Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 21

Effiziente Verarbeitung von Anfragen nach Polygon-Enthaltensein im Kontext von OSM-Daten

Sascha Meusel

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. Stefan Funke
Betreuer:	Prof. Dr. Stefan Funke
begonnen am:	20. Juni 2012
beendet am:	20. Dezember 2012
CR-Klassifikation:	E.1, H.3.3, I.1.2

Kurzfassung

Diese Arbeit beschäftigt sich mit den Flächen-Informationen von OpenStreetMap-Daten und die effiziente Ermittlung der zu einer konkreten Koordinate vorhandenen relevanten OpenStreetMap-Flächendaten.

Die Flächen liegen in OpenStreetMap als sogenannte *ways* und *relations* vor. Ein Weg ist dann eine Fläche, wenn der Weg ein Polygon bildet und mit bestimmten *tags* (Schlüssel-Wert-Paare) markiert ist. Eine Relation ist dann eine Fläche, wenn die Relation mit bestimmten Tags markiert ist und mindestens ein Polygon enthält, also einen geschlossen Weg oder eine Kombination von Wegen enthält, die zusammen ein Polygon bilden. Die Informationen zu einer Fläche liegen als Tags des entsprechenden Weges oder der entsprechenden Relation vor. In dieser Arbeit wurde deshalb auch ein C++-Programm entwickelt, welches bestimmte Polygone und zugehörige Flächen-Informationen aus den OpenStreetMap-Daten extrahiert. Unterstützt wird das C++-Programm durch ein Python-Skript, welches die OpenStreetMap-Daten in ein leichter parsbares Textformat exportiert.

Desweiteren wurde in dieser Arbeit ein Programm entwickelt, welches die Polygone als Eingabedaten nimmt und daraus eine Datenstruktur aufbaut, die bei einer Anfrage mit einer Koordinate effizient die Polygone ermitteln kann, die diese Koordinate enthalten. Das Ergebnis auf eine Anfrage besteht aus den Polygonen in Koordinatenform und aus den zugehörigen Flächen-Informationen. Als Datenstruktur wurde ein Quadtree verwendet, der zur Reduzierung von Point-In-Polygon-Berechnungen eingesetzt wird.

Das Quadtree-Programm wurde anschließend in die Anwendung MapViewer der Abteilung Algorithmik am Insitut für Formale Methoden der Informatik der Universität Stuttgart eingebaut. Dadurch lassen sich die Anfragen über eine Graphische Oberfläche stellen, wobei die Polygone aus dem Ergebnis direkt grafisch auf einer OpenStreetMap-Karte angezeigt werden.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel gibt es einen kurzen Überblick auf die Gliederung der Bachelorarbeit und es wird das Thema der Arbeit näher vorgestellt. Dabei werde ich zuerst die Problemstellung allgemein erläutern, ohne auf die Besonderheiten einzugehen, auf die man dabei mit den Daten von OpenStreetMap beachten muss. Anschließend stelle ich OpenStreetMap vor, was insbesondere für diejenigen Leser interessant sein dürfte, die bisher wenig oder gar nichts über OpenStreetMap wissen. Als letzten Punkt der Einleitung werde ich kurz erläutern, mit welcher Vorgehensweise das Problem im Kontext von OpenStreetMap in dieser Arbeit gelöst wird.

1.1 Gliederung der Bachelorarbeit

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Extrahierung der Polygone aus OpenStreetMap: Dieses Kapitel beschäftigt sich mit dem Extrahieren der Polygone aus den OpenStreetMap-Daten. Es wird aufgezeigt, wie die Struktur der OpenStreetMap-Daten im Detail aussieht, wie ich die benötigten OpenStreetMap-Daten platzsparend aus Protocolbuffer Binary Format-Dateien extrahiere, und wie ich aus den OpenStreetMap-Daten die Polygon-Daten erzeuge.

Kapitel 3 – Quadtree-Datenstruktur: beschreibt die Eigenschaften eines Quadtrees und den daraus folgenden Nutzen für die Arbeit, und erläutert, wie ich den Quadtree aus den Polygon-Daten erzeuge, welche Struktur der Quadtree hat und wie die Anfragen auf dem Quadtree funktionieren.

1.2 Allgemeine Problemstellung

Gegeben ist eine Menge von Polygonen in einer zweidimensionalen Ebene. Für gegebene Koordinaten soll effizient ermittelt werden, in welchen Polygonen die Koordinaten jeweils liegen. Eine Koordinate besteht dabei aus einem zweidimensionalen Tupel. Ein Polygon kann als eine geordnete Sequenz von Koordinaten definiert werden, wobei die Koordinaten dann einen geschlossenen Ring bilden.

Um das Problem zu lösen, sollen die Polygone in einer geeigneten Datenstruktur gespeichert werden, so dass für beliebige Koordinaten die entsprechenden Polygone effizient gefunden werden. Die Idee dabei ist, dass die Datenstruktur im Voraus mit allen gegebenen Polygonen

erzeugt wird und zeitliche Dauer dafür vernachlässigbar ist. Sobald die Datenstruktur stehe, sollen beliebig viele Anfragen Koordinaten an die Datenstruktur gesendet werden können.

Die Datenstruktur könnte die räumliche Position und Größe der Polygone ausnutzen, um für jede Anfrage frühzeitig zu identifizieren, welche Polygone aufjedenfall die angefragte Koordinate enthalten, welche Polygone zu weit weg von der Koordinate sind, um die Koordinate enthalten zu können, und welche Polygone die Koordinate eventuell enthalten. Für den letzten Fall müssten dann teurere Point-In-Polygon-Berechnungen ausgeführt werden. Für solche Point-In-Polygon-Berechnungen kommt unter anderem der Ray-Casting-Algorithmus in Frage.

Jedem Polygon sind außerdem Informationen zugeordnet, die bei einer Anfrage im Ergebnis zusammen mit den Polygonen ausgegeben werden sollen.

1.3 Was ist OpenStreetMap?

OpenStreetMap ist ein Projekt zur Bereitstellung von freien geographischen Daten und Kartenmaterial. Vergleichbar mit Wikipedia kann jeder Daten beitragen und Anwendungen entwickeln, die die Daten nutzen. Dabei müssen aber gewisse lizenz- und urheberrechtliche Sachen beachtet werden. So darf das Beitragen von Daten nicht Rechte (Lizenzen, Urheberrecht) anderer verletzen.

OpenStreetMap-Daten bestehen hauptsächlich aus Knoten (nodes), Wegen (ways) und Relationen (relations). Jeder Knoten besitzt außerdem eine Koordinate, die aus einem Latitude- und einem Longitude-Wert nach dem Standard WGS 84 besteht. Wege sind Sequenzen aus Knoten stellen Straßen sowie Grenzen und Gebäude-Umrisse dar.

Das Kartenmaterial wird mit OSM-Renderern erzeugt, welche aus den OSM-Daten die Karten-Bild-Dateien erzeugen. So erzeugtes Kartenmaterial kann man zum Beispiel auf `http://www.openstreetmap.org/index.html` betrachten.

Zur Bearbeitung der OpenStreetMap-Daten existieren verschiedene Tools wie JOSM und Osmosis. JOSM ist eine Java-Anwendung, durch welche man über eine graphische Oberfläche die Elemente der Daten bearbeiten kann. Die Elemente werden entsprechend ihren Koordinaten graphisch dargestellt und können so auch intuitiv positioniert werden. Mit Osmosis kann man unter anderem gewünschte Daten aus den OpenStreetMap-Daten extrahieren, beispielsweise ausgewählt nach bestimmten Tags oder gewünschten geographischen Bereichen. Auch lassen sich damit aus OSM-XML-Daten speicherplatz-sparende PBF-Dateien (Protocollbuffer Binary Format-Dateien) erzeugen, und umgekehrt.

1.4 Das Problem im Kontext von OpenStreetMap

Um eine entsprechende Datenstruktur aufzubauen, sind Polygon-Daten nötig. Anstatt zuerst zufallsgenerierte Polygone zu verwenden, habe ich in der Arbeit die Polygone aus den OSM-

Daten erzeugt. Es gibt dabei diverse Besonderheiten zu beachten. Deswegen ist eine sinnvolle Polygon-Extraktion aus den OSM-Daten entscheidend für die Nützlichkeit der Resultate. Das Kapitel Extrahierung der Polygone aus OpenStreetMap beschäftigt sich mit diesem Prozess. Dabei werden zwei Programme verwendet: ein Python-Skript zum Extrahieren der wesentlichen OSM-Daten (ohne Metadaten) aus einer PBF-Datei sowie ein C++ Programm zur Erzeugung der Polygone.

Das Programm für die Datenstruktur ist dann weitesgehend unabhängig von OSM-Besonderheiten. Als Koordinaten wurden jedoch weiterhin WSG 84-Breiten- und Längengrade verwendet und sind auf 7 Nachkommastellen begrenzt. Wenn andere Koordinatensysteme unterstützt werden sollen, müsste das Programm eventuell angepasst werden. Die den Polygonen zugeordneten Informationen werden für jedes Polygon einzeln als String in der Datenstruktur gespeichert, wodurch die Informationen nicht an eine bestimmte Struktur gebunden sind.

Als Grundidee für die Datenstruktur wurde von Professor Funke ein Quadtree vorgeschlagen und von mir so auch umgesetzt. Näheres dazu findet sich im Kapitel Quadtree-Datenstruktur. Die Quadtree-Datenstruktur wurde dabei in das Java-Programm MapViewer eingebaut und zeigt eine mögliche Anwendung der Datenstruktur als geographischen Informationsdienst.

2 Extrahierung der Polygone aus OpenStreetMap

2.1 Struktur der OpenStreetMap-Daten

Die OpenStreetMap-Daten sind in verschiedenen Präsentationen verfügbar. Die vom OpenStreetMap-Projekt genutzten Präsentationen sind unter anderem OSM XML[1] (OpenStreetMap Extensible Markup Language), PBF[2] (Protocollbuffer Binary Format) und als Daten von PostgreSQL-Datenbanken[3]. Die Formate der jeweiligen Präsentationen sind im OpenStreetMap-Wiki spezifiziert. Für die OSM-XML-Daten und PostgreSQL-Daten sind außerdem verschiedene Variationen vorhanden, die für unterschiedliche Anwendungsgebiete entworfen wurden.

OSM-XML-Daten sind XML-Dateien mit einem spezifizierten Schema, welche üblicherweise die Dateiendung **.osm** haben. Da die PostgreSQL-Daten für diese Arbeit nicht verwendet wurden, werde ich auf diese Präsentationsform der OpenStreetMap-Daten nicht weiter eingehen. Das PBF-Format beschreibe ich im Abschnitt **2.2 Der PBF-Parser** näher.

2.1.1 Beispiel einer OSM-XML-Datei

Um sich einen Überblick von der Datenstruktur und deren Elemente der OpenStreetMap-Daten zu verschaffen, kann man OSM XML-Dateien betrachten. Im OpenStreetMap-Wiki gibt es einen Auszug aus einer OSM-XML-Datei, an welchem man die Struktur von OSM-Daten herauslesen kann. Diesen Auszug findet man auch im Listing 2.1.

[1] http://wiki.openstreetmap.org/wiki/OSM_XML
[2] http://wiki.openstreetmap.org/wiki/PBF_Format
[3] http://wiki.openstreetmap.org/wiki/PostgreSQL

Listing 2.1 Beispiel für eine OSM-XML-Datei
Quelle: http://wiki.openstreetmap.org/wiki/OSM_XML

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="12.2524800"/>
 <node id="298884269" lat="54.0901746" lon="12.2482632" user="SvenHRO" uid="46882"
     visible="true" version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
 <node id="261728686" lat="54.0906309" lon="12.2441924" user="PikoWinter" uid="36744"
     visible="true" version="1" changeset="323878" timestamp="2008-05-03T13:39:23Z"/>
 <node id="1831881213" version="1" changeset="12370172" lat="54.0900666" lon="12.2539381"
     user="lafkor" uid="75625" visible="true" timestamp="2012-07-20T09:43:19Z">
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
 </node>
 ...
 <node id="298884272" lat="54.0901447" lon="12.2516513" user="SvenHRO" uid="46882"
     visible="true" version="1" changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
 <way id="26659127" user="Masch" uid="55988" visible="true" version="5" changeset="4142606"
     timestamp="2010-03-16T11:47:08Z">
  <nd ref="292403538"/>
  <nd ref="298884289"/>
  ...
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Straße"/>
 </way>
 <relation id="56688" user="kmvar" uid="56190" visible="true" version="28"
     changeset="6947637" timestamp="2011-01-12T14:23:49Z">
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="node" ref="364933006" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  <member type="node" ref="249673494" role=""/>
  <tag k="name" v="Küstenbus Linie 123"/>
  <tag k="network" v="VVW"/>
  <tag k="operator" v="Regionalverkehr Küste"/>
  <tag k="ref" v="123"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
 </relation>
 ...
</osm>
```

2.1.2 Elemente der OpenStreetMap-Daten

Die OpenStreetMap-Daten bestehen im wesentlichen aus drei verschiedenen Element-Arten[4]: den Nodes (auch Punkte oder Knoten genannt), Ways (auch als Wege oder Linien bezeichnet) und den Relations (beziehungsweise Relationen). Alle drei Element-Arten besitzen die folgenden Attribute, die für OpenStreetMap benötigte Metadaten enthalten:

- **id:** Ein ganzzahliger Wert, der zur Referenzierung von Elementen dient. Zu beachten ist, dass für jede der drei Element-Arten der ganze Zahlenraum zur Verfügung steht. Das bedeutet, dass in den OpenStreetMap-Daten gleichzeitig ein Knoten, ein Weg und eine Relation existieren dürfen, die alle die ID 42 haben. Um ein Element zu referenzieren, muss man somit auch den Typ des zu referenzierenden Elementes kennen. Es dürfen aber nie zwei Elemente derselben Art die gleiche ID haben.
- **user:** Der Benutzername von der Person, die zuletzt das Element geändert hat.
- **uid:** Der ID von dem Benutzer, welcher zuletzt das Element geändert hat.
- **timestamp:** Der Zeitpunkt, an dem zuletzt das Element geändert wurde.
- **visible:** Gibt an, ob das Element in der Datenbank gelöscht ist.
- **version:** Die Versionsnummer von dem Element.
- **changeset:** Die Nummer des Changesets (eine Menge von Änderungen), in welcher das Element geändert oder erzeugt wurde.

Die Elemente können außerdem sogenannte Tags[5] enthalten. Ein Tag ist ein Schlüssel-Wert-Paar, wobei der Schlüssel und der Wert vom Datentyp Zeichenketten (auch Strings genannt) sind. Die Tags dienen dazu die Eigenschaften der Elemente zu beschreiben.

Nodes

Die OpenStreetMap-Knoten[6] dienen zur Definierung von Positionen auf der Weltkarte. Sie sind die einzigen Elemente in OpenStreetMap, die Koordinaten-Daten enthalten. Als Koordinaten-System wird das standardmäßige WGS 84-System verwendet. Eine Koordinate besteht dabei aus einem Breitengrad (engl. Latitude) und einem Längengrad (engl. Longitude). Jeder Knoten enthält dafür die Attribute **lat** (für den Breitengrad) und **lon** (für den Längengrad).

Der Breitengrad gibt die Entfernung zum Äquator in Nord-Süd-Richtung in Grad an. Bei den OpenStreetMap-Daten darf der Wert zwischen einschließlich -90 Grad und einschließlich +90 Grad liegen, mit einer Genauigkeit von 7 Nachkommastellen. Eine positive Gradzahl

[4] http://wiki.openstreetmap.org/wiki/Elements
[5] http://wiki.openstreetmap.org/wiki/Tags
[6] http://wiki.openstreetmap.org/wiki/Node

bedeutet, dass die Koordinate nördlich des Äquators ist, für Koordinaten südlich des Äquators ist die Gradzahl negativ.

Der Längengrad gibt die Entfernung zum Nullmeridian in West-Ost-Richtung in Grad an. Der Nullmeridian ist der festgelegte Grad 0 geographischer Länge und verläuft durch den Londoner Stadtteil Greenwich[7] in Großbritannien. Bei den OpenStreetMap-Daten darf der Wert zwischen einschließlich -180 Grad und einschließlich +180 Grad liegen, mit einer Genauigkeit von 7 Nachkommastellen. Eine positive Gradzahl bedeutet, dass die Koordinate östlich des Nullmeridian ist, für Koordinaten westlich des Nullmeridians ist die Gradzahl negativ.

Ways

Die OpenStreetMap-Wege[8] sind geordnete Listen von Knoten, die mit ihrer Knoten-ID referenziert werden. Zwei in der Knoten-Liste aufeinanderfolgende Knoten definieren dabei ein Segment, also eine Linie zwischen den Knoten. Da ein OpenStreetMap-Weg mindestens 2 Knoten enthält, definiert ein Weg also eine Liste an Segmenten. Mit den Wegen lassen sich dadurch zum Beispiel Straßen, Gebäude-Umrisse und Flächen-Grenzen definieren.

Damit ein Weg auch eine Fläche definiert, muss die erste Knoten-ID in der Knoten-Liste identisch mit der letzten Knoten-ID sein. Ein solcher Weg wird auch als geschlossener Weg bezeichnet. So kann die Fläche von zum Beispiel Gebäuden, Wald-Flächen oder Stadt-Grenzen definiert werden. Da Wege aber maximal 2000 Knoten enthalten dürfen, kann man größere Flächen nicht ausschließlich mit einem Weg definieren, dafür gibt es aber Relations.

Geschlossene Wege können neben Flächen auch Kreisverkehre und Straßenringe definieren. Um eine Fläche von anderen Arten von geschlossenen Wegen zu unterscheiden, muss man dann die Tags des Weges näher betrachten.

Relations

OpenStreetMap-Relationen[9] sind geordnete Listen von sogenannten Member-Elementen. Member-Elemente kommen nur innerhalb von Relationen vor und bestehen aus den Attributen **ref**, **type** und **role**. Das Attribut ref enthält eine Knoten-, Weg- oder Relations-ID und referenziert dadurch auf das entsprechende Element. Das Attribut type definiert die Art des zu referenzierenden Elementes. Das Attribut role enthält ein Wert abhängig vom Typ der Relation und abhängig vom Zweck des Elementes innerhalb der Relation. Der Typ einer Relation wird mit einem Tag mit dem Schlüssel **type** definiert.

[7] http://de.wikipedia.org/wiki/Greenwich_(London)
[8] http://wiki.openstreetmap.org/wiki/Way
[9] http://wiki.openstreetmap.org/wiki/Relation

Relationen können für verschiedenste Zwecke eingesetzt werden. So gibt es Relationen für die Kennzeichnung von Busrouten oder auch für die Kennzeichnung von Straßenabschnitten, auf denen kein Wenden[10] erlaubt ist. Die meisten Relationen sind aber für diese Arbeit uninteressant, da sie keine Flächen definieren. Für diese Bachelorarbeit nützlich sind dagegen die Relationen vom Typ **multipolygon**, **boundary** und **site**. Was man bei den verschiedenen existierenden Relationen-Typen und deren Rollen beachten muss, sollte im OSM-Wiki für die Relationen-Typen[11] spezifiziert sein.

Multipolygon-Relationen[12] dienen zum Kennzeichnen beliebiger Flächen. Weitere Eigenschaften werden den Relationen dann über Tags gegeben. Multipolygone ermöglichen es, dass in einer Fläche Löcher vorkommen können. Ein Beispiel wäre die Fläche von dem Bundesland Brandenburg, welche das Bundesland Berlin vollständig umschließt. Dadurch enthält die Brandenburg-Fläche ein Loch. Die äußere Grenze eines solchen Multipolygons wird mit Membern vom Typ Weg definiert, welche die Rolle **outer** besitzen und zusammen einen oder mehrere geschlossene Ringe bilden. Die inneren Grenzen werden analog gebildet, wobei aber die Rolle der dafür benutzten Member **inner** ist. Zu beachten ist, dass Member vom Typ Knoten in Multipolygonen nicht zum Bilden der Polygongrenzen verwendet werden können. Bei anderen Relationen könnte es eventuell möglich sein, aber die übliche OSM-Praxis dafür ist das Benutzen von Wegen.

Boundary-Relationen[13] kennzeichnen Grenzen von zum Beispiel Staaten, Bundesländern oder Landkreisen. Zu beachten ist, dass man beim type-Tag der Relation mittlerweile nicht mehr den Wert boundary benutzen soll, sondern den Wert **multipolygon**. Dass es sich bei der Relation um eine Boundary-Relation handelt, erkennt man dann an einem Tag mit dem Schlüssel **boundary**. Bei den Membern und Rollen wird das gleiche System wie bei den Multipolygon-Relationen verwendet. Zusätzlich gibt es noch die Rollen **admin_centre** und **label**. Für jede der beiden Rollen darf maximal ein Knoten-Member existieren, der diese Rolle besitzt. Die erste Rolle dient zur Definierung der Position des administrativen Zentrums der Relation, bei einem Bundesland wäre das die Hauptstadt des Bundeslandes. Die zweite Rolle definiert die Position, an der ein Renderer den Schriftzug für die Relation auf der Karte zeichnet.

Site-Relationen[14] sind zum Definieren von Gelände-Flächen und zugehörigen Gebäuden von zum Beispiel Schulen gedacht. Dieser Typ hat einen anderen Aufbau der Rollen als die beiden oben genannten Relationen und so wurde der Typ der Einfachheit halber in der Arbeit nicht weiter behandelt.

[10] http://wiki.openstreetmap.org/wiki/Relation:restriction
[11] http://wiki.openstreetmap.org/wiki/Types_of_relation
[12] http://wiki.openstreetmap.org/wiki/Relation:multipolygon
[13] http://wiki.openstreetmap.org/wiki/Relation:boundary
[14] http://wiki.openstreetmap.org/wiki/Relations/Proposed/Site

2.2 Der PBF-Parser

Die OpenStreetMap-Daten enthalten eine Menge an Meta-Daten, die für die Arbeit nicht benötigt werden. In dieser Arbeit werden die OpenStreetMap-Daten mit einem PBF-Parser auf die wesentlichen OSM-Elemente ohne Meta-Daten reduziert. Dieser Abschnitt beschäftigt sich damit, was PBF-Dateien eigentlich sind und was der PBF-Parser genau tut.

2.2.1 Vergleich zwischen PBF und OSM-XML

Ein Nachteil von OSM XML (und XML allgemein) ist der Overhead an Daten, welcher durch die Syntax und menschenlesbare Form entsteht. Für jedes OpenStreetMap-Element steht der vollständige Typename und die vollständigen Attributsnamen in der OSM-XML-Datei. Da es nur eine überschaubare Anzahl verschiedener OpenStreetMap-Elemente und Attribute gibt, könnte man jeden Element- und Attributsnamen jeweils durch einen einzelnen Buchstaben ersetzen und man würde dadurch den Overhead schon um einiges senken. Desweiteren könnte man unter anderem die Koordinaten, Timestamps und IDs in binärer Form speichern. Eine ID braucht in Textform für jede Ziffer mindestens ein Byte, in Binärform kann man mit 4 Bytes Zahlen im Bereich von 0 bis $2^{32} = 4294967296$ speichern.

Potential zur Komprimierung der Daten ist also vorhanden. Genutzt wird dieses Potential durch das Protocollbuffer Binary Format (PBF). Durch das Format spart man einiges an Speicherplatz ein und eine PBF-Datei mit den OSM-Daten der gesamten Erde soll auch 5 mal schneller zu schreiben und 6 mal schneller zu lesen sein als eine mit GZip komprimierte OSM-XML-Datei mit den OSM-Daten der gesamten Erde, dies steht jedenfalls auf der OSM-Wiki-Seite zum PBF-Format[15].

Erhältlich sind die PBF-Dateien, welche üblicherweise die Dateiendung **.pbf** haben, auf den Webseiten `http://planet.openstreetmap.org/pbf/` und `http://download.geofabrik.de`. Außerdem kann man mit dem Tool Osmosis zum Beispiel aus den OSM-XML-Dateien PBF-Dateien erzeugen.

2.2.2 Der ursprüngliche PBF-Parser

Die Implementierung für das Auslesen der PBF-Dateien ist im Vergleich zu XML-Dateien komplizierter. Auf der Webseite `http://pbf.raggedred.net/` existiert jedoch ein Python-Programm, welches aus PBF-Dateien OSM-XML-Dateien erzeugt. Das Python-Programm liest eine PBF-Datei blockweise und schreibt die in dem zuletzt gelesenen PBF-Block enthaltenen OSM-Elemente als XML-Daten in die Ausgabe-Datei.

Der ursprüngliche PBF-Parser besitzt für die OSM-Element-Arten Nodes, Ways, Relations und Relation-Member jeweils eine Klasse. Wenn ein OSM-Element in einem PBF-Block eingelesen wurde, erzeugt das Programm ein Objekt von der entsprechenden Klasse und fügt

[15] `http://wiki.openstreetmap.org/wiki/PBF_Format`

dem Objekt die eingelesenen Daten hinzu. Dann wird die Methode **xout** des entsprechenden Objekts aufgerufen, welche die Daten des OSM-Elementes im OSM-XML-Format in die Ausgabe-Datei schreibt. Sobald ein Element geschrieben wurde, wird das Objekt nicht mehr benötigt und kann vom Garbage-Collector von Python gelöscht werden. Speicherintensiv ist das Python-Programm dadurch nicht.

2.2.3 Durchgeführte Änderungen am PBF-Parser

Ich habe von Professor Funke eine Variante des Python-Programmes erhalten, welche anstelle von OSM-XML-Daten ein leicht parsbares Text-Format ausgibt. Das Text-Format enthält die für die Arbeit wesentlichen OSM-Daten, die OSM-Meta-Daten gehören nicht dazu. Eingebaut wurde die veränderte Ausgabe in die Methoden **xout** der Klassen für die OSM-Element-Arten.

Mir erschien das Erzeugen der OSM-Element-Objekte in dem Python-Programm für meine Arbeit jedoch nicht besonders sinnvoll, da die abzudeckende Funktionalität recht überschaubar ist und es nicht geplant ist, weitere Features einzubauen. Ich habe das ursprüngliche Python-Programm soweit angepasst, dass keine OSM-Element-Objekte mehr erzeugt werden und die Klassen werden somit nicht mehr benötigt. Dafür habe ich die Ausgabe der Daten an die Programm-Code-Stelle verschoben, an der die Objekte in den beiden anderen PBF-Parser-Versionen erzeugt wurden. Das Ziel dabei war, die doch relativ lange Ausführzeit für große PBF-Dateien etwas zu reduzieren. Ob dies aber wirklich etwas gebracht hat, habe ich nicht weiter überprüft.

Für die Erzeugung der Polygon-Daten aus dem Text-Format ist es im nächsten Schritt erforderlich, zuerst die Relationen, dann die Wege und zuletzt die Knoten einzulesen. Ansonsten müsste man alle Knoten im Arbeitsspeicher behalten, was bei den Daten für ganz Deutschland bei 4 GigaByte Arbeitsspeicher schon problematisch werden kann.

Bei den verwendeten PBF-Dateien werden aber zuerst die Knoten ausgegeben, dann die Wege und zuletzt die Relationen. Würden im PBF-Parser die Knoten im Arbeitsspeicher zwischengespeichert werden, um sie erst am Ende auszugeben, würde es dann im PBF-Parser zu hoher Arbeitsspeicher-Auslastung kommen. Am sinnvollsten erschien es mir daher, drei verschiende Dateien auszugeben: eine Datei für die Knoten, eine für die Wege und eine weitere für die Relationen. Den PBF-Parser habe ich dann dementsprechend angepasst.

2.2.4 Ausgabeformat

Die Datei für die Knoten hat standardmäßig die Datei-Endung **.nodes.ptxt**. Jede Zeile beinhalten die Daten von genau einem Knoten. Das Format sieht folgendermaßen aus:

```
node <Knoten-ID> <Breitengrad> <Längengrad>
```

Die Datei für die Wege hat standardmäßig die Datei-Endung **.ways.ptxt**. Jede Zeile beinhalten die Daten von genau einem Weg. Das Format sieht folgendermaßen aus:

```
way <Weg-ID> [<Knoten-ID>][ "<Tag-Schlüssel>" "<Tag-Wert>"]
```

Die Datei für die Relationen hat standardmäßig die Datei-Endung **.rels.ptxt**. Jede Zeile beinhalten die Daten von genau einer Relation. Das Format sieht folgendermaßen aus:

```
rel <Relation-ID>[ <Knoten-/Weg-/Relation-ID> <OSM-Element-Typ> "<Rolle>"]
[ "<Tag-Schlüssel>" "<Tag-Wert>"]
```

Der oben vorkommende Zeilenumbruch gehört nicht zum Format, die beiden Ausdrücke in den eckigen Klammern folgen direkt hintereinander. Dazwischen ist außerdem auch kein Leerzeichen.

Die eckigen Klammern bedeuten, dass das Format innerhalb der eckigen Klammern sich beliebig oft wiederholen darf. Das bedeuted beispielsweise, dass die Anzahl der Tags nicht beschränkt ist. Es können auch gar keine Tags vorhanden sein.

2.3 Erzeugung der Polygone

Dank dem PBF-Parser liegen die nötigen OSM-Daten nun in einem Text-Format vor, welches man mit der Standard-Bibliothek von C++ relativ einfach parsen kann. Für die Extraktion der Polygon-Daten wurde mir insbesondere aus Speicher-Effizienzgründen empfohlen, das Extraktionsprogramm in C++ zu entwickeln. Da ich jedoch nicht so vertraut mit der Programmiersprache C++ war, bekam ich von Professor Funke ein in C++ geschriebenes Programm, welches ein sehr ähnliches Text-Format parst und die Daten von Knoten und Wegen im Arbeitsspeicher hält. Relationen wurden in dem Programm aber noch nicht eingelesen, außerdem fehlte die Behandlung von Escape-Sequenzen beim Parsen der Tags. Beides konnte ich aber anhand der vorhandenen Code-Basis leicht umsetzen.

Das Ziel des Polygon-Extraktions-Progammes ist es, aus den OSM-Daten im Text-Format mit den drei Dateien die Polygon-Daten in einer möglichst OSM-unabhängigen Form auszugeben.

2.3.1 Einlesen der OSM-Text-Daten

Während der Implementierung der Polygon-Extraktions-Funktionalität habe ich festgestellt, dass es bei OSM-Daten in der Größenordung von Deutschland bezüglich des Arbeitsspeichers viel zu unpraktisch ist, alle Knoten im Arbeitsspeicher zu halten. Das Programm muss also frühzeitig bestimmen, welche Knoten für die Polygone gebraucht werden. Und dafür muss frühzeitig festgestellt werden, welche Wege als Polygone oder als Teil-Abschnitte von Polygonen in Frage kommen.

Um diese Notwendigkeiten zu erfüllen, werden zuerst die Relationen eingelesen. Es werden dabei nur die Relationen gespeichert, die gültige Polygone definieren könnten und mit den gewünschten Tags gekennzeichnet sind. Damit eine Multipolygon- oder Boundary-Relation ein gültiges Polygon definieren kann, muss mindestens Member-Weg mit der Rolle outer enthalten sein. Die zu überprüfenden Tags hängen davon ab, welche Polygon-Daten man haben möchte. OSM-Daten mit den Tag-Schlüsseln **boundary** oder **place** oder auch mit den Schlüssel-Wert-Paaren **type - boundary** oder **landuse - residential** kennzeichnen gewöhnlich Flächen-Daten.

Mit den gespeicherten Relationen und mithilfe einer beim Relationen-Einlesen erstellten HashMap kann das Programm nun nachschauen, ob ein Weg von einer Relation benötigt wird. Dies wird nun beim Einlesen der Wege benötigt. Dabei werden nur die Wege gespeichert, die selbst ein Polygon definieren (also geschlossen sind) und mit den gewünschten Tags gekennzeichnet sind(siehe Absatz zum Einlesen von Relationen), oder von einer Relation benötigt werden. In einer HashMap werden die Knoten-IDs gespeichert, die von Wegen benötigt werden.

Anschließend kann man die Knoten einlesen. Das Programm schaut in der Knoten-HashMap nach, ob die ID des aktuell eingelesenen Knotens vorhanden ist und wenn ja, wird der Knoten mit der ID und den Koordinaten gespeichert.

Falls die eingelesenen Daten nicht vollständig sind, könnte es sein, dass Wege existieren, bei denen zu einer referenzierten Knoten-ID keine Daten in der Knoten-Text-Datei vorhanden waren. Solche Wege können nicht verwendet werden und werden daher aus der im Programm gespeicherten Weg-Liste herausgelöscht. Relationen können ebenfalls Wege enthalten, bei denen die Knoten-Informationen fehlen, oder bei denen die Weg-Informationen selbst fehlen. Die entsprechenden Member-Wege werden dann aus der entsprechenden Relation gelöscht. Relationen, die anschließend keine Member mehr enthalten, können anschließend auch gelöscht werden. Durch diese Löschaktionen kann man ein paar Spezialfälle loswerden, die man so später nicht mehr behandeln muss.

2.3.2 Bestimmung der Polygone

Nach dem Einlesen aller Daten muss das Programm nun die Polygone finden, die in den Relationen enthalten sind. Bei aus mehreren Wegen zusammengesetzte Polygone ist es so spezifiziert, dass zwei Wege nur dann miteinander verbunden sind, wenn beide Wege miteinander mindestens einen identischen Endknoten haben. Mit Endknoten bezeichne ich hier den ersten und den letzten Knoten in der Knoten-Liste des Weges. Die Knoten eines Weges zwischen den Endknoten müssen dadurch nicht betrachtet werden, während das Programm alle definierten Polygone einer Relation sucht. Dadurch können alle geschlossenen Member-Wege einer Relation als Polygone gezählt werden und müssen nicht mit den anderen Member-Wegen in der Relation abgeglichen werden. Allerdings findet eine Trennung von Outer- und Inner-Polygonen statt. Einerseits kann man damit die Polygon-Informationen mit der entsprechenden Rolle zusätzlich auszeichnen, andererseits sollte mindestens ein valides Outer-Polygon in der Relation vorhanden sein. Wenn dies nicht der Fall ist, wird keines der in der Relation definierten Polygone gespeichert.

Das Zusammensetzen der nicht geschlossenen Member-Wege zu Polygonen wird mit einem Depth-First-Search-inspirierten Algorithmus bewerkstelligt. Eine äußere Schleife iteriert über alle Weg-Member mit der gleichen Rolle, wobei für den Algorithmus die ID und die Endknoten relevant sind. Jeder Schleifenschritt betrachtet einen der Weg-Member als Start-Weg, so dass jeder Weg-Member genau einmal ein Start-Weg ist. Im einem Schleifenschritt werden alle Polygone ermittelt, die den Start-Weg enthalten. Ein Polygon ist gefunden, wenn die Tiefensuche den ersten Knoten des Start-Weges wieder findet. Es wird sichergestellt, dass in dem Pfad der Tiefensuche kein Weg-Member doppelt vorkommt, und dass keiner der schon betrachteten Start-Wege in den aktuellen Pfad der Tiefensuche verwendet wird. Durch den Algorithmus sollten alle vorhandenen Polygone gefunden werden und kein Polygon doppelt im Ergebnis vorkommen.

2.3.3 Ausgabeformat

Das Programm zur Extraktion der Polygon-Daten erzeugt zwei Dateien. Die erste Datei ist die Datei **segments.sgs**, welche alle Segmente der Polygone enthält. Das Format ist für

den MapViewer gedacht und enthält in der ersten Zeile die Anzahl aller Segmente. In jeder weiteren Zeile ist ein Segment definiert nach folgendem Format:

```
<1. Breitengrad> <1. Längengrad> <2. Breitengrad> <2. Längengrad> 1 1
```

Ein Segment besteht also aus zwei Knoten-Koordinaten nach WSG 84. Die letzten beiden Ziffern stellen die Farbe und Linien-Breite der Segmente ein, hier werden rote Linien der Pixel-Breite 1 definiert.

Die zweite vom Programm erzeugte Datei ist die Datei **polygons.ply** und enthält die Polygone, die der Quadtree-Datenstruktur als Eingabe dienen. Die Datei ist nach folgendem Format aufgebaut:

```
<Anzahl der Knoten>
<Liste der Knoten-Koordinaten>
<Anzahl der Polygone>
<Anzahl der Polygon-Knoten>
<Liste der Polygone>
```

Gespeichert werden nur Knoten, die auch von Polygonen verwendet werden. Den Knoten werden außerdem neue IDs zugewiesen, die der Position des Knotens in der Liste der Knoten-Koordinaten entspricht. In der Liste der Koordinaten enthält jede Zeile die Koordinaten von genau einem Knoten. Die Anzahl der Polygon-Knoten entspricht der Summe, die sich ergibt, wenn man über alle Polygone iteriert und die Knotenanzahl von jedem Polygon aufsummiert. Der Wert wird von der Quadtree-Datenstruktur zum Erstellen eines Offset-Arrays benötigt.

Die Liste der Polygone enthält in jeder Zeile ein Polygon nach folgendem Format definiert:

```
<Anzahl der Knoten des Polygons> <Liste der Knoten-IDs> <Polygon-Data-String>
```

Die Liste der Knoten-IDs enthält eine durch einzelne Leerzeichen getrennte Liste von Knoten-IDs, über die da Polygon definiert ist. Die erste Knoten-ID ist mit der letzten Knoten-ID identisch (dies vereinfacht die Benutzung der Polygone).

3 Quadtree-Datenstruktur

3.1 Wieso eine Quadtree-Datenstruktur?

Der Betreuer meine Bachelorarbeit, Professor Funke, hat mir als Datenstruktur einen Quadtree vorgeschlagen. Für einen kurzen Überblick über die Quadtree-Eigenschaften kann man die entsprechende Wikipedia-Seite[1] aufsuchen.

Die Eigenschaften einer solchen Tree-Datenstruktur erschienen mir sehr gut geeignet für die Lösung des Problems der Bachelorarbeit. Um dies zu erläutern, folgt zuerst die Grundidee zur Verwendung des Quadtrees in dieser Arbeit, die von Professor Funke aufgestellt wurde.

3.1.1 Grundidee für die Konstruktion des Quadtrees

- Die Konstruktion des Quadtrees:
  Solange die Zelle von von mehr als O(1) Polygonen geschnitten wird, wird die aktuelle Zelle weiter unterteilt bis in den Blättern immer nur O(1) Polygone geschnitten werden. Die Unterteilung kann dadurch stattfinden, dass die Zelle in vier gleich große Quadranten aufgeteilt wird.
- In jeder Zelle des Quadtrees werden alle Polygone gespeichert, die die Zelle komplett enthalten und noch nicht in einem Elterknoten der Zelle gespeichert ist.
- In jedem Blatt werden alle Polygone gespeichert, welche die Zelle schneiden, aber getrennt von den durch obige Definition gespeicherte Zellen-Polygone.

Mit dem Schneiden von Zelle und Polygon ist die Schnittmenge der beiden Fläche gemeint, nicht das Kreuzen der Zellen- und Polygon-Grenzen.

[1] http://en.wikipedia.org/wiki/Quadtree

3.1.2 Grundidee für Anfragen auf dem Quadtree

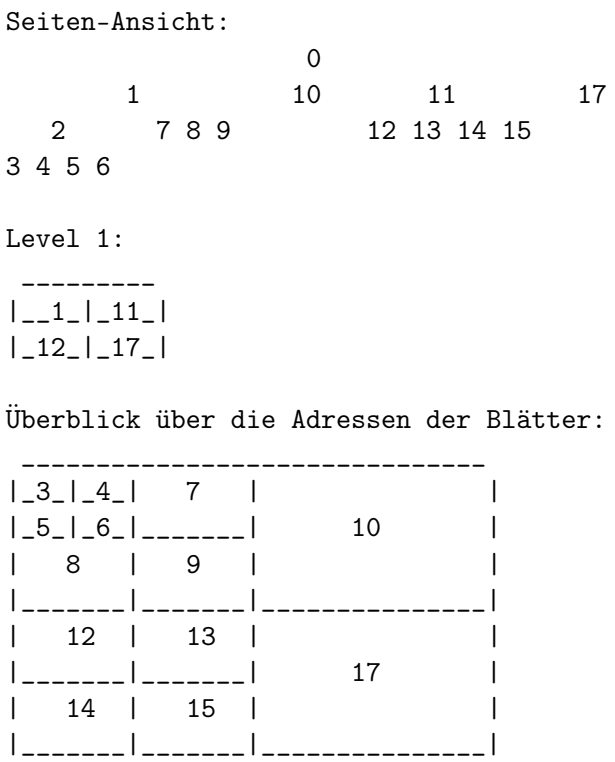
- Der Quadtree wird mit dem Punkt von der Wurzel bis zu einem Blatt durchlaufen.
- Auf dem Weg werden alle Polygone gemerkt, die in den besuchten Zellen gespeichert sind.
- Beim Erreichen eines Blattes: Für alle O(1) Polygone checken, ob die Anfrage-Koordinate enthalten ist

3.2 Struktur und Konstruktion des Quadtrees

Für die Konstruktion des Quadtrees werden als wesentliche Datenstrukturen Arrays für die Breiten- und Längengrade der Knoten benötigt, wobei der Index der neuen ID des Knotens entspricht. Die Koordinaten werden als Integer-Werte gespeichert, indem die Koordinaten mit 10.000.000 multipliziert werden. Dies ist aufgrund der Beschränkung auf 7 Nachkommastellen bei den Koordinaten möglich und soll die Berechnungen durch Vermeidung von Fließkomma-Arithmetik beschleunigen.

Des weiteren werden die Knoten der Polygone mit einer Offset-Array-Datenstruktur gespeichert und die Flächen-Informationen der Polygone in einem String-Array abgelegt. Eine Offset-Array-Datenstruktur besteht aus zwei Arrays, im ersten Array kann man mit der entsprechenden ID nachschauen, in welchem Bereich im zweiten Array sich zum Beispiel die Polygon-Daten zu einer Polygon-ID befinden.

Die Nummerierung der Quadtree-Zellen erfolgt nach der Konstruktionsreihenfolge, siehe folgende Abbildungen.



Dadurch können die IDs für die Zellen- und Blatt-Daten klein bleiben und Offset-Array-Datenstrukturen für die Speicherung der Polygon-Daten verwendet werden. Über eine weitere Offset-Array-Datenstruktur werden die Kind-Zellen zu einer Zellen-ID gespeichert.

3.3 Der Ray-Casting-Algorithmus

Um Herauszufinden, ob ein Punkt in einem Polygon liegt, wird in dieser Arbeit der Ray-Casting-Algorithmus[2] verwendet. Die mathematische Idee dahinter: Vom Anfragepunkt wird in eine Richtung ein Strahl gesendet, und wenn der Strahl eine ungerade Anzahl an Segmenten durchkreuzt, ist der Punkt im Polygon. Ansonsten ist der Punkt außerhalb des Polygons.

Um herauszufinden, ob sich eine Zelle mit einem Polygon schneidet, verwende ich den Algorithmus zusätzlich in einer anderen Variante. Die Zellen-Grenze wird im wesentlichen als Strahl interpretiert, der aber nach einer gewissen Strecke endet. Dies ist möglich, da in dem von mir implementierten Quadtree die Zellen-Grenzen entweder horizontal oder vertikal sind.

Bei der Konstruktion des Quadtrees wird außerdem der Ray-Casting-Algorithmus verwendet, um festzustellen, ob eine Zelle wirklich in einem Polygon enthalten ist. Dazu darf das Polygon keine der Zellen-Grenzen schneiden, und eine Point-in-Polygon-Berechnung von einem Ecke der Zelle aus muss ergeben, dass die Ecke innerhalb des Polygons ist.

3.4 Point-in-Polygons-Anfragen auf dem Quadtree

Auf dem fertig gebauten Quadtree wird bei jeder Anfrage der Quadtree von der Wurzel bis zu einem Blatt rekursiv durchlaufen. Anhand der Anfrage-Koordinaten wird in jeder erreichten Zelle des Quadtrees berechnet, in welche der 4 Quadranten einer Zelle der Algorithmus hinabsteigen gehen muss. Unterwegs werden alle entsprechenden Polygon-IDs gesammelt und im Blatt werden auf die speziellen Blatt-Polygone Ray-Casting-Berechnungen ausgeführt.

3.5 Beispiel-Anwendung mit dem Map-Viewer

Der Quadtree wurde in zur einfachen Demonstration der Funktionalität in der Programmiersprache Java in den MapViewer eingebaut. Der MapViewer ist ein NetBeans IDE Projekt und auf der Webseite der Abteilung Algorithmik des Instituts für Formale Methoden der Informatik von der Universität Stuttgart als Download verfügbar.

Der MapViewer dient zur Abbildung von Segmenten auf der OpenStreetMap-Karte. Das Integrieren des Quadtrees für das Stellen von Anfragen per Mausklick auf der Karte und das Anzeigen der Ergebnisse war recht einfach. Außerdem ist der Quadtree vom MapViewer-Code ausreichend abgekapselt, so dass der Quadtree auch in anderen Projekten verwendet werden kann.

[2] http://rosettacode.org/wiki/Ray-casting_algorithm

Literaturverzeichnis

Alle URLs wurden zuletzt am 19. 12. 2012 geprüft.

Als Quellen wurden im Wesentlichen die Webseiten unter den folgende URLs verwendet.

http://wiki.openstreetmap.org/wiki/Main_Page

http://en.wikipedia.org/wiki/Quadtree

http://rosettacode.org/wiki/Ray-casting_algorithm

Für nähere Informationen zu den verwendeten Quellen sind die Fußnoten innerhalb dieser Arbeit zu betrachten.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Sascha Meusel)