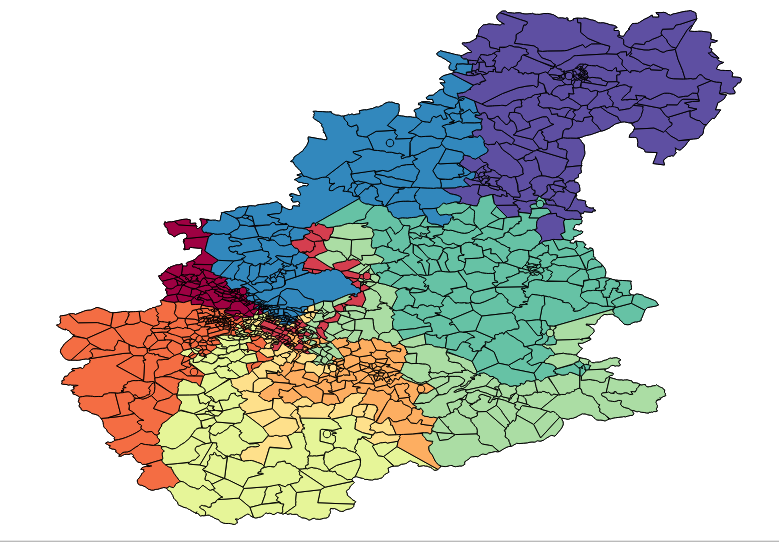
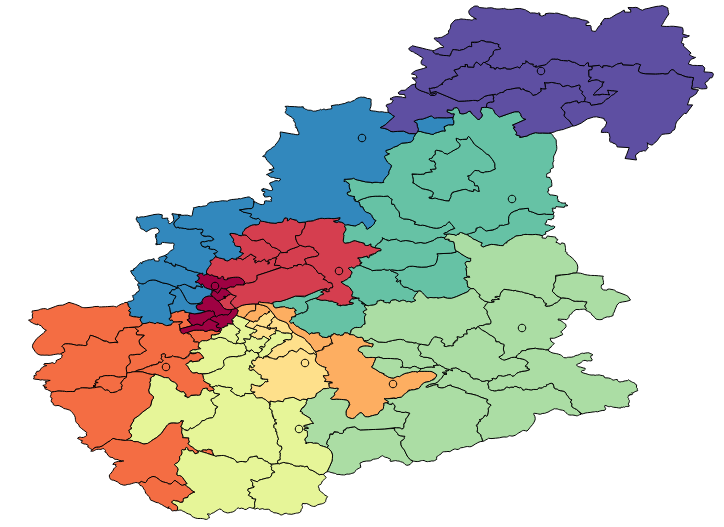
Notizen

* Analysen zeigten, dass DistCrit der potentiell beste Ansatz ist
* Probleme: Zusammenhängigkeit
  + Prüfen, ob eines einzeln per Check, ob Nachbar vorhanden
  + Problem: Es können gruppen von Gebieten abgetrennt sein; somit nutzen von Graphen notwendig --> welche Gebiete erreiche ich, gibt es gebiete die nicht erreicht werden, dann kein Zusammenhang
* Problem beim Umsortieren ist das Bilden von Nasen/Schläuchen





* Überlegen einer Wichtungsfunktion für die Kompaktheit notwendig
* Probleme bereitet auch das Umsortieren in den Fällen, wenn Gebiete schleifenartig immer wieder zu den gleichen Standorten verteilt werden; dann wird irgendwann abgebrochen mit einer höheren Differenz der Crit Sum
* Ansätze für Kompaktheitsmaße sind auf der Seite des KITs zu finden, muss nun implementiert werden

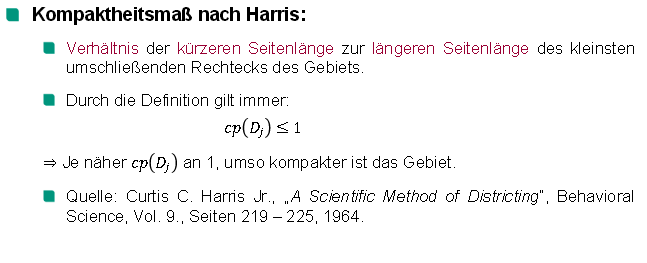
Weitere Aufgaben:

* Klärung der Qualitätskriterien:
  + Muss immer zusammenhängend sein?
  + Wichtungsmöglichkeit überlegen von Kompaktheit und Gleichverteilung
  + Refactoring/Umschreiben auf OOP
  + Gebiet abändern und rechts oben eine PLZ hinzunehmen, damit Zuordnung der PLZ zu „falschem“ Standort, wenn nur die Luftliniendistanz und nicht die Zusammenhängigkeit beachtet wird
* Ähnlich wie scoreberechnung? Ähnliche herangehenensweise sinnvoll? Für jedes Kriterium ein Maß, was unterschiedlich gewichtet werden kann und daraus gleichung erstellen zb 0,5\*Kompaktheit+0,4\*Gleichverteilung+0.1\*Zusammenhang möglichst klein

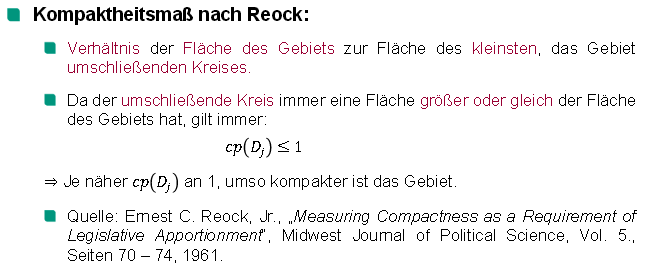
29.5.2015

* Refactoring ist durchgeführt
* Gedanken zum Routing
  + Einschränkung der Straßen, die verwendet werden dürfen: Gesamtgeometrie des Gebietes erzeugen, st\_contains(gebiet, welche Straße)
  + Routing darauf ausführen
  + Distanz/Fahrtzeit ermitteln und abspeichern
* Qualitätskriterien:
  + Unbedingt zusammenhängend
  + Kompaktheit
  + Gleichverteilung

Kompaktheit

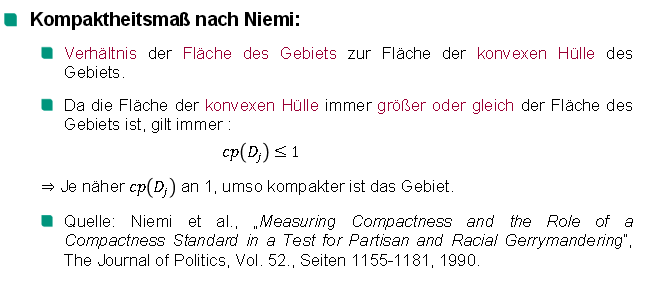


Nur ein Datenbankzugriff notwendig: St\_evelope erzeugt bounding box



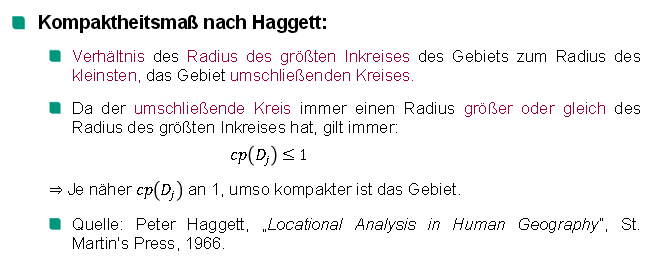
4 Datenbankzugriffe notwendig:

* Bildung des Gesamtgebietes ST\_union
* Flächenberechnung ST\_Area
* Erzeugung des kleinsten Umkreises, ST\_MinimumBoundingCircle
* Flächenberechnung des Kreises, ST\_Area



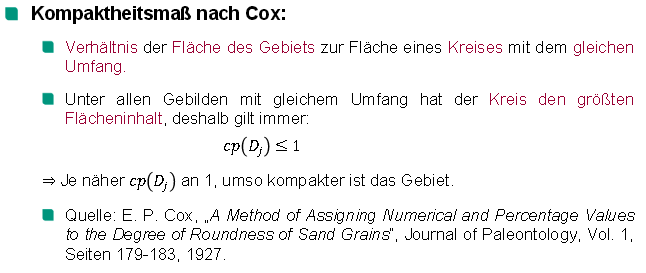
4 Datenbankzugriffe notwendig:

* Bildung des Gesamtgebietes ST\_union
* Flächenberechnung ST\_Area
* Erzeugung der konvexen Hülle, ST\_convexhull
* Flächenberechnung des Kreises, ST\_Area



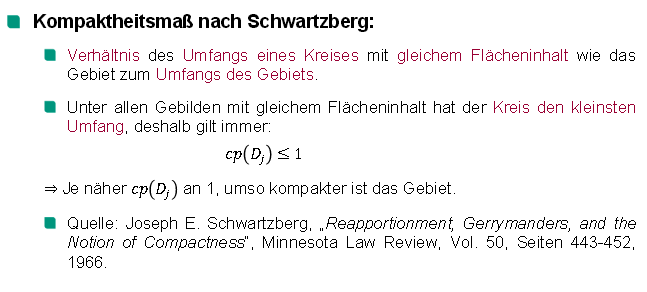
3 Datenbankzugriffe notwendig:

* Erzeugung des größten Inkreises, ?
* Flächenberechnung ST\_Area
* Erzeugung des kleinsten Umkreises, ST\_MinimumBoundingCircle
* Radiusberechnung



3 Datenbankzugriffe, 4 Berechnungen notwendig:

* Bildung des Gesamtgebietes ST\_union
* Flächenberechnung ST\_Area
* Umfangsberechnung ST\_Perimeter
* Erzeugung eines Kreises mit gleichen Umfang wie Fläche
* Flächenberechnung des Kreises



3 Datenbankzugriffe, 4 Berechnungen notwendig:

* Bildung des Gesamtgebietes ST\_union
* Umfangsberechnung, ST\_Perimeter
* Flächenberechnung, ST\_Area
* Erzeugung eines Kreises mit Flächeninhalt wie Umfang des Gebietes
* Umfangsberechnung des Kreises

Quelle: <http://dol.ior.kit.edu/downloads/Ergebnisse_Visueller_Kompaktheits-Test.pdf>

<http://www.diedreifragezeichen.de/bachelorarbeit/Seminarvorlage.pdf>

* untersucht verschiedene Kompaktheitsmaße auf ihre genauigkeit, etc. Es gibt Länge zu Breite Maße, Fläche zu Fläche-Maße und Flächenträgheitsmaße
* Länge zu breite zb Harris: Nachteil, dass Extrempunkte sehr hohe Wichtung haben
* Fläche zu Fläche: Extrempunkte fallen nicht so sehr ins Gewicht, aber sind dennoch beeinflussend
* Umfang zu Fläche-Maße
* Flächenträgheitsmaße: Durch verschiedene Punktwichtung fallen Extrempunkte nicht so sehr ins Gewicht, dafür extremer rechenaufwand notwendig
* Umfang zu Fläche-Maß (Cox, Schwarzberg) benötigen am wenigsten Datenbankzugriffe mit etwa gleich gutem Ergebnis wie Fläche zu Fläche, sodass diese gewählt werden

Cox:

Verhältnis der Fläche des Gebietes zur Fläche eines Kreises mit gleichem Umfang wie das Gebiet

* ST\_union der geometrien
* ST\_Area zur Flächenberechung = A\_Gebiet
* ST\_Perimeter zur Umfangsberechnung
* A\_Kreis = Pi \* Radius \* Radius
* C <= 1, je näher an 1 um so kompakter

Gleichverteilung

* Mittelwert der Summe
* Abweichung vom Mittelwert, Betrag nehmen, da Gebiet auch zu hohe Summe haben kann

Wichtungsfunktion

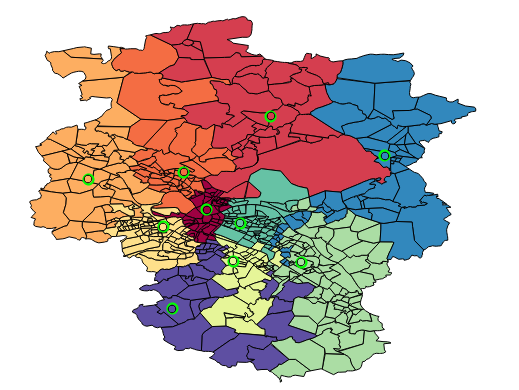
* Für jeden Standort ausführen, d.h. für den der Geometrie abgibt und der, der annimmt
* Prüfen der Veränderung des Wichtungswertes im Vergleich zum Wert vor der Veränderung; alt – neu, da so eine Verbesserung oder Verschlechterung erkennbar ist; Wert<0 = Verbesserung; Wert>0 = Verschlechterung
* Veränderung von beiden, sollte möglichst klein sein

09. Juni 15

* Gebiet auf kleineres, optimaleres Gebiet angepasst
* Änderung des Algorithmus, welches Gebiet abgibt oder hinzunimmt, Möglichkeiten:
  + Per Zufall: Problem, dass Ergebnis nicht reproduzierbar ist
  + Kleinstes/größtest: langes Umsortieren am Ende nur noch mit wenigen werten, ohne das alle betrachtet werden, was aber notwendig wäre
  + Alle, Schritt für Schritt: geht, aber umsortieren von Gebieten, wo es eigentlich nicht notwendig ist
  + Alle, Schritt für Schritt, aber Prüfung, ob notwendig: Es wird jedes Gebiet genommen, geprüft, ob die Abweichung zum Mittelwert zu groß, wenn ja, dann umsortierung: dieser Algorithmus wird nun verwendet

10. Juni

* Problem der Nasen konnte behoben werden



Ursache:

* Fehler in Algorithmus bei Simulation des Tausches wurde die ID beim Polygon der allocatedLoc nicht geändert
* Fehler in Berechnung; absolutwert der Veränderung wurde genommen, wodurch eine unterscheidung ob verbesserung oder verschlechterung möglich, außerdem waren die werte der Differenz bei der Berechnung des Unterschiedes verkehrt herum
* Erkenntnis: Wichtungswerte bei Funktion sind egal, voraussgesetzt, der Schwellwert ist hoch genug, dann kommt stets das gleiche Ergebnis heraus; ist der schwellwert niedrig, muss die wichtung angepasst werden, damit ein ergebnis erzielt werden kann

Greenfield:

Quelle:

<https://de.wikipedia.org/wiki/Orthodrome>

bzw

Geoinformatik in Theorie und Praxis, Norbert de Lange, Seite 145 Mitte

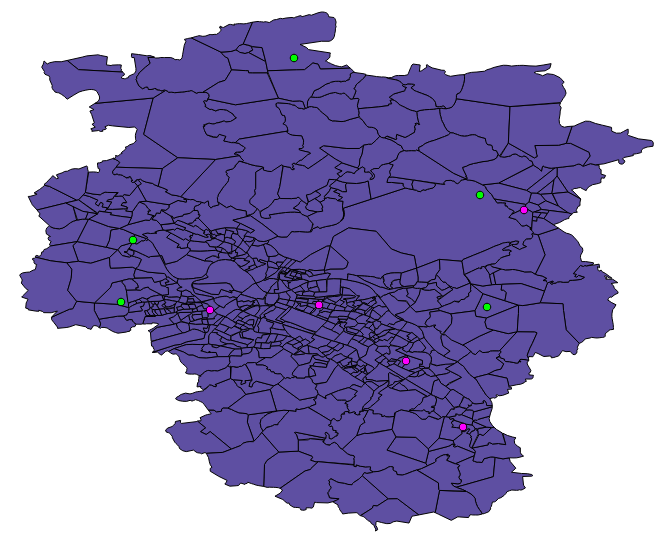
13.07.2015 – Whitespot

Der Ansatz für whitespot ist wie folgt:

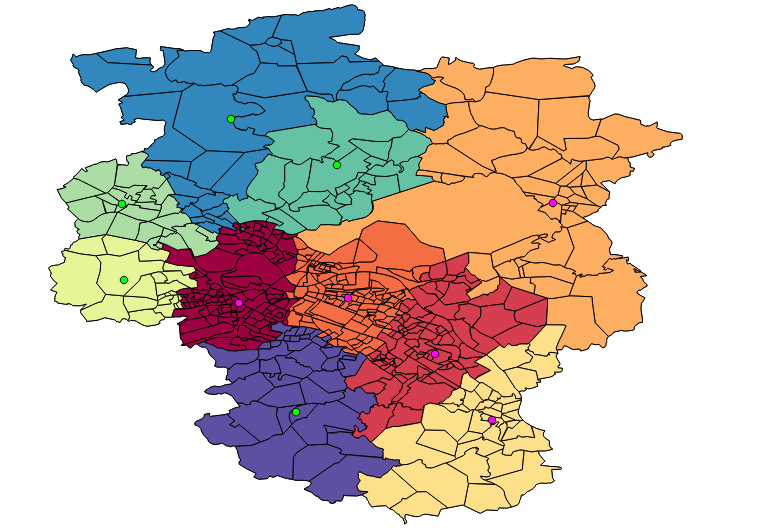
1. Erzeugung der Gebiete der gegebenen Standorte
2. Erzeugung der Gebiet für neue Standorte --> Erzeugung von Startlocations
3. Reset Areas; use all locations (new + given) and do area segmentation

Zwei verschiedene Ansätze bei 2.:

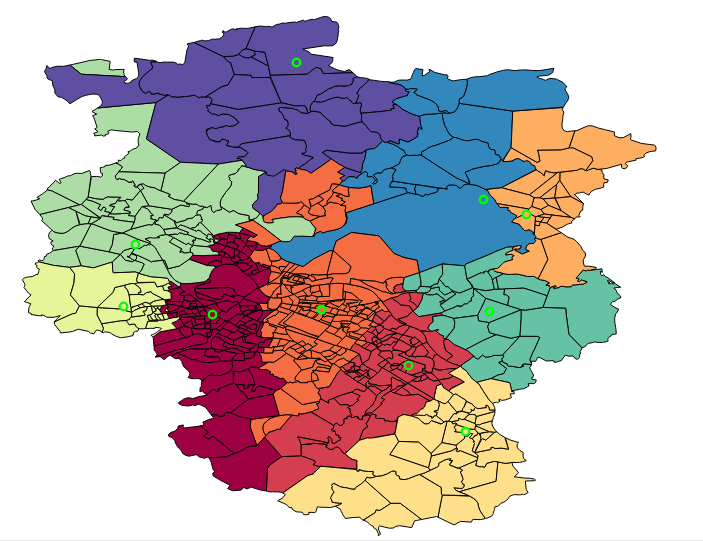
* Entweder Nutzung eines Randpolygons als Start zur Erzeugung des Gebietes
* Oder Nutzung eines beliebigen noch nicht verteilten Polygons: Problem: Punkte/Standorte liegen evtl zu dicht beieinander und dann auch noch zu viele dicht nebeneinander



Ergebnis könnte eigentlich so aussehen, aber durch Standorte nicht möglich



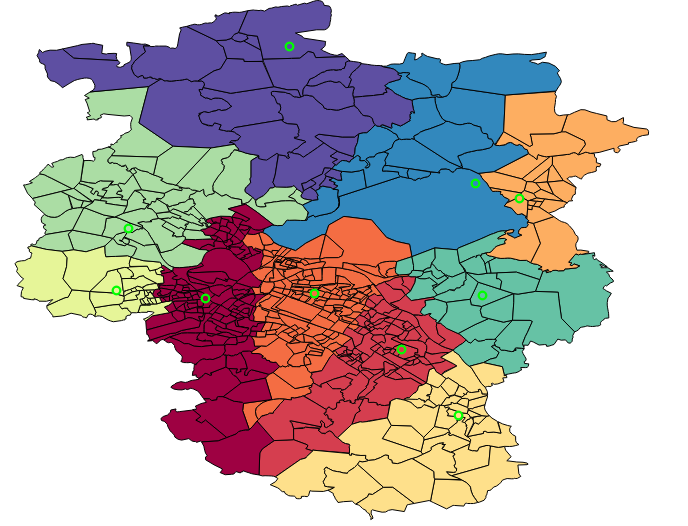
Problem erkannt bei allocatePolygonsByDistance: Es ist nicht gewährleistet, dass zusammenhängende Gebiete entstehen. Wenn die Standorte ungünstig liegen, entstehen Inseln (siehe Bild)



Lösungsidee:

1. Zuordnung wie zuvor nach Distanz
2. Anschließend check Unity
   1. Für jeden Standort
   2. Wenn es Inseln gibt, kleineren Teil des Graphen bestimmen --> den Teil bestimmen, der nicht am Homepoly liegt = Graphen bei Homepoly starten lassen, alle Nachbarn bestimmen, somit bleibt automatisch der Teil übrig, der abseits liegt, wenn kein Zusammenhang
   3. Kleineren Teil dem Standort zuordnen, der am 2. Nächsten, dabei Check Unity (notfalls 3.,4.,… nächsten zuordnen)

Ergebnis:



21. Juli

Beide Whitespotansätze sind nun implementiert. Es ist erkennbar, dass der Ansatz mit Nutzung von Randpolygonen das bessere Ergebnis liefert: Rechenzeit ist kürzer + Gebiete sind besser geformt, also wird dieser Ansatz genommen.

Nachdem Whitespot nun auch fertig ist, beginnt Refactoring.

* Zusammenfassung von Funktionen, die zu einem bestimmten Teilbereich gehören:
  + Initialisation: getNrOrSum, initPolygones, initNeighbours, initCentroids
  + AreaSegmenation: determineHomePoly, initDistance,allocByDist, initCrit, checkUnity, checkThreshold
  + VisualizeResults: writePolys, showCritResults
* Alle Funktionen sind in jedem algorithmus vorhanden, sodass nun einfach übergeordnete Funktion aufgerufen werden kann, die alle unter Funktionen aufruft. Diese können nun als private gesetzt werden
* Auskommentieren der Funktionen
* Kürzen des Quellcodes / Löschen von doppeltem Code

Für initDistances gibt es zwei Ansätze: initDistanceStDist oder initDistanceStCentroid

initDistanceStDist nutzt die PostGis distanz-funktion, es wird dabei der Punkt der Geometrie genommen, der am nächsten ist. Quellcode:

**public** **static** **double** calculateDistStDistance(**int** i, String geometry,

Statement stmt) **throws** SQLException {

**double** distance;

Location loc = *locationContainer*.getLocation(i);

StringBuffer sb = **new** StringBuffer();

sb.append("SELECT st\_distance(ST\_GeomFromText('POINT(" + loc.getLon()

+ " " + loc.getLat() + ")'),ST\_GeomFromText('" + geometry

+ "'));");

distance = *calculateDist*(sb, stmt);

**return** distance;

}

initDistanceStCentroid nutzt die PostGIS centroid Funktion, somit wird die Distanz zum Centroid des Polygones betrachtet. Später kann statt dieser die initialisierten Centroide genutzt werden, so dass Datenbankzugriffe gespart werden. Quellcode:

**public** **static** **double** calculateDistStCentroid(**int** i, String geometry,

Statement stmt) **throws** SQLException {

**double** distance;

Location loc = *locationContainer*.getLocation(i);

StringBuffer sb = **new** StringBuffer();

sb.append("SELECT ST\_AsText(st\_centroid(ST\_GeomFromText('" + geometry

+ "')));");

ResultSet d = stmt.executeQuery(sb.toString());

d.next();

String centroid = d.getString(1);

sb = **new** StringBuffer();

sb.append("SELECT st\_distance(ST\_GeomFromText('POINT(" + loc.getLon()

+ " " + loc.getLat() + ")'),ST\_GeomFromText('" + centroid

+ "'));");

distance = *calculateDist*(sb, stmt);

**return** distance;

}

Verbesserungsideen:

* Nach Location Ermittlung Gebiete neu verteilen, um Nasen zu vermeiden? Somit nur die erzeugten Standorte nutzen und danach neu verteilen; --> Beim aktuellen Algorithmus nicht mehr notwendig, da durch Nutzung der Distanz anstatt der Kompaktheit keine Nasen mehr gebildet werden. Somit Gefahr von Nasen nicht zu groß, so dass höhere Rechenzeit nicht im Verhältnis zum Nutzen steht
* Wichtung möglich bei Gebietsverteilung algorithmus nach Distanzen oder Anwendung eines anderen Algorithmus zb GetsTrueNearest
* Bessere Bestimmung des critAverage bei Greenfield bzw besseres Abbruchkriterium
* Threshold value nicht über Prozent, sondern über Varianz etc.
* Homepolys of greenfield can be rearranged
* initDistance über Centroids anstatt ST\_Distance

Noch notwendige Schritte:

* Veränderung des Thresholds zb mit nutzung der varianz anstatt Prozent
* Abhängigkeit erzeugen zwischen schwellwert und wichtung
* Abbruchbedingung einfügen, schwellwert im notfall erhöhen
* Performanceverbesserung ideen: Critsumme nach größten sortieren und dann alle step by step durchgehen in der Hoffnung, der Abbruch wird schneller erreicht
* orderbyID beim Einlesen der Polygone einbauen bzw beim Auslesen der Randpolygone, um zufällige Auswahl zu vermeiden