Projektdokumentation im Modul Semantic Web

Semantische Suchmaschine für das Web of Data: Trip-Planung für Sehenswürdigkeiten

`https://github.com/pnancke/trip-planner`
Matthias Zober

31.07.2016

Aufgabenstellung Umsetzung eines Trip-Planers, welcher unter Berücksichtigung von Sehenswürdigkeiten eine semantische Suchmaschine für das Web of Data anbieten soll.

1 Inhaltliche Interpretation der Aufgabenstellung

Routenplanung beschäftigt sich mit dem Finden einer optimalen Strecke. Übliche Abfragen eines Routenplaners werten Strecken anhand ihrer Längen oder ihrer benötigten Reisezeit aus. Doch sind meistens die Orte, die man durch eine Route erreicht, entscheidender als die Länge der Strecke. Stellt man sich also eine Expedition vor, bei der man möglichst viele Sehenswürdigkeiten (Point of Interest - POI) finden und erkunden will, scheitern die üblichen Suchmaschinen. Da die meisten Routenplaner semantische Abfrage wie "Welche POIs liegen auf den kürzesten Weg von A zu B" nicht berücksichtigen. Ziel ist es, einen Routenplaner zu entwerfen, der zu einer angemessenen Zeit die eben gestellte Frage beantworten kann und dabei eine einfache und benutzerfreundliche Bedienung zulässt.

Für das Planen eines Trips, ist es primär notwendig eine kürzeste Route berechnen zu können und POIs in einem gegebenen Bereich zu finden. Dieser Trip-Planer verwendet jedoch noch andere Datenquellen, um beispielsweise eine erleichterte Eingabe für den Benutzer zu ermöglichen. Welche Datenquellen im Detail verwendet werden und welche Überlegungen konkret bei der Umsetzung berücksichtigt worden, wird in der folgenden Produktdokumentation verdeutlicht.

2 Relevante Datenquellen

Anfangs werden in diesem Kapitel die Datenquellen für die erleichterte Bedienung des Trip-Planers präsentiert. Danach folgen Schnittstellen die für die Routen-Abfrage und die Abfrage der POIs verwendet worden.

2.1 Erleichterte Bedienung

Bevor eine optimale Route für einen Trip geplant werden kann, müssen mindestens Start- und Zielpunkt von dem Nutzer angegeben werden. Da die Angabe von bloßen Geo-Koordinaten sehr unhandlich erscheint, soll eine vereinfachte Eingabe durch zwei separate Suchfelder für die Eingabe der Namen ermöglicht werden. Eine Möglichkeit zur erheblichen Vereinfachung dieser Eingabe ist es, eine Autocompletion API zu benutzen. Da damit der Nutzer in Echtzeit Namensvorschläge auf seine eingegebenen Namensvorschläge erhält.

2.1.1 Autocompletion

Die verwendete Autocompletion API, namens *Photon*, stellt eine speziell für *OpenStreetMap (OSM)*[1] Daten ausgerichteten Open-Source-Geocoder dar, der basierend auf *elasticsearch*[2] eine effiziente, hoch skalierbare Suchplattform anbietet.

Link	`https://photon.komoot.de/api/?q=ADRESSE`
GitHub	`https://github.com/komoot/photon`
Datenformat	JSON
Schnittstelle	Rest-API
Lizenz	Apache License, Version 2.0
Open Data	⋆⋆⋆⋆⋆

2.1.2 Textinterpretation

Eine andere Möglichkeit die Eingabe für den Benutzer stark zu vereinfachen bietet die Textinterpretation vom angegebene Adresstext, damit können die benötigten Koordinaten direkt aus dem eingegebenen Text ermittelt werden. Dadurch dass die bloße Interpretation des angegebenen Texts weniger Aufwand für den Nutzer bedeutet, da lediglich

[1] http://www.openstreetmap.org/
[2] https://www.elastic.co/de/

ein Kürzel der Adresse eingegeben werden muss, ist es auch sinnvoll diese Technologie für die vereinfachte Eingabe zu verwenden.

Die verwendete API zur Textsuche für Geo-Koordinaten nennt sich *Nominatim*. Sie basiert auch wie *Photon* auf den *OSM* Daten und wird sogar von *OSM* selbst für die Geo-Suche verwendet.

Link	`http://nominatim.openstreetmap.org/search?q=ADRESSE`
GitHub	`https://github.com/twain47/Nominatim`
Datenformat	HTML
Schnittstelle	Rest-API
Lizenz	GPLv2
Open Data	⋆⋆⋆⋆⋆

2.2 Routing

Nach der Eingabe des Start- und Zielorts, kann die kürzeste Route zwischen diesen ermittelt werden, um später einen optimalen Trip zu ermitteln.

Hierfür wird eine Routing API verwendet, mit dem Namen *YourNavigation*, sie basiert auch auf dem Kartenmaterial der *OSM* und berechnet die kürzeste oder schnellste Route zwischen den zwei angegebenen Geo-Koordinaten. Problematisch an dieser Schnittstelle ist, dass sie keine weiteren Wegpunkte zwischen Start- und Zielpunkt zulässt und damit eine Einschränkung bei der semantischen Suche von einer optimalen Route darstellt. Dennoch verwenden wir diese API, da sie *Open-Source* ist und zu einer angemessenen Zeit eine Route finden kann.

Link	`http://www.yournavigation.org/api`
Datenformat	KML (ähnlich zu XML)
Schnittstelle	Rest-API
Lizenz	BSD
Open Data	⋆⋆⋆

2.3 Sehenswürdigkeiten

Wenn durch die Routing-API eine kürzeste Route gefunden wurde, können POIs in Nähe des Streckenabschnitts gesucht werden. Für diese Suche wird passend auf die Kartendaten von *OSM* zugegriffen, da sie zum einen eine große zuverlässige Open-Source-Quelle

für Kartenmaterial darstellt (mit 4,9 Milliarden GPS-Punkte[3]) und zum anderen alle anderen verwendeten APIs *OSM* als Basis verwenden. *OSM* - Daten bringen nicht nur den Vorteil, dass auf Kartendaten der gesamten Welt zugegriffen werden kann, auch werden Orte verschiedener Kategorie getaggt. Damit kann man eine Suchanfrage z.B. für alle POIs stellen.

OSM-Daten können über die Overpass-XAPI direkt gefiltert abgerufen werden oder lokal gehalten werden[4] und durch ein Command-Line-Tool, namens *osmctools*[5], entsprechend lokal gefiltert werden. Nachteilig an der Overpass-XAPI ist, dass Bereiche nur durch Vierecke abgefragt werden können. Außerdem benötigen größere Abfragen sehr viel Zeit, so dass diese API als Quelle für eine einmalige Berechnung ausreichen mag, jedoch für ständige Abfragen weniger geeignet ist.

Link	`http://www.overpass-api.de/api/xapi?`
Wiki	`http://wiki.openstreetmap.org/wiki/Overpass_API`
Datenformat	OSM (XML)
Schnittstelle	Rest-API
Lizenz	Affero GPL v3
Open Data	⋆⋆⋆⋆⋆

3 Extraktion der Daten

Der Trip-Planer wurde unter dem Framework *Groovy on Grails*[6] programmiert. Da der Trip-Planer eine dynamische Suchmaschine darstellen soll, werden Suchanfragen durch *Groovy*-Programm-Code in Echtzeit aufbereitet, extrahiert und beantwortet.

3.1 REST-Anfragen

Groovy bietet leichtgewichtige Umsetzungen für *REST*-Anfragen, die von allen APIs benötigt werden, als auch für die Extraktion von *XML, HTML, JSON* und weiteren Formaten an.

Im folgenden wird ein Code-Beispiel gezeigt, welches die REST-Abfrage in *Groovy* zeigt.

[3]Stand Dezember 2015: `http://www.openstreetmap.org/stats/data_stats.html`
[4]Download unter `http://download.geofabrik.de/`
[5]`https://github.com/mapsme/osmctools`
[6]https://grails.org/

Code-Beispiel 1: REST-Abfrage in Groovy

```
String url =
    "http://nominatim.openstreetmap.org/search?q=$q&format=xml&addressdetails=0&limit=1"
response = new RestBuilder().get(this.url).responseEntity
String xmlContent = response.metaPropertyValues.get(2).value
HttpStatus status = response.metaPropertyValues.get(3).value as HttpStatus
```

Wie durch Codebeispiel 1 gezeigt wird, ist es sehr einfach REST-Anfragen (wie GET) mit *Groovy* zu realisieren und dabei den Inhalt und den Status der Http-Antwort zu extrahieren.

3.2 Datenextraktion

Im Trip-Planer werden Daten aus dem REST-Anfragen extrahiert und weiterverarbeitet. Mit dem nächsten Codebeispiel sei ein vereinfachtes *KML*-Dokument gegeben, welches die Routen-Koordinaten der *YourNavigation-API* übermitteln soll.

Code-Beispiel 2: vereinfachte KML mit Routen-Koordinaten

```
<kml xmlns="http://earth.google.com/kml/2.0">
    <Document>
        <Folder>
            <Placemark>
                <LineString>
                    <coordinates>
                        11.458035,51.997480 11.464201,51.998336 11.888245,52.306744
                    </coordinates>
                </LineString>
            </Placemark>
        </Folder>
    </Document>
</kml>
```

Mit dem folgenden *Groovy*-Programmcode, kann aus der gegebenen *KML* (`kml` in Codebeispiel 3) eine Liste von Zeichenketten extrahiert werden, diese Liste enthält dann die drei angegebenen Koordinaten.

Code-Beispiel 3: Extraktion der Routen-Koordinaten mit XmlSlurper

```
def xml = new XmlSlurper().parseText(kml)
List<String> routeCoordinates =
    xml.Document.Folder.Placemark.LineString.coordinates.text().trim().tokenize("\n")
```

Durch den `XmlSlurper`[7] kann im Codebeispiel 3 direkt zu den Kindknoten mit dem angegebenen Namen navigiert werden. So kann der Inhalt des gesamten `Folder`, beispielsweise mit `xml.Document.Folder` extrahiert werden. Nicht nur Inhalte von XML-ähnlichen Dokumenten können durch *Groovy* Bibliotheken extrahiert werden, auch können JSON ähnliche Dokumente durch den *JsonSlurper*[8] verarbeitet werden.

Das nächste Kapitel beschäftigt sich mit der Konzeption, den Herausforderungen und der letztendlichen Umsetzung des Trip-Planers.

4 Umsetzung des Trip-Planers

Wie anfangs beschrieben wurde, besteht die grundlegende Aufgabenstellung einen Trip-Planer zu entwerfen der einen optimalen Trip in endlicher Zeit berechnen kann. Da subjektive Meinungen über die gefundene Route des Trip-Planers ausgeschlossen werden, wird ein Trip als optimal definiert, wenn er in einer annehmbaren Zeit erfüllt werden kann und möglichst viele POIs abdeckt.

Damit die gefundene Route nicht zu viel Zeit erfordert, wird erst die kürzeste Route zwischen Start- und Zielort ermittelt. Anhand dieser kürzesten Route können POIs in der Nähe der kürzesten Route gefunden werden. Am Ende wird der eigentliche Trip aus der Verbindung der gefundenen Sehenswürdigkeiten berechnet. Da viele Sehenswürdigkeiten bereist werden sollen, aber dennoch eine Route ähnlich zum kürzesten Weg berechnet werden soll, befasst sich die erste Aufgabe mit dem Filtern der relevanten Sehenswürdigkeiten.

4.1 Sehenswürdigkeiten filtern

Sehenswürdigkeiten sind relevant für den Trip-Planer, wenn sie nicht an entlegenen Orten sind. Ferner ist es auch vom Vorteil, wenn sich andere Sehenswürdigkeiten in der Nähe aufhalten. Es kann davon ausgegangen werden, dass an einem Ort mit vielen POIs öfters bedeutsame Orte vorkommen (z.B. in Großstädten), als in Ortschaften in denen vereinzelt POIs zu finden sind (z.B. Dörfer). Für die Filterung, ist es deshalb notwendig, nicht nur in der Nähe der kürzesten Route zu suchen, sondern auch die POIs heraus zu filtern, die keine POIs in ihrer unmittelbaren Umgebung besitzen. Folglich wird erst das Clustering betrachtet, gefolgt von der Abfrage eines Bereiches entlang der Route (Buffer).

[7] http://docs.groovy-lang.org/latest/html/api/groovy/util/XmlSlurper.html
[8] http://docs.groovy-lang.org/latest/html/gapi/groovy/json/JsonSlurper.html

4.1.1 Clustering

Für das Clustering der POIs wird das *ELKI Data Mining Framework*[EA08] verwendet, welches unter Nutzung des *K-Means Algorithmus*, Cluster mit einem definierten Maximaldurchmesser aus der gegebenen Menge aller POIs berechnet.
In der folgenden Abbildung ist ein Vorher-Nachher Vergleich für das Clustering zu sehen.

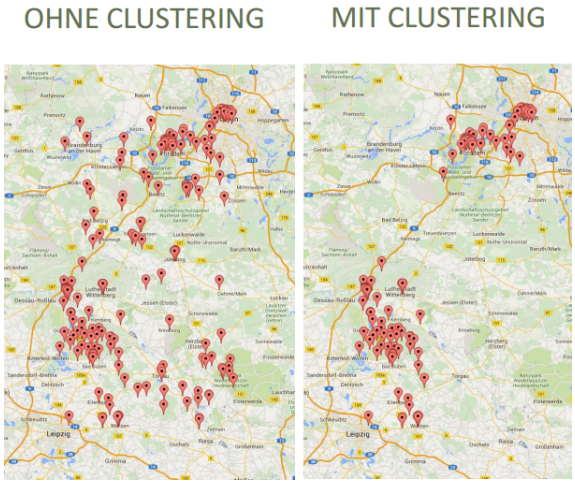

Abbildung 1: Vergleich mit und ohne Clustering

Wie man durch Abbildung 1 sieht, werden weiter entfernte POIs aus dem Bild heraus gefiltert und störende Zerstreuungen entfernt. Ein weiterer Vorteil des Clusterings ist die Information über das Clusterzentrum, so kann man diese für die Erzeugung des resultierenden Trips verbinden, anstatt jeden einzelnen POI zu verbinden. Damit wird dem Nutzer die Möglichkeit gegeben, Bereiche mit hoher POI-Dichte zu erkunden, anstatt statisch jedem Punkt zu bereisen. Diese Vereinfachung erleichtert das Routing für den resultierenden Trip erheblich .

Weiter entfernte POIs können nur schlecht durch das Clustering gefiltert werden, wie durch die folgende Darstellung gezeigt wird.

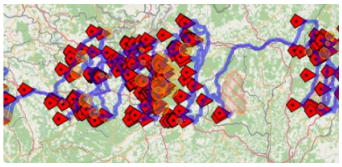

Abbildung 2: Route ohne Filterung weit entfernter POIs

Für die Filterung weit entfernter POIs ist eine passende Suchanfrage mit einem sog. Buffer notwendig, diese werden als nächstes näher erläutert.

4.1.2 Abfrage eines Buffers

Aus den genannten Gründen von Kapitel 2.3, kann für die Umsetzung einer Buffer-Abfrage die *Overpass-API* nicht verwendet werden. Stattdessen wird auf die lokale Speicherung der *OSM*-Kartendaten, innerhalb einer *MySQL 5.6* Datenbank zurückgegriffen, da sich daraus nicht nur Performance-Vorteile ergeben, sondern auch geometrische Abfragen unterstützt werden.
Auf die Kartendaten kann innerhalb der Anwendung, durch die Frameworks *Hibernate* und *GORM* zugegriffen werden. *GORM* stellt das Objekt-Relationale Mapping für *Grails* Anwendung dar, *Hibernate* hingegen kann eine direkte Verbindung zur Datenbank herstellen. Bevor eine Abfrage getätigt werden kann, muss jedoch ein Buffer-Polygon für die geometrische Abfrage erzeugt werden.
Das folgende Codebeispiel zeigt den Programm-Code für die serverseitige Erstellung des Buffer-Polygons, mithilfe des Frameworks *vividsolution-jts*[9], welches Java-Repräsentation von geometrischen Objekten im *MySQL*-Stil bereitstellt.

Code-Beispiel 4: serverseitige Polygon-Darstellung des Buffers

```
LineString lineString = factory.createLineString(route);
Geometry buffer = lineString.buffer(area)
List<Point> bufferShape = buffer.getCoordinates().toList().stream()
     .map { c -> new Point(c.x, c.y) }
     .collect(Collectors.toList())
bufferShape.add(bufferShape.first())
```

Das resultierende geometrische Objekt (`bufferShape`) verkörpert ein Polygon, welches einen Buffer (`buffer`) von einer Route (erst `route`, dann `lineString`) mit einem Abstand `area` repräsentiert. Damit entspricht das erzeugte Polygon den gesuchten Bereich entlang einer Route. Nach der Konstruktion des Polygons folgt die *Hibernate*-Abfrage, wie im folgenden Codefragment dargestellt.

Code-Beispiel 5: Beispiel Datenbankabfrage mittels Hibernate

```
query = "SELECT * FROM point_of_interest (point) "+
        "WHERE ST_CONTAINS(GeomFromText(:polygon), point)"
polygon = 'Polygon((0 0,0 1,1 1,1 0,0 0))' // vorher erzeugtes Polygon
List<PointOfInterest> result = new ArrayList<>()
PointOfInterest.withSession { SQLQuery sqlQuery = it.createSQLQuery(query)
    result = sqlQuery.addEntity(PointOfInterest).setString('polygon', polygon).list() }
```

[9] http://www.vividsolutions.com/jts/jtshome.htm

Wie im Codebeispiel 5 zu sehen ist, kann das `SELECT`-Statement durch eine Session von der Klasse `PointOfInterest` erzeugt werden. Diese Klasse stellt eine Domain-Klasse des Frameworks *GORM* dar. Sie besitzt alle Felder der Datenbank-Tabelle `point_of_interest` und verkörpert damit einen POI. Da *GORM* selbst eine Erweiterung des *Hibernate*-Frameworks darstellt, kann man beide Bibliotheken kombiniert einsetzen und alle POIs innerhalb des Buffers (`result`) abfragen.

4.2 Performance-Verbesserungen

Der Trip-Planer soll möglichst schnell eine Route finden, deshalb wurden während der Umsetzung Performance-Optimierungen vorgenommen. Die erste Optimierung wurde an der Datenbankabfrage vorgenommen, da *MySQL 5.6* neben den geometrischen Abfragen auch Performance-optimierte Spatial Indizes eingeführt hat. Diese Indizes werden verwendet, um weniger Vergleiche für eine Anfrage zu benötigen und damit schließlich die Performance zu erhöhen.

4.2.1 POI-Abfrage

Für die Performance-Verbesserung musste zunächst eine Datenbank-Tabelle für die POIs (`point_of_interest`) erzeugt werden. Diese Tabelle muss ihre Geo-Koordinaten nun in einem Attribut mit dem Typ `Point` abspeichern, da dieser Spatial Indizes unterstützt, ferner muss das Attribut auch als `SPATIAL INDEX` markiert werden und der `ENGINE`-Typ `MyISAM` gesetzt werden. Das nächste Codefragment zeigt das `CREATE-TABLE`-Statement.

Code-Beispiel 6: Beispiel für CREATE-TABLE-Statement

```
CREATE TABLE point_of_interest (
 poi_id BIGINT unsigned NOT NULL auto_increment , PRIMARY KEY (poi_id)
 , point POINT NOT NULL , SPATIAL INDEX(point)
 , wikipedia TEXT, ... // Wikipedia Integration moeglich durch den Wiki-Tag
) ENGINE=MyISAM;
```

Nachdem die Datenbank neu konfiguriert wurde, kann eine optimierte Abfrage gestellt werden. Der folgende Code zeigt das veränderte `SELECT`-Statement.

Code-Beispiel 7: optimiertes SELECT-Statement

```
SELECT * FROM point_of_interest FORCE INDEX (point)
WHERE ST_CONTAINS(GeomFromText(:polygon), point) = TRUE
```

Das optimierte Statement verwendet Spatial Indizes, dadurch vergleicht die Funktion `ST_CONTAINS` nicht mehr alle POIs, sondern wesentlich weniger, wodurch die Performance drastisch verbessert wird. Eine Abfrage von Leipzig nach München würde zehn Sekunden benötigen, wenn alle POIs von ganz Deutschland (ca. 16000) untereinander verglichen werden. Verwendet man Spatial Indizes benötigt die gleiche Abfrage nur 370 Vergleiche und drei Millisekunden.
Nicht nur die POI-Abfrage konnte verbessert werden, sondern auch die Performance des Routings.

4.2.2 Routing

Für das Routing wurde die *YourNavigation* API (siehe Kapitel 2.2) verwendet, diese stellt eine Open-Source-Lösung dar. Der Nachteil an dieser API ist, dass sie keine Zwischenstopps zulässt und sich deshalb schlecht für das verbinden aller POI-Clusterzentren anbietet, außerdem weist sie schwankende Abfragezeiten auf.
Dennoch haben wir uns für diese API entschieden, da auf andere APIs mit besseren Abfragezeiten leider nicht zugegriffen werden kann, da diese nur kommerziell nutzbar sind.
Um dennoch eine Route in einer annehmbaren Zeit zu berechnen, wird die Berechnung des Trips parallel ausgeführt. Die parallele Ausführung wird realisiert, indem die komplette Abfrage geteilt wird in kleinere Abfragen. Diese besitzen nur noch ein Start- und Zielpunkt und können parallel von der Routing-API verarbeitet werden. Damit die Last der Routing-API nicht überschritten wird, wird zusätzlich nach einer gewissen Anzahl von Requests synchronisiert.

4.3 Weboberfläche

Die Weboberfläche wurde mit Hilfe von *Groovy-Server-Pages (GSPs)* programmiert, diese gehören zum *Grails*-Standard und bietet die Möglichkeit an, Webinhalte über *HTML* darzustellen und Zugriff auf Server-Funktionalitäten der definierten Web-Controller zu erhalten.
Der Inhalt der Weboberfläche besteht aus einem Eingabefeld und einer Weltkarte, welche die gefundene Route anzeigen soll. Die Visualisierung der Weltkarte, wurde mit Hilfe des leichtgewichtigen *JS*-Framework *OpenLayers*[10] realisiert, dieses Framework realisiert auch die Weboberfläche von *OSM*.
Im folgenden wird die Weboberfläche anhand einer umzusetzenden Funktionalität präsentiert.

[10] http://openlayers.org/

Verstellbarer Suchbereich Der Suchbereich entlang des kürzesten Wegs kann je nach Bedarf verstellt werden, dies hat den Nutzen, dass bei kleineren Expeditionen weit entferntere POIs heraus gefiltert, bei einem größeren diese wieder betrachtet werden können. Der Bereich ist von einem Durchmesser von acht bis 40 Kilometer verstellbar in fünf Abstufungen. In der folgenden Abbildung sieht man die Weboberfläche im Vergleich, zum einen wird eine Abfrage von 8 (1) und zum anderen von 40 Kilometern (2) im Überblick dargestellt.

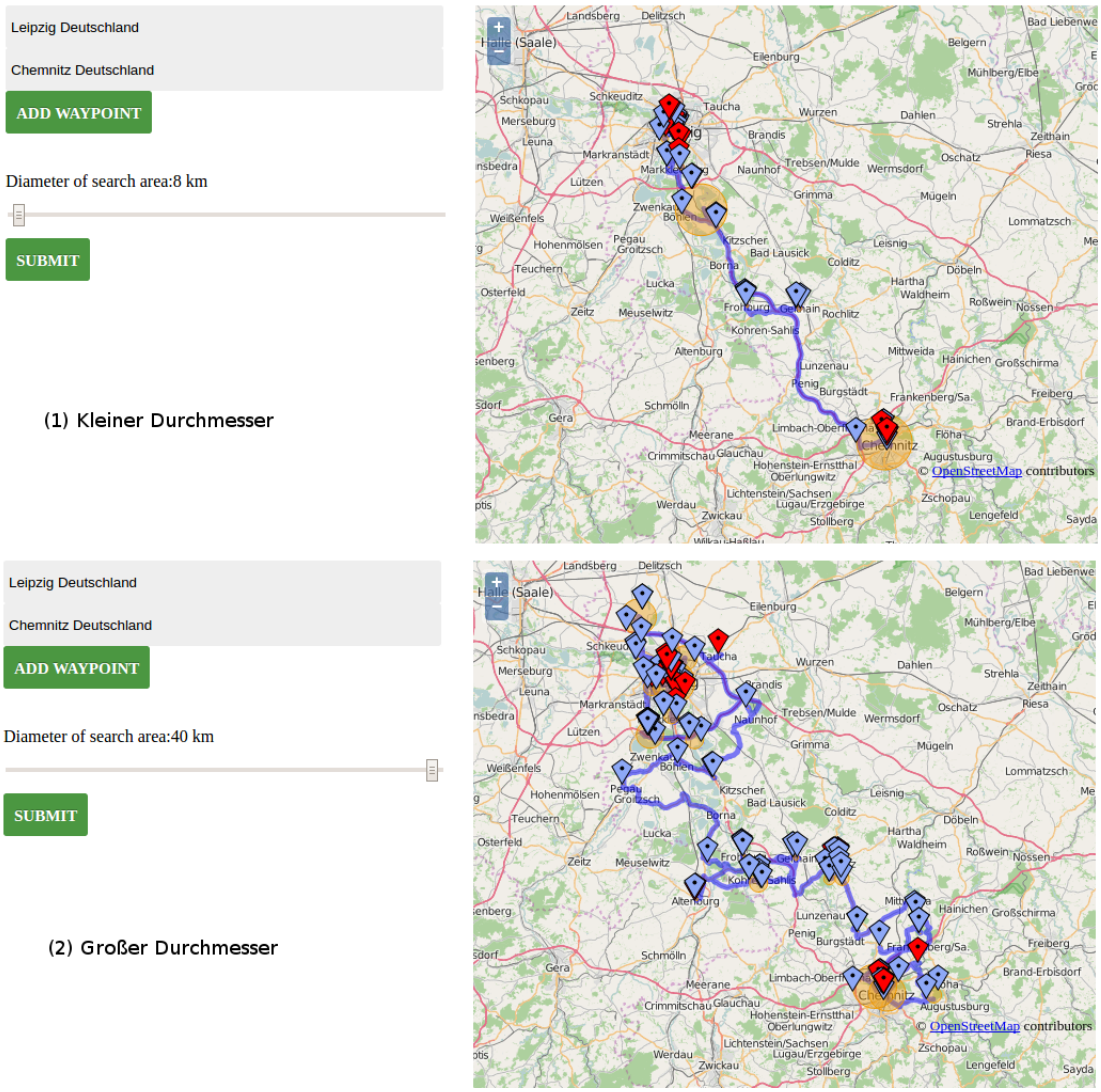

Abbildung 3: Vergleich kleiner und großer Suchbereich

5 Zusammenfassung

Die Aufgabenstellung einen Trip-Planer umzusetzen, wurde durch das Semantic Web Projekt erfüllt. Der Trip-Planer kann folglich eine optimierten Trip planen, der eine schnelle Route vom Start- zum Zielpunkt erfüllt und dennoch möglichst viele POIs beinhaltet. Neben dieser Hauptaufgabe, kann der Suchbereich einer Abfrage in verschiedenen Größen variiert werden. Ferner kann nicht nur der Name, sondern auch die Wikipedia-Verknüpfung zu den gefundenen POIs angeben werden (Vgl. Codebeispiel 6 - Wikipedia Link). Der Nutzer hat außerdem die Möglichkeit Zwischenstopps in die Route einzubeziehen.
Damit der Trip-Planer in Echtzeit eingesetzt werden kann, wurden Performance-Optimierungen beim Filtern der POIs und beim Routing vorgenommen und erfolgreich umgesetzt.
Der Trip-Planer verwendet viele verschiedenen Web-APIs zum Finden des optimalen Trips und stellt im Sinne des Web of Data eine semantische Suchmaschine dar.

Abbildungsverzeichnis

Verzeichnis der Code-Beispiele

Literatur

[EA08] Arthur Zimek Elke Achtert, Hans-Peter Kriegel. ELKI: A Software System for Evaluation of Subspace Clustering Algorithms. In *Proc. 20th International Conference on Scientific and Statistical Database Management (SSDBM 2008)*, Seiten 580–585, Hong Kong, China, 2008.