University of Zurich[UZH]

**Informatics and Sustainability Research**

# Evaluation von Darstellungsformen der Transaktionen in einem agentenbasierten Modell

Bachelorarbeit 31. März 2015

**Silvio Fankhauser**
Langwies, Schweiz

Student-ID: 10-719-649
silvio.fankhauser@uzh.ch

Betreuer: Stefan Holm

Prof. Dr. Lorenz M. Hilty
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ce

# Danksagungen

Danken möchte ich in erster Linie meinem Betreuer Stefan Holm, der mich mit Rat und Tat unterstützt hat. Seine eingebrachten Impulse hatten einen wesentlichen Einfluss auf den Verlauf und das Ergebnis dieser Arbeit. Ausserdem möchte ich mich bei Prof. Dr. Lorenz Hilty bedanken, dass ich die Arbeit am Lehrstuhl für Informatik und Nachhaltigkeit schreiben durfte.

Ein besonderer Dank geht natürlich auch an meine Eltern, ohne deren Unterstützung ich mein Studium an der Universität Zürich niemals hätte verwirklichen können.

# Zusammenfassung

Ziel dieser Arbeit ist es, verschiedene Darstellungsweisen und Clustering-Algorithmen für eine agentenbasiertes Modell eines Marktes zu evaluieren. In unserem Falle handelt es sich um ein Modell des Holzmarktes, welches von der eidgenössischen Forschungsanstalt für Wald, Schnee und Landschaft (WSL) entwickelt wurde. Die erarbeiteten und implementierten Varianten werden anhand verschiedener Parameter verglichen und bewertet. Es zeigt sich, dass die jeweiligen Ergebnisse stark von den Eingabedaten abhängen und dementsprechend keine allgemein gültige optimale Lösung zu finden ist. Je grösser die Anzahl Transaktionen ist, desto schwieriger ist es, die Daten für den Endnutzer übersichtlich darzustellen. Dies gilt auch für Transaktionen über grosse geografische Distanzen. Auch die Clustering-Algorithmen hängen stark von diesen Faktoren ab.

# Abstract

The goal of this thesis is to evaluate different modes of representation and clustering algorithms for agent-based model of a market. In our case, it is a research of the timber market, which was developed by the Swiss Federal Institute for Forest, Snow and Landscape Research (WSL). The developed and implemented variants are compared and evaluated using various parameters. It shows that the results are highly dependent on the input data. There is no suitable optimal solution for all kind of input data. The greater the number of transactions, the harder it is to display the data intelligible. This also applies for transactions over large geographical distances. The clustering algorithms strongly depend on all these factors as well.

# Inhaltsverzeichnis

# 1

# Einleitung

Um die zukünftige Situation auf dem Holzmarkt besser abschätzen zu können, entwickelte die eidgenössische Forschungsanstalt für Wald, Schnee und Landschaft (WSL) ein Modell, welche den Handel der Güter und das Verhalten der verschiedenen Beteiligten auf dem Markt darstellen soll. Es handelt sich dabei um ein agentenbasiertes Modell, welches das Verhalten der einzelnen Akteure über einen bestimmten Zeitraum abbildet. Als Agenten dienen in diesem Fall beispielsweise Holzproduzenten als Verkäufer oder Sägereien als Käufer. Die agentenbasierte Modellierung (ABM) modelliert Individuen als Softwareagenten, lässt diese miteinander interagieren und aggregiert anschliessend diese Interaktionen zu einem Gesamtverhalten. Prozesse und Veränderungen können somit gleichzeitig aus der Mikro- wie auch der Makroperspektive näher untersucht werden, und es werden Szenarioanalysen im Sinne von Wenn-dann- Fragestellungen möglich (F. Kostadinov, 2012, S. 422). Die einzelnen Verkäufe werden dabei als Tabelle im csv-Format (character-separated value) abgespeichert.

Nun sind die generierten Daten auf diese Weise nur schwer zu lesen und gar zu interpretieren. Der Zweck dieser Arbeit ist es, für diese Daten geeignete Darstellungsformen zu finden. Die grosse Menge an generierten Daten erschwert dies allerdings wesentlich, da es sich meist um mehrere tausend Transaktionen pro Simulation handelt. Deshalb werden in dieser Arbeit neben verschiedenen Darstellungsformen auch einige Varianten evaluiert, wie man diese Transaktionen zu Gruppen zusammenfassen kann und welchen Einfluss dies auf die gewählte Darstellungsform hat.

# 2

# Wahl der technischen Infrastruktur

Als Ausgangslage für diese Arbeit dienten die bereits erwähnten csv-Dateien. Die einzelnen Transaktionen werden darin mit den Koordinaten des Verkäufers, des Käufers, sowie einer Zahl, welche die gehandelte Warenmenge wiedergeben, dargestellt. Bei jeder Transaktion wird das jeweilige Datum festgehalten (siehe Abb. 2.1).

| #date | buyerX | buyerY | sellerX | sellerY | amount |
|---|---|---|---|---|---|
| 01.01.01 | 636.0853 | 242.86864 | 656.0736 | 228.92879 | 600 |
| 01.01.01 | 665.9073 | 236.66357 | 654.1337 | 228.92879 | 600 |
| 01.02.01 | 666.8395 | 269.13116 | 644.88 | 228.92879 | 600 |
| 01.02.01 | 655.0041 | 258.19754 | 641.1477 | 228.92879 | 600 |
| 01.02.01 | 629.739 | 270.28357 | 655.1242 | 228.92879 | 600 |

Abb. 2.1: Ausgangsdaten im csv-Format

Um diese Daten für den Endnutzer verständlich zu machen, musste es unter anderem möglich sein, sie auf einer Karte visualisieren zu können. Die Variante, die Informationen in einem Webbrowser mittels HTML- und Javascript-Datein darzustellen, erschien im Vergleich zu anderen Möglichkeiten am flexibelsten. Als Alternative kommen beispielsweise Java Swing oder JavaFX infrage. Mittels dieser Frameworks können grafische Benutzeroberflächen erstellt werden. Allerdings kann man mit unserer gewählten Technologien nicht direkt auf das Filesystem eines Rechners zugreifen, weshalb eine reine Javascript-Applikation nicht zur Debatte stand. Um die Infrastruktur möglichst schlank zu halten, fiel die Entscheidung nicht auf einen Webserver oder eine eigene Datenbank, sondern auf ein "Backend" mittels eines in Java geschriebenen Programmes. Dieses berechnet die benötigten Daten und generiert Javascript-Dateien, welche dann in einem Webbrowser interpretiert werden können.

Für die Darstellung der Karte fiel die Entscheidung auf Google-Maps. Dessen API (Application Programming Interface) ist sehr gut dokumentiert und bietet alle benötigten Funktionen für diese Arbeit. Auf diese Art ist es möglich, entsprechende Simulationsergebnisse in der ganzen Schweiz anzeigen zu lassen. Beim Einsatz anderer Frameworks, welche beispielsweise svg-Dateien (scalable vector graphics) als Grundlage für ihre Karten haben, wäre der Wechsel von einem Kanton zum anderen wesentlich umständlicher. Die Karten müssten bei jedem regionalen Wechsel neu eingebunden werden, was einen erheblichen Aufwand bedeuten würde. Die Beschränkung auf die Schweiz besteht deshalb,

weil die übergebenen Koordinaten auf dem ausschliesslich in der Schweiz verwendeten CH1903 System basieren. Eine Umstellung auf eine weltweit übliche Norm wäre aber ohne grossen Aufwand durchführbar.

# 3

# Darstellungsformen von Transaktionen

## 3.1 Graphen

Um die Transaktionen visualisieren zu können, bieten sich gerichtete Graphen an, welche man auf einer Landkarte darstellt. Graphen sind anschauliche Darstellungen von "Beziehungsgeflechten". Diese werden aus Knoten zusammengesetzt, die gegebenenfalls durch Kanten verbunden sind. (siehe Schöning, 2008, S.31) Die Knoten der Graphen entsprechen jeweils den Agenten, während die Kanten als Pfeile dargestellt werden. Um den Informationsgehalt dieser Pfeile zu maximieren, kann beispielsweise die Farbe oder Linienstärke angepasst werden. In diesem Fall sind folgende die Attribute der Transaktionen mit folgenden Eigenschaften verknüpft:

- Der Verkäufer bildet jeweils den Anfangsknoten, der Käufer den Endknoten des gerichteten Graphen. Die Kante wird als Pfeil dargestellt.
- Mittels Farbe wird die Entfernung der beiden involvierten Agenten hervorgehoben. Grosse geografische Entfernungen entsprechen einem roten Farbton, während kurze Distanzen in grün gehalten werden. Dazwischen existieren verschiedene Farbabstufungen. Dies hilft optisch bei der Unterscheidung von sich kreuzenden oder sehr nahe beieinander liegenden Kanten.
- Die gehandelte Menge hat einen Einfluss auf die Linienstärke der Pfeile. Je mehr Waren gehandelt werden, desto dicker wird die Kante gezeichnet. Ausserdem werden die bereits erwähnten Farben bei grossen Mengen kräftig dargestellt, während sie bei kleineren Mengen blass werden.

Das grösste Problem bei dieser Darstellungsweise ist die Anzahl der zu zeichnenden Pfeile. Durch deren Menge kann die Karte sehr schnell unübersichtlich werden. Aus diesem Grunde versuchen wir, möglichst viele Pfeile zusammenzufassen, ohne Informationen zu verlieren.

### 3.1.1 Zusammenfassen von Pfeilen

Die Grundidee hinter diesem Algorithmus ist es, Transaktionen welche den gleichen Abnehmer für ihre Waren haben, zusammenzufassen und als einzelnen Pfeil zum Endknoten zu zeichnen. Im folgenden werden zwei Varianten vorgestellt, welche dieses Ziel verfolgen:

Variante 1:

Um zwei Kanten miteinander zu verbinden, benötigt man in erster Linie einen geeigneten Verbindungspunkt. Bei dieser ersten Variante werden die Koordinaten wie folgt berechnet:

$$P_4(x_4, y_4) = ((P_1(x_1, y_1) + P_2(x_2, y_2))/2 + P_3(x_3, y_3))/2 \tag{3.1}$$

$P_1$ = Verkäufer 1, $P_2$ = Verkäufer 2, $P_3$ = Käufer, $P_4$ = Kombinationspunkt

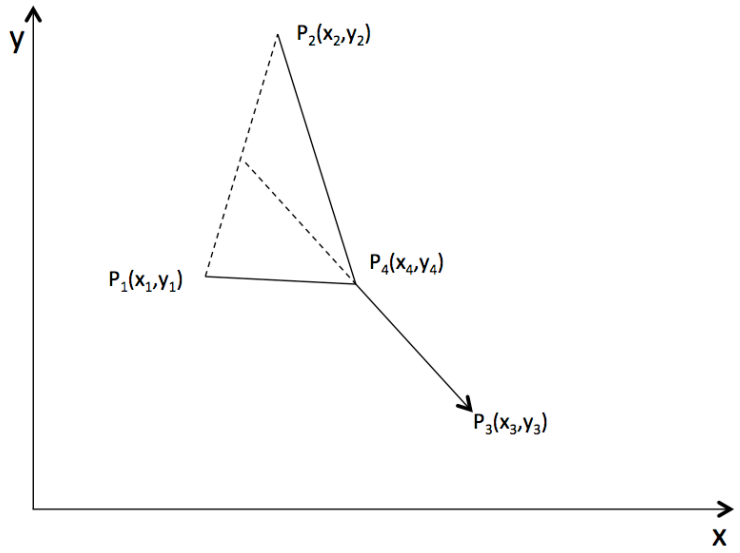



Abb. 3.1: Berechnung des Verbindungspunktes (Variante 1)

Durch ein mehrmaliges Wiederholen können mehrere Kanten zu einer baumähnlichen Struktur zusammengefügt werden. Um einen solchen "Baum" zu erzeugen, welcher in sich selbst möglichst keine Überschneidungen aufweist, müssen die Verkäufer zuerst geordnet und unterteilt werden. Dafür wird das Koordinatensystem ausgehend vom Käufer in vier Quadranten unterteilt und jeder Käufer einem dieser Quadranten zugeteilt. Anschliessend werden die Punkte abhängig von ihrem jeweiligen Quadranten geordnet. Dann kann dann der jeweilige Verbindungspunkt berechnet werden (siehe Abb. 3.2).

Variante 2:

Anstatt wie bei Variante 1 den Verbindungspunkt nach einer bestimmten Formel zu berechnen, werden zwei Graphen dann verbunden, wenn ihr Abstand unter eine bestimmte Grenze fällt (siehe Abb. 3.3). Durch das Anpassen dieses Abstandes (d) kann dieser Punkt entweder näher zum Käufer oder Verkäufer verschoben werden. Um die Distanz approximativ zu bestimmen, werden auf den zu verbindenden Graphen jeweils sieben Punkte berechnet. Anschliessend werden vom Anfang des Graphen her die Distanzen der jeweiligen Punkte berechnet und, sobald der Wert unter demjenigen der Distanz d

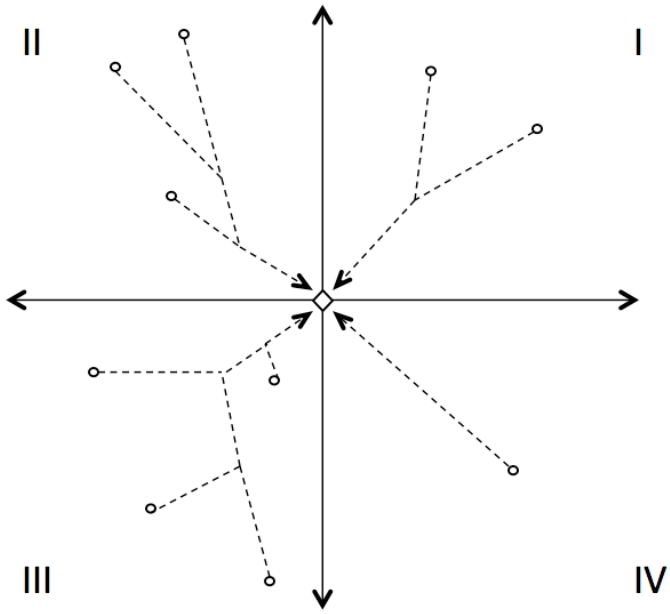



Abb. 3.2: Verbinden von Graphen in Quadranten

liegt, miteinander kombiniert. Wie bei der ersten Variante werden auch hier die vier Quadranten gebildet und die Punkte entsprechend sortiert.

## 3.2 Heatmap

Eine Heatmap eignet sich bestens, wenn es herauszufinden gilt, wo Ballungsgebiete bestehen (siehe Abb. 3.2). Die Google-Maps API erlaubt es, gewichtete Punkte auf der Karte einzuzeichnen und mit verschiedenen Farbtönen einzufärben. In unserem Fall entspricht die Gewichtung den jeweils gehandelten Mengen. Die Skala, mit welcher die Farben eingeteilt werden, passt sich dynamisch den Eingabewerten an. Der höchste Wert wird rot eingefärbt und geht dann mit abnehmender Grösse langsam in grün über. Auf diese Weise können wir leicht erkennen, in welchen Regionen grosse Mengen gehandelt werden. Anhand der Heatmap können wir aber nicht erkennen, welche Agenten miteinander interagieren. Ausserdem verbinden sich nahe beieinander liegende Punkte miteinander. Wie schnell dies geschieht, können wir im Quellcode anpassen. Bei einer hohen Zoomstufe lassen sich die Punkte dennoch gut auseinanderhalten, da sich auch dieser Wert dynamisch an den Zoomfaktor anpasst. Diese Darstellungsform eignet sich insbesondere für grosse Datenmengen.

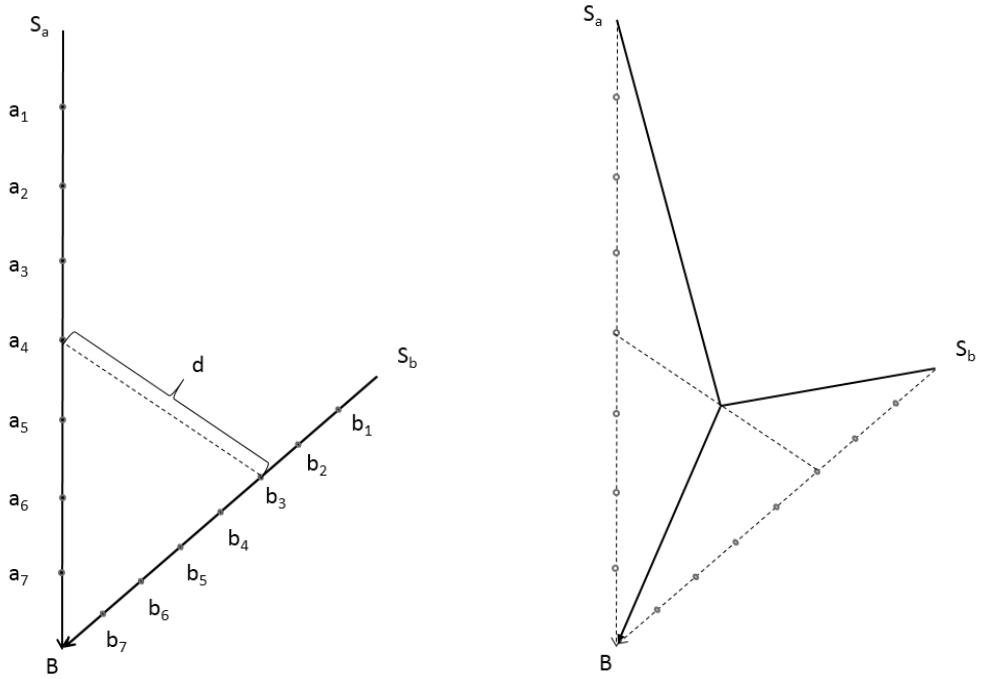



Abb. 3.3: Kombinieren von Graphen (Variante 2)

## 3.3 Einzugsgebiet eines Käufers

Die Grundidee dieses Verfahrens ist es, ein Polygon zu zeichnen, das alle Verkäufer umrandet, welche den gleichen Käufer beliefern. Gerade bei Käufern, welche von vielen verschiedenen Agenten beliefert werden, kann diese Darstellungsweise übersichtlicher sein als mit den bereits beschriebenen Graphen. Leider geht hierbei die Information verloren, welche Mengen zwischen den Agenten transferiert wird. Ausserdem überschneiden sich bei einer grossen Anzahl Agenten die Polygone, was die Darstellung sehr unübersichtlich machen kann. Aus diesem Grunde lassen sich bei der Implementation die Polygone einzeln ein- und ausblenden. Um das Polygon zu zeichnen müssen wir vorerst alle Agenten finden, welche den äussersten Rand der entsprechenden Agenten bilden. Dazu nutzen wir den "Convex hull - Algorithmus", welcher im Kapitel 3.3.1 genauer beschrieben wird. Da wir bei diesem Polygon nur einen Rahmen um die äussersten Agenten ziehen, werden Verkäufer, welche innerhalb dieses Polygons liegen, nicht angezeigt. Aus diesem Grund werden die Agenten zusätzlich mit Markern versehen, damit diese Informationen nicht verloren gehen.

### 3.3.1 Convex hull-Algorithmus

Ziel dieses Algorithmus ist es, aus einer Menge von Punkten diejenigen herauszufiltern, welche die äusserste konvexe Grenze um die gesamte Menge bilden. Basierend auf

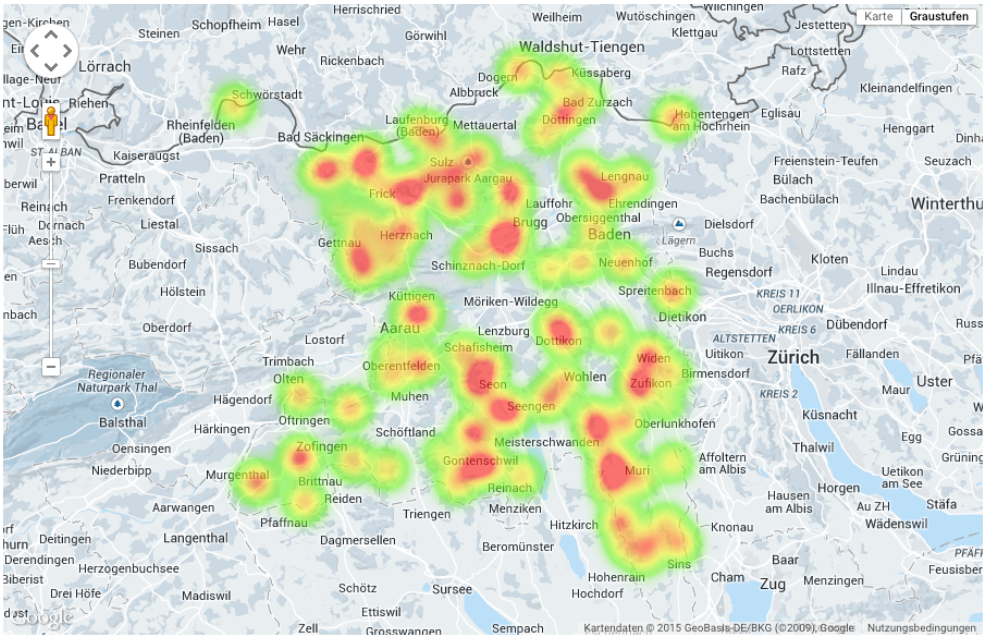



Abb. 3.4: Heatmap mit geringer Anzahl Agenten

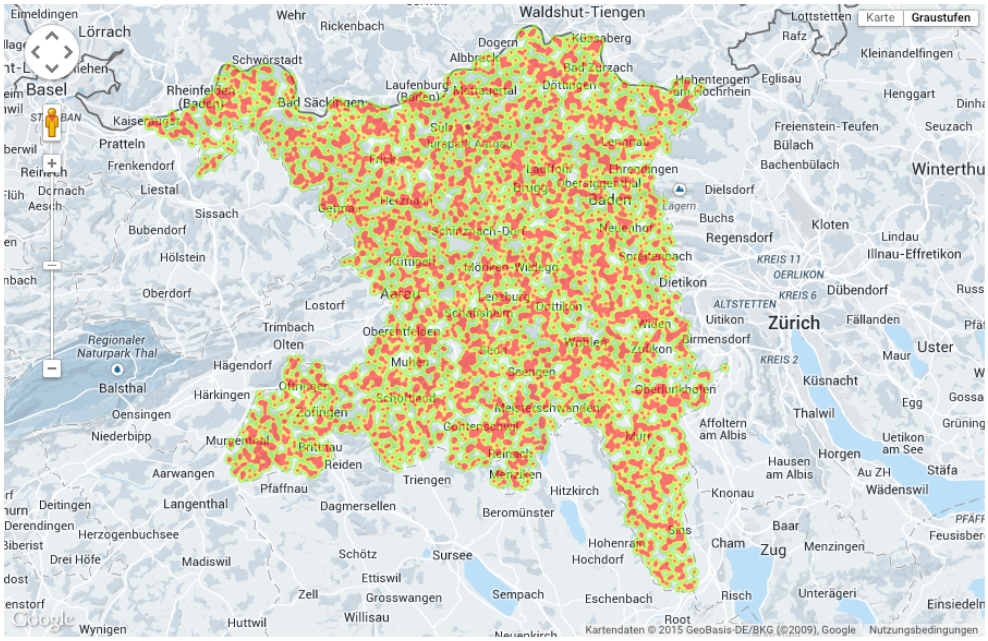



Abb. 3.5: Heatmap mit grosser Anzahl Agenten

einem Algorithmus von Andrew (Andrew, 1979, S. 216-219) wurde der entsprechende Algorithmus implementiert (siehe wikibooks.org (2014)). Eingabeparameter ist ein Array

mit Punkten, welches zuerst nach den x-Wert geordnet wird. Anschliessend daran wird eine Liste (hull) initialisiert, welche jeweils die Punkte des äusseren Rahmens beinhalten wird. In einem nächsten Schritt wird durch die Eingabeliste iteriert. Solange der nächste Punkt der Eingabeliste keine Richtungsänderung gegen den Uhrzeigersinn aufweist (in der Methode cross(); geprüft) , wird der Punkt der hull-Liste hinzugefügt.

```
private static double cross(Point O, Point A, Point B) {
   return (A.getxCoord() - O.getxCoord()) * (B.getyCoord() - O.getyCoord())
       - (A.getyCoord() - O.getyCoord()) * (B.getxCoord() - O.getxCoord());
}

private static Point[] convex_hull(Point[] points) {
   if (points.length > 1) {
      int n = points.length, k = 0;
      Point[] hull = new Point[2 * n];
      Arrays.sort(points);
      // Build lower hull
      for (int i = 0; i < n; ++i) {
         while (k >= 2 && cross(hull[k - 2], hull[k - 1], points[i]) <= 0)
            k--;
         hull[k++] = points[i];
      }
      // Build upper hull
      for (int i = n - 2, t = k + 1; i >= 0; i--) {
         while (k >= t && cross(hull[k - 2], hull[k - 1], points[i]) <= 0)
            k--;
         hull[k++] = points[i];
      }
      if (k > 1) {
         hull = Arrays.copyOfRange(hull, 0, k - 1); // remove non-hull
             vertices after k; remove k - 1 which is a duplicate
      }
      return hull;
   } else if (points.length <= 1) {
      return points;
   } else{
      return null;
   }
}
```

## 3.4 Sankey-Diagramm

In einem Sankeydiagramm werden Flüsse (in unserem Fall Warenflüsse) von einem Agenten zum anderen dargestellt. Um dafür die Google Charts API nutzen zu können, muss zuerst ein geeignetes Datenformat hergestellt werden. Dafür erstellen wir eine Tabelle (google.visualization.DataTable), welche jeweils den Verkäufer, den Käufer und

den gehandelten Betrag enthält. Diese Informationen können wir direkt aus der zu Beginn bereitgestellten csv-Datei beziehen. Wie auch bereits schon bei der Heatmap, werden die Daten direkt in unserem Browser bearbeitet und gerendert. Es werden keinerlei Daten zu einem Server übermittelt. Die Menge der gehandelten Güter wird prozentual auf die zur Verfügung stehende Fläche aufgeteilt. Dies erlaubt es, einen sehr guten Überblick zu erhalten, wie viel einzelne Agenten untereinander handeln. Allerdings ist die absolute Menge nicht mehr ablesbar. Um zu identifizieren, um welche Agenten es sich handelt, sind die jeweiligen Koordinaten angeschrieben. Für den Endnutzer ist es ohne Aufwand kaum möglich herauszufinden, wo die allfälligen Punkte auf der Karte zu finden sind. Ausserdem wird bei der Implementation auch ein Multi-Level-Diagramm gezeichnet, falls Zwischenhändler existieren. Bei unserem Modell des Holzmarktes übernehmen die Agenten aber jeweils nur eine einzelne Rolle. Ein Käufer wird seine Waren also nicht weiter verkaufen. In den Abbildungen 3.6 und 3.7 kann man erkennen, dass mit zunehmender Anzahl Agenten die Darstellung unübersichtlich werden kann.

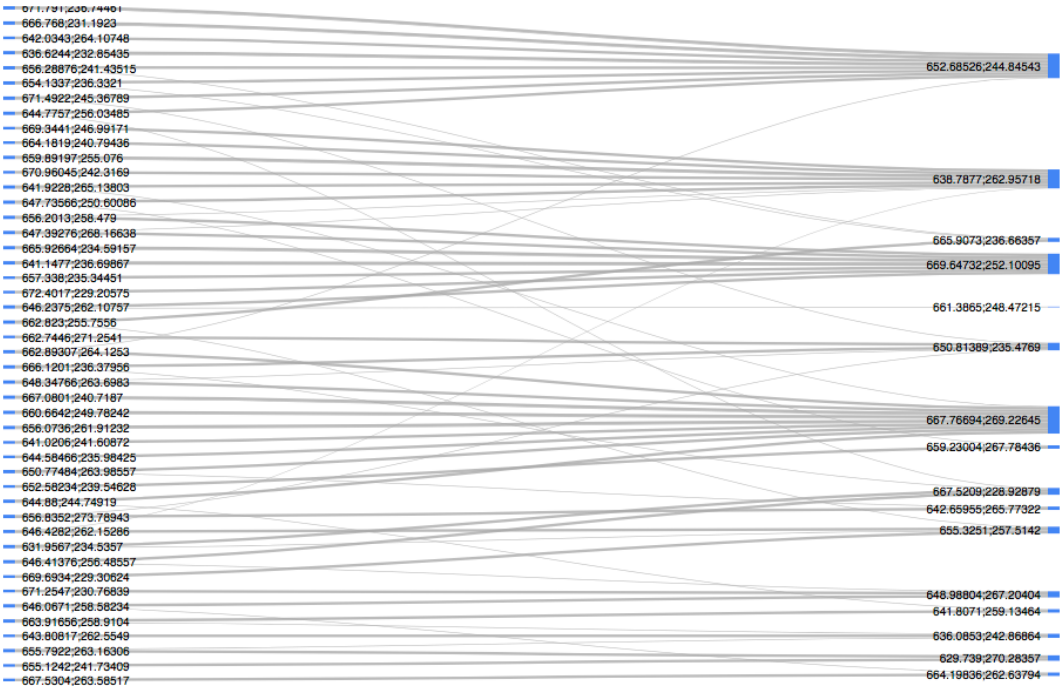



Abb. 3.6: Sankey-Diagramm eines einzelnen Zeitpunktes


Abb. 3.7: Sankey-Diagramm von allen Transaktionen

# 4

# Clusteringverfahren

Um die Verkäufe eines Simulationsdurchganges mit einer grossen Anzahl Transaktionen auf einer Karte übersichtlich darstellen zu können, muss der Umfang der Daten reduziert werden. Eine mögliche Variante ist das Zusammenfassen ähnlicher Datenpunkte zu Clustern. Ziel ist es hierbei, die Transaktionen als Pfeile auf der Karte einzeichnen zu können und deren Anzahl klein zu halten. Dabei sollen möglichst wenig Informationen aus den ursprünglichen Daten verloren gehen.

## 4.1 Nearest-Neighbor-Clustering

Die Grundidee dieses Verfahrens ist es, diejenigen Verkäufer zu kombinieren, welche direkt nebeneinander liegen und gleichzeitig auch den gleichen Abnehmer für ihre Ware haben. So kann man diese Transaktionen schon nahe an ihrem Ursprung zusammenfassen und die Anzahl einzelner Pfeile zu einem Käufer dezimieren. In einem ersten Schritt gilt es nun also, den nächsten Nachbarn eines Agenten identifizieren zu können, was im Abschnitt 4.1.1 genauer beschrieben ist. Falls nun der benachbarte Verkäufer den gleichen Käufer für seine Waren hat, können diese Agenten zu einem Cluster zusammengefasst werden.

### 4.1.1 Nearest-Neighbor-Algorithmus

Den nächsten Nachbarn eines Punktes zu finden, erscheint auf den ersten Blick trivial. Allerdings erschwert sich diese Aufgabe beträchtlich, je mehr Punkte es zu vergleichen gilt. Bei mehreren tausend Punkten kann die Berechnung des nächsten Nachbarn schnell zu einer massiven Erhöhung der Laufzeit führen. Um diese Problematik möglichst zu umgehen, wurde dazu eine Abwandlung eines k-nearest-neighbor-Algorithmus implementiert (vgl. I. H. Witten, 2011, S.132-138). Ziel dieses Algorithmus ist es, Datenpunkte mit den selben Eigenschaften einer bestimmten Anzahl Clustern zuzuordnen. Bei diesem Verfahren entstehen aufgrund der begrenzten Anzahl Clustern Fehlzuweisungen. Während beim ursprünglichen Algorithmus die Anzahl Cluster im Vornherein angegeben werden muss, wird bei dieser Variante eine vorerst unbekannte Anzahl gebildet.

Zuerst werden alle Punkten in einem kD-Baum angeordnet. Dieser beinhaltet die Punkte eines k-dimensionalen Raumes. Dabei steht k für die Anzahl Attribute (in diesem

Fall die x- und y-Koordinate eines Agenten). Um einen solchen Baum aufzubauen, wird die Menge der Punkte vorerst nach ihrem x-Wert sortiert und anschliessend das Element, welche den Median aller Punkte markiert, gesucht. Dieses Element bildet den ersten Knoten. Anschliessend wird diese Prozedur mit den jeweils dabei entstandenen Hälften wiederholt. Allerdings werden die Punkte immer abwechslungsweise jeweils nach ihrem x-Wert oder ihrem y-Wert sortiert, um den entsprechenden Median des übrig gebliebenen Raumes zu finden. Dies führt man fort, bis jeder Punkt im Baum abgebildet ist (siehe Abb. 4.1).

```
Data: int tiefe, punkteArray
Result: kD-Baum
ordne die Punkte nach der Grösse ihrer x-Koordinate;
root = Median-Punkt;
linkesArray = Punkte kleiner als Median;
rechtesArray = Punkte grösser als Median;
if punkteArray.size ==1 then
    füge den Punkt zum kDBaum hinzu;
end
if punkteArray.isEmpty() then
    return;
end
if punkteArray.size > 1 then
    if tiefe % 2 == 0 then
        sortiere punkteArray nach x-Wert;
        füge alle Punkte links vom Median zu linkesArray hinzu;
        füge alle Punkte rechts vom Median zu rechtesArray hinzu;
        füge den Median-Punkt dem kD-Baum hinzu;
    end
    if tiefe % 2 == 1 then
        sortiere punkteArray nach x-Wert;
        füge alle Punkte links vom Median zu linkesArray hinzu;
        füge alle Punkte rechts vom Median zu rechtesArray hinzu;
        füge den Median-Punkt dem kD-Baum hinzu;
    end
    if !linkesArray.IsEmpty() then
        starte diese Methode mit Parametern: tiefe+1, linkesArray
    end
    if !rechtesArray.IsEmpty() then
        starte diese Methode mit Parametern: tiefe+1, rechtesArray
    end
end
```

**Algorithm 1:** Aufbau eines kD-Baumes

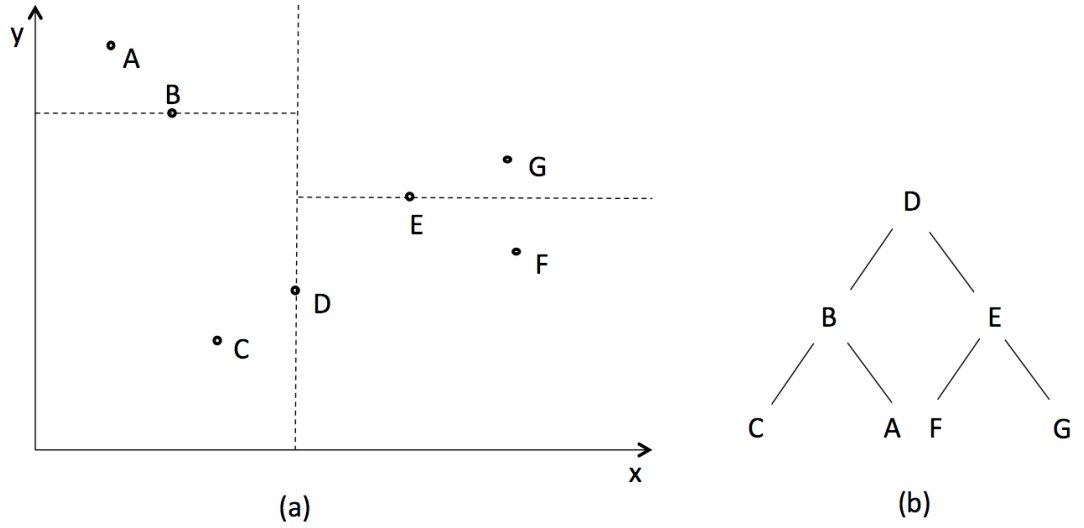



Abb. 4.1: Aufbau eines kD-Baumes

Um nun den nächsten Nachbarn eines Punktes (in unserem Beispiel Punkt F) finden zu können, nehmen wir als erste Annahme den Eltern-Knoten des Punktes (falls es die Wurzel des Baumes ist, eines seiner Kind-Knoten). Dies entspricht in unserem Fall Punkt E. Als nächstes bilden wir einen Kreis mit dem Mittelpunkt F und einem Radius mit dem Abstand von F und E. Der entstandene Kreis schneidet zwei der Rechtecke, in welche unser Koordinatensystem durch den kD-Baum eingeteilt wurde. Beide Teilbäume müssen wir auf Punkte untersuchen, welche näher an unserem Ursprungspunkt liegen, als unsere erste Annahme. Auf unserer Suche finden wir Punkt G, welcher der nächste Nachbar unseres Punktes F ist. (siehe Abb. 4.2).

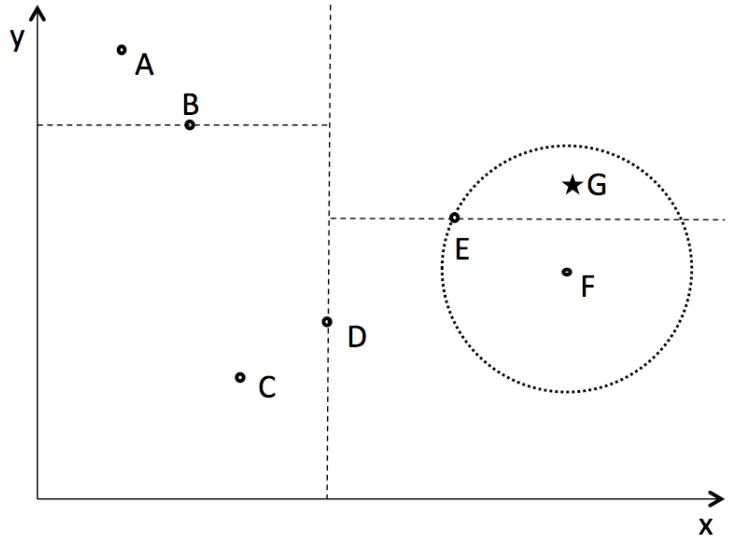



Abb. 4.2: Suche nach dem nächsten Nachbarn

**Data**: Point p
**Result**: Point nearestNeighbor
Suche den Punkt p im Kd-Baum;
KdTreeNode nearestNode = root;
double bestDistance = maxValue;
**if** *p == root* **then**
    nearestNode = p.getLeftChild();
    bestDistance = computeDistance(p, nearestNode);
**else**
    nearestNode = p.getParent();
    bestDistance = computeDistance(p, nearestNode);
**end**
Berechne die jeweils oberste und unterste Grenze, welche der x und y-Wert aufgrund der bestDistance noch annehmen darf;
**while** *ein Nachbar (child oder parent) des Knoten liegt innerhalb der Grenzen* **do**
    Berechne für diesen Punkt jeweils die Distanz;
    **if** *this.distance < bestDistance* **then**
        nearestNode = this.node;
        berechne neue Grenzen;
        Kontrolliere die Nachbarn des neuen nearestNode;
    **end**
**end**

**Algorithm 2:** Suche nach dem nächsten Nachbarn

Durch die Datenstruktur, welche der eines binären Suchbaumes entspricht, kann die Laufzeit der Suche auf mindestens $\log_2 n$ (n= Anzahl der Knoten) verkürzt werden. (I. H. Witten, 2011, S.134)

### 4.1.2 knn-Clustering Verfahren

Um nun Cluster zu bilden, wird untersucht, ob der nächste Nachbar des gewählten Punktes den gleichen Abnehmer für seine Waren hat. Wenn dies der Fall ist, werden diese beiden Transaktionen zu einem Cluster zusammengefügt. Darin werden zunächst sämtliche Daten in einer internen Liste abgespeichert. Da die Beziehung des "nächsten Nachbarn" surjektiv und nicht zwingendermassen symmetrisch ist, können durchaus noch weitere Transaktionen zum Cluster hinzugefügt werden. Falls der nächste Nachbar nicht dem gleichen Käufer liefert, wird ein neuer Cluster gebildet. Sobald alle Transaktionen in einem Cluster abgebildet werden, kann nun das geometrische Zentrum aller beinhalteten Käufer berechnet werden. Statt nun von jedem Käufer aus einen Graphen zum Käufer auf die Karte zu zeichnen, werden diese nun im Zentrum des Clusters zusammengefasst und ein einzelner Pfeil führt zum Käufer (siehe Abb. 4.3). Um die Effizienz zu steigern, kann dieses Verfahren mehrfach ausgeführt werden. Bei den weiteren Durchlaufen wird dann nicht mehr von den einzelnen Käufern ausgegangen, sondern von den neu entstandenen Cluster-Centern.

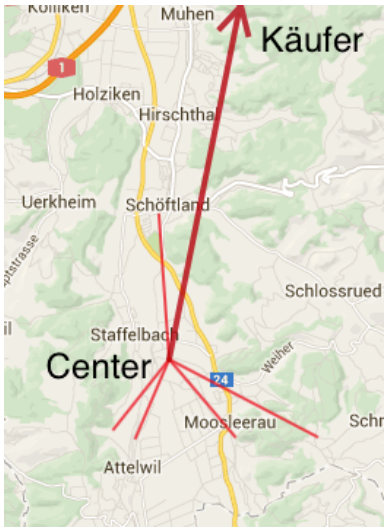



Abb. 4.3: Zusammenfassen der Graphen zu einem Cluster

### 4.1.3 Resultate und Analyse des knn-Clusterings

Dieses Verfahren ist stark davon abhängig, wie viele Verkäufer den gleichen Käufer beliefern. Sobald diese Voraussetzung nicht gegeben ist und nur sehr wenige Agenten den gleichen Abnehmer haben, können aus naheliegenden Gründen auch nur kleine

Cluster gebildet werden. In den Diagrammen 4.4, 4.5 und 4.6 kann man erkennen, dass die Effizienz dieses Clusteringverfahrens stark von den Eingabewerten abhängt. Für diese drei Messungen wurden jeweils verschiedene Simulationsergebnisse eingelesen und analysiert:

1. Simulation mit durchschnittlich 45 unterschiedlichen Käufern und 45 Verkäufern pro Monat, die Distanzen zwischen den Agenten ist relativ gross.

2. Simulation mit durchschnittlich 100 unterschiedlichen Käufern und 100 Verkäufern pro Monat, die Distanzen zwischen den Agenten ist etwas kleiner als in der ersten Simulation

3. Simulation mit durchschnittlich 4300 unterschiedlichen Käufern und 4600 Verkäufern pro Monat, die Distanzen von Verkäufer zu Käufer betragen nur wenige Kilometer.

Um das Verhältnis der zwischen der Anzahl der Käufer und Verkäufer zu ändern, wurden jeweils im Radius von 5km und 10km Radius alle Käufer zusammengefasst.

Auf den Grafiken können wir erkennen, dass die Anzahl Cluster abnimmt, je weniger Käufer wir haben. In der Simulation 3 lagen die Käufer nahe beieinander und konnten dementsprechend stark zusammengefasst werden. Man sieht auch, dass ein mehrmaliges Wiederholen des Algorithmus das Ergebnis verbessert, der Nutzen aber von Mal zu Mal abnimmt. Ein Vorteil dieser Variante liegt sicherlich darin, dass mit den ursprünglichen Daten keine Informationen verloren gehen. Falls man nun aber alle Käufer in einem bestimmten Radius durch einen einzelnen Agenten ersetzt, verliert man zwar Informationen, kann aber unter Umständen die Übersichtlichkeit der Darstellung erhöhen. Es handelt sich dabei um einen klassischen Trade-off zwischen Kosten und Nutzen.

## 4.2 Raster-Clustering

Eine weitere Variante besteht darin, Agenten und Transaktionen regional zusammenzufassen. Für unsere Zwecke legen wir ein Raster über unsere Karte und ordnen die Agenten den entsprechenden Zellen im Raster zu. Statt nun für jede einzelne Transaktion einen Pfeil zeichnen zu müssen, können wir nun die Transaktionen einer Zelle zusammenfassen (siehe 4.7). Hier werden die beiden Verkäufer (S1) dem gleichen Quadrat zugeordnet und die Transaktion nur mit einem einzelnen Pfeil aufgezeichnet. Die jeweiligen Mengen werden dabei addiert. Die Knoten werden jeweils in der Mitte der entsprechenden Quadrate gesetzt.

### 4.2.1 Resultate und Analyse des Raster-Clusterings

Das Zusammenfassen in einem Raster dieser Art ist verhältnismässig einfach. Durch die Wahl der Quadratgrösse a, kann ohne grossen Aufwand die Anzahl der dargestellten Pfeile variiert werden. Auch in diesem Beispiel verwendeten wir die in Abschnitt 4.1.3

beschriebenen Eingabedaten. Auf der Abbildung 4.8 erkennt man, dass ein gröberes Raster die Anzahl der eingezeichneten Transaktionen verringert. Wenn die Transaktionen nur über wenige Kilometer gehen, so wie in Simulation 3, verringert sich die Anzahl drastisch.

Auch in diesem Verfahren kann man erhebliche Unterschiede bei den Resultaten erkennen. Bei sehr kurzen Distanzen (Simulation 3) kann es vorkommen, dass beide Agenten, also der Käufer und der Verkäufer, im gleichen Quadrat zu liegen kommen. In diesem Fall wird die Transaktion überhaupt nicht dargestellt. Ein weiteres Problem liegt bei den Überlagerungen der Graphen. Da wir ein orthogonales Raster verwenden, liegen auch die Agenten alle auf entsprechenden Linien. Durch diese Anordnung kann es öfters vorkommen, dass mehrere Graphen übereinander zu liegen kommen. Um dies zu verhindern, könnte man die Agenten nicht in einem quadratischen Raster, sondern beispielsweise in Gemeinden oder Bezirken zusammenfassen. Dies müsste allerdings wieder für jede Simulation einer neuen Region manuell neu angepasst werden und kann mit der für diese Arbeit gewählten Infrastruktur nicht generisch erfolgen.

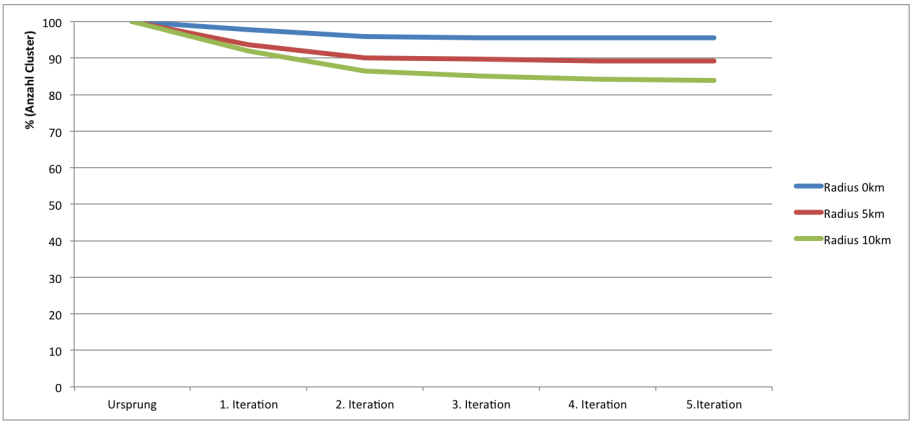



Abb. 4.4: Zusammenfassen von Pfeilen mit mehreren Iterationen (Simulation 1 mit 45 Verkäufern)

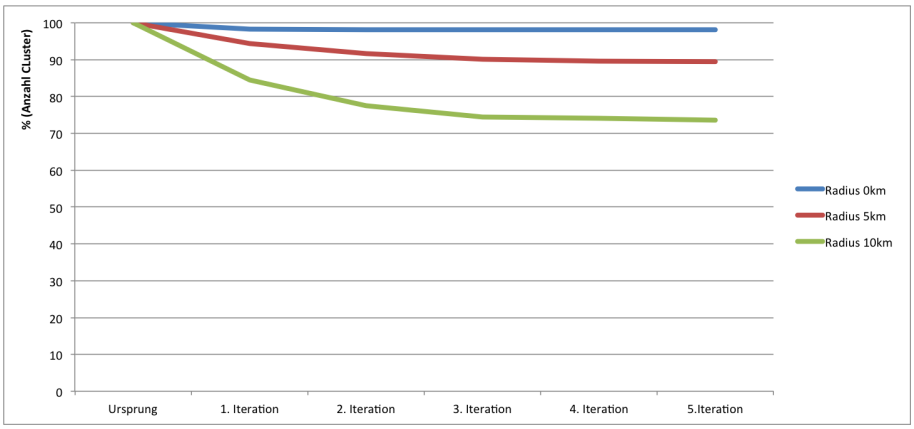



Abb. 4.5: Zusammenfassen von Pfeilen mit mehreren Iterationen (Simulation 2 mit 100 Verkäufern)

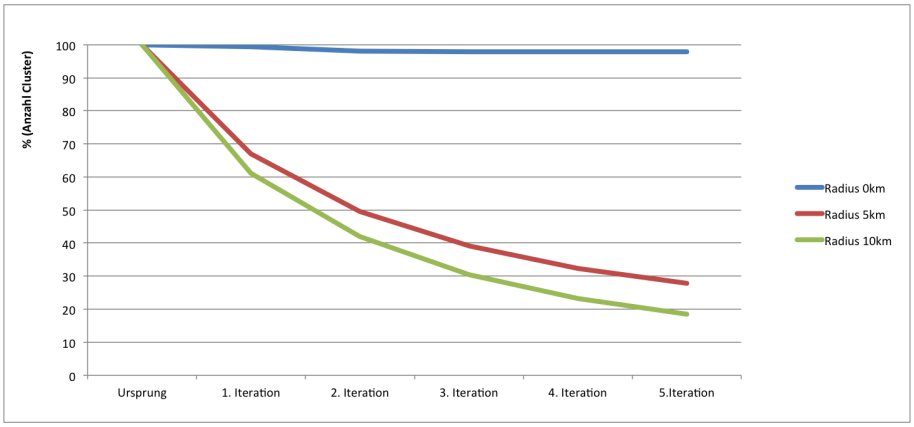



Abb. 4.6: Zusammenfassen von Pfeilen mit mehreren Iterationen (Simulation 3 mit 4600 Verkäufern)

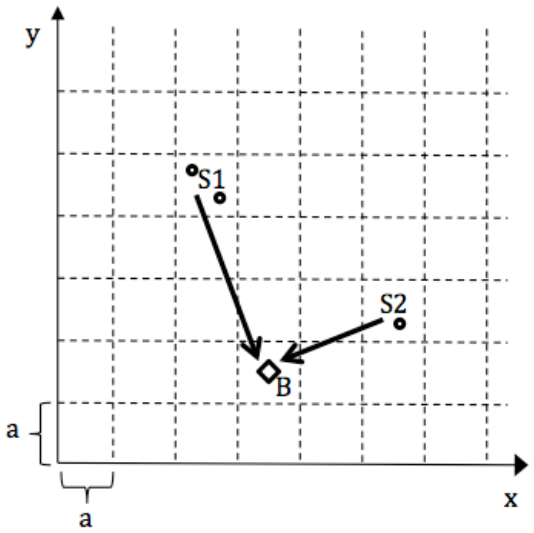



Abb. 4.7: Zusammenfassen von Agenten in einem Raster

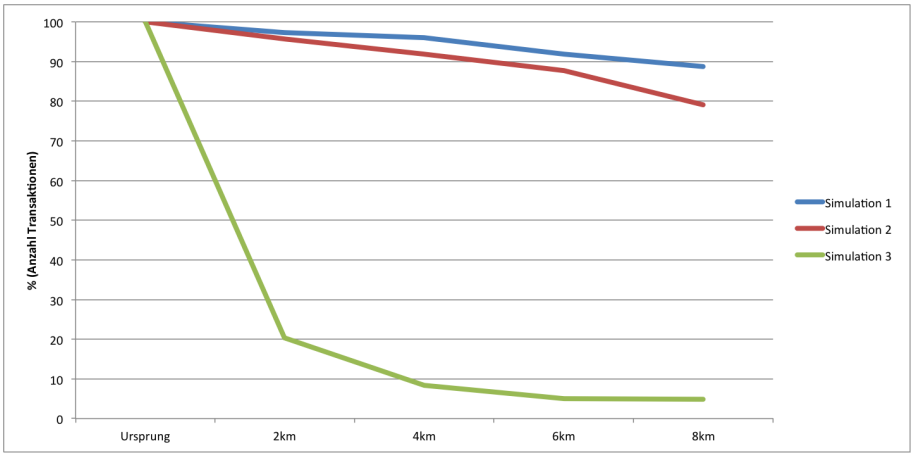



Abb. 4.8: Reduktion der Anzahl Transaktionen durch das Raster-Clustering

# 5

# Evaluation der Verfahren

Die vorgestellten Darstellungsformen haben jeweils beträchtliche Unterschiede in verschiedenen Bereichen. Die Eingabewerte der Simulation haben einen wesentlichen Einfluss darauf. Um die Darstellungsformen besser miteinander vergleichen zu können, werden folgende Punkte jeweils näher betrachtet:

- Übersichtlichkeit:
  Dies ist wohl das wichtigste Kriterium. Ein zentrales Ziel dieser Arbeit, ist es, die generierten Daten übersichtlich und für den Endnutzer einfach und schnell verständlich zu machen.
- Informationsgehalt/-verlust:
  Bei diesem Aspekt wird betrachtet, ob alle Informationen welche ursprünglich in den Eingabedaten enthalten waren, dargestellt werden. Durch Zusammenfassen von Informationen (sei es der geographische Ort oder der gehandelte Betrag) kann der Informationsgehalt sinken.
- Flexibilität:
  Die Eingabedaten unterscheiden sich teilweise massiv. Hier wird verglichen, wie gut man die Darstellungsform an beispielsweise sehr grosse Datenmengen oder unterschiedlich grosse geografische Distanzen zwischen den Agenten anpassen kann.
- Rechenzeit:
  Durch die gewählte Infrastruktur werden die meisten Daten vor der Visualisierung berechnet und für den Web-Browser aufbereitet. Die Berechnungen dauern für die einzelnen Darstellungsarten unterschiedlich lange und reagieren teilweise sehr stark auf grösser werdende Datenmengen.

## 5.1 Evaluation: Darstellung mit Pfeilen

- Übersichtlichkeit:
  Mit Pfeilen lassen sich die Daten auf einer Karte darstellen und mit den bereits vorgestellten Attributen wie Farben oder Linienstärke für den Betrachter gut verständlich darstellen. Allerdings verschlechtert sich die Übersicht mit zunehmender

Anzahl Transaktionen sehr schnell. Zwar kann dem mittels Zusammenfassen der Pfeile entgegen gewirkt werden, aber auch das ist nur bis zu einem gewissen Grad. Die Übersicht leidet hauptsächlich wegen der sich überschneidenden Linien. Demzufolge spielt nicht nur die Anzahl, sondern auch die geografische Verteilung der Agenten eine wesentliche Rolle. Liegen diese weit auseinander, erhöhen sich Überschneidungen zwangsläufig.

- Informationsgehalt/-Verlust:
  Im Wesentlichen kann man sämtliche Transaktionen ohne Informationen zu verlieren darstellen. Falls man nun aber aufgrund vieler Eingabedaten deren Menge einschränken will indem man beispielsweise verschiedene Punkte in einem bestimmten Radius zusammenfasst, sinkt der Informationsgehalt. Hier entsteht ein Trade-off zwischen der Übersichtlichkeit und dem Informationsgehalt.

- Flexibilität:
  Durch das Zusammenfassen von Punkten wie zum Beispiel im Raster-Clustering (siehe 4.2) kann die Anzahl Graphen angepasst werden. Ausserdem stehen hier zwei verschiedene Varianten zur Verfügung, wie Graphen miteinander verbunden werden können (eine davon statisch, die andere flexibel). Es existieren also durchaus Varianten, mit denen die Vektoren nach Gutdünken angepasst werden können.

- Rechenzeit:
  Die Berechnung der Graphen, insbesondere die Kombination derselben, ist vergleichsweise rechenintensiv. Es sind Algorithmen implementiert in welchen mehrfach über verschiedene Listen iteriert wird. In dieser Hinsicht kann die Implementation noch optimiert werden. Die Anbindung einer indexierten Datenbank ist eine denkbare Möglichkeit.

## 5.2 Evaluation: Heatmap

- Übersichtlichkeit:
  Auf der Heatmap kann die Menge der gehandelten Güter pro Agent dargestellt werden. Wahlweise können Verkäufer, Käufer oder beide eingeblendet werden. Höhere Verkaufsmengen werden rot dargestellt, während kleinere grün eingefärbt werden. Es ist leicht erkennbar, wo sich Gebiete mit hohen Verkaufszahlen befinden. Die Mengen werden relativ zur grössten gehandelten Menge berechnet. Diese Darstellungsform ist im Vergleich zu den anderen Varianten sehr übersichtlich.

- Informationsgehalt/-Verlust:
  Die Information welcher Agent mit wem interagiert geht auf der Heatmap komplett verloren. Die Karte passt sich auf ein Hinein- oder Hinauszogen automatisch an. Bei einem kleinen Zoofaktor werden nahe beieinander liegende Punkte zusammengefasst, während auf einer höheren Zoomstufe die Agenten alle einzeln dargestellt werden.

- Flexibilität:
  Die Heatmap passt sich flexibel an die Eingabedaten an. Die Farbtöne werden relativ zum höchsten gehandelten Wert gesetzt. Ausserdem lassen sich auch sehr grosse Datenmengen darstellen. Wie schnell sich jeweils nahe beieinander liegende Punkte verbinden kann manuell im Quellcode angepasst werden.

- Rechenzeit:
  Die eigentliche Darstellung wird von Google-Maps übernommen. Es müssen nur die jeweiligen Objekte in Javascript generiert werden. Die Rechenzeit ist gering, da die ursprünglichen Transaktionen ohne andere Mutationen (wie zum Beispiel Clustering) direkt in die entsprechenden Objekte umgewandelt werden.

## 5.3 Evaluation: Einzugsgebiet

- Übersichtlichkeit:
  Für jeden Käufer wird ein Polygon gezeichnet, welches die äussersten Verkäufer umspannt. Bei unserer Implementation ist es möglich, diese Polygone einzeln anzeigen zu lassen. Wenn man alle gleichzeitig einblendet, überschneiden sie sich stark und es wird sehr schwierig zu erkennen, welches Polygon zu welchem Käufer gehört. Dieses Problem verschärft sich mit grösser werdender Anzahl der Käufer. Die einzelnen Verkäufer werden ausserdem mit Markern versehen, da sie auch innerhalb des gezeichneten Polygons sein können.

- Informationsgehalt/-Verlust:
  Die Verkaufsmenge wird bei dieser Darstellung ignoriert. Es werden nur die geografischen Zusammenhänge aufgezeigt. Die entsprechenden Daten bleiben aber erhalten, ausser man fasst die Käufer wie im Kapitel 4.1.3 erwähnt, zusammen.

- Flexibilität:
  Dadurch, dass man einzelne Polygone ein- und ausblenden kann, ist diese Darstellungsform recht flexibel. Wenn nun allerdings die Anzahl der Agenten allzu gross wird, verdecken die eingezeichneten Marker das Sichtfeld.

- Rechenzeit:
  Der Flaschenhals der Rechenzeit liegt bei der Berechnung des Polygons, welches mit einer Rechenzeit von $O(nlogn)$ zu Buche schlägt. Im Vergleich zu der Berechnung der anderen Varianten ist dies eher wenig.

## 5.4 Evaluation: Sankey-Diagramm

- Übersichtlichkeit:
  Das Sankey-Diagramm bietet eine gute Übersicht der einzelnen Agenten und deren Handelspartner. Man kann sehr einfach erkennen, mit wem wie viele Waren ausgetauscht werden. Die Menge der gehandelten Waren wird im Diagramm nicht

absolut, sondern relativ im Vergleich zum gehandelten Gesamtvolumen berechnet. Aus diesem Grund kann bei einer hohen Anzahl Agenten die Übersicht verloren gehen.

- Informationsgehalt/-Verlust:
  Wie erwähnt, wird die gehandelte Menge relativ zum Gesamtvolumen dargestellt. Demzufolge geht hier etwas Information verloren. Die geografische Lage wird bei unserer Implementation anhand der Koordinaten bei den jeweiligen Agenten aufgeführt, aber für den Betrachter ist es nicht ohne Weiteres ersichtlich, wo sich der einzelne Agent befindet.

- Flexibilität:
  Die Darstellungsform ist bis zu einer bestimmten Anzahl involvierter Agenten sehr flexibel. Mit der Google Charts API lassen sich auch Multilevel-Diagramme aufzeichnen, für den Fall dass ein Käufer seine Waren an einen weiteren Agenten weiter verkauft. Die API bietet ausserdem eine grosse Anzahl Optionen, mit denen man sein Diagramm für spezifische Zwecke anpassen kann.

- Rechenzeit:
  Die Daten werden im Browser berechnet und gerendert. Es werden keine Daten an einen Server geschickt. Die Diagramme reagieren aber trotzdem schnell auf Veränderungen.

## 5.5 Evaluation: Nearest-Neigbor-Clustering

- Übersichtlichkeit:
  Das Clusteringverfahren ist sehr stark von den Eingabedaten abhängig. Es ist nur dann wirkungsvoll, wenn mehrere nahegelegene Verkäufer den gleichen Käufer beliefern. Falls dies der Fall ist, können bereits nahe bei den Käufern Pfeile zusammengefasst werden, was die Übersichtlichkeit erhöht. In der Praxis ist die Wirksamkeit aufgrund der Kriterien allerdings nur von geringem Nutzen.

- Informationsgehalt/-Verlust:
  Bei diesem Clusteringverfahren bleiben grundsätzlich alle Informationen erhalten, ausser man fasst nahe gelegene Käufer zusammen. Der Nutzen des Verfahrens ist aber sehr stark von den Eingabedaten abhängig.

- Flexibilität:
  Die Wirksamkeit des Verfahrens kann verbessert werden, indem Käufer zusammengefasst werden. Wie in Abschnitt 4.1.3 beschrieben, kann dies einen grossen Einfluss auf das Ergebnis haben, bedeutet aber gleichzeitig auch einen Informationsverlust. Ansonsten ist der Algorithmus in dieser Form absolut unflexibel.

- Rechenzeit:
  Das Clusteringverfahren benötigt, insbesondere bei vielen Agenten und mehreren Durchläufen, im Vergleich zu den anderen Varianten am meisten Zeit.

## 5.6 Evaluation: Raster-Clustering

- Übersichtlichkeit:
  Auch hier hängt die Übersichtlichkeit von den Eingabedaten ab. Wir haben allerdings durch die Wahl der Rastergrösse ein Instrument, mit welchem die Anzahl der gezeichneten Pfeile stark reduzieren werden kann. Durch die Wahl eines rechtwinkligen Rasters liegen die Agenten der Pfeile jeweils auf entsprechenden Linien und können sich so gegenseitig überlagern. Durch andere Formen könnte diese Problematik umgangen werden.

- Informationsgehalt/-Verlust:
  Je grösser die gewählten Quadrate sind, umso mehr Informationen über die exakten geographischen Standorte gehen verloren. Wenn eine Transaktion vollständig in einem einzelnen Quadrat zu liegen kommt, wird sie komplett ignoriert. Der Informationsverlust ist entsprechend abhängig, wie stark man die Anzahl Graphen durch ein gröberes Raster verringern will.

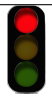
- Flexibilität:
  Durch die Wahl einer grossen Quadratlänge, kann die Anzahl benötigter Graphen stark verringert werden. Die einzelnen Rasterelemente müssen nicht zwangsläufig Quadraten entsprechen, sondern können theoretisch jede erdenkliche Form haben.

- Rechenzeit:
  Bei unserer Implementation ist es wegen der quadratischen Felder sehr einfach, Agenten einem bestimmten Raster zuzuordnen. Dementsprechend gering fällt auch die Rechenzeit aus. Bei der Wahl einer komplexeren Form kann sich dies allerdings verschlechtern.

## 5.7 Evaluation: Gesamtübersicht

In untenstehende Tabelle werden die einzelnen vorgestellten Methoden zusammenfassend dargestellt. Die Farben haben dabei folgende Bedeutung:

- Grün: gut, entspricht den Erwartungen
- Gelb: genügend, aber nicht in allen Situationen den Erwartungen entsprechend
- Rot: mangelhaft, die Resultate entsprechen nicht den Erwartungen

| | Übersichtlich-keit | Informations-gehalt | Flexibilität | Rechenzeit |
|---|---|---|---|---|
| Graphen |  |  |  |  |
| Heatmap |  |  |  |  |
| Einzugsgebiet |  |  |  |  |
| Sankey-Diagramm |  |  |  |  |
| Nearest-Neighbor-Clustering |  |  |  |  |
| Raster-Clustering |  |  |  |  |

# 6

# Kombinationsmöglichkeiten der Verfahren

Die einzelnen ausgearbeiteten Verfahren haben jeweils ihre individuellen Vor- und Nachteile. Es liegt entsprechend nahe, die Verfahren miteinander zu kombinieren, um etwaige Schwächen zu kompensieren und die Vorteile nutzen zu können. Das Ziel ist es, die Übersichtlichkeit zu maximieren, ohne Informationen aus den ursprünglichen Daten zu verlieren.

### Clusteringalgorithmen & Graphen

Eine naheliegende Möglichkeit ist es, die Varianten zu kombinieren, welche mit Pfeilen dargestellt werden. Bei den beiden Clusteringverfahren werden die Resultate jeweils entsprechend eingezeichnet. Diese Pfeile können nun mit den beiden in Kapitel 3.1.1 vorgestellten Algorithmen kombiniert werden. Beim Raster-Clustering kann so das Problem mit den sich überdeckenden Linien etwas verringert werden, da die Punkte, wo die Pfeile miteinander verbunden werden nicht parallel zum Raster liegen, sondern spezifisch für jeden Pfeil berechnet werden. Auch beim Nearest-Neigbor-Clustering können die Pfeile miteinander kombiniert werden. Hier werden allerdings nur die Pfeile vom Cluster-Zentrum bis zum Käufer berücksichtigt. Auf der Abbildung 6.1 können wir erkennen, dass auch durch die Kombination dieser Varianten die Übersichtlichkeit nicht optimal ist. Es ist immer noch der Fall, dass sich die Pfeile überschneiden und so die Gesamtübersicht erheblich beeinträchtigen.

### Sankey-Diagramm & Graphen

Da auf dem Sankey-Diagramm nur schwer nachzuvollziehen ist wo sich die entsprechenden Agenten befinden, wäre es möglich, dass bei der Auswahl eines spezifischen Agenten oder einer Transaktion die entsprechenden Kanten dynamisch auf der Karte eingezeichnet werden. Dies würde auch die Übersicht der Karte selbst erhöhen, da nicht alle Pfeile gleichzeitig dargestellt werden. Einige Informationen würden entsprechend doppelt dargestellt. Das Sankey-Diagramm würde demzufolge als Gesamtübersicht dienen, während die Graphen auf der Karte die im Sankey-Diagramm verlorenen oder nur schwer zu entnehmenden Informationen anzeigen. Im Rahmen dieser Arbeit konnte diese Funktion aufgrund der gewählten Softwarearchitektur allerdings nicht implementiert werden, wäre aber durchaus umsetzbar.

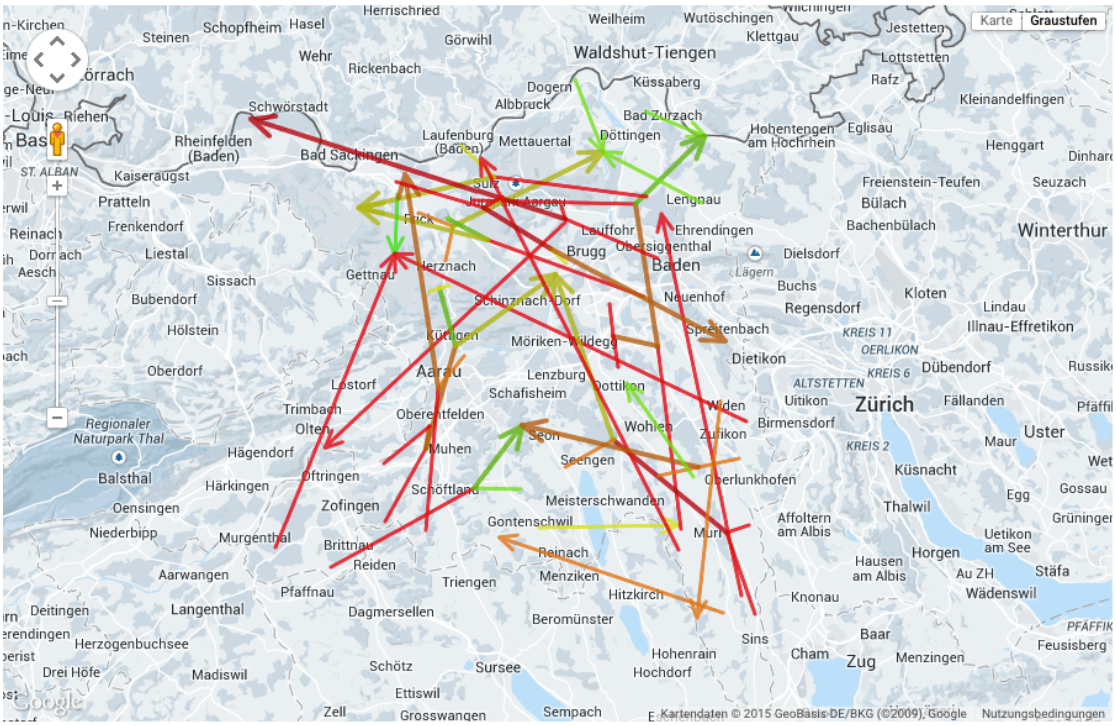



Abb. 6.1: Kombinieren von Graphen (Variante 2) mit Nearest-Neighbor-Clustering

### Heatmap & Einzugsgebiet

Die einzige namhafte Schwäche der Heatmap ist die fehlende Visualisierung davon, welche Agenten miteinander interagieren. Durch die Kombination mit der Anzeige des Einzugsgebietes, kann dieses Manko beseitigt werden. Die beiden Varianten ergänzen sich in Sachen Stärken und Schwächen sehr gut. Bei einer grossen Anzahl Agenten, insbesondere vieler verschiedener Käufer, kann die Darstellung durch die vielen Marker dennoch unübersichtlich werden. Die Verknüpfung dieser beiden Varianten liefert meines Erachtens das beste Ergebnis. Sämtliche Informationen bleiben erhalten und die Übersichtlichkeit ist gut. Es benötigt viele Agenten auf engstem Raum um die Übersichtlichkeit zu beeinträchtigen.

### Heatmap & Graphen

Anstatt wie bei der vorangegangenen Variante die Polygone, werden hier Pfeile eingezeichnet. Auch hier kann man diese dynamisch ein- und ausblenden lassen. Im direkten Vergleich zur Darstellung des Einzugsgebietes, bietet diese Art der Darstellung keinerlei Vorteile. Die Pfeile sind weniger übersichtlich und liefern auch keine zusätzlichen Informationen. Der gehandelte Betrag wird redundant bereits in der Heatmap visualisiert. Aus diesem Grund wurde diese Kombination nicht implementiert.

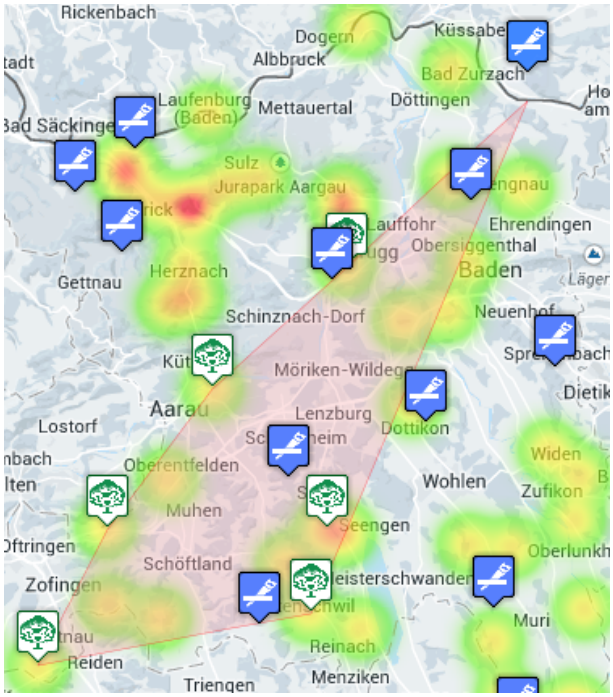



Abb. 6.2: Kombination der Heatmap und der Anzeige des Einzugsgebietes

# 7

# Implementation

Das Ziel bei der Wahl der Technologien war es, die Infrastruktur möglichst schlank zu halten. Die Applikation sollte direkt aus der Programmierumgebung Eclipse heraus gestartet werden können und einfach zu bedienen sein. Unsere Applikation besteht aus zwei Teilen. Das Backend, welches in Java programmiert ist, liest die csv-Dateien ein und berechnet die Ergebnisse der in dieser Arbeit vorgestellten Verfahren. Diese Ergebnisse werden anschliessend in eigens dafür generierte Javascript-Dateien geschrieben (siehe Abb. 7.1). Allerdings bedeutet es auch, dass gerade bei grossen Datenmengen das Erstellen der Javascript-Dateien entsprechend Zeit in Anspruch nimmt. Alternativ könnten diese Daten auch in eine Datenbank geschrieben werden. Die grafische Darstellung wird vom Webbrowser übernommen. Wir benutzen die Karten von Google-Maps, da deren API eine Vielzahl für uns nützlicher Funktionen beinhaltet. Wir haben vom Browser aus keine Möglichkeit, Parameter für unsere Algorithmen zu setzen. Dies muss direkt im Quellcode in der Klasse "Driver.java" geschehen.

Der grobe Ablauf sieht folgendermassen aus:

1. Einlesen der csv-Datei und umwandeln der Werte in Transactions.
2. Kombinieren der Käufer im eingegebenen Radius (sofern er grösser ist als 0).
3. Bilden des Kd-Baumes mit allen gefundenen Koordinaten der Agenten.
4. Aufsplitten der Transactions in Batches (welche jeweils alle Transactions eines bestimmten Datums beinhalten).
5. Bilden von Clustern mit dem nn-Clustering-Algorithmus.
6. Wiederholen des nn-Algorithmus, sofern die Anzahl Iterationen grösser als 0 ist.
7. Berechnen und zusammenfügen der Graphen mit der ausgewählten Variante. Eingabedaten sind hierbei die einzelnen Cluster-Listen pro Batch.
8. Berechnen der neuen Koordinaten, welche durch das Raster-Clustering definiert werden. Bilden einer neuen Cluster-Liste.
9. Berechnen der Graphen vom Raster-Clustering. Auch hier mit der entsprechenden Variante zum Kombinieren.

10. Schreiben der entsprechenden Javascript-Datei durch einen File-Writer.

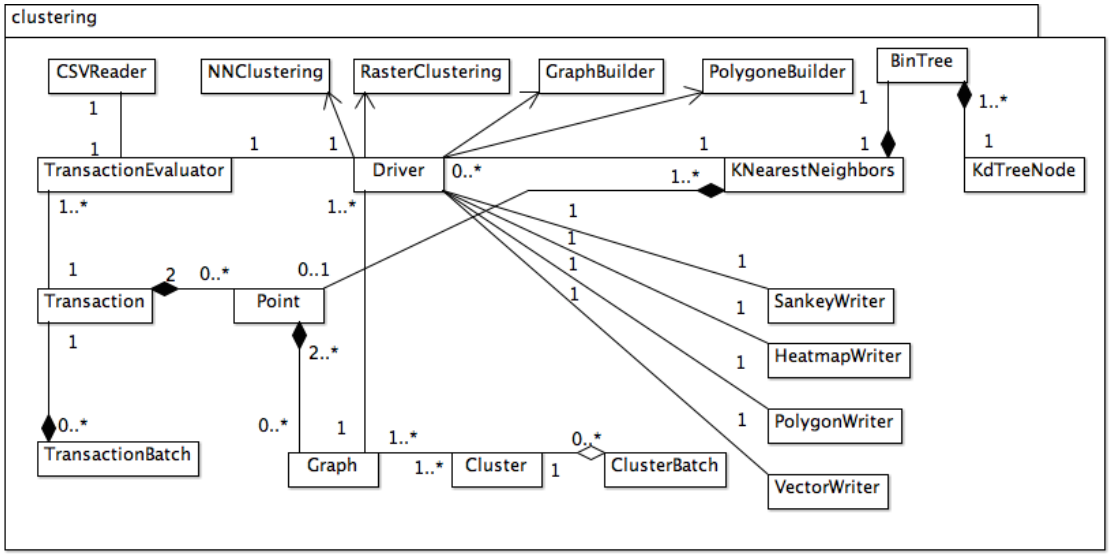



Abb. 7.1: Übersicht über den Ablauf des Programmes

Die gesamte Implementation umfasst rund 3100 Linien Code, wobei ein Grossteil davon auf das in Java programmierte Backend fällt. Neben den generierten Dateien, umfasst das Frontend die Dateien Startpage.html, Startpage.js und Startpage.css. Ausserdem wird die Javascript-Bibliothek jQuery benutzt, um Funktionen wie beispielsweise den Schieber für das Datum zu integrieren. Das Programm lässt sich einfach direkt aus der Programmierumgebung Eclipse heraus starten. Sobald die benötigten Daten generiert wurden, erscheint auf der Konsole ein Link, welchen der Nutzer mit seinem bevorzugten Webbrowser öffnen kann. Auf der Abbildung 7.2 ist ersichtlich, wie Benutzeroberfläche aussieht. Folgende Auswahlvarianten stehen dabei zur Verfügung:

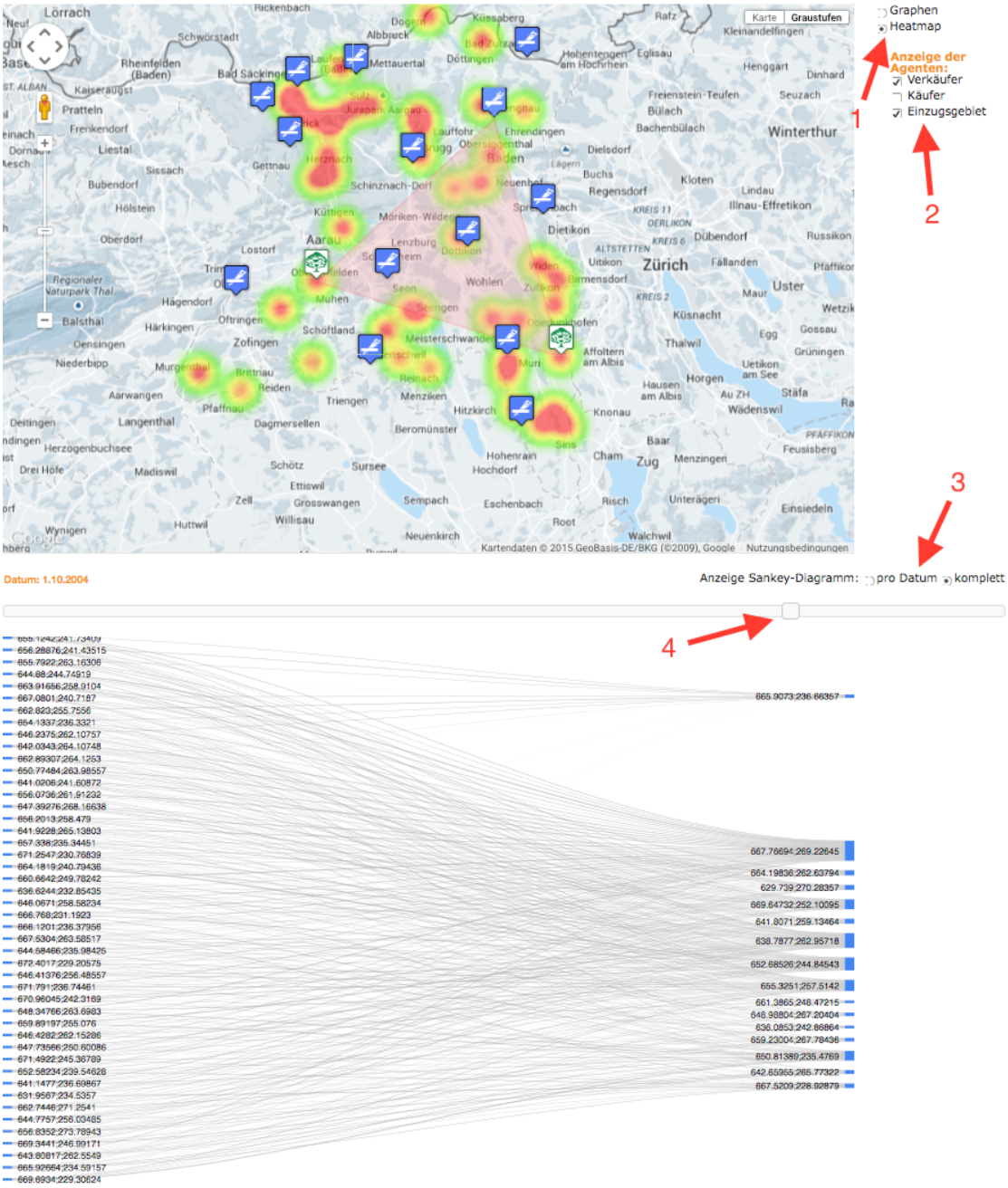



Abb. 7.2: Grafische Benutzeroberfläche

1. Hier kann der Nutzer auswählen, ob er die Daten auf der Karte als Graphen oder als Heatmap anzeigen lassen will. Entsprechend werden dann die Möglichkeiten unter Punkt 2 angepasst.

2. Bei diesen Boxen können spezifische Varianten angezeigt werden.
    - nn-Clustering: Anzeige der Ergebnisse des nn-Clusteringsverfahrens
    - Ergebnisse des Raster-Clusterings mit der entsprechenden Länge der Quadrate
    - Käufer: Anzeige aller Käufer auf der Heatmap
    - Verkäufer: Anzeige aller Verkäufer auf der Heatmap
    - Einzugsgebiet: Einblenden der Marker für das Einzugsgebiet. Durch das Klicken auf einen Marker wird das entsprechende Einzugsgebiet ein- oder ausgeblendet.
3. Auswahl der Anzeige für das Sankey-Diagramm: Entweder wird nur ein Batch angezeigt, oder eine Gesamtübersicht über sämtliche Transaktionen.
4. Schieber für die Auswahl eines Batches. Die Daten des gewählten Datums werden sowohl auf der Karte, als auch auf dem Sankey-Diagramm angepasst (sofern diese Option ausgewählt ist)

Die Benutzeroberfläche kann mit den meisten gängigen Browsern aufgerufen werden. Getestet wurde dies mit Safari 8.0 , Google Chrome 41.0.2272.101m und Firefox 35.0.1. Der Internet Explorer unterstützt nicht alle benötigten Funktionen.

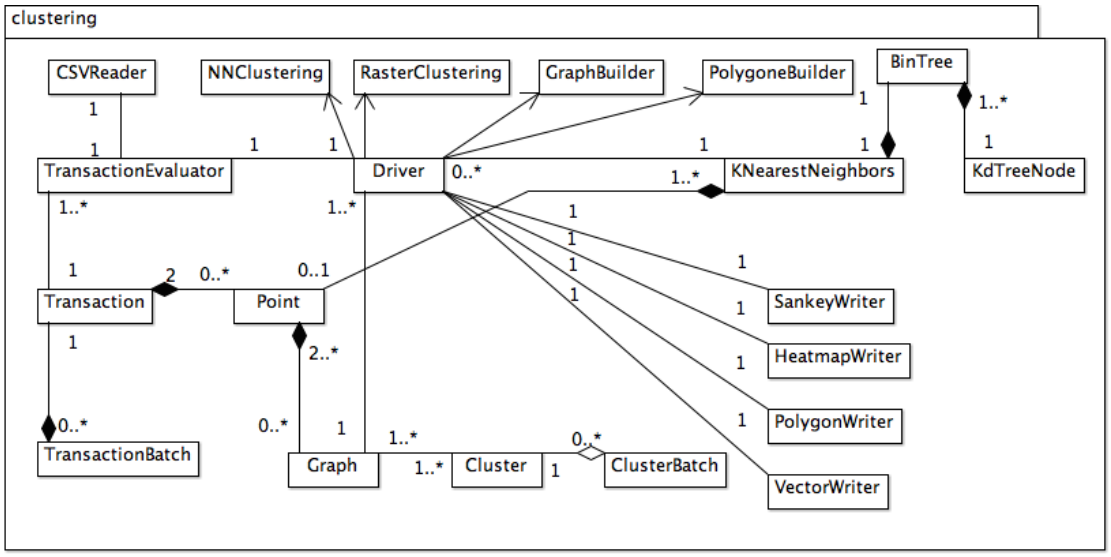



Abb. 7.3: vereinfachtes Klassendiagramm des Backendes

# 8

# Fazit

Alle behandelten Varianten haben ihre Stärken und Schwächen. Was sie allerdings gemeinsam haben ist die Tatsache, dass die jeweiligen Ergebnisse stark von den Eingabedaten abhängen. So lässt sich keine allgemein gültige, optimale Vorgehensweise bestimmen. Da die jeweiligen Ausgangsdaten zu Beginn für den Endnutzer eher kryptisch sind, ist es schwierig, auf Anhieb die geeigneten Parameter für die einzelnen Varianten zu setzen. Ausser der Heatmap bekunden alle anderen Varianten Probleme bei steigender Anzahl Transaktionen. Erhöht sich diese zu stark, leidet die Übersichtlichkeit massiv. Durch ein dynamisches Verhalten der Darstellung kann die Übersichtlichkeit in vielen Fällen erhöht werden. Falls der Nutzer auf eine statische Darstellung, zum Beispiel in ausgedruckter Form, angewiesen ist, ist dies nutzlos. In diesem Fall hilft nur ein Zusammenfassen von Informationen, was gezwungenermassen zu einem Informationsverlust führt.

Die gewählte Infrastruktur bietet den Vorteil, dass kein Server oder keine Datenbank benötigt werden, hat aber auch dementsprechend Nachteile. Es wäre wünschenswert, wenn die Eingaben für die jeweiligen Algorithmen direkt über die Benutzeroberfläche getätigt werden könnte. Der Einsatz einer Datenbank würde die Manipulation und Abfrage der Daten um einiges vereinfachen. Eine sehr grosse Anzahl Transaktionen erhöht die Rechenzeit massiv und kann im schlimmsten Fall sogar zum Absturz des Programmes führen. Der Flaschenhals ist hierbei der FileWriter, welcher aufgrund eines zu kleinen Heapspaces (OutOfMemoryError) zum Abbruch des Programmes führt, da die Javascript Dateien mehrere Tausend Zeilen umfassen können.

Meiner Meinung nach ist die Heatmap in Kombination mit dem Einzugsgebiet die benutzerfreundlichste Variante. Die Darstellung als Pfeile liegt dagegen hinter meinen Erwartungen zurück und bedarf einer Überarbeitung für ein befriedigenderes Ergebnis. Als Eingabedaten dienten für diese Arbeit jeweils Simulationsergebnisse für den Kanton Aargau. Für zukünftige Analysen wären sicherlich auch Daten aus anderen Gebieten oder gar einer gesamtschweizerische Simulation interessant. Auch eine etwas aufwändigere Infrastruktur mit einem Server und angebundener Datenbank müsste bei einer Weiterentwicklung ins Auge gefasst werden.

# Referenzen

**Andrew, A.M.** 1979. "Another efficient algorithm for convex hulls in two dimensions." *Information Processing Letters 9.*

**F. Kostadinov, B. Steubing, S. Holm.** 2012. "Vorgehen zur agentenbasierten Modellierung eines schweizerischen Waldenergieholzmarktes." *Schweizerische Zeitschrift für Forstwesen*, , (163, 10).

**I. H. Witten, E. Frank, M. A. Hall.** 2011. *Data Mining - Practical Machine Learning Tools and Techniques.* . 3. ed., Morgan Kaufmann.

**Schöning, U.** 2008. *Ideen der Informatik - Grundlegende Modelle und Konzepte der Theoretischen Informatik.* . 3. ed., Oldenbourg.

**wikibooks.org.** 2014. "Convex Hull Algorithm." *http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain*, Accessed 18.3.2015.

# A

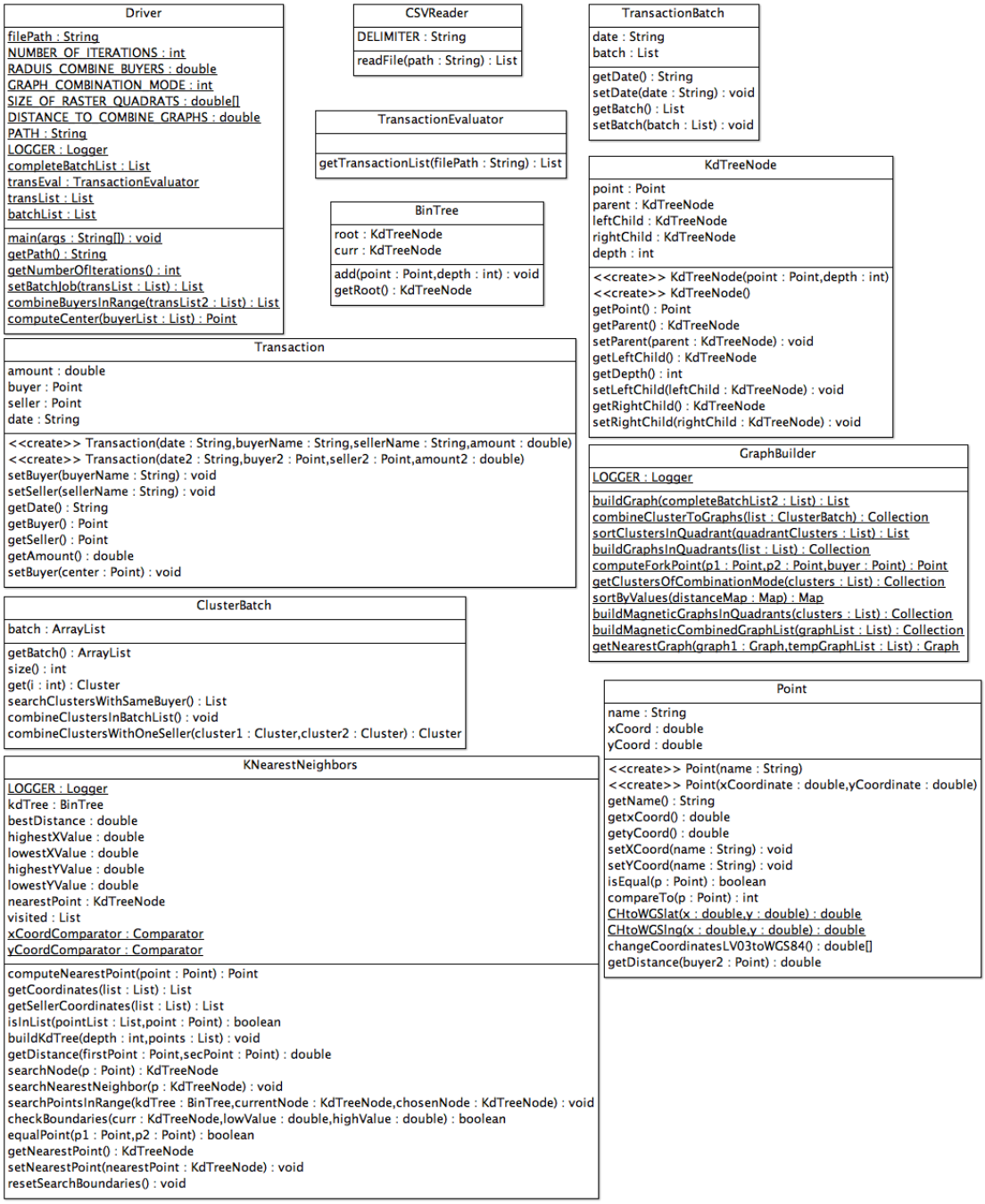
# Appendix

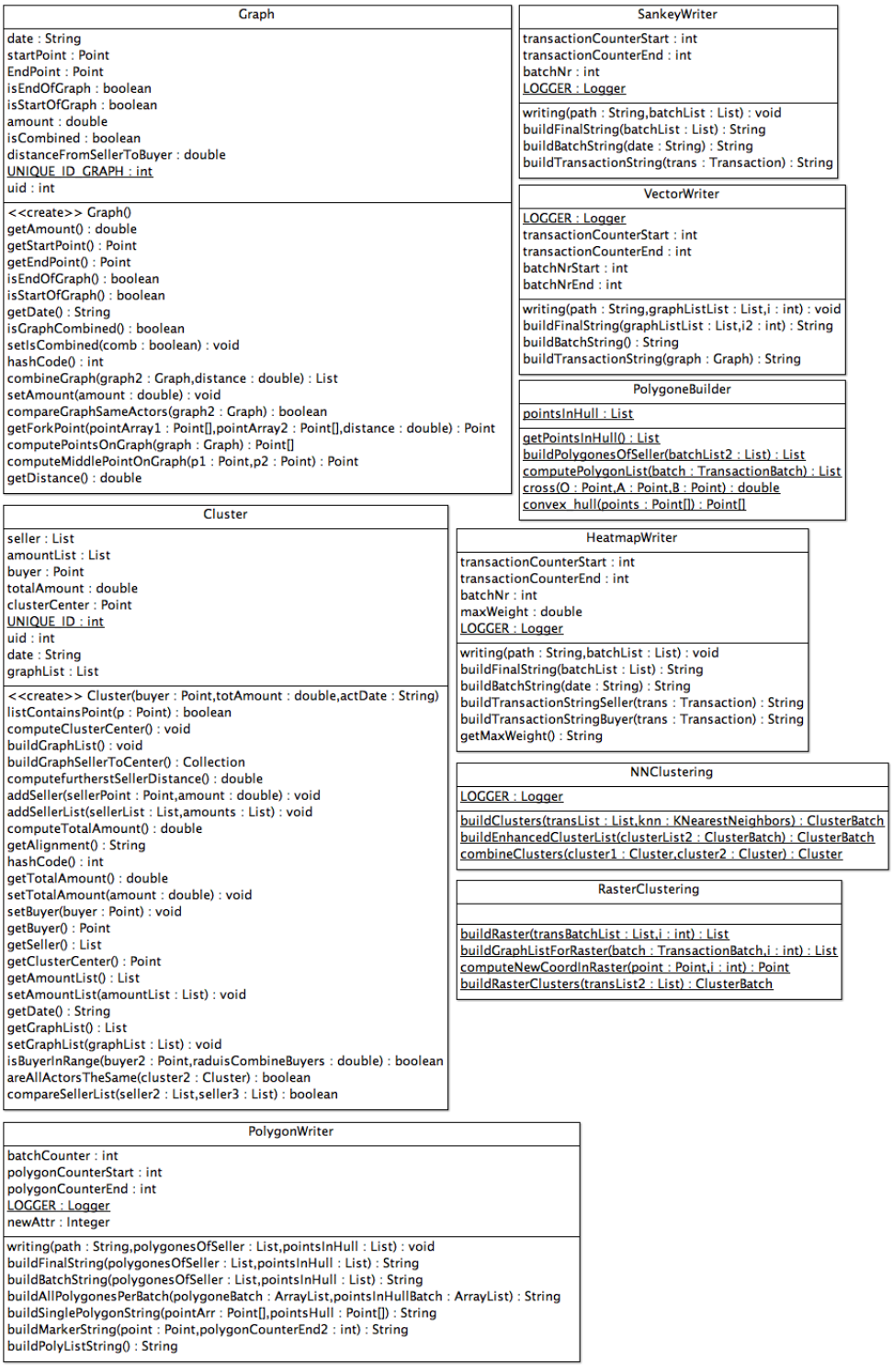
## A.1 Parameter

Folgende Parameter können vom Nutzer in der Klasse Driver.java angepasst werden:

- int NUMBER_OF_ITERATIONS: Bei diesem Parameter kann festgelegt werden, wie viel mal der Nearest-Neighbor-Clustering-Algorithmus ausgeführt werden soll. Diese Zahl muss zwingend grösser als 0 sein.
- RADIUS_COMBINE_BUYER: Hier kann der Radius (in km) festgelegt werden, in welchem Käufer kombiniert werden sollen. Alle entsprechenden Agenten werden zusammengefasst und durch einen neuen Punkt ersetzt, welcher dem geometrischen Zentrum aller dieser Käufer entspricht.
- GRAPH_COMBINATION_MODE: Auf welche Art die Graphen miteinander kombiniert werden, hängt von diesem Parameter ab (1 für die Variante 1, 2 für Variante 2).
- SIZE_OF_RASTER_QUADRATS: Um die Seitenlänge für unsere Quadrate im Rasterclustering zu verändern, können die Parameter in diesem double Array verändert werden. Die Anzeige auf der Benutzeroberfläche müsste entsprechend von Hand angepasst werden, sofern die Werte verändert werden.
- DISTANCE_TO_COMBINE_GRAPHS: Bei der Variante 2 der Kombination von Graphen, kann die Distanz, welche eine Fusion auslöst, gewählt werden. Sie entspricht diesem Parameter (in Kilometern)
- PATH: Hier wird der Pfad auf die einzulesende cvs-Datei gesetzt.
- RADIUS: Diese Variable befindet sich in der Startpage.js-Datei. Damit kann man variieren, wie schnell sich die Datenpunkte auf der Heatmap verbinden. Je grösser dieser Wert ist, umso schneller geschieht dies.

# A.2 Übersicht aller implementierten Klassen




Graph
date : String
startPoint : Point
EndPoint : Point
isEndOfGraph : boolean
isStartOfGraph : boolean
amount : double
isCombined : boolean
distanceFromSellerToBuyer : double
UNIQUE_ID_GRAPH : int
uid : int
<<create>> Graph()
getAmount() : double
getStartPoint() : Point
getEndPoint() : Point
isEndOfGraph() : boolean
isStartOfGraph() : boolean
getDate() : String
isGraphCombined() : boolean
setIsCombined(comb : boolean) : void
hashCode() : int
combineGraph(graph2 : Graph,distance : double) : List
setAmount(amount : double) : void
compareGraphSameActors(graph2 : Graph) : boolean
getForkPoint(pointArray1 : Point[],pointArray2 : Point[],distance : double) : Point
computePointsOnGraph(graph : Graph) : Point[]
computeMiddlePointOnGraph(p1 : Point,p2 : Point) : Point
getDistance() : double
SankeyWriter
transactionCounterStart : int
transactionCounterEnd : int
batchNr : int
LOGGER : Logger
writing(path : String,batchList : List) : void
buildFinalString(batchList : List) : String
buildBatchString(date : String) : String
buildTransactionString(trans : Transaction) : String
VectorWriter
LOGGER : Logger
transactionCounterStart : int
transactionCounterEnd : int
batchNrStart : int
batchNrEnd : int
writing(path : String,graphListList : List,i : int) : void
buildFinalString(graphListList : List,i2 : int) : String
buildBatchString() : String
buildTransactionString(graph : Graph) : String
PolygoneBuilder
pointsInHull : List
getPointsInHull() : List
buildPolygonesOfSeller(batchList2 : List) : List
computePolygonList(batch : TransactionBatch) : List
cross(O : Point,A : Point,B : Point) : double
convex_hull(points : Point[]) : Point[]
Cluster
seller : List
amountList : List
buyer : Point
totalAmount : double
clusterCenter : Point
UNIQUE_ID : int
uid : int
date : String
graphList : List
<<create>> Cluster(buyer : Point,totAmount : double,actDate : String)
listContainsPoint(p : Point) : boolean
computeClusterCenter() : void
buildGraphList() : void
buildGraphSellerToCenter() : Collection
computefurtherstSellerDistance() : double
addSeller(sellerPoint : Point,amount : double) : void
addSellerList(sellerList : List,amounts : List) : void
computeTotalAmount() : double
getAlignment() : String
hashCode() : int
getTotalAmount() : double
setTotalAmount(amount : double) : void
setBuyer(buyer : Point) : void
getBuyer() : Point
getSeller() : List
getClusterCenter() : Point
getAmountList() : List
setAmountList(amountList : List) : void
getDate() : String
getGraphList() : List
setGraphList(graphList : List) : void
isBuyerInRange(buyer2 : Point,raduisCombineBuyers : double) : boolean
areAllActorsTheSame(cluster2 : Cluster) : boolean
compareSellerList(seller2 : List,seller3 : List) : boolean
HeatmapWriter
transactionCounterStart : int
transactionCounterEnd : int
batchNr : int
maxWeight : double
LOGGER : Logger
writing(path : String,batchList : List) : void
buildFinalString(batchList : List) : String
buildBatchString(date : String) : String
buildTransactionStringSeller(trans : Transaction) : String
buildTransactionStringBuyer(trans : Transaction) : String
getMaxWeight() : String
NNClustering
LOGGER : Logger
buildClusters(transList : List,knn : KNearestNeighbors) : ClusterBatch
buildEnhancedClusterList(clusterList2 : ClusterBatch) : ClusterBatch
combineClusters(cluster1 : Cluster,cluster2 : Cluster) : Cluster
RasterClustering
buildRaster(transBatchList : List,i : int) : List
buildGraphListForRaster(batch : TransactionBatch,i : int) : List
computeNewCoordInRaster(point : Point,i : int) : Point
buildRasterClusters(transList2 : List) : ClusterBatch
PolygonWriter
batchCounter : int
polygonCounterStart : int
polygonCounterEnd : int
LOGGER : Logger
newAttr : Integer
writing(path : String,polygonesOfSeller : List,pointsInHull : List) : void
buildFinalString(polygonesOfSeller : List,pointsInHull : List) : String
buildBatchString(polygonesOfSeller : List,pointsInHull : List) : String
buildAllPolygonesPerBatch(polygoneBatch : ArrayList,pointsInHullBatch : ArrayList) : String
buildSinglePolygonString(pointArr : Point[],pointsHull : Point[]) : String
buildMarkerString(point : Point,polygonCounterEnd2 : int) : String
buildPolyListString() : String

## A.3 Analyse der Clustering-Verfahren

Daten der Diagramme 4.4, 4.5, 4.6 und 4.8

Nearest-Neighbor-Clustering: Anzahl Cluster nach x Iterationen

Simulation 1

Radius 0

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 43 | 42 | 42 | 42 | 42 | 42 |
| 46 | 46 | 45 | 44 | 44 | 44 |
| 43 | 42 | 40 | 40 | 40 | 40 |
| 45 | 43 | 42 | 42 | 42 | 42 |
| 45 | 44 | 44 | 44 | 44 | 44 |

Radius 5

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 43 | 40 | 39 | 39 | 39 | 39 |
| 46 | 43 | 42 | 42 | 42 | 42 |
| 43 | 42 | 39 | 39 | 39 | 39 |
| 45 | 40 | 38 | 37 | 36 | 36 |
| 45 | 43 | 42 | 42 | 42 | 42 |

Radius 10

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 43 | 39 | 37 | 37 | 37 | 37 |
| 46 | 42 | 41 | 41 | 41 | 41 |
| 43 | 41 | 38 | 38 | 38 | 38 |
| 45 | 39 | 37 | 36 | 35 | 34 |
| 45 | 43 | 39 | 37 | 36 | 36 |

% (Anzahl Cluster)

| | Ursp. | 1. It. | 2. It. | 3. It. | 4. It. | 5.It. |
|---|---|---|---|---|---|---|
| R: 0km | 100 | 97.73 | 95.92 | 95.49 | 95.49 | 95.49 |
| R: 5km | 100 | 93.72 | 90.09 | 89.65 | 89.20 | 89.20 |
| R: 10km | 100 | 91.91 | 86.48 | 85.15 | 84.26 | 83.82 |

Simulation 2

Radius 0

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 105 | 102 | 102 | 102 | 102 | 102 |
| 116 | 112 | 112 | 112 | 112 | 112 |
| 90 | 89 | 89 | 89 | 89 | 89 |
| 95 | 95 | 95 | 95 | 95 | 95 |
| 89 | 88 | 87 | 87 | 87 | 87 |
| 108 | 107 | 107 | 107 | 107 | 107 |

Radius 5

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 105 | 96 | 96 | 96 | 96 | 96 |
| 116 | 106 | 101 | 99 | 98 | 97 |
| 90 | 87 | 83 | 79 | 78 | 78 |
| 95 | 92 | 91 | 91 | 91 | 91 |
| 89 | 85 | 82 | 80 | 80 | 80 |
| 108 | 97 | 95 | 95 | 95 | 95 |

Radius 10

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 105 | 91 | 83 | 79 | 79 | 79 |
| 116 | 97 | 86 | 82 | 81 | 80 |
| 90 | 74 | 69 | 66 | 66 | 66 |
| 95 | 77 | 73 | 71 | 70 | 68 |
| 89 | 79 | 72 | 70 | 70 | 70 |
| 108 | 84 | 81 | 80 | 80 | 80 |

% (Anzahl Cluster)

| | Ursp. | 1. It. | 2. It. | 3. It. | 4. It. | 5.It. |
|---|---|---|---|---|---|---|
| R: 0km | 100 | 98.29 | 98.06 | 98.06 | 98.06 | 98.06 |
| R: 5km | 100 | 94.36 | 91.72 | 90.04 | 89.65 | 89.47 |
| R: 10km | 100 | 84.46 | 77.51 | 74.52 | 74.14 | 73.55 |

Simulation 3

Radius 0

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 4268 | 4224 | 4184 | 4175 | 4169 | 4167 |
| 4320 | 4269 | 4247 | 4245 | 4245 | 4245 |
| 4348 | 4296 | 4263 | 4253 | 4247 | 4244 |
| 4325 | 4278 | 4243 | 4236 | 4234 | 4234 |
| 4384 | 4442 | 4301 | 4291 | 4283 | 4281 |

Radius 5

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 4268 | 2892 | 2145 | 1698 | 1420 | 1224 |
| 4320 | 2914 | 2174 | 1720 | 1433 | 1240 |
| 4348 | 2923 | 2155 | 1696 | 1393 | 1195 |
| 4325 | 2904 | 2127 | 1677 | 1387 | 1195 |
| 4384 | 2876 | 2126 | 1680 | 1377 | 1167 |

Radius 10

| Ursprung | Iteration 1 | Iteration 2 | Iteration 3 | Iteration 4 | Iteration 5 |
|---|---|---|---|---|---|
| 4268 | 2647 | 1813 | 1321 | 1004 | 788 |
| 4320 | 2610 | 1786 | 1290 | 991 | 787 |
| 4348 | 2669 | 1823 | 1324 | 1006 | 793 |
| 4325 | 2652 | 1831 | 1320 | 1018 | 824 |
| 4384 | 2651 | 1828 | 1338 | 1016 | 807 |

% (Anzahl Cluster)

| | Ursp. | 1. It. | 2. It. | 3. It. | 4. It. | 5.It. |
|---|---|---|---|---|---|---|
| R: 0km | 100 | 99.36 | 98.11 | 97.94 | 97.84 | 97.81 |
| R: 5km | 100 | 67.03 | 49.56 | 39.14 | 32.39 | 27.82 |
| R: 10km | 100 | 61.12 | 41.95 | 30.46 | 23.26 | 18.47 |

Raster-Clustering
1. Simulation

| | Ursprung | 2km | 4km | 6km | 8km |
|---|---|---|---|---|---|
| 1. Batch | 43 | 41 | 40 | 38 | 36 |
| 2. Batch | 46 | 45 | 43 | 42 | 41 |
| 3. Batch | 43 | 42 | 42 | 41 | 38 |
| 4. Batch | 45 | 44 | 44 | 42 | 40 |
| 5. Batch | 45 | 44 | 44 | 41 | 42 |
| Durchschnitt | 44.4 | 43.2 | 42.6 | 40.8 | 39.4 |
| Prozentual | 100 | 97.22 | 95.94 | 91.89 | 88.73 |

2. Simulation

| | Ursprung | 2km | 4km | 6km | 8km |
|---|---|---|---|---|---|
| 1. Batch | 105 | 100 | 98 | 93 | 81 |
| 2. Batch | 116 | 110 | 101 | 98 | 85 |
| 3. Batch | 90 | 86 | 84 | 79 | 72 |
| 4. Batch | 96 | 94 | 90 | 86 | 80 |
| 5. Batch | 89 | 85 | 83 | 79 | 74 |
| Durchschnitt | 99.2 | 95 | 91.2 | 87 | 78.4 |
| Prozentual | 100 | 95.76 | 91.93 | 87.70 | 79.03 |

3. Simulation

| | Ursprung | 2km | 4km | 6km | 8km |
|---|---|---|---|---|---|
| 1. Batch | 4268 | 867 | 352 | 217 | 146 |
| 2. Batch | 4320 | 882 | 369 | 222 | 453 |
| 3. Batch | 4348 | 890 | 371 | 220 | 148 |
| 4. Batch | 4325 | 884 | 362 | 214 | 146 |
| 5. Batch | 4384 | 872 | 352 | 210 | 145 |
| Durchschnitt | 4329 | 879 | 361.2 | 216.6 | 207.6 |
| Prozentual | 100 | 20.30 | 8.34 | 5.00 | 4.79 |

% (Anzahl Cluster)

| | Ursprung | 2km | 4km | 6km | 8km |
|---|---|---|---|---|---|
| Simulation 1 | 100 | 97.22 | 95.94 | 91.89 | 88.73 |
| Simulation 2 | 100 | 95.76 | 91.93 | 87.70 | 79.03 |
| Simulation 3 | 100 | 20.30 | 8.34 | 5.00 | 4.79 |

# Bildverzeichnis