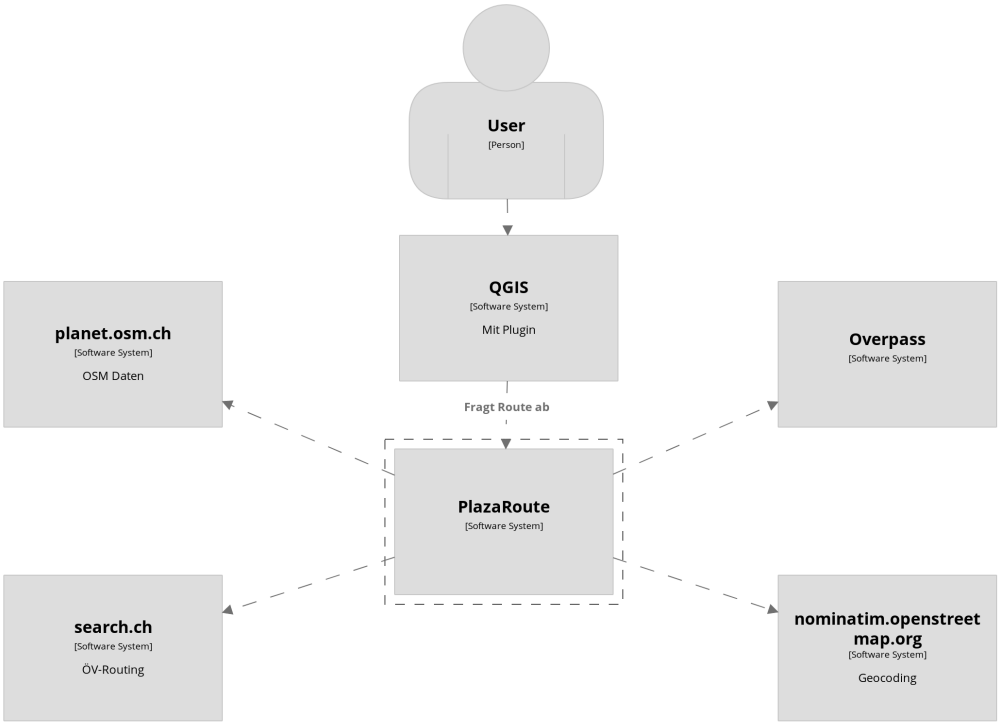

Abbildung 24: System PlazaRoute im Kontext mit Umsystemen; Grafik erstellt mit *Structurizr Express*[31]

4.2 PlazaRoute Container

In Abbildung 25 zoomen wir in das System PlazaRoute hinein und teilen es in drei Container auf, die logisch voneinander getrennt sind. So könnten die Container auch verteilt deployed werden.

In den nachfolgenden Abschnitten werden die drei Container näher beleuchtet.

4.2.1 Plaza Vorverarbeitung

Bevor die Routing-Engine ihre Routing-Funktion ausführen kann, muss diese zuerst aus dem OSM-Datensatz einen Routing-Graph generieren. Unsere Hauptaufgabe besteht darin, diesen Routing-Graphen für Fussgänger-Routing über Flächen im urbanen Raum zu optimieren. Ein Ansatz wäre, den Graphen nach dem Generieren zu verändern. Dies ist aber schwer realisierbar, da die meisten Routing-Engines die Graphen in eigenen (binären) Datenstrukturen ablegen. So wäre unsere Implementation auch stark an eine einzelne Routing-Engine gekoppelt.

Ein zweiter Ansatz die Integration unserer Optimierung in die Verarbeitung der Routing-Engines selbst. Auch da wären wir wieder stark an eine spezifische Routing-Engine gekoppelt. Wir haben uns stattdessen entschieden, beim Input der OSM-Daten anzusetzen. Dazu werden die Rohdaten

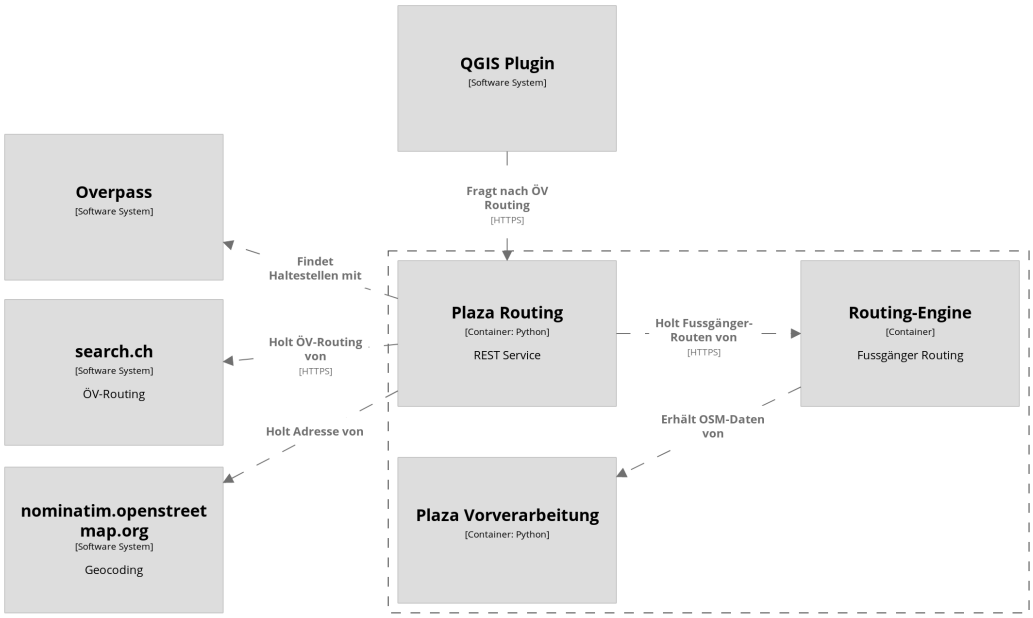

Abbildung 25: Container Diagramm von PlazaRoute; Das System PlazaRoute befindet sich in der eingerahmten Box; Grafik erstellt mit *Structurizr Express*[31]

zuerst eingelesen und nach Fussgänger-Flächen abgesucht (OSM Importer). Mit unserem Algorithmus werden neue Fusswege eingetragen (Plaza Optimizer). Diese neu erzeugten Kartendaten werden dann wieder mit den ursprünglichen Rohdaten verschmelzt (OSM Merger). Erst dann generiert die Routing-Engine daraus den Routing-Graphen. Das Vorgehen ist in Abbildung 26 schematisch aufgezeigt.

Abbildung 27 zeigt die einzelnen Komponenten für die Vorverarbeitung der OSM-Daten auf. Darin werden die Komponenten des Containers Plaza Vorverarbeitung (siehe Abbildung 25) in der gestrichelten Box dargestellt.

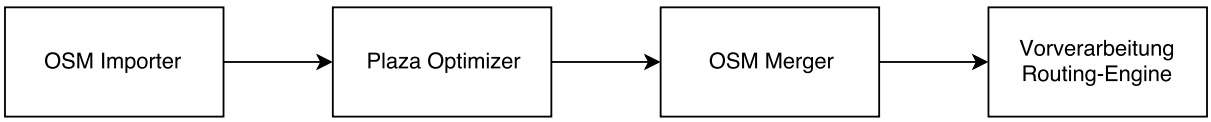

Abbildung 26: Datenfluss-Diagramm der Vorverarbeitung von OSM-Daten bis zur Übergabe an die Routing-Engine; Grafik erstellt mit *draw.io*

OSM Importer

Für unser optimiertes Routing wird in regelmässigen Abständen der neueste OSM-Datensatz [8] der Schweiz geladen. Die OSM-Importer Komponente liest das komplette für uns relevante Kartenmaterial (z.B. die Schweiz) als Protocolbuffer Binary Format (PBF) ein und sucht dabei nach Flächen, die wir bearbeiten wollen.

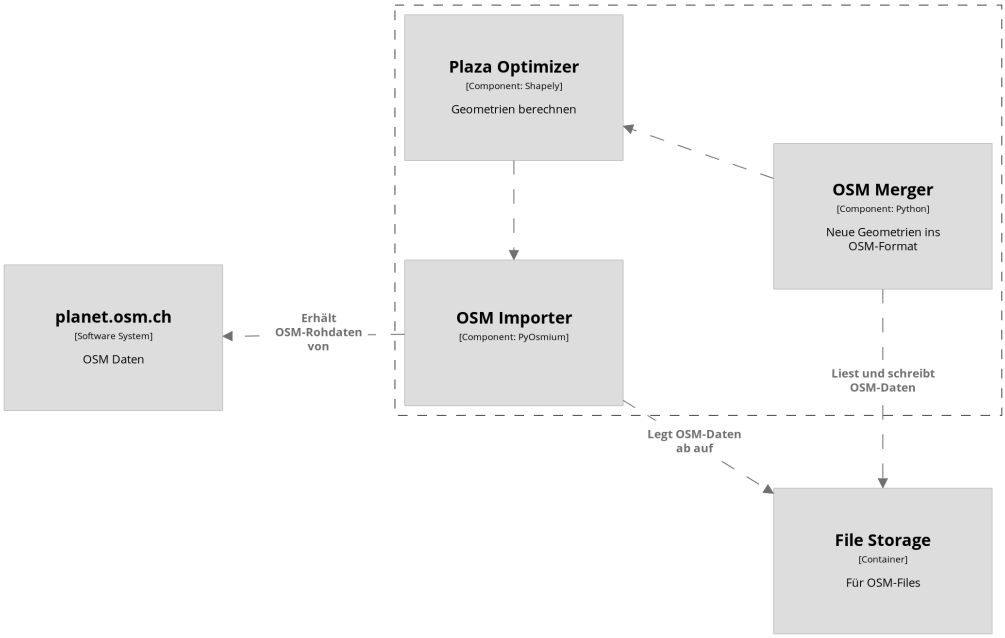

Abbildung 27: Komponentendiagramm der Plaza Vorverarbeitung (eingerahmt); Grafik erstellt mit *Structurizr Express*[31]

Dazu werden *Osmium* und die dazugehörigen Python-Bindings *pyOsmium*[32] verwendet. Osmium erkennt automatisch Flächen aus OSM Multipolygone oder Relationen. Mit einem eigenen Handler können wir dabei gleich das Einlesen des Files auf die für uns interessanten OSM-Objeke beschränken.

Plaza Optimizer

Die mit Osmium importierten OSM-Daten sind noch reine OSM-Objekte, auf denen keine Geometrie-Berechnungen angewendet werden können. Dazu wird die Python-Library *Shapely*[33] verwendet. Shapely kann mit Geometrien umgehen und Algorithmen von Geometry Engine, Open Source (GEOS) wie `intersection` und `contains` darauf anwenden.

Um die mit Osmium importierten Objekte in Shapely zu verwenden, werden diese ins Well-known Binary (WKB) Format übersetzt und Shapely übergeben, wie in Listing 4 gezeigt.

```
1 # convert to WKB
2 wkbfab = osmium.geom.WKBFactory()
3 wkb = wkbfab.create_linestring(osmium_area)
4
5 # load into shapely object
6 shapely_area = shapely.wkb.loads(wkb, hex=True)
```

Listing 4: Übergabe von Osmium-Objekten zu Shapely für die Weiterverarbeitung

OSM Merger

Der OSM Merger ist dafür verantwortlich, unsere erzeugten Geometrien (Fusswege) wieder in das OSM-Kartenmaterial einzupflegen, um es anschliessend der Routing-Engine zur Verarbeitung zum Routing-Graphen zu übergeben.

Die durch unseren Algorithmus erzeugten Wege durch Flächen (in Shapely Datenstrukturen) sollen nun wieder zurück ins OSM-Format geschrieben werden. Dazu wird wie beim OSM Importer Osmium verwendet, mit dem Geometrien in eine OSM-Datei geschrieben werden können.

In einem weiteren Schritt müssen unsere optimierten Wege in das bestehende Strassennetz eingebunden werden, damit die Routing-Engine diese auch beachtet. Dazu werden die Einstiegspunkte der von uns erzeugten Fusswege in den bestehenden Strassen und Wegen referenziert, die an diesem Punkt auf die Fläche treffen. Somit werden sie topologisch direkt miteinander verbunden.

Als letzten Schritt führen wir die erzeugten Fusswege und die modifizierten Strassen wieder mit der ”grossen”OSM-Datei zusammen, welche ganz am Anfang importiert wurde. Dazu wird das Command-Line-Tool Osmosis [34] verwendet.

4.2.2 Plaza Routing

Der Plaza Routing Container (siehe Abbildung 25) ist für das Koordinieren und Verarbeiten von Routing-Anfragen verantwortlich. Er bietet für das QGIS-Plugin eine API an. Mit Hilfe von verschiedenen Drittsystemen und der Routing-Engine (für Fussgänger-Routing) wird eine komplette Route mit Fahrplan erstellt und dem QGIS-Plugin oder anderen Konsumenten übergeben.

In diesem Abschnitt wird vertieft in die einzelne Bestandteile von Plaza Routing eingegangen. So sind nachfolgend die einzelnen Schichten und Konstrukte aufgeführt und beschrieben. Zur Übersicht findet man das Schichten-Diagramm in Abbildung 28. Die Verantwortlichkeiten der Drittsystemen wird in den jeweiligen Schichten angesprochen.

app

Hierbei handelt es sich um den zentralen Einstiegspunkt in die Komponente *PlazaRouting*. Die Komponenten wird in dieser Schicht konfiguriert und initialisiert.

api

In dieser Schicht wird eine Web-API für den gesamten externen Einstieg in das Plaza Routing exponiert. Diese API wird primär vom QGIS-Plugin verwendet. Eine Einschränkung bezüglich der Konsumenten gibt es nicht.

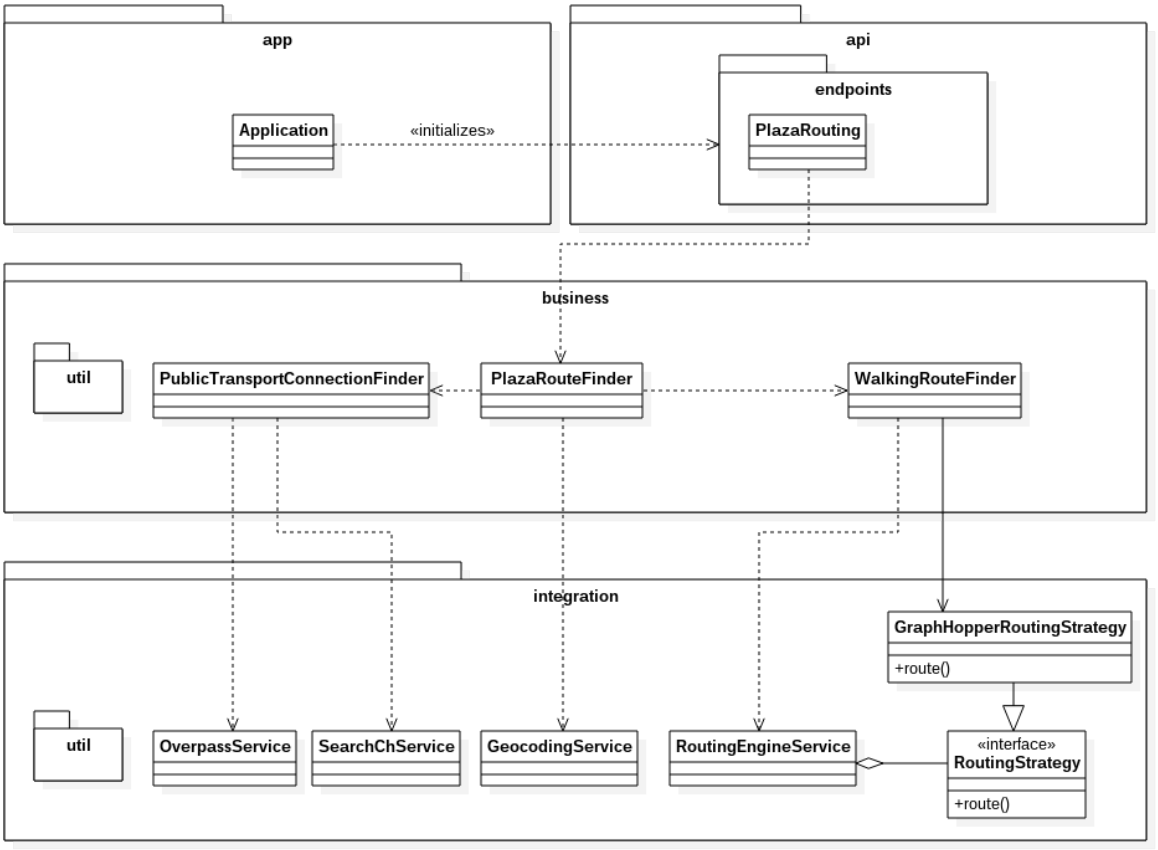

Abbildung 28: Schichten-Diagramm Plaza Routing

Die OpenAPI Specification (OAS) [35] der API ist unter [36] verfügbar. SwaggerUI kann unter [37] eingesehen werden. Die API-Spezifikation wird in einem Git-Repository gehalten [38]. Änderungen werden somit automatisch auf der Github-Page [36] publiziert. Dies hat die Vorteile, dass es keine Abweichung zwischen Spezifikation und Dokumentation geben kann, mit Swagger [39] die Schnittstelle bereits ideal beschrieben ist und die Haltung in einem öffentlichen Repository eine Diskussions-Plattform für Konsumenten bietet.

business

In der Business-Schicht ist die Business-Logik der API-Abfragen vorhanden. Das PlazaRouting wird logisch in zwei Bereiche *PublicTransportConnectionFinder* und *WalkingRouteFinder* aufgeteilt. Die Koordination übernimmt dabei *PlazaRouteFinder*. So wird das Design-Prinzip *Separation of Concerns* ideal umgesetzt, da *PublicTransportConnectionFinder* nur mit Services kommuniziert, welche für das ÖV-Routing notwendig sind und *WalkingRouteFinder* die Kommunikation mit der Routing-Engine übernimmt.

integration

In den nachfolgenden Abschnitten sind die Integration-Services und ihre Notwendigkeit beschrieben. In dieser Architektur werden Komponenten, welche mit Drittsystemen oder -komponenten kommu-

nizieren als *Services* bezeichnet und der Integration-Schicht angegliedert.

Search.ch Service

Der *SearchChService* kommuniziert mit dem Fahrplan-API von search.ch [9] und bezieht die Fahrplan-Daten für eine bestimmte Ausgangsstation und Destination.

Overpass Service

Der *OverpassService* übernimmt mithilfe der QL die Kommunikation mit Overpass [10]. Der Hauptfokus liegt auf dem Extrahieren von ÖV-Haltestellen in einem gegebenen Umkreis und der geografischen Position von Kanten basierend auf Daten, welche aus dem Search.ch Service gewonnen werden.

Geocoding Service

Der *GeocodingService* liefert mithilfe von Nominatim [11] für eine Adresse eine Koordinate.

Routing-Engine Service

Der *RoutingEngineService* ist für das Fussgänger-Routing zuständig und kommuniziert mit einer Routing-Engine. Die Architektur ist mit dem Strategy-Pattern [40] so gewählt, dass die Routing-Engine einfach ausgetauscht werden kann.

4.2.3 Routing-Engine

In diesem Container ist die Routing-Engine angesiedelt, welche im Kapitel 3.1 evaluiert wurde. Sie ist reiner Lieferant und operiert auf den OSM-Daten, welche der Container Plaza Vorverarbeitung liefert.

4.3 QGIS-Plugin

Das QGIS-Plugin ist unabhängig vom System PlazaRoute und läuft auf dem Client des Benutzers. Die Kommunikation mit dem Container Plaza Routing erfolgt über einen REST-Service auf Level 2 des Maturity Model [41]. Das Plugin ist reiner Konsument und zeigt auf, welchen Mehrwert PlazaRoute bieten kann. In Abbildung 29 ist das Klassen-Diagramm zu sehen. Herausheben kann man, dass das Plugin in fünf Komponenten zerlegt wird. *PlazaRouteRoutingService* kommuniziert mit dem REST-Service (siehe Abschnitt api). *PlazaRouteMapTool* übernimmt die Interaktion mit der Karte und führt das Zeichnen der Route mit *PlazaRouteRouteDrawer* durch. *PlazaRouteDirectionsGenerator* generiert eine Navigationsanweisung für die zurückgelieferte Route. Die zentrale Steuerung übernimmt das Dockwidget.

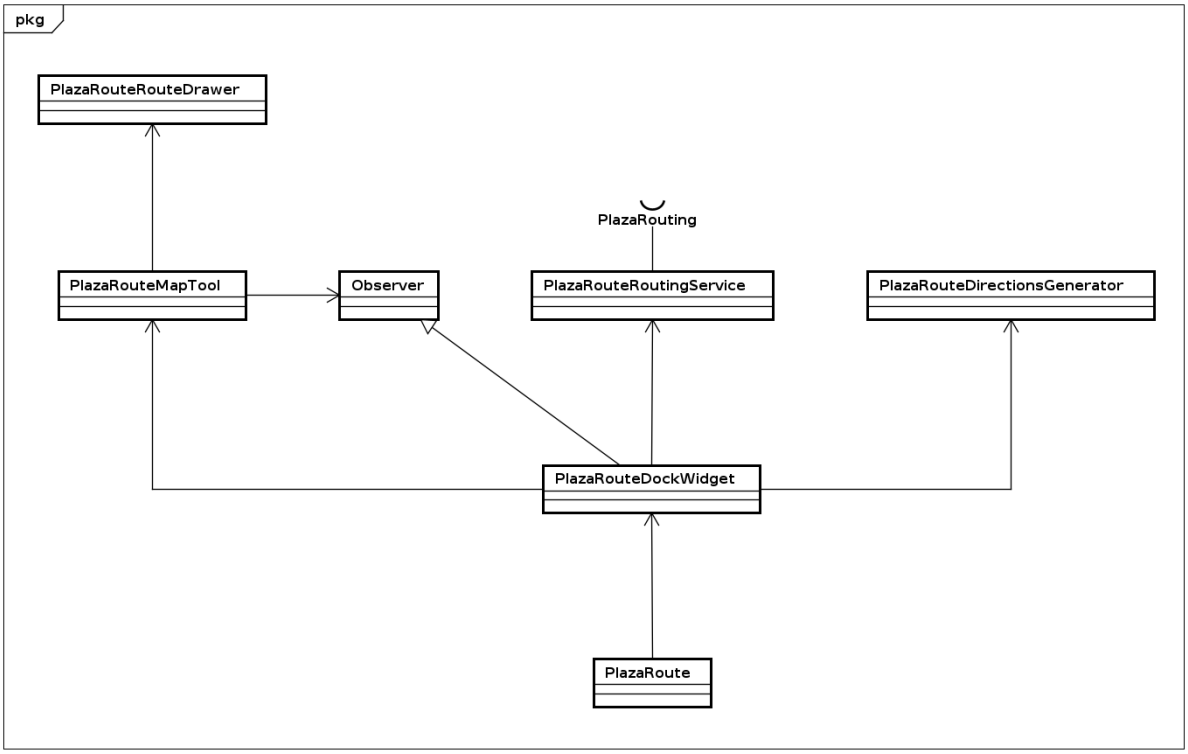

Abbildung 29: Klassen-Diagramm Plaza Route QGIS-Plugin

5 Implementation

5.1 PlazaRoute Container

In diesem Kapitel wird die Implementation der einzelnen Container beschrieben, wie sie in 4.2 definiert wurden.

5.1.1 Plaza Vorverarbeitung

In diesem Kapitel wird die Umsetzung der Plaza Vorverarbeitung, wie sie in Kapitel 4.2.1 beschrieben ist, erläutert. Dabei wird das Zusammenspiel der verschiedenen Komponenten aufgezeigt und einige Implementationsdetails diskutiert.

Die Code-Dokumentation der Plaza Vorverarbeitung ist unter [42] veröffentlicht.

Struktur und Ablauf der Komponenten

Die Python-Pakete wurden anhand der Architektur und des Datenflusses, wie in Abbildung 26 zu sehen, strukturiert. So wurde für jede Komponente *OSM Importer*, *Plaza Optimizer* und *OSM Merger* ein eigenes Python-Subpackage erstellt.

Angestossen wird die Vorverarbeitung vom `__main__.py` file, das vom Benutzer mit der Kommandozeile aufgerufen wird.

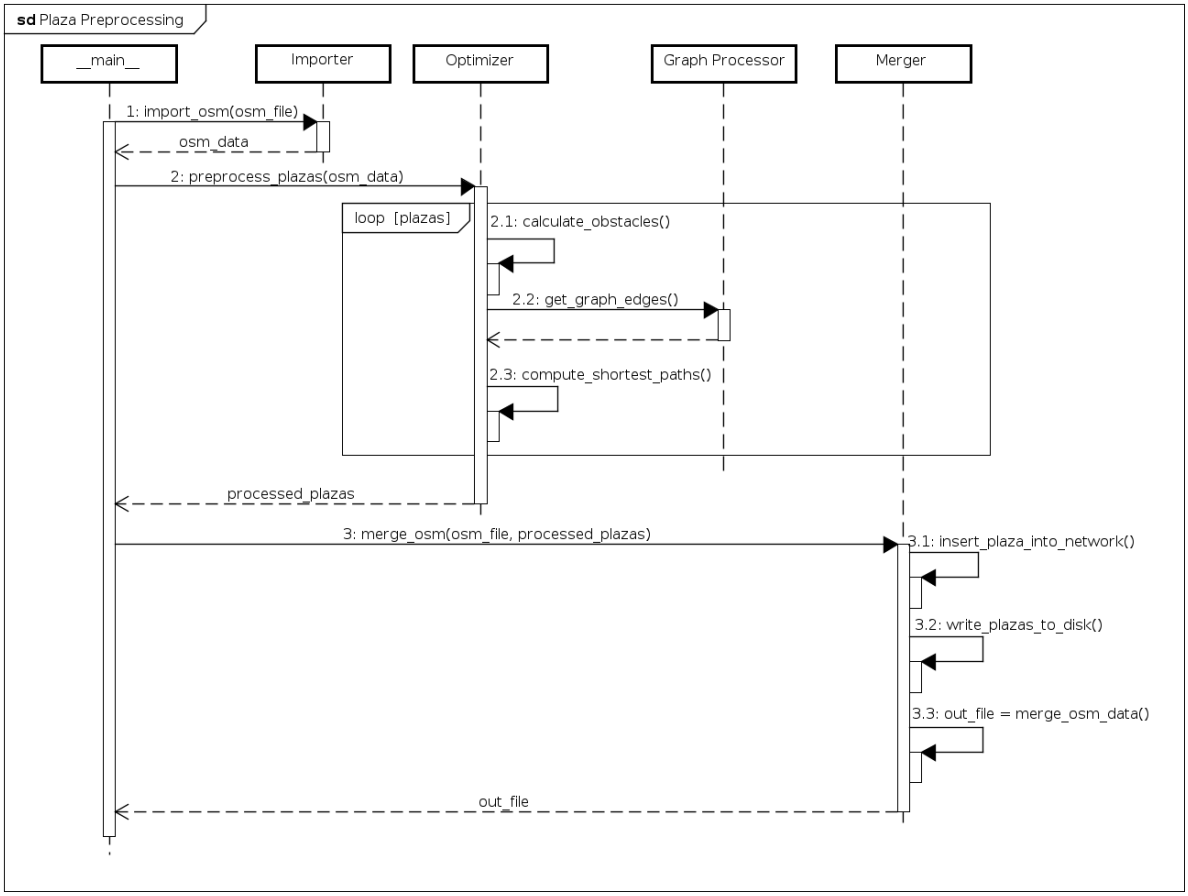

Abbildung 30: Sequenz-Diagramm Plaza Vorverarbeitung

In Abbildung 30 ist der Ablauf in einem Sequenz-Diagramm dargestellt. Dies entspricht nicht komplett dem Programmcode, sondern soll nur zur Veranschaulichung des Zusammenspiels der einzelnen Komponenten dienen.

Im Folgenden wird auf die einzelnen Komponenten eingegangen.

Importer

Das Importer-Package liest aus einer gegebenen OSM-Datei (Dateityp OSM oder PBF) die für uns relevanten Daten mit Hilfe von Pyosmium [32] ein. Während dem Einlesen werden die benötigten Daten direkt heraus gefiltert. Für uns relevant sind vor allem Plazas, die anhand von konfigurierbaren Tags bestimmt werden. Dies ist in Listing 5 veranschaulicht. Zusätzlich werden alle Strassen benötigt, um die Plazas später wieder an das Strassennetz anzuschliessen (siehe Abschnitt Merger). Punkte und Gebäude werden für die Berechnung von Hindernissen auf den Plazas genutzt.

Alle ausgelesenen Daten werden in ein `OSMHolder` Objekt verpackt und zurück gegeben.

```
class PlazaHandler(osmium.SimpleHandler):
    def __init__(self):
        osmium.SimpleHandler.__init__(self)

    def area(self, a):
        if isPlaza(a):
            # import as plaza
```

Listing 5: Einlesen von OSM Daten mithilfe von *Osmium*; Filterung auf für uns relevante Flächen

Optimizer

Der Optimizer übernimmt die Kernfunktion der Vorverarbeitung, die Optimierung von Fussgänger-Flächen.

Das `OSMHolder`-Objekt aus dem Importer wird dem Optimizer übergeben. Als erstes wird für Strassen, Punkte und Gebäude je ein R-Tree [43] Index erstellt. Dies erlaubt es uns, effizient geometrische Objekte zu finden, die sich mit dem Plaza schneiden. Ohne einen Index müsste für jedes Objekt (Linie, Punkt oder Gebäude) jeweils der komplette Datensatz durchsucht werden ($O(n)$). Mit einer Index-Suche ($O(\log n)$) werden die Datensätze auf ein paar wenige mögliche Treffer eingegrenzt.

Für jedes Plaza werden folgende Schritte ausgeführt:

1. Es werden alle Einstiegspunkte berechnet. Dazu werden alle Linien gesucht, die das Plaza schneiden oder dieses berühren. Der Schnittpunkt dieser Linien mit dem äusseren Polygon der Fläche bildet jeweils einen Einstiegspunkt. Um Rechenfehler auszugleichen, werden die Schnittpunkte mit einem Buffer von einigen Zentimeter um das Polygon herum gesucht. Plazas, die weniger als zwei Einstiegspunkte haben, werden direkt verworfen, da auf diesen mit unserem Algorithmus gar keine kürzesten Wege berechnet werden könnten.

2. Linien, die das Plaza schneiden und einen Einstiegspunkt bilden, werden mit der OSM-ID zwischengespeichert. Die jeweiligen Einstiegspunkte werden später vom Merger in diese Linien eingefügt.

3. Alle Hindernisse auf dem Plaza werden ausgeschnitten. Dies kann ein Gebäude sein, das auf dem Platz steht, oder ein Hindernis wie z.B. ein Brunnen. Solche Hindernisse werden mit einem Quadrat mit konfigurierbaren Radius ausgeschnitten. Sollte nach diesem Schritt das Plaza komplett von Gebäuden oder Hindernissen verdeckt sein, wird es verworfen.

4. Nun findet die eigentliche Optimierung statt, das Berechnen eines Graphen über das Plaza. Als Knotenpunkte des Graphen werden die Einstiegspunkte sowie alle Eckpunkte des Polygons (dementsprechend auch von Hindernissen) beachtet. Es sind die beiden Verfahren SpiderWeb-Graph und Visibility-Graph implementiert. Das Verfahren kann über das Strategy-Pattern [40] konfiguriert werden.

5. Um die Anzahl Kanten des Graphen zu verringern, werden aus den generierten Graphen alle kürzesten Wege von jedem Einstiegspunkt zu jedem anderen Einstiegspunkt berechnet. Dies kann mit dem Dijkstra- [29] oder A*-Algorithmus [28] durchgeführt werden. Alle Kanten, die nicht auf diesen kürzesten Pfade sind, werden verworfen.

6. Für den SpiderWeb-Graphen wird zusätzlich die Anzahl Knotenpunkte in den kürzesten Pfaden reduziert, indem die Linien geglättet werden. Dazu wird mit Hilfe von Shapely [33] der Douglas-Peucker Algorithmus [22] verwendet.

Merger

Die Aufgabe des Merger ist es, die vom Optimizer neu erstellten Geometrien wieder mit den originalen OSM-Daten zu verschmelzen.

1. In einem ersten Schritt werden die erstellten Linien im PBF-Format als Nodes und Ways in eine temporäre Datei gespeichert.

2. Als nächstes werden die Einstiegspunkte in das Strassennetz eingebunden. Nur wenn unsere Linien einen gemeinsamen Punkt mit dem bestehenden Strassennetz haben, werden sie vom Router überhaupt für das Routing beachtet. Da sich jeder Einstiegspunkt mit einer bestehenden Linie kreuzt, wird dieser jeweils als zusätzlicher Punkt dem OSM-Way hinzugefügt.

3. Die im vorherigen Schritt modifizierten Ways werden ebenfalls in eine temporäre Datei abgespeichert. Dazu wird die Versionsnummer erhöht, damit die bestehenden Ways schlussendlich überschrieben werden.

4. Nun werden alle in den vorherigen Schritten erstellten temporären OSM-Dateien mit der originalen Datei zusammen geschmolzen, die ursprünglich importiert wurde. Dazu wird das externe Tool Osmosis [34] aufgerufen.

Als Endresultat liegt nun eine OSM-Datei mit den kompletten Kartendaten vor, die um zusätzliche Linien für die Optimierung über Fussgänger-Flächen ergänzt wurde. Die Datei kann nun etwa einer Routing-Engine übergeben werden.

Konfiguration

Um die Vorverarbeitung möglichst flexibel und erweiterbar zu machen, wird eine Konfigurationsdatei im YAML-Format verwendet. Es ist unter anderem konfigurierbar, welche Algorithmen (Visibility-Graph oder SpiderWeb-Graph) für die Berechnung der Graphen verwendet wird. Es können ebenfalls die OSM-Tags bestimmt werden, die der Importer verwendet, um zu klassifizieren, was z.B. als Plaza oder als Hindernis gilt. Listing 6 zeigt einen Ausschnitt dieser Konfiguration.

```
tag-filter:
  plaza: # what counts as a plaza
    includes:
      tag-key-values: # tags with specific values
        - or:
          - highway: pedestrian
        - or:
          - highway: footway
          - area: yes
    excludes: # no tags should match
      tag-key-values:
        - or:
          - area: no
```

Listing 6: Ausschnitt der Konfigurationsdatei von Plaza Preprocessing

5.1.2 Plaza Routing

In den folgenden Unterkapitel wird die Umsetzung der Plaza Routing Architektur, welche im Kapitel 4.2.2 definiert ist, erläutert. Dabei handelt es sich um eine Flask-Applikation[44].

app

Die Flask-Applikation[44] wird in dieser Komponente konfiguriert und gestartet. So wird unter anderem das Logging aufgesetzt und die API, welche im Abschnitt api beschrieben ist, initialisiert.

Ebenfalls werden hier Error-Handler definiert, welche dafür sorgen, dass der Konsument der API eine ansprechende und informative Fehlermeldung erhält und sicherstellt, dass keine Implementationsdetails der Applikation exponiert werden.

api

In diesem Abschnitt wird auf die Umsetzung der api-Schicht in Abbildung 28 und auf den Abschnitt api in der Architektur Bezug genommen.

Dabei wird mithilfe von Flask-RESTPlus [45], einer Flask-Extension, eine Web-API exponiert. Diese besteht aktuell aus einem Representational state transfer (REST)-Service, welcher im Abschnitt PlazaRouting genauer erläutert ist. Die Extension hat den Vorteil, dass automatisch eine Swagger-Dokumentation [39] generiert wird, mit welcher unter anderem API-Clients in verschiedenen Programmiersprachen generiert werden können. Diese ist unter [37] verfügbar. Ebenfalls ist sofort ersichtlich, wie das Response-Model der API aussieht und in welchem Format welche Parameter übergeben werden müssen. Dies vereinfacht die Handhabung mit der API massiv.

Diese Komponente kann ohne Probleme mit zusätzlichen REST-Services analog zu PlazaRouting er-

gänzt werden.

PlazaRouting

In der Komponente *PlazaRouting* wird der REST-Service für das Routing exponiert. Im Bezug auf das Maturity Model [41] von Leonard Richardson befinden wir uns auf Level 2. Im REST-Service werden die übergebenen Parameter entgegen genommen, automatisch in das verlangte Format geparsed und geprüft, ob die Parameter, welche zwingend verlangt werden, auch übergeben wurden. Diese Information wird dann weiter an die Business-Schicht gereicht, welche im nächsten Abschnitt aufgeschlüsselt wird.

Zusammenspiel und Verantwortlichkeit der Business- und Service-Schicht

In Abbildung 31 sind die Interaktionen zwischen den Komponenten in einem Sequenz-Diagramm aufgeschlüsselt. Dabei wurde zur Wahrung der Übersichtlichkeit das Holen aller möglichen Routen und das Optimieren der besten Route in ein Unter-Sequenz-Diagramm in Abbildung 32 respektive Abbildung 34 ausgelagert. Es geht primär um die Verantwortlichkeiten der Services und nicht um die Logik, die dahinter liegt. Wann sie ins Spiel kommen und für was sie benötigt werden, ist so gut ersichtlich. Aus diesem Grund entsprechen die Diagramme nicht exakt dem Programmcode.

Die Ausgangsposition und Destination können entweder als Koordinate oder Adresse übergeben werden. Falls es sich um eine Adresse handelt, wird ein Geocoding durchgeführt, um die Koordinate zu extrahieren. Dies erklärt den zweimaligen Zugriff auf das *Geocoding* im Sequenz-Diagramm 31.

Man sieht in Abbildung 32 gut, dass im Loop für eine Ausgangs-ÖV-Haltestelle die Verbindung von search.ch [9] geholt (siehe Abschnitt ÖV-Haltestellen eruieren) und zwei Mal eine Fussgänger-Routing durchgeführt wird, jeweils vom aktuellen Standort zur ersten Kante und von der letzten Kante zur Destination. Diese Daten sind die Grundlagen für die Entscheidungsfindung der besten Route, welche in Abschnitt Beste Route eruieren beschrieben ist. *Optimize Route Combination* ist im Abschnitt Route optimieren genauer beschrieben.

ÖV-Haltestellen eruieren

ÖV-Haltestellen werden mit Overpass [10] aus OSM bezogen. Dabei wird um den aktuellen Standort eine Bouding Box gezogen. Die Bouding Box ist konfigurierbar und entspricht einer zumutbaren Laufdistanz. In dieser Fläche werden Nodes und Relationen mit Tags, welche ÖV-Haltestellen identifizieren (*"public_transport"="stop_position"*, *"type"="public_transport"* etc.), gefiltert und deren *uic_ref* zurückgegeben.

Beste Route eruieren

Aufgrund der möglichen Routen, welche wie in Abbildung 32 gewonnen werden, muss man nun ent-

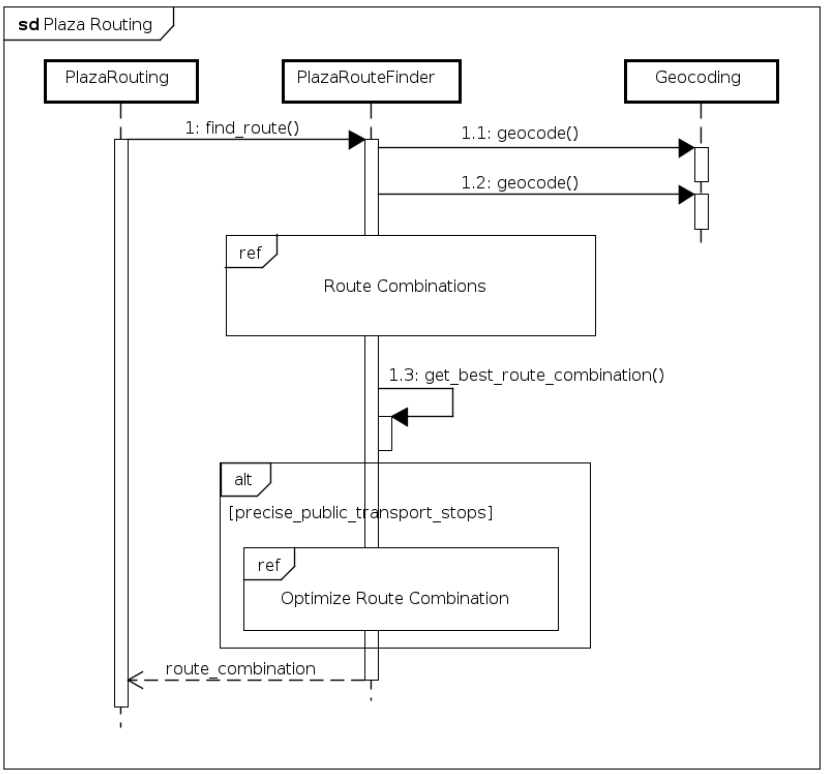

Abbildung 31: Plaza Routing Sequenz-Diagramm Übersicht

scheiden, welche Route dem User konkret retourniert wird. Dazu wurde eine Kosten-Matrix erstellt. Diese ist in Tabelle 7 sichtbar.

	Gewicht
Gehzeit	2
Dauer der ÖV-Verbindung	1
Anzahl der ÖV-Teilstrecken	7 * 60

Tabelle 7: Kosten-Matrix

Die Dauer der Fussstrecke wird doppelt gewichtet. Ein einmaliges Umsteigen schlägt mit 7 Minuten ins Gewicht. So lassen sich die Kosten aufgrund der Zeit, welche man für jeden Faktor benötigt, berechnen.

Eine Verbindung, bei welcher man 5 Minuten geht, 15 Minuten fährt und zwei Mal umsteigen muss, wird mit Totalkosten von $5*60*2+15*60+2*7*60 = 2340$ mit den anderen möglichen Verbindungen verglichen. Schlussendlich wird die Verbindung mit den niedrigsten Kosten retourniert.

Route optimieren

Search.ch [9] liefert in einer ÖV-Verbindung für zwei gegenüberliegenden Kanten (je eine für jede Fahrrichtung) eine Koordinate zurück, welche beispielsweise direkt auf der Hauptstrasse liegen kann (siehe Abbildung 33). Dies ist für ein Fussgänger-Routing suboptimal.

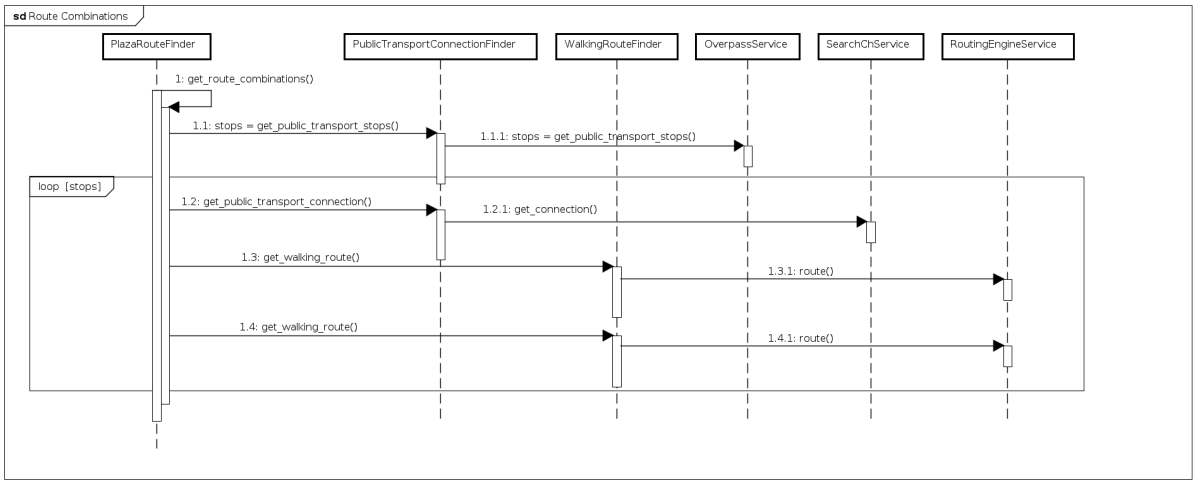

Abbildung 32: Plaza Routing Sequenz-Diagramm Route Combinations

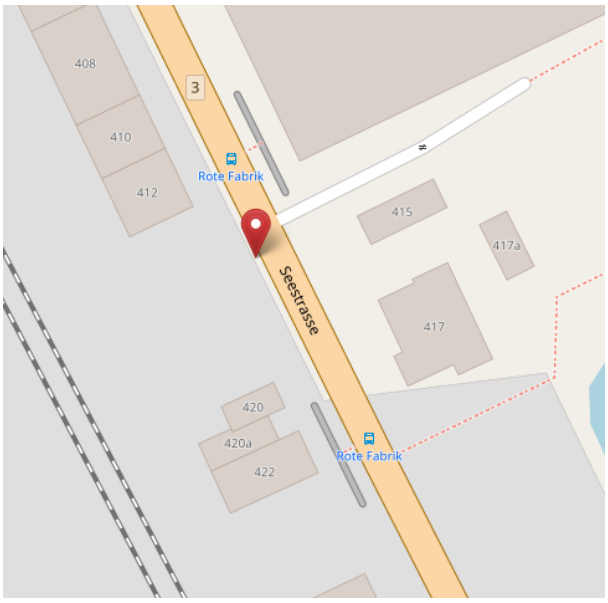

Abbildung 33: eine Koordinate für zwei Kanten der Roten Fabrik, Zürich, Schweiz; openstreetmap.org; Screenshots aufgenommen am 25.11.2017

Aus diesem Grund werden diese Koordinaten nun von Overpass [10] bezogen. Der Ablauf ist schematisch in Abbildung 34 dargestellt. So wird für jede ÖV-Teilstrecke die Koordinate für die Einstieg- und Ausstiegskante geladen. Nach dem Laden muss nochmals das Fussgänger-Routing durchgeführt werden, da sich die Koordinate der ersten und letzten Kante verändert haben kann.

Da ÖV-Haltestellen und -Linien in OSM jedoch nicht konsistent gemappt sind, wird das Recovery Blocks Pattern [46] angewendet, um dem User so oft wie möglich ein genaues Resultat liefern zu können. Mit nicht konsistent meinen wir, dass sie nicht zwingend den Vorgaben in [47] folgen. Durch Tests lassen sich jedoch Trends erkennen, wie Mapper grundsätzlich mit ÖV-Haltestellen und -Linien umgehen.

Vorgängig muss klargestellt werden, dass wenn die Rede vo einer ÖV-Linie ist, grundsätzlich eine Buslinie oder ähnliches, welche in OSM als Relation [47] abgebildet ist, gemeint ist. Kanten werden als

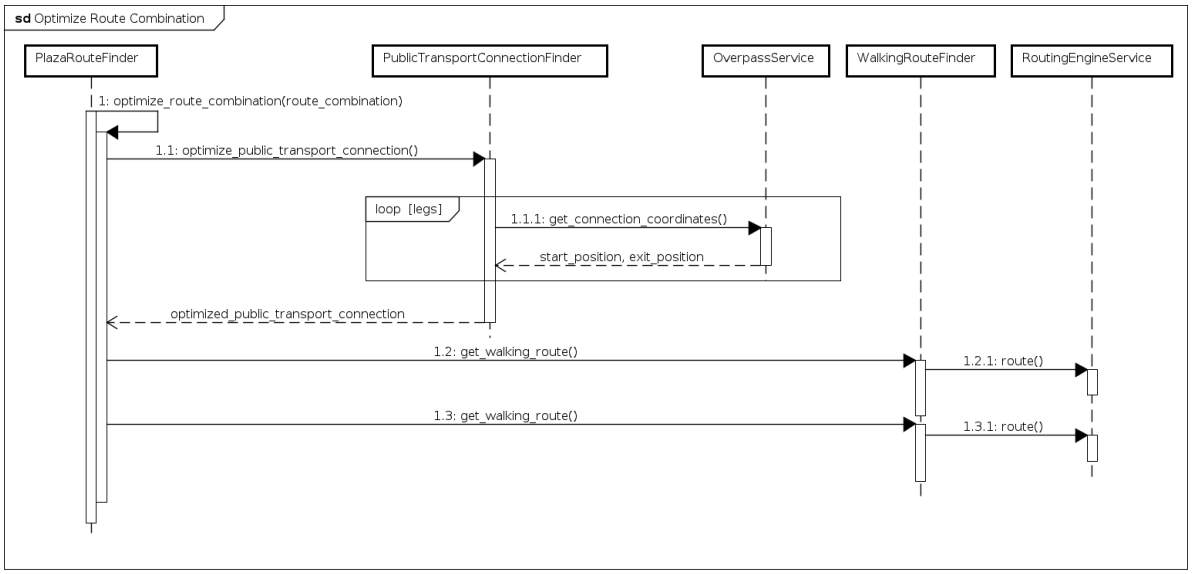

Abbildung 34: Plaza Routing Sequenz-Diagramm Optimize Route Combination

Node gemappt.

In einem ersten Schritt werden die Daten über die bekannte ÖV-Linie geladen. Search.ch [9] liefert eine *uic_ref* für die initiale ÖV-Haltestelle und eine, bei welcher man aussteigt. Üblicherweise erhält man zwei Relationen zurück, je eine für jede Fahrrichtung. Mit den beiden *uic_ref*s der ÖV-Haltestellen kann man nun die Members der Relation durchgehen und die ÖV-Linie selektieren, bei welcher die *uic_ref* der initialen ÖV-Haltestelle vor der *uic_ref* der Ausstiegshaltestelle kommt. Der Member mit der *uic_ref* der initialen ÖV-Haltestelle in der selektierten ÖV-Linie entspricht in diesem Fall der gesuchten Kante in die richtige Fahrtrichtung. Anmerken muss man, dass es natürlich mit einem Risiko verbunden ist, so die korrekte Koordinate zu eruieren. Entspricht die Reihenfolge der Members nicht der tatsächlichen Reihenfolge der ÖV-Haltestellen in einer ÖV-Linie, kommt aber bei der Falschen die *uic_ref* der initialen ÖV-Haltestelle vor der *uic_ref* der Ausstiegshaltestelle, kann es vorkommen, dass man die falsche Kante erwischt. Mit diesem Restrisiko leben wir. Falls bei beiden Relationen die gleiche *uic_ref* zuerst kommt oder die gemappten OSM-Daten für dieses Vorgehen nicht ausreichen, wird auf eine alternative Variante ausgewichen.

Ist die erste Variante nicht erfolgreich, werden die Relationen mit der *uic_ref* der initialen ÖV-Haltestelle geladen. Kanten mit der gleichen *uic_ref* gehören normalerweise zu einer Relation, welche auch selbst diese *uic_ref* besitzt. Über diese Relation können die möglichen initialen Ausgangskante und die ÖV-Linien geladen werden. In einem weiteren Schritt werden die Ausstiegskanten in einem ähnlichen Verfahren für die *uic_ref* der Ausstiegshaltestelle geladen. Im idealen Fall hat man nun die Einstiegskanten, ÖV-Linien und Ausstiegskanten für beide Fahrtrichtungen. Mit diesen Daten kann nun die korrekte Einstiegs- und Ausstiegskante der konkreten ÖV-Linie zugeordnet werden (durch ihre Zugehörigkeit als Member). Da man die Einstiegs- und Ausstiegskante separat geladen hat, ist es eine Leichtigkeit, die ÖV-Linie in die richtige Fahrtrichtung zu eruieren.

Führen diese zwei Varianten nicht zum Ziel, wird die Koordinate von search.ch zurückgegeben.

Das verbesserte Resultat von Abbildung 33 ist in Abbildung 35 ersichtlich. In Abbildung 38 sieht man, wie sich die Optimierung auf den konkreten Anwendungsfall auswirkt.

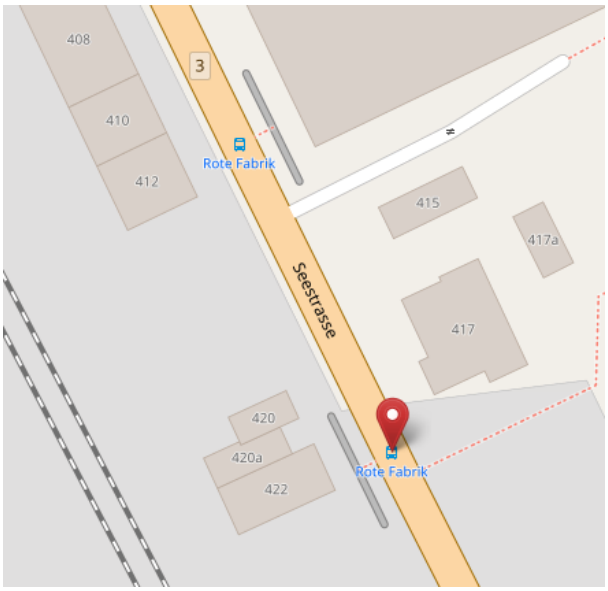

Abbildung 35: Eine mit Overpass geladenen Koordinate der Roten Fabrik in Richtung Seerose, Zürich, Schweiz; openstreetmap.org; Screenshots aufgenommen am 25.11.2017

Die erhöhte Genauigkeit kommt nicht ohne Performanzeinbusse. Im Worst-Case wird drei Mal eine Overpass-Abfrage [10] für jede ÖV-Teilstrecke durchgeführt und schlussendlich doch die Fallback-Koordinate von search.ch [9] retourniert. Aus diesem Grund kann man beim Aufruf des REST-Service ein optionales Flag setzen, welches dem User die Antwort auf die Frage überlässt, ob er diese Genauigkeit will oder mit der Koordinate von search.ch [9] zufrieden ist. Bei gesetztem Flag beschränken sich die Zugriffe auf Overpass [10] auf das Holen der ÖV-Haltestellen.

Route-Kombination oder nur Fussgänger-Route

Bevor überhaupt ein ÖV-Routing durchgeführt wird, wird geprüft, ob das Ziel in einem konfigurierbaren Grenzwert zu Fuss erreichbar ist. Wenn dies zutrifft, wird nur das Fussgänger-Routing retourniert. Dadurch werden die Drittsysteme geschont und die Abfragezeit ist minimal. Search.ch [9] gibt für kleinere Routen ebenfalls nur ein Fussgänger-Routing zurück, somit wäre diese Abfrage ohne diesen Grenzwert überflüssig.

Nach dem durchgeführten multimodalen Routing, sprich der Kombination aus dem Fussgänger- und ÖV-Routing, wird überprüft, ob die Dauer des Fussweg vom Start zum Ziel kleiner ist als die Summe aus der Dauer des Fussweg zur Ausgangshaltestelle, der Wartezeit auf die ÖV-Verbindung, des ÖV-Routing und des Fussweg zum Ziel. Ist man zu Fuss schneller, wird das Fussgänger-Routing retourniert. Dies ist in Abbildung 36 dargestellt. In der linken Grafik wird das Routing mit der Startzeit 15:46 Uhr durchgeführt. Bei der retournierten ÖV-Verbindung muss man *240 s* warten und hat für das Fussgänger- und ÖV-Routing zusammen *829 s*. In der rechten Grafik startet man um 14:12 Uhr und wartet auf die gleiche

ÖV-Linie *480 s*. Zu Fuss ist das Ziel in *1137 s* erreichbar. Links erhält man die Gesamtdauer von *1069 s* und rechts *1309 s*. Aus diesem Grund wird rechts nur das Fussgänger-Routing retourniert, da *1309 s > 1137 s* zutrifft.

Es lässt sich nun darüber streiten, ob man dies dem Benutzer zumuten kann. Wir haben entschieden, dass wir die schnellstmögliche Variante anbieten möchten und nicht die Bequemste. Im Kapitel 9.2 wird darauf eingegangen, dass man in einer weiteren Version die Benutzer-Präferenzen definieren kann.

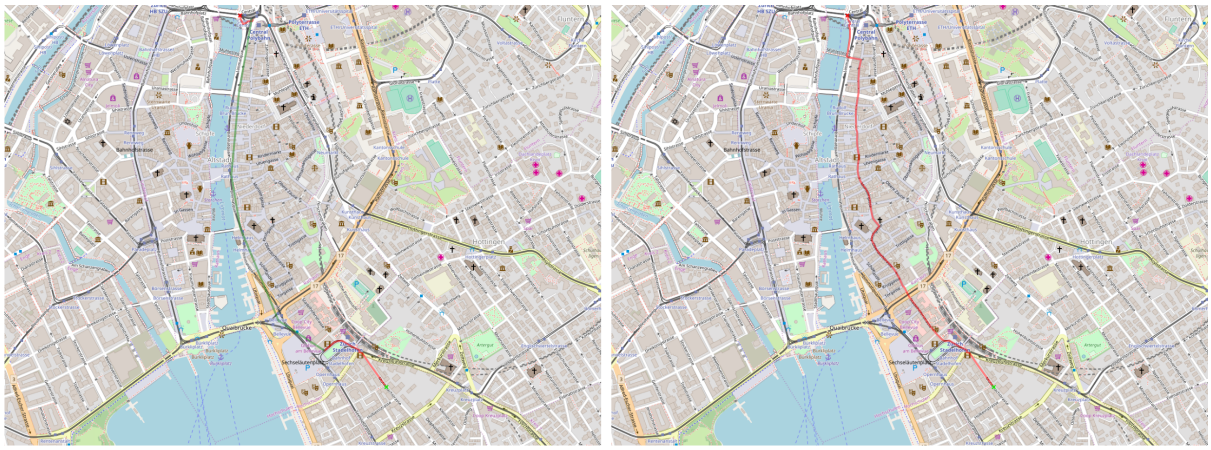

Abbildung 36: Entscheidung, ob Route-Kombination oder nur Fussgänger-Route retourniert wird; Dauer des Fussweg: 1137 s, Links: Startzeit 15:46, Kombination 829 s, Wartezeit 240 s, Total 1069 s, Rechts: Startzeit 14:12, Kombination 829 s, Wartezeit 480 s, Total 1309 s

Konfiguration

Die Konfiguration der Plaza Routing Komponente erfolgt durch ein Python-Konfigurationsfile im Root-Verzeichnis des Python-Packages. Dort können unter anderem die APIs zu den benötigten Services angepasst werden. Ausserdem ist konfigurierbar, welche Grenzwerte für das Finden der besten Route (siehe Abschnitt Beste Route eruieren) verwendet werden und mit welchem Buffer die Bouding Box für das Beziehen der ÖV-Haltestellen berechnet wird (siehe Abschnitt ÖV-Haltestellen eruieren).

5.2 QGIS-Plugin

Das QGIS-Plugin Plaza Route [24] fügt sich nahtlos als *Dockwidget* in die QGIS-Umgebung ein. Dabei wird über Python-Bindings (PyQt4) mit dem Qt-Framework gearbeitet. In Abbildung 37 sieht man das Plugin in Aktion. Die Basis wurde mit dem QGIS-Plugin Builder [48] gelegt. Beim Entwickeln des Plugin wurde Wert darauf gelegt, dass man Funktionalität bietet, welche bei vergleichbaren Produkten verfügbar ist. So lassen sich die Start und die Destination über ein Kontextmenü direkt auf der Karte auswählen. Das Plugin lässt Koordinaten sowie Adressen zu. Die Koordinaten können in einem beliebigen System, welches QGIS kennt, angegeben werden. Dies ermöglicht eine unkomplizierte Anwendung, da das Plug-in keine speziellen Anforderungen an das System und den Benutzer stellt. Das Plugin wird installiert, eine Karte wird geöffnet und das Routing kann bereits durchgeführt werden.

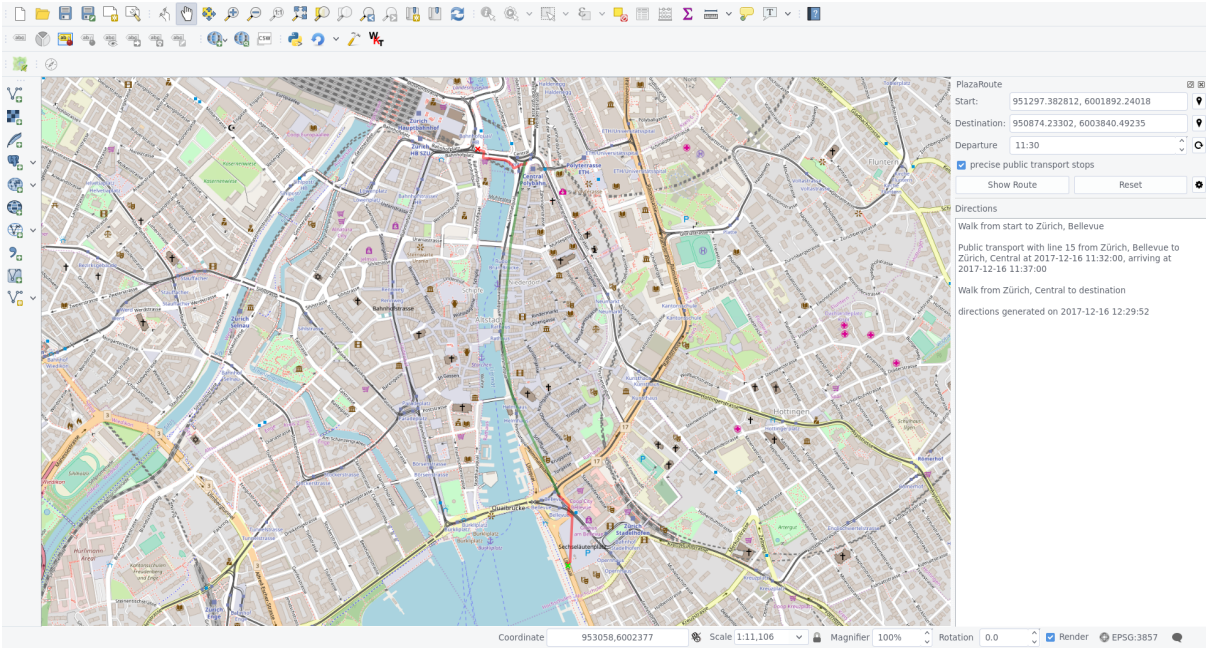

Abbildung 37: QGIS-Plugin Plaza Route

Die Integration der Karte erfolgt über *QgsMapTool*. Dadurch können Klicks auf der Karte abgefangen und an das Dockwidget übergeben werden. Dabei wird das Observer-Pattern [40] eingesetzt, um eine loose Kopplung sicherzustellen. Das Dockwidget registriert sich dabei bei *PlazaRouteMapTool*, einer Implementation des *QgsMapTool*, und wird somit von dieser über die Auswahl auf der Karte informiert.

Die Kommunikation mit dem REST-Service, welcher im Abschnitt api beschrieben ist, erfolgt in *PlazaRouteRoutingService* über den *QNetworkAccessManager*, damit auch Proxy-Einstellungen im QGIS des Users berücksichtigt werden.

Die Karte wird über *QgsRubberBand* modifiziert. *QgsRubberBand* nutzt man für das Zeichnen von transienten Geometrien. In Listing 7 ist die Nutzung schematisch aufgezeigt. Die Geometrien werden dabei auf die Karte gelegt und verschwinden beim Beenden von QGIS. Für eine ÖV-Verbindung werden die Stationen mit Fluglinien verbunden.

```
1  from qgis.gui import QgsRubberBand
2
3  canvas = self.iface.mapCanvas()
4
5  rubber_band = QgsRubberBand(canvas, geometry_type)
6  rubber_band.setColor(color)
7  rubber_band.setWidth(width)
8
9  rubber_band.addPoint(start)
10 rubber_band.addPoint(destination)
```

Listing 7: QgsRubberBand Pseudocode

Im Abschnitt Route optimieren wurde auf die Problematik mit den ungenauen Koordinaten von search.ch [9] eingegangen. In Abbildung 38 ist nun der Unterschied zwischen dem ungsetzten und gesetzten *precise public transport stop* Flag sichtbar. Mit dem gesetzten Flag werden die Koordinaten optimiert.

Abbildung 38: Vergleich der search.ch Koordinaten (links) und optimierten Koordinaten (rechts)

Neben der Route wird dem Benutzer auch eine einfache Navigationsanweisung (siehe Abbildung 39) angezeigt, welche aus der erhaltenen Response generiert wird.

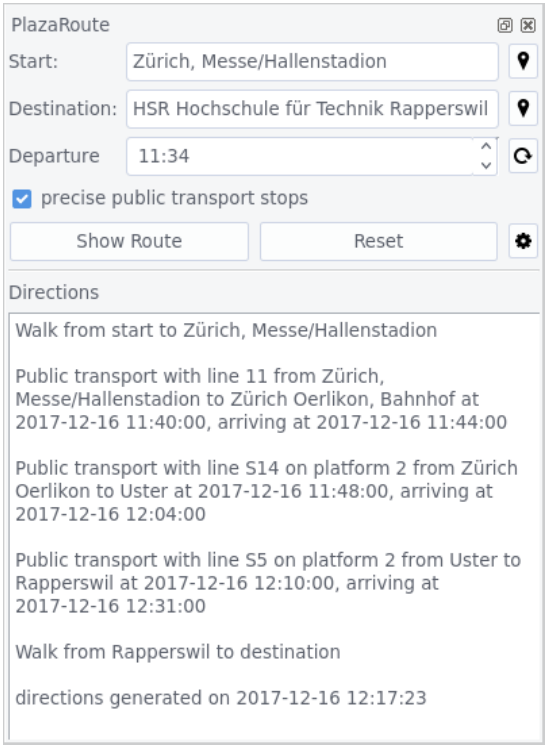

Abbildung 39: QGIS-Plugin Plaza Route mit Navigationsanweisung

6 Error-Handling Policy

Grundsätzlich folgt das System dem Motto, dass dem Benutzer keine Internas exponiert werden, sondern im Fehlerfall eine kurze und prägnante Meldung zurückgegeben wird, was schief gelaufen ist. Im Folgenden sind dazu die Exception und Logging Policy aufgeführt.

6.1 Exception Policy

Aufgrund der unterschiedlichen Anforderung der Komponenten *Plaza Vorverarbeitung* und *Plaza Routing* an das System und den Benutzer werden die Exception Policies für beide getrennt betrachtet.

6.1.1 Plaza Vorverarbeitung

Die Komponente wird als Command-Line-Tool verwendet. Das hat den Vorteil, dass der Benutzer nicht schnell mit Fehlermeldungen überfordert ist.

Verwendet werden die eingebauten Exceptions *ValueError* und *RuntimeError*, sowie die Exceptions von *Argparse*, die beim Parsen von Kommandozeilen-Argumenten auftreten.

Die Parameter werden beim Aufruf direkt von *Argparse* validiert, der Benutzer sollte also nur die *Argparse* Exceptions sehen. *RuntimeError* und *ValueError* werden innerhalb des Programms verwendet, um sicher zu stellen, dass alle benötigten Daten eines vorherigen Schrittes vorhanden sind. Wenn ein solcher Fehler auftritt, deutet dies auf ein Programmierfehler hin, nicht ein Fehler der Benutzereingabe.

6.1.2 Plaza Routing

Plaza Routing definiert zwei eigene Typen an Exceptions, namentlich *ValidationError* und *ServiceError*, und setzt auf die eingebaute *ValueError*.

ServiceError

ServiceError werden für Fehler beim Aufruf von Fremdsystemen in der Integration-Schicht verwendet. So kann es sein, dass ein Fremdsystem temporär offline ist oder sich Breaking-Changes in der API ereignet haben. In diesem Fall wird die Exception geloggt und ein ServiceError geworfen.

ValidationError

ValidationError wird für Fehler verwendet, welche aufgrund von Benutzereingaben geworfen werden. Dazu gehört beispielsweise der Aufruf des API mit ungültigen Parameter (z.B. falsch formatierte Koordinaten). Das Werfen dieser Exception beschränkt sich jedoch nicht nur auf die Business-Schicht. Auch die Integration-Schicht kann gebrauch von dieser machen, falls die Parameter für unsere Seite in Ordnung sind, für ein Fremdsystem jedoch nicht. So kann beispielsweise eine Routing-Engine für eine Koordinate den Fehler "Point is out of bounds"werfen, falls die Koordinate nicht auf der Karte gefunden wird, auf welcher die Routing-Engine agiert. Dies tritt beispielsweise auf, wenn man eine Koordinate aus Deutschland dem Service übergibt, die Routing-Engine jedoch nur die OSM-Datei der Schweiz kennt.

Diese Exception hat die Eigenschaft, dass die Message in der Exception nach aussen, sprich dem Benutzer, exponiert werden kann. Somit muss diese mit klaren und ohne internen Implementierungdetails versehenen Messages versehen werden. Dies wird aus dem Grund gemacht, da der Fehler auf Seiten des Benutzers liegt und er somit sofort sehen kann, wie er das System richtig bedienen kann.

ValueError

Fehler auf Seiten von Plaza Routing werden mit ValueError geworfen. Dabei sollen diese mit informativen Details versehen werden, so dass bei der Fehlerbehebung genug Informationen für das Reproduzieren vorhanden sind. Diese werden nicht direkt nach aussen exponiert.

Error-Handlers

In der api-Schicht werden Error-Handler registiert. Für die zwei eigenen Exceptions (ServiceError und ValidationError) existieren je zwei Error-Handler, welche die geworfenen Exceptions abfangen und verarbeiten. Als letztes wird ein Default-Handler registriert, welcher sicherstellt, dass alle Exceptions gefangen werden und dass nur definierte Meldungen an den Benutzer exponiert werden. Die Verwendung ist in Listing 8 dargestellt.

```
1  @api.errorhandler(ValidationError)
2  def validation_error_handler(e):
3      return {'message': str(e)}, 400
4
5  @api.errorhandler(ServiceError)
6  def service_error_handler(e):
7      return {'message': 'third party system is temporarily unavailable'}, 503
8
9  @api.errorhandler
10 def default_error_handler(e):
11     return {'message': 'plaza route is temporarily unavailable'}, 500
```

Listing 8: Error-Handler

Man sieht in Listing 8 gut, dass die Message der *ValidationError* direkt retourniert wird und für die

anderen Typen vordefinierte Meldungen zurückgegeben werden.

Verwendete HTTP Status Codes

Folgende HTTP Status Codes sollen verwendet werden:

- *200 OK*
- *400 Bad Request*: Fehler auf Seiten des Benutzer
- *500 Internal Server Error*: Fehler auf Seiten PlazaRouting
- *503 Service Unavailable*: Fehler, welche aufgrund fehlerhaften Fremdsystem auftreten (nicht-verfügbar, Änderungen im API)

6.2 Logging Policy

Beide Komponenten führen ein Log-File, welches zur Information oder im Fehlerfall zur Hand gezogen werden kann. Für das Logging wird die Python-Library *logging* verwendet. Dadurch sind die Standard-Log-Level (`ERROR`, `WARNING`, `INFO`, etc.) verfügbar.

Exceptions müssen zwingend mit Stack-Trace protokolliert werden, falls diese nicht korrekt behandelt werden können und ein Fehler-Recovery nicht möglich ist. Das Logging von Exceptions wird dort durchgeführt, wo der Fehler auftritt und gefangen wird.

Dem Entwickler steht es frei, die anderen Log-Levels nach Belieben einzusetzen.

7 Tests

7.1 Strategie

Grundsätzliches Ziel war es, beim Entdecken von Grenzfällen und Fehlern nach dem Test-Driven Development (TDD)-Zyklus vorzugehen. Für das Testing wurden Unit- und Integration-Tests grosszügig eingesetzt. Durch die Gegebenheiten der Problem-Domäne ist es von besonderer Wichtigkeit, dass möglichst breit und viel getestet wird. Dadurch kann den Problemen, welche durch ein ”fehlendes”OSM-Datenmodell auftreten, entgegen gewirkt werden. Die Tests werden automatisiert bei jedem Commit mit Continuous Integration durch CircleCI [49] ausgeführt. So ist sichergestellt, dass Builds, welche auf einem Feature-Branch fehlschlagen, nicht in den Master gemerged werden. Im nachfolgenden sind die

Tests für die Komponenten *Plaza Vorverarbeitung* und *Plaza Routing* getrennt aufgeschlüsselt, da die Applikationen mit unterschiedlichen Bedingungen zu kämpfen haben.

7.2 Plaza Vorverarbeitung

Bei *Plaza Vorverarbeitung* liegt der Fokus auf dem korrekten Zusammenspiel der Komponenten. Die Abbildung 26 zeigt dessen Datenfluss auf. Aus diesem Grund haben wir den Fokus auf Integration-Tests gelegt. So wird unter anderem beim Testen des Optimizer auch der Importer verwendet, da das Mocken der internen in-memory Datenstruktur, welche der Importer liefert, keinen Sinn macht, da dieser aufgrund der vordefinierten OSM-Dateien deterministische Daten liefert. In Fällen wo diese Annahme nicht zutrifft und Funktionen in Isolation getestet werden können, werden Unit-Tests, wie in Listing 9 sichtbar, eingesetzt.

```
1  def test_compute_dijkstra_shortest_paths():
2      graph_edges = [LineString([(0, 0), (0, 1)]), LineString([(0, 1), (1, 1)]),
3                     LineString([(1, 1), (1, 0)]), LineString([(0, 0), (1, 0)]),
4                     LineString([(0, 0), (1, 1)]), LineString([(1, 1), (2, 1)])]
5
6      entry_points = [Point((0, 0)), Point((2, 1)), Point((1, 0))]
7      expected_lines = [[(0, 0), (1, 1), (2, 1)], [(0, 0), (1, 0)],
8                        [(1, 0), (1, 1), (2, 1)]]
9
10     graph = shortest_paths.create_graph(graph_edges)
11     lines = shortest_paths.compute_dijkstra_shortest_paths(graph, entry_points)
12     assert expected_lines == [list(line.coords) for line in lines]
```

Listing 9: Unit-Test Shortest Path

In Listing 10 ist aufgeführt, wie das Unit-Testing gehandhabt wird. Durch Fixtures kann der Tests in diesem Fall auf vier verschiedene Arten von möglichen Konfigurationen getestet werden, beispielsweise eine Visibility-Graph Vorverarbeitung, welche den A* [28] als Shortest-Path-Algorithmus verwendet.

7.3 Plaza Routing

Plaza Routing verwendet Fremdsysteme für unterschiedliche Zwecke. Unter anderem wird search.ch [9] für das Abfragen der ÖV-Verbindungen benötigt. Werden nun Tests direkt auf dem Fremdsystem durchgeführt, kann man nicht mit einem deterministischen Verhalten rechnen. Die Werte sind in diesem konkreten Fall datumsabhängig. So werden beispielsweise am Sonntag nicht die gleichen Verbindungen angeboten wie unter der Woche. Auch die Abfrage auf ein konkretes Datum ist nur beschränkt möglich, da die Daten nur einen gewissen Zeitraum in die Vergangenheit abrufbar sind. Bei Overpass [10] liegt die Problematik bei den zugrundeliegenden Daten. Es liegt in der Natur der OSM-Daten, dass

```python
@pytest.fixture(params=['visibility', 'spiderweb'])
def process_strategy(request):
    if request.param == 'visibility':
        return VisibilityGraphProcessor(visibility_delta_m=0.1)
    elif request.param == 'spiderweb':
        return SpiderWebGraphProcessor(spacing_m=5, visibility_delta_m=0.1)


@pytest.fixture(params=['astar', 'dijkstra'])
def shortest_path_strategy(request):
    if request.param == 'astar':
        return shortest_paths.compute_dijkstra_shortest_paths
    elif request.param == 'dijkstra':
        return shortest_paths.compute_astar_shortest_paths

def test_optimized_lines_inside_plaza(process_strategy,
                                      shortest_path_strategy, config):
    # test PlazaPreprocessor
    # ...
```

Listing 10: Integration-Test Plaza Preprocessor

sie sich stetig verändern. Stabile Tests sind in diesem Fall nicht möglich, wenn man wie in Kap. 5.1.2 von Koordinaten abhängig ist und diese direkt auf dem Fremdsystem überprüfen will. Diese Erfahrung wurde im Verlauf des Projekts gemacht. So wurden nachträglich Mocks eingeführt, welche stabile Tests ermöglichen. In Listing 11 ist zu sehen, wie mit *monkeypatch* [50] für search.ch [9] aufgrund der übergebenen Parameter die erwartete Antwort geliefert wird.

```python
def mock_get_connection(monkeypatch):
    monkeypatch.setattr(search_ch_service, "_query",
                        lambda payload:
                        utils.get_file(_get_filename(payload), 'search_ch'))
```

Listing 11: Mock search.ch

Durch diesen Ansatz lassen sich Unit- und Integration-Tests auf einer stabilen Datenbasis durchführen.

7.3.1 Healthchecks

Da alle Services in den Tests gemockt werden, ist es unabdingbar, dass mit Healthchecks die Fremdsysteme geprüft werden. So ist sichergestellt, dass der Aufruf auf das konkreten System mit den bewährten Parameter eine erfolgreiche Rückmeldung liefert und das Fremdsystem verfügbar ist. Ob die Werte ansatzweise sinnvoll sind, wird durch die jeweiligen Parser der Services sichergestellt.

7.4 Fazit

7.4.1 TDD

Das Vorgehen nach dem TDD-Zyklus hat sich vor allem beim Arbeiten mit Fremdsystemen extrem bewährt. Herauszuheben ist hier das Optimieren von Routen, welches im Abschnitt Route optimieren behandelt wurde. Hier wurde intensiv mit Overpass [10] kommuniziert. Durch die Gegebenheit der OSM-Daten und den Freiheiten der Mapper sind Grenzfälle und unerwartete Abbildungen der Daten keine Seltenheit. In diesen Fällen hat man Tests erstellt, welche das Verhalten reproduzieren und auf das zu Erwartende prüfen und konnte so die Logik modifizieren, damit das wie im Implementationskapitel beschriebene fehlertolerante System umgesetzt werden konnte.

7.4.2 Tests mit Fremdsystemen

Beim Arbeiten mit den Fremdsystemen hat sich gezeigt, dass es sich lohnt, die Resultate der Service-Aufrufe von Anfang an zu mocken, wenn man im Vorhinein bereits weiss, dass die zugrundeliegende Daten transient sind. Der Aufwand ist in diesem Fall nicht zu unterschätzen, hat sich aber in unserem Fall definitiv gelohnt.

7.4.3 Python Typensystem

Vorallem bei einer Sprache mit einem dynamischen Typensystem wie bei Python ist es wichtig, dass ausgiebig getestet wird. Ändert sich beispielsweise die Struktur eines Rückgabewerts, sind die Auswirkungen dieser Änderungen nicht sofort sichtbar. Wird diese Stelle jedoch zusätzlich getestet, ist jederzeit klar, ob es sich um Breaking-Changes handelt. Mit den in Python 3 eingeführten ”Type Hints”konnte dieses Problem aber schon etwas eingedämmt werden.

8 Infrastruktur

8.1 Continuous Integration

In unserem Projekt wird Continuous Integration (CI) einerseits für das automatische Erstellen dieser Dokumentation mit LaTeX verwendet, und andererseits, um automatische Tests für die Backend-Implementation auszuführen. Beides wird im Folgenden kurz beschrieben.

8.1.1 Erstellen der LaTeX-Dokumentation

Bei jedem Git Commit in das Repository der Dokumentation [51] wird mit Continuous Integration ein PDF erstellt. So wird sichergestellt, dass die Dokumentation keine syntaktischen Fehler enthält. Das PDF wird in unser internes JIRA hochgeladen und direkt dem Task angehängt, der zum Commit gehört.

Für diese Continuous Integration haben wir uns für Travis CI [52] entschieden, da die Integration mit Github gut funktioniert und das Produkt für Open-Source-Projekte kostenlos ist. Allerdings bietet Travis CI keine direkte Unterstützung für eine LaTeX-Umgebung, die LaTeX-Pakete aus den Repositories sind ausserdem veraltet. So muss bei jedem Build die LaTeX-Umgebung kompiliert werden, damit immer die neueste Version aller Pakete verwendet werden kann.

8.1.2 CI für Backend

Mit CI für das Backend des Prototyps werden bei jedem Commit in das Repository [23] alle Tests ausgeführt. So wird sichergestellt, dass die Tests auch auf einem isolierten System mit den definierten Abhängigkeiten korrekt durchlaufen. Für jeden Merge in den Master-Branch des Repositories wird zusätzlich mit Sphinx [53] eine automatische Dokumentation der Python-Pakete generiert. Diese ist unter [42] veröffentlicht.

Anders als bei 8.1.1 funktioniert hier Travis CI für unseren Fall nicht. Eine Abhängigkeit unserer Implementation ist Pyosmium [32], das selbst die *boost* Library für C++-Komponenten verwendet. Leider sind die Python-Bindings dafür in der Umgebung von Travis CI veraltet und wären nur sehr aufwändig selbst zu kompilieren. Stattdessen sind wir auf CircleCI [49] ausgewichen, das für einen parallelen CI-Job kostenlos ist. Dies reicht für unsere Zwecke völlig aus. CircleCI bietet die Möglichkeit, eigene Docker-Container zu verwenden. So können wir ein eigenes Docker-Image bauen, das *boost* bereits in der neuesten Version vorinstalliert hat, womit es mit den Python-Abhängigkeiten keine Probleme mehr gibt.

8.2 Deployment

Um das Deployment möglichst zu vereinfachen, werden für die einzelnen Container von PlazaRoute (siehe 4.2) jeweils eigene Docker-Images erstellt.

8.2.1 Plaza Vorverarbeitung

Die Plaza Vorverarbeitung wird periodisch von Hand oder automatisch ausgeführt. Um die Installation und das Dependency-Management zu vereinfachen, wurde ein Docker-Image erstellt und auf Docker Hub [54] hochgeladen, damit es direkt verwendet werden kann. Die Benutzung ist im Github Repository [23] dokumentiert.

8.2.2 Plaza Routing

Das Plaza Routing stellt das Backend dar und bietet ein API an, die vom QGIS-Plugin benutzt wird. Neben den Fremdsystemen braucht es dazu eine lokale Instanz der Routing-Engine, in unserem Fall GraphHopper [5].

Für Plaza Routing und GraphHopper wurde je ein Docker-Image erstellt. Da Flask [44] nur ein minimaler Applikations-Server mitbringt, wird unsere Flask-Applikation auf einem *uWSGI* [55] Server gestartet. Als dritter Container benutzen wir *Nginx* [56], der als Reverse Proxy konfiguriert ist und alle Anfragen an den Plaza Routing Container weiterleitet. So ist der Applikations-Server zusätzlich abgeschirmt.

Um das Management der drei Container zu vereinfachen, wurde Docker Compose [57] verwendet. Die Benutzung ist im Github Repository [23] dokumentiert.

9 Resultate und Weiterentwicklung

9.1 Resultate

Mit unserer Implementation und Optimierung der Algorithmen zur Flächen-Traversierung kann gezeigt werden, wie ein natürlicheres Routing für Fussgänger, insbesondere über offene Flächen im urbanen Raum, erreicht werden kann. Mit dem Plaza Routing Backend wurde dabei eine praktische Anwendung umgesetzt, die das Routing mit öffentlichen Transportmitteln zusammen mit optimiertem Fussgänger-Routing ermöglicht. Insbesondere haben wir Ansätze gezeigt, wie für ÖV-Haltestellen genaue Koordinaten berechnet werden können. Die Routen können mit dem QGIS-Plugin bezogen und visualisiert werden.

9.2 Möglichkeiten der Weiterentwicklung

9.2.1 Plaza Vorverarbeitung

In Zukunft könnte unsere Implementation der Vorverarbeitung dazu dienen, die Optimierung auf Fussgänger-Routing in die Graphen-Berechnung von bestehenden Routing-Engines einzubinden. Somit würde der separate Schritt der Vorverarbeitung von OSM-Daten entfallen, was durch den ersparten Aufwand des Lesens und Schreibens von OSM-Daten die Effizienz deutlich steigern würde.

In der jetzigen Implementation werden die kürzesten Wege nur zwischen Paaren von Einstiegspunkten berechnet. Dies hat zur Folge, dass auf einer Fläche mindestens zwei Einstiegspunkte vorhanden sein müssen, um Pfade berechnen zu können. Es gibt aber einige Fussgänger-Flächen, wo keine Strasse oder Weg den Platz angrenzt oder schneidet, wodurch keine Einstiegspunkte gefunden werden. In der Realität wäre der Platz aber mit grosser Wahrscheinlichkeit problemlos begehbar. Eine mögliche Lösung ist in Kapitel 2.3.1 beschrieben.

Während der Arbeit sind noch folgende Ideen und Ansätze aufgekommen:

- Wenn zwei Fussgänger-Flächen direkt aneinander angrenzen, sollte dies für die Verarbeitung berücksichtigt werden. Lösungsansätze dazu sind in Kapitel 2.2.5 beschrieben.

- In der jetzigen Implementation wird die meiste Rechenzeit verwendet, um die OSM-Daten für die Vorverarbeitung zu importieren. Es würde sich anbieten, die Daten in einer sepparaten Datenstruktur, z.B. PostGIS, zu halten, damit der Import-Schritt effizienter wird.

- Die Verarbeitung der einzelnen Flächen sind grundsätzlich unabhängig voneinander. So würde es sich anbieten, die Vorverarbeitung zu parallelisieren, um Multi-Core Systeme effizienter ausnutzen zu können.

- Python eignet sich gut als Sprache für den Prototypen. Für eine performante Verarbeitung würde sich eine Implementation mit einer Sprache wie C++ allerdings besser eignen. Es könnten auch nur Teile des Pythons-Code in C++ implementiert werden.

9.2.2 Backend und QGIS-Plugin

Während der Entwicklung von Plaza Routing und dem QGIS-Plugin sind folgende Verbesserungsmöglichkeiten aufgekommen:

- Bei der Auswahl der optimalen ÖV-Route werden gewisse Annahmen über die Gewichtung von Laufwegen und der Wartezeit auf ÖV-Verbindungen getroffen. Benutzer haben allerdings un-

terschiedliche Präferenzen, beispielsweise dass sie lieber etwas länger laufen, anstatt auf eine Verbindung zu warten. Für diese Parameter könnten optimale Durchschnittswerte gefunden werden. Eine weitere Möglichkeit wäre es, dass der Benutzer seine Präferenzen selber konfigurieren kann.

- Vom Startpunkt aus wird in einem fixen Umfeld nach ÖV-Haltestellen gesucht. Pro gefundene ÖV-Haltestelle wird eine Route berechnet. Es wäre unter Umständen sinnvoller, die ÖV-Haltestellen in einem grösseren Umkreis zu suchen, aber nur ÖV-Verbindungen für die am nähesten liegenden ÖV-Haltestellen in Betracht zu ziehen.

- Bei ÖV-Verbindungen werden momentan nur die Koordinaten der Zwischenhaltestellen geliefert. In der Visualisierung werden zwischen diesen Koordinaten Geraden gezogen. Für eine genauere Darstellung ist es denkbar, ein separates Routing für ÖV-Verbindungen zu realisieren, das exakt den Strassen und Schienen des öffentlichen Verkehrsmittels folgt.

10 Projektmanagement

10.1 Vorgehen

Für die Studienarbeit wurde das agile Vorgehen SCRUM in Kombination mit Rational Unified Process (RUP) gewählt. Gründe für diese Entscheidung sind, dass das agile Vorgehen der noch zu Beginn offenen Aufgabenstellung entgegenkommt, welche dann iterativ finalisiert werden kann und dass die unbekannte Problem-Domäne sowie der theoretische Fokus der Arbeit so besser gehandhabt und schneller reagiert werden kann. Die wöchentlichen Besprechungen und Reviews mit dem Betreuer ist ein weiterer Grund für diese Entscheidung. Die Kombination mit RUP ermöglicht es, dass Projekt in einzelne Phasen aufzuteilen, um so das Ziel und die Zeit nicht aus den Augen zu verlieren.

10.1.1 Entwicklung

Der Source-Code der Implementation wie auch diese Arbeit wird mit Git verwaltet und ist auf Github abgelegt. Die Entwicklung und das Dokumentieren erfolgt nach dem Github-Flow. Der Master-Branch ist auf allen Repositories während der ganzen Zeit gesperrt, so dass er nur über Pull-Requests bearbeitet werden kann. Für jede User-Story wird ein Branch erstellt. Ist die User-Story implementiert, wird ein Pull-Request erstellt und dem anderen Projekt-Mitglied zum Review übergeben. Wird der Pull-Request akzeptiert, wird der Feature-Branch in den Master gemerged. Dieses Vorgehen hat den Vorteil, dass alle Änderungen, welche in den Master gelangen, ein Review durchlaufen müssen und so die Qualität hochgehalten werden kann.

10.2 Zeitplanung

Die Arbeitspakte und Zeit wird mithilfe von Jira verwaltet. Für alle Tätigkeiten werden User-Stories im Backlog erfasst, priorisiert und geschätzt. Die Schätzung der User-Stories erfolgte mit Story Points. Die Arbeitszeitverbuchung wurde auf Arbeitspaket-Stufe mit Stunden gemacht.

10.2.1 Phasen / Iterationen und Meilensteine

Die Studienarbeit wird in die RUP-Phasen (Inception, Elaboration, Construction, Transition) aufgeteilt. Dabei wird jedoch eine von der gängigen Norm abweichende Aufteilung gewählt. Durch den theoretischen Fokus der Arbeit wird der Elaboration das grösste Zeitbudget zugeordnet. Dies ist auch der Grund warum mit einwöchigen Sprints gearbeitet wird. Es wird zusätzlich vom Standard-RUP-Prozess abgewichen. Gegen Ende des Projekts wird nochmals eine zusätzliche Elaboration-Phase durchgeführt. Dieser zweite theoretische Teil wird nicht in der ersten Elaboration gemacht, damit der wichtigste Use Case *Fussgänger-Routing über offene Flächen im urbanen Raum* so sicher umgesetzt wird und gegen Ende des Projekts bei idealem Projektverlauf noch Zeit für die theoretische Aufbereitung von *Stand Fussgänger-Routing über offene Strassen* und *Stand Flächen-Traversierung über Berge/Strände* bleibt.

Sprint	**Sprint 0**	**Sprint 1**	**Sprint 2**
Phase	Inception	Inception	Elaboration
Milestones	*Aufgabenstellung*	*Aufgabenstellung*	*Stand Fussgänger-Routing über offene Flächen*
Arbeitspakete			
	1. Aufgabestellung-Brainstorming 2. JIRA aufsetzen 3. LATEX Doc aufsetzen 4. Vorgehen definieren	1. Aufgabenstellung finalisieren 2. Travis aufsetzen 3. Python/Docker Know-How aufbauen 4. Einarbeitung QGIS	1. Visibility-Graph Know-How sammeln 2. Visibility-Graph QGIS Test 3. SpiderWeb-Graph Know-How sammeln

Sprint	Sprint 3	Sprint 4	Sprint 5
Phase	Elaboration	Elaboration	Elaboration
Milestones	*Stand Fussgänger-Routing über offene Flächen*	*Backend Prototype*	*Backend Prototype*
Arbeitspakete	1. Visibility-Graph QGIS Test 2. SpiderWeb-Graph QGIS Test 3. Skeleton-Graph Know-How sammeln	1. Evaluation Routing Engines 2. Evaluation Overpass Anbindung 3. Konzept Plaza Vorverarbeitung erstellen	1. Architektur Backend 2. Evaluation Search.ch Anbindung 3. Plaza Vorverarbeitung umsetzen

Sprint	Sprint 6	Sprint 7	Sprint 8
Phase	Construction	Construction	Construction
Milestones	*Backend Prototype*	*Backend Prototype*	*Backend/Frontend Prototype*
Arbeitspakete	1. Anbindung Fremdsysteme 2. Plaza Vorverarbeitung umsetzen 3. API definieren	1. Kommunikation mit Fremdsystemen koordinieren 2. QGIS-Plugin Grundstruktur	1. Vergleich Flächentraversierungsvarianten durchführen 2. QGIS-Plugin Eingabemaske 3. Stand Start-/Endpunkt auf Flächen

Sprint	**Sprint 9**	**Sprint 10**	**Sprint 11**
Phase	Construction	Elaboration	Elaboration
Milestones	*Prototyp Frontend*	*Stand Fussgänger-Routing über offene Strassen*	*Stand Flächen-Traversierung über Berge/Strände*
Arbeitspakete	1. QGIS-Plugin beste Route anzeigen 2. Stand Flächentraversierung bei zwei benachbarten Polygonen	1. Stand Fussgänger-Routing über offene Strassen Recherche 2. Überführung in Docker-Container	1. Stand Flächen-Traversierung über Berge/Strände Recherche

Sprint	**Sprint 12**	**Sprint 13**
Phase	Transition	Transition
Milestones	*Abgabe SA*	*Abgabe SA*
Arbeitspakete	1. Restarbeiten 2. Doku aufräumen 3. Abgabe vorbereiten	1. Abgabe SA

Tabelle 8: Phasen / Iterationen und Meilensteine

10.3 Risiken

Tabelle 9: Risiken

ID	Risiko	max. Schaden [h]	WSK	gewichteter Schaden
1	nicht-existentielle Flächenverarbeitungs-algorithmen	32	40%	12.8
2	Schwierigkeiten mit der Implementation der Algorithmen	16	20%	3.2
3	Technologie eignet sich nicht für die Vorverarbeitung	32	20%	6.4
4	viele Abhängigkeiten auf Fremdsysteme	8	20%	1.6
5	Fremdsysteme bieten nicht die geforderte Funktionalität	32	30%	9.6
6	Scope zu gross angesetzt	24	30%	7.2

Das Hauptrisiko der Arbeit ist der grosse theoretische Aspekt, welcher viele Unbekannten bietet. So ist nicht klar, ob der aktuelle Stand der Technik genügend Material als Grundlage liefert, um das Problem der Traversierung von Flächen zu beheben. Auch ist nicht bekannt, ob sich Python als Technologie eignet, um grosse Mengen an OSM-Daten zu verarbeiten. Die Arbeit mit Fremdsystemen birgt in vieler Hinsicht Risiken. Ob die Fremdsysteme die gewünschten Funktionalität in dem Umfang liefert, welcher benötigt wird, muss zuerst abgeklärt werden. Dass eine Routing-Engine beispielsweise problemlos auf den von uns vorverarbeitenden OSM operieren kann, muss zwingend schnellstmöglich sichergestellt werden, da dies massgeblich verantwortlich für den Erfolg der Arbeit ist.

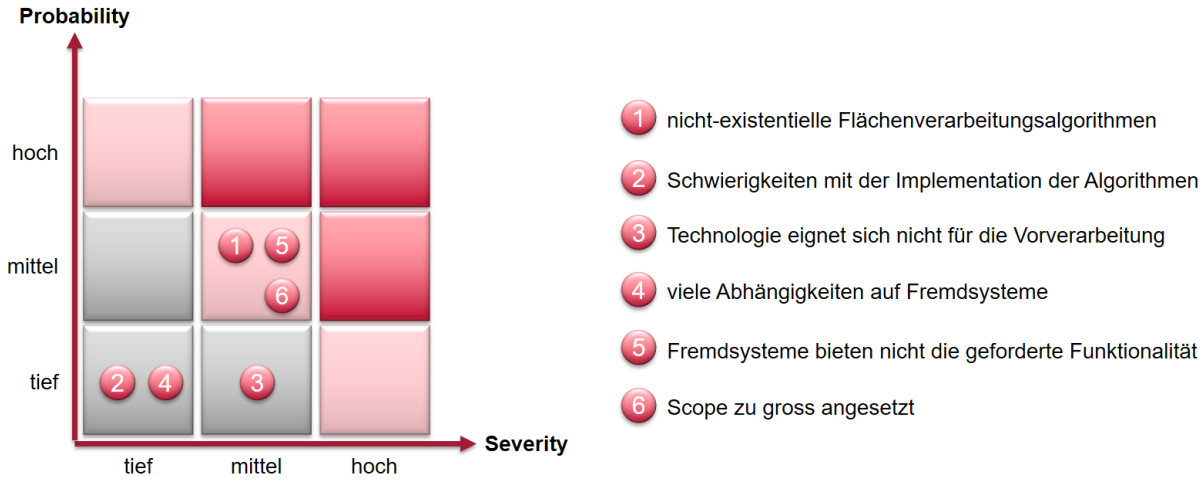

Abbildung 40: Risiko Analyse

10.3.1 Umgang mit Risiken

Um dem Hauptrisiko des Projekts, sprich die Evaluation, Optimierung und Implementation der Flächenverarbeitungsalgorithmen entgegen zu wirken, ist die erste Elaboration-Phase 4 Sprints lang und die Umsetzung der Algorithmen zu Beginn der Construction-Phase angesiedelt, um so genügend Spiel-

raum zu haben. Treten in der ersten Elaboration-Phase Probleme mit diesem Punkt auf oder man sieht, dass sich Python für diesen Zweck nicht eignet, muss dementsprechend der Scope des Frontends (QGIS-Plugin) oder der zweite theoretische Teil gekürzt werden. Dies aus dem einfachen Grund, da ohne diese Grundlage auch keine Visualisierung und weitere theoretische Aufbereitung Sinn macht. Nach der Elaboration ist auch bekannt, ob alle Fremdsysteme die Anforderungen erfüllen. Ist dies nicht der Fall, können bei search.ch [9] Change-Requests beantragt werden. Je nach Priorisierung dieser durch search.ch muss der Zeitplan aktualisiert und der Scope gekürzt werden. Kann die Routing-Engine auf den vorverarbeitenden OSM-Daten nicht operieren, so muss weiteren Aufwand in Anpassung der Verarbeitung der OSM-Daten gesteckt werden, was ebenfalls eine Kürzung des Scopes zur Folge hat.

10.3.2 Konsequenz

Tritt eines der Hauptrisiken (WSK > 9) ein, muss der Prototyp (QGIS-Plugin) oder die zweite theoretische Aufarbeitung aus dem Scope gestrichen werden. Die kleineren Risiken vermindern die Funktionsweise des Prototyps.

10.4 Team, Rollen und Verantwortlichkeiten

Tabelle 10: Team, Rollen und Verantwortlichkeiten

	Rolle	**Verantwortlichkeiten**
Prof. Stefan Keller	Betreuer	
Robin Suter	Autor	Plaza Preprocessing
Jonas Matter	Autor	PlazaRouting Backend, QGIS-Plugin

11 Softwaredokumentation

Die Installations- und Benutzeranleitungen zu den Komponenten PlazaRoute und QGIS-Plugin werden in den jeweiligen Github-Repositories [23] [24] gehalten.

Für einen reibungslosen Einstieg sind hier die Verweise auf die Anleitungen der einzelnen Komponenten aufgeführt.

11.1 Plaza Vorverarbeitung

Die Installations- und Benutzeranleitungen für *Plaza Vorverarbeitung* befindet sich unter [58].

11.2 Plaza Routing und Routing-Engine

Die Installations- und Benutzeranleitungen für *Plaza Routing* und der Routing-Engine befindet sich unter [59].

Die API-Spezifikation, die von Plaza Routing implementiert wird, ist unter [36] aufrufbar.

11.3 QGIS-Plugin

Die Installations- und Benutzeranleitungen für das QGIS-Plugin befindet sich unter [24].

Glossar

Bouding Box Ein Rechteck, welches durch zwei Längen- und Breitengrade definiert ist und nomalerweise dem Format *min Longitude , min Latitude , max Longitude , max Latitude* folgt. Im Kontext von Geometrien beschreibt eine *minimale Bounding Box* das kleinstmögliche Rechteck, dass alle Punkte einer oder mehrerer Geometrien beinhaltet.

Contraction Hierarchies Ein Ansatz zur Optimierung von Routing, in dem der Routing-Graph hierarchisch erzeugt wird.

Einstiegspunkt Ein Punkt auf dem Rand eines äusseren Polygons einer Fussgängerfläche, der sich mit einer bestehenden Strasse oder mit einem Fussweg schneidet oder eine solche Linie berührt.

Geocoding Der Prozess, einer Postadresse eine Koordinate zuzuordnen. Der umgekehrte Weg, das Bestimmen einer Postadresse aus einer Koordinate, nennt sich *Reverse Geocoding*.

Kante Eine ÖV-Haltestelle kann mehrere Plattformen umfassen. Normalerweise gehören zu einer ÖV-Haltestelle zwei Plattformen, je eine in jede Fahrtrichtung. Eine dieser Plattformen wird allgemein als Kante bezeichnet. Nicht zu verwechseln mit einer Kante eines Graphen, die eine Verbindung zwischen zwei Graph-Knoten beschreibt.

Node Ein geografischer Punkt im Modell von OpenStreetMap.

OpenStreetMap Ein Community-Projekt mit dem Ziel, eine frei verfügbare Karte der Erde zu erstellen, die von jedem bearbeitet und ergänzt werden kann.

Plaza Wird in dieser Arbeit als Synonym für Fussgänger-Fläche verwendet.

QGIS Ein frei verfügbares Geoinformationssystem für den Desktop für die Anzeige, Bearbeitung und Analyse von geografischen Daten.

Routing-Engine Eine Software, die aus Kartendaten einen Graphen aufbereitet und Funktionalitäten anbietet, um auf diesen Routen zu berechnen.

Shortest-Path Ein Begriff aus der Graphentheorie. Ein Shortest-Path beschreibt den kürzest möglichen Pfad zwischen zwei Knoten in einem gewichteten Graphen.

Tag Ein Schlüssel-Wert-Paar, das in OpenStreetMap einem Objekt zugewiesen wird, um dieses zu beschreiben. Tags können z.B. beschreiben, ob es sich bei einer Strasse um eine Haupt- oder Nebenstrasse handelt.

Way Eine Linie in OpenStreetMap, die gebildet wird, in dem mehrere Nodes verknüpft werden. Wenn ein Way einen geschlossenen Ring bildet, kann er auch eine Fläche beschreiben.

Abkürzungsverzeichnis

OSM OpenStreetMap

GIS Geographic Information System

RUP Rational Unified Process

PBF Protocolbuffer Binary Format

GEOS Geometry Engine, Open Source

WKB Well-known Binary

API Application Programming Interface

QL Query Language

UML Unified Modeling Language

REST Representational state transfer

OAS OpenAPI Specification

CI Continuous Integration

TDD Test-Driven Development

Literaturverzeichnis

[1] “Area — openstreetmap wiki,,” 2016, [Online; accessed 14-Oktober-2017]. [Online]. Available: https://wiki.openstreetmap.org/wiki/Area

[2] “Relation:multipolygon — openstreetmap wiki,,” 2017, [Online; accessed 15-Oktober-2017]. [Online]. Available: https://wiki.openstreetmap.org/wiki/Relation:multipolygon

[3] “Opengis simple features specification for ole/com — revision 1.1,” Open GIS Consortium, Inc., Standard, May 1999.

[4] “Key:barrier — openstreetmap wiki,,” 2017, [Online; accessed 19-Dezember-2017]. [Online]. Available: https://wiki.openstreetmap.org/wiki/Key:barrier

[5] “Graphhopper,” 2017, [Online; accessed 25-Oktober-2017]. [Online]. Available: https://www.graphhopper.com/

[6] A. Graser, “Integrating open spaces into openstreetmap routing graphs for realistic crossing behaviour in pedestrian navigation,” vol. 1, pp. 217–230, 06 2016.

[7] D. Dzafic, S. Klug, D. Franke, and S. Kowalewski, “Routing über flächen mit spiderwebgraph,” 07 2015.

[8] “Osm data for switzerland,” 2017, [Online; accessed 20-Oktober-2017]. [Online]. Available: https://planet.osm.ch/

[9] “Fahrplan - api spezifikation - search.ch,” 2017, [Online; accessed 29-Oktober-2017]. [Online]. Available: https://timetable.search.ch/api/help

[10] O. Wiki, “Overpass api — openstreetmap wiki,,” 2017, [Online; accessed 19-November-2017]. [Online]. Available: http://wiki.openstreetmap.org/w/index.php?title=Overpass_API&oldid=1525290

[11] “Openstreetmap nominatim,” 2017, [Online; accessed 20-December-2017]. [Online]. Available: https://nominatim.openstreetmap.org/

[12] T. Lozano-Pérez and M. A. Wesley, “An algorithm for planning collision-free paths among polyhedral obstacles,” *Commun. ACM*, vol. 22, no. 10, pp. 560–570, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/359156.359164

[13] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner, *A Novel Type of Skeleton for Polygons.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 752–761. [Online]. Available: https://doi.org/10.1007/978-3-642-80350-5_65

[14] D. H. Douglas, "Least-cost path in gis using an accumulated cost surface and slopelines," *Cartographica: the international journal for Geographic Information and Geovisualization*, vol. 31, no. 3, pp. 37–51, 1994.

[15] "Cost path analysis," 2016, [Online; accessed 5-December-2017]. [Online]. Available: http://wiki.gis.com/wiki/index.php/Cost_Path_Analysis

[16] P. Bolstad, *GIS Fundamentals: A First Text on Geographic Information Systems.* Eider Press, 2005.

[17] M. J. De Smith, M. F. Goodchild, and P. A. Longley, "Geospatial analysis," *Matador*, 2009.

[18] "Sidewalks — openstreetmap wiki,," 2017, [Online; accessed 18-Dezember-2017]. [Online]. Available: http://wiki.openstreetmap.org/wiki/Sidewalks

[19] N. Lang, "Entwicklung eines datenmodells zur fußgängernavigation auf basis von openstreetmap-daten," 2016, [Online; accessed 18-Dezember-2017]. [Online]. Available: https://github.com/Nathanael-L/pedro

[20] "Improving sidewalks globally in openstreetmap — mapbox," 2017, [Online; accessed 18-Dezember-2017]. [Online]. Available: https://blog.mapbox.com/improving-sidewalks-globally-in-openstreetmap-216bf34cca22

[21] S. Bühler and S. Kurath, "Extraction of crosswalks from aerial images," HSR Hochschule für Technik Rapperswil, Student Research Project, 2015. [Online]. Available: http://eprints.hsr.ch/id/eprint/485

[22] D. H. DOUGLAS and T. K. PEUCKER, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[23] "Github: Plazaroute," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://github.com/PlazaRoute/plazaroute

[24] "Github: Plazaroute," 2017, [Online; accessed 15-December-2017]. [Online]. Available: https://github.com/PlazaRoute/qgis-plugin

[25] F. Ramm, "Routing engines für openstreetmap," vol. 1, pp. 12–31, 03 2017.

[26] "Osrm," 2017, [Online; accessed 13-November-2017]. [Online]. Available: http://project-osrm.org/

[27] "Valhalla," 2017, [Online; accessed 25-Oktober-2017]. [Online]. Available: https://github.com/valhalla

[28] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, July 1968.

[29] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec 1959. [Online]. Available: https://doi.org/10.1007/BF01386390

[30] "C4 model," 2017, [Online; accessed 25-Oktober-2017]. [Online]. Available: https://c4model.com/

[31] "Structurizr," 2017, [Online; accessed 25-Oktober-2017]. [Online]. Available: https://structurizr.com/

[32] "Pyosmium," 2017, [Online; accessed 22-Oktober-2017]. [Online]. Available: http://osmcode.org/pyosmium/

[33] "Shapely," 2017, [Online; accessed 22-Oktober-2017]. [Online]. Available: https://github.com/Toblerity/Shapely

[34] "Osmosis," 2017, [Online; accessed 07-November-2017]. [Online]. Available: https://github.com/openstreetmap/osmosis

[35] "Openapi specification," 2017, [Online; accessed 19-November-2017]. [Online]. Available: https://swagger.io/specification

[36] "Plaza routing api spezifikaiton," 2017, [Online; accessed 19-November-2017]. [Online]. Available: https://plazaroute.github.io/api

[37] "Plaza routing api swaggerui," 2017, [Online; accessed 19-November-2017]. [Online]. Available: https://plazaroute.github.io/api/swagger-ui

[38] "Github: Plazaroute/api," 2017, [Online; accessed 24-November-2017]. [Online]. Available: https://github.com/PlazaRoute/api

[39] "Swagger," 2017, [Online; accessed 19-November-2017]. [Online]. Available: https://swagger.io

[40] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.

[41] "Maturity model," 2017, [Online; accessed 02-December-2017]. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html

[42] "Plazaroute api-dokumentation," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://plazaroute.github.io/plazaroute/

[43] A. Guttman, "R-trees: A dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984. [Online]. Available: http://doi.acm.org/10.1145/971697.602266

[44] "Flask," 2017, [Online; accessed 19-November-2017]. [Online]. Available: http://flask.pocoo.org

[45] "Flask-restplus," 2017, [Online; accessed 19-November-2017]. [Online]. Available: https://flask-restplus.readthedocs.io/en/stable

[46] R. Hanmer, *Patterns for Fault Tolerant Software*. John Wiley & Sons, Ltd, 2007.

[47] "Relation — openstreetmap wiki,," 2017, [Online; accessed 25-November-2017]. [Online]. Available: http://wiki.openstreetmap.org/w/index.php?title=Relation&oldid=1481354

[48] "Qgis plugin builder," 2017, [Online; accessed 07-December-2017]. [Online]. Available: https://g-sherman.github.io/Qgis-Plugin-Builder/

[49] "Circleci," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://circleci.com/

[50] "pytest," 2017, [Online; accessed 15-December-2017]. [Online]. Available: https://docs.pytest.org/en/latest/

[51] "Github: Plazaroute/doc," 2017, [Online; accessed 24-November-2017]. [Online]. Available: https://github.com/PlazaRoute/doc

[52] "Travis ci," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://travis-ci.org/

[53] "Sphinx doc," 2017, [Online; accessed 26-November-2017]. [Online]. Available: http://www.sphinx-doc.org/

[54] "Docker hub: Plazaroute," 2017, [Online; accessed 08-December-2017]. [Online]. Available: https://hub.docker.com/u/plazaroute/

[55] "uwsgi," 2017, [Online; accessed 08-December-2017]. [Online]. Available: https://uwsgi-docs.readthedocs.io/en/latest/

[56] "Nginx," 2017, [Online; accessed 08-December-2017]. [Online]. Available: https://nginx.org/

[57] "Docker compose," 2017, [Online; accessed 08-December-2017]. [Online]. Available: https://docs.docker.com/compose/overview/

[58] "Github: Plazaroute plaza vorverarbeitung," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://github.com/PlazaRoute/plazaroute/blob/master/plaza_preprocessing/README.md

[59] "Github: Plazaroute plaza routing," 2017, [Online; accessed 26-November-2017]. [Online]. Available: https://github.com/PlazaRoute/plazaroute/blob/master/plaza_routing/README.md

[60] "Tourpl - einfach touren planen..." 2017, [Online; accessed 20-Oktober-2017]. [Online]. Available: http://tourpl.ch/

[61] "Geometa lab at hsr," 2017, [Online; accessed 20-Oktober-2017]. [Online]. Available: https://www.ifs.hsr.ch/index.php?id=12520

Abbildungsverzeichnis

Tabellenverzeichnis

Code Listings