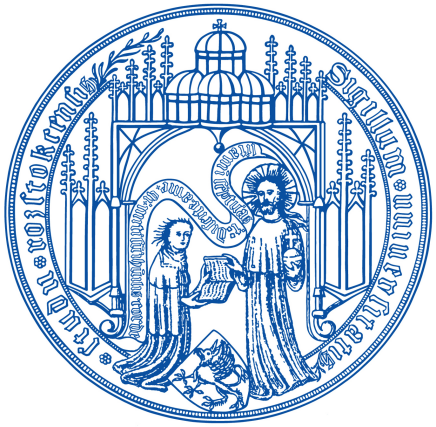
Automatische Generierung von Domänengeneralisierungshierarchien

Bachelorarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Richard Dabels
Matrikelnummer: 214206647
geboren am: 21. Juli 1995 in Ribnitz-Damgarten
Erstgutachter: Prof. Dr. rer. nat. habil. Andreas Heuer
Zweitgutachter: Dr.-Ing. Holger Meyer
Betreuer: Hannes Grunert
Abgabedatum: 7. September 2017

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Vorwort

In unser auf Kommunikationstechnologien basierenden Gesellschaft gehört es zur gängigen Praxis, Informationen aller Art über Personen zu sammeln und diese auszuwerten. Staatliche Einrichtungen sowie private Unternehmen verfügen über große Datenbestände, die heutzutage auch mit Data-Mining-Verfahren zur automatisierten Wissensgewinnung genutzt werden können. Häufig stehen diese Praktiken jedoch im Widerspruch zu dem natürlichen Bedürfnis der Betroffenen, selbst über ihre Daten entscheiden zu können. Anonymisierungstechniken zur Erhaltung der Privatsphäre sind mit Hilfe von Maßen wie der k-Anonymität, l-Diversity und t-Closeness bereits dazu in der Lage, qualitative Aussagen über die Identifizierbarkeit von Individuen in Datensätzen zu treffen und diese so zu generalisieren, dass die Identifizierung gar nicht mehr oder nur noch erschwert möglich ist. Teilaspekte dieser Techniken bedürfen jedoch weiterer Entwicklung, um sich besser auf dynamische Informationsbestände anpassen zu können.

Um die genannten Anonymisierungstechniken anzuwenden, müssen Informationen über die Generalisierung einzelner Attribute in einem Datensatz vorliegen. Diese Informationen lassen sich durch Ontologien darstellen und werden, meist per Hand, als Domänengeneralisierungshierarchien in bestehende Systeme implementiert [Mül16]. Ein Problem, welches bei der manuellen Speicherung dieser Hierarchien auftreten kann, ist, dass sie bei dynamischen Datensätzen dazu neigen, keine optimale Anonymisierung zu ermöglichen. In diesem Fall werden Attribute bis zur Unbrauchbarkeit anonymisiert, da die verwendeten Ontologien für die neue Verwendung keine passenden Generalisierungen mehr zur Verfügung stellt. Die generalisierten Datensätze sind dann für deren eigentlich angedachte Verwendung oft bedeutungslos. Dies würde sich in den meisten Fällen jedoch vermeiden lassen, wenn die für die Anonymisierung genutzten Ontologien weiter an den Verwendungszweck angepasst werden würden, sodass eine feingranulare Generalisierung möglich ist.

Diese Arbeit beschäftigt sich mit der automatischen Erzeugung von Domänengeneralisierungshierarchien, um eine Grundlage für optimale Generalisierungen zu schaffen. Dabei werden bestehende Lösungsansätze zur Anonymisierung von Datensätzen untersucht, sowie Clustering-Verfahren und Ontologien angewendet, um optimale Generalisierungsbäume zu konstruieren. Als Quelle für Ontologien sollen dabei domänenspezifische Repositories herangezogen werden.

1.2 Fälle des Datenmissbrauches

Moderne Informationsverarbeitungssysteme, mit denen sich Daten schneller und effizienter verarbeiten lassen, erleichtern dem Menschen den Alltag enorm. Die Gefahren, welche damit einhergehen, sind den meisten Nutzern dagegen jedoch oft nicht bewusst. Dies führt zu teilweise fahrlässigen Umgang mit Informationen. Betroffene sind sich dabei meist überhaupt nicht darüber im Klaren, wie mit ihren Daten gearbeitet wird.

Als Beispiel lässt sich der Schutz – oder die Verletzung – der Privatsphäre in Krankenhäusern heranziehen. Hier werden, verglichen mit anderen Organisationen, unverhältnismäßig viele private Informationen über Patienten gesammelt. Die meisten dieser haben sehr hohe Erwartungen an die Wahrung ihrer Privatsphäre; nur Fachkräfte, welche sich direkt mit der Behandlung einer Person beschäftigen, sollen Zugriff auf intime Daten haben. In der Praxis ist dies jedoch oft nicht der Fall. Bereits innerhalb eines Krankenhauses ist es keine Seltenheit, dass Angestellte, welche nicht mit den Patienten arbeiten, Zugriff auf deren ganze Datensätze haben. Des Weiteren schließen sich Krankenhäuser zu Netzwerken zusammen, um ihr Gesundheitssystem untereinander besser zu koordinieren. Dies bedeutet auch, dass darin involvierten Organisationen der Zugriff auf persönliche Details von Patienten erlaubt wird. Zusätzlich haben auch Apotheken, Versicherungen und Beratungsunternehmen teilweisen Zugriff [Swe97]. Mehrfacher Datenmissbrauch ist Folge dieser Praxis, wie Latanya Sweeney vom Massachusetts Institute of Technology an mehreren Beispielen zeigt:

Die *Fortune* ist eine US-amerikanische Zeitschrift, welche jährlich die 500 erfolgreichsten Unternehmen der Welt in einer Liste, die „Fortune 500“, zusammenfasst. Was ein Teil dieser Unternehmen jedoch nicht ihrem Glück überlässt, ist die Wahl ihrer Angestellten. 1995 nahmen 87 der Fortune 500 an einer Befragungen teil bei der sich herausstellte, dass 35% dieser Unternehmen Entscheidungen über ihre Angestellten anhand medizinischer Unterlagen treffen [Swe97].
1996 berichtete die National Association of Health Data Organizations (NAHDO), dass 37 US-amerikanische Staaten Mandate erteilt hätten, die das Sammeln von Krankenhausdaten ermöglichen. Nachdem Anonymisierungen erfolgt sind, wurden diese Informationen für wissenschaftliche Zwecke weiterverwendet oder verkauft. Die Art der Anonymisierung war hier jedoch das Problem: Daten wurden als anonym angenommen, nachdem Individuen nicht mehr direkt identifizierbar waren. Die Entfernung direkter Identifikatoren (z.B. ausweisende IDs) war, und ist allgemein, aber nicht ausreichend um Anonymität zu gewährleisten. Hierfür reicht in der Regel bereits das Vorhandensein sogenannter *Quasi-Identifikatoren.*
In Zeiten der Not ist die Aufnahme von Krediten bei Banken nicht unüblich. In Zeiten der Krankheit ist die Aufnahme von Krediten wohlmöglich sogar notwendig. Wenn die Interessen der Bank im Kontrast zu denen der Patienten stehen, dann ist Diskretion nötiger denn je. Trotzdem ist es 1995 einem Banker in Maryland durch den Abgleich einer Liste von Krebspatienten mit Daten seiner eigenen Kunden möglich gewesen, Individuen anhand außergewöhnlich hoher Kredite zu identifizieren. Dies führte in diesem Fall dazu, dass Kredite der Betroffenen zurückverlangt wurden [Swe97].

Als letztes Beispiel für mögliche De-Anonymisierung soll eine Wahlliste von 1997 dienen. Diese Wahl fand in Cambridge, Massachusetts statt und umfasste eine Gesamtzahl von 54805 Wählern. Attribute, welche zur direkten Identifikation von Teilnehmern führt, werden im Folgenden nicht beachtet.
Alleine durch Angabe des Geburtstags sind 12% der Wähler direkt identifizierbar. Wird dieser Information noch das Geschlecht hinzugefügt, reicht dies bereits für 29%. Werden statt dem Geschlecht der ZIP-Code zur Auswertung herangezogen, ist es bereits möglich, 69% der Wählerschaft zu identifizieren und bei Beachtung der Adresse und des Geburtstags beachtliche 97% [Swe97].

Diese Beispiele zeigen, wie kritisch der Umgang mit sensiblen und komplexen Daten ist. Anonymisierung von Daten ist prinzipiell kein intuitiver Prozess und es fällt Verantwortlichen oft schwer Anonymität

festzustellen. Aus diesem Grund sind Formalismen erforderlich, die Datensätze für den Menschen beurteilen. In Form der k-Anonymität [Sam01], l-Diversity [MKGV07] und t-Closeness [LLV07] sind bereits solide Grundlagen erarbeitet, die in vielen der oben genannten Fälle höchstwahrscheinlich hilfreich gewesen wären. Diese drei Techniken ermöglichen es, qualitative Aussagen über die Anonymität von Datenbeständen zu treffen. Das in dieser Arbeit vorgestellte System soll dabei Hand in Hand mit den genannten Verfahren arbeiten, um die Ergebnisse von Anonymisierungen weiter zu verbessern.

1.3 Privatsphäre und Anonymität im Recht

Da in profitorientierten Unternehmen aus wirtschaftlichen Gründen die Privatsphäre von Personen nicht immer die höchste Priorität erhält, müssen gesetzliche Maßnahmen zu deren Schutz getroffen werden. Diese können für verschiedene Länder, Staaten oder einen Staatenverbund festgelegt werden.

In Deutschland wird das Recht auf Privatsphäre unter anderem aus Artikel 1 des Grundgesetzes abgeleitet – *„Die Würde des Menschen ist unantastbar. […]“* [Deu17]. Mit dieser Grundlage ist das Recht auf informelle Selbstbestimmung herleitbar.
Es existieren auch konkretere Bestimmungen, welche besonders für die Anonymisierung und Generalisierung als Leitfaden dienen. Diese sind für Deutschland in §3a des Bundesdatenschutzgesetzes festgelegt:

> *Die Erhebung, Verarbeitung und Nutzung personenbezogener Daten und die Auswahl und Gestaltung von Datenverarbeitungssystemen sind an dem Ziel auszurichten, so wenig personenbezogene Daten wie möglich zu erheben, zu verarbeiten oder zu nutzen. Insbesondere sind personenbezogene Daten zu anonymisieren oder zu pseudonymisieren, soweit dies nach dem Verwendungszweck möglich ist und keinen im Verhältnis zu dem angestrebten Schutzzweck unverhältnismäßigen Aufwand erfordert* [Bun15].

Auch europaweit existieren Gesetze, welche sich auf den Datenschutz der Bürger beziehen. Als Beispiel hierfür soll Artikel 8, Absatz 1 der EU-Grundrechtcharta dienen [Eur00]. Hier wird festgelegt: *„Jede Person hat das Recht auf Schutz der sie betreffenden personenbezogenen Daten.“*

Überlegungen zur Wahrung der Privatsphäre sind jedoch nicht nur ein Phänomen der digitalen Gesellschaft. Einer der ersten Artikel bezüglich des Datenschutzes ist „The Right to Privacy“ [BW90], welcher von Samuel Warren und Louis Brandeis 1890 veröffentlicht wurde. Das formulierte „right to be let alone“ besaß bereits vor über 100 Jahren Relevanz, da durch die Entwicklung der Fotografie, und dessen Missbrauch, neue Herausforderungen in der Rechtsprechung entstanden sind. Ein Recht auf Privatsphäre wurde als natürlicher nächster Schritt in der Evolution des Rechtswesens gesehen, da das Potential unerwünschter Konsequenzen für Betroffene bei unfreiwilligen Veröffentlichungen, unter anderem in Zeitschriften, erkannt wurde.

Mit den in dieser Arbeit vorgestellten Verfahren soll es letztendlich ermöglicht werden, optimale Anonymisierung von Datenbeständen zu erzeugen. Prinzipiell impliziert dies auch, dass die erzeugten Datensätze gesetzliche Vorgaben erfüllen. Es wird jedoch hiermit darauf hingewiesen, dass das entstandene Produkt nicht im direkten Hinblick auf die Gültigkeit im Gesetz entwickelt wird und somit auch keine Garantien in Bezug auf dieses gegeben werden.

1.4 Aufbau

Ziel dieser Arbeit ist es, einen eigenen Ansatz zur automatischen Generierung von Domänengeneralisierungshierarchien zu finden. Hierzu sollen verschiedene Techniken und Methoden, wie Ontologien und

Clustering, verwendet werden. Um eine Übersicht über die bereits in der Wissenschaft untersuchten Verfahren zu gewinnen, werden in Kapitel 2 Datenschutz-Systeme und -Maße betrachtet, erläutert, sowie deren Vor- und Nachteile abgewogen. Auch wird das in der Praxis weniger besprochene hierarchische Clustering als unter Umständen für dieses Projekt nutzbares Verfahren ausführlich erklärt.

Kapitel 3 wird sich mit in der Praxis verwendeten und bewährten Systemen beschäftigen. Es werden Nutzen und Probleme derzeit genutzter Domänengeneralisierungshierarchien veranschaulicht, um den Vorteil eines automatischen Verfahrens zu verdeutlichen. Für dessen Umsetzung werden Ontologien erklärt, wie sie derzeit im Semantic Web Verwendung finden.

In der Erklärung des Konzeptes in Kapitel 4 werden die Anforderungen an das hier zu erarbeitende System noch einmal aufgezeigt. Sich daraus ableitende Softwarekonzepte und die damit verbundene Zielfindung sind in diesem Kapitel auch dargestellt. Zusätzlich wird entschieden und begründet, welche Verfahren in der Software Anwendung finden und wie die Implementierung in das bestehende PArADISE-Projekt [1] der Universität Rostock erfolgt.

Die auf dem Konzept aufbauenden Implementierungen sind im darauf folgenden Kapitel 5 erläutert. Auf diese Arbeit folgende, mögliche Weiterentwicklungen des Projektes finden im letzten Kapitel 6 Ansprache.

[1] Privacy AwaRe Assitive Distributed Information System Environment

Kapitel 2

Stand der Forschung

Dieses Kapitel behandelt Konzepte, welche entweder keine breite Anwendung in aktuellen Systemen finden, oder aufgrund fortlaufender Forschung hauptsächlich in wissenschaftlichen Kreisen bekannt sind. Ziel ist das Finden neuer Technologien, welche für die durch diese Arbeit gegebene Problemstellung hilfreich seien könnten sowie die Erkenntnisgewinnung aus bereits erarbeiteten Ansätzen, welche sich aus verschiedenen Gründen in der Praxis nicht durchsetzen konnten.

2.1 Datafly

In Folge der in Kapitel 1 genannten Verstöße in medizinischen Bereichen wurde 1997 die Software „Datafly“ von Latanya Sweeney entwickelt [Swe97]. Dies ist ein Programm, welches zwischen dem Nutzer und einem Datenbank-Server alle gestellten Anfragen verarbeitet. Anonymität wird gewährleistet, wenn sich in einer Tabelle ein Individuum nicht von k anderen Individuen unterscheidet. Hierzu werden Spalten entweder, im Fall von numerischen Werten, generalisiert, oder ganz gelöscht. Zu sehen ist diese Arbeitsweise in Abbildung 2.1.

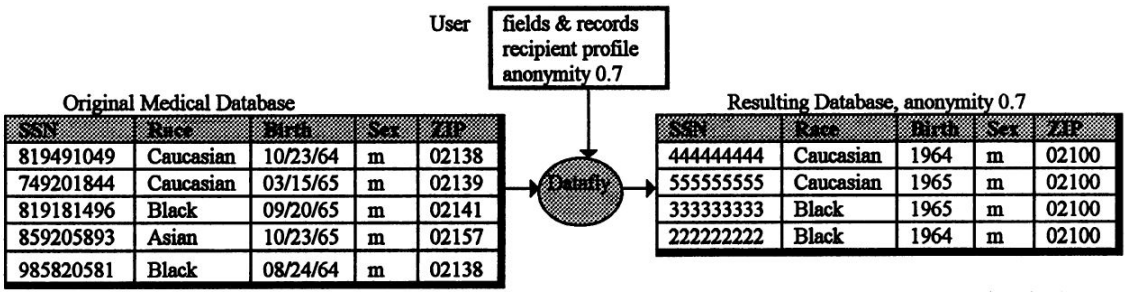


Abbildung 2.1: Funktionsweise von Datafly nach [Swe97]

In diesem Beispiel wird ein Tupel ganz und gar aus der Relation gelöscht. Dieses Vorgehen ist in Datafly notwendig, da keine Generalisierungshierarchien für nicht-numerische Werte verwendet werden. Eine Generalisierung ist damit nicht möglich und das ganze Tupel muss gelöscht werden, da sich sonst kein anonymisierter Datensatz erzeugen lässt. Aus diesem Grund werden in Datafly Datensätze oft zu sehr generalisiert. Es fällt dem Verfahren auch schwer, mit „Ausreißern“ umzugehen. Wenn diese Elemente Attribute besitzen, welche weder numerisch sind, noch häufig genug vorkommen um Anonymität zu gewährleisten, resultiert das in der Löschung derjenigen Information. Allgemein fehlt es an formalen

Grundlagen zur Anonymisierung. Die zu aggressive Generalisierung ist auch Kritikpunkt in der Arbeit von Pierangela Samarati, welche 2001 in Zusammenarbeit mit Sweeney das Konzept der k-Anonymität entwickelt hat [Sam01].

2.2 Anonymitätsmaße

Die Anonymität eines Datensatzes kann anhand verschiedener Methoden bestimmt werden. Dabei hat sich herausgestellt, dass kein einzelnes der hier vorgestellten Verfahren Datenschutz vollständig garantieren kann [MKGV07, LLV07]. Aus diesem Grund ist es ratsam, die verschiedenen Techniken miteinander zu kombinieren.

In der Praxis reicht oft unvollständiges Wissen aus, um Tupel in Relationen ganz oder fast zu identifizieren. Dieses unvollständige Wissen ist jedoch für Unternehmen meist gut genug, um wichtige Rückschlüsse zu ziehen, wie die Fälle in Kapitel 1 zeigen. Kandidaten für De-Anonymisierungsangriffe sind dabei diejenigen Attribute, welche aus verschiedenen Datensätzen kreuzreferenziert werden können. Manche Daten über Alter, Geschlecht oder Familienstand von Personen können dabei auch aus öffentlichen Quellen frei einsehbar sein [Dal86]. Es kann dabei jedoch vorkommen, dass durch die Kombination verschiedener nicht-identifizierender Attribute die Identifizierung von Individuen trotzdem ermöglicht wird. Diese Attribute werden als *Quasi-Identifikatoren* – eine Menge von Eigenschaften, welche ein Tupel fast eindeutig identifiziert [Dal86] – bezeichnet.

Um diese De-Anonymisierungsangriffe [Gol17] zu vermeiden, werden die folgend erklärten Verfahren eingesetzt. Diese arbeiten jeweils mit den eben definierten Quasi-Identifikatoren, deren Anonymisierung zusammen mit der der sensitiven Attribute die Anonymität der Relation sicherstellen. Für jede der Techniken soll die Wirkung mit Tabelle 2.1 anschaulich erklärt werden. In dieser sind die Quasi-Identifikatoren die Attribute *Studiengang*, *Geburtsdatum*, *PLZ* und *Ort*. Das *Geburtsdatum* beispielsweise reicht alleine bereits zur eindeutigen Identifizierung eines Tupels. Die Kombination der Attribute *Studiengang* und *PLZ*, beziehungsweise *Ort*, ist auch aussagekräftig genug, um die meisten der Tupel zu identifizieren. Die *Matrikelnummer* ist zwar der Schlüssel der Tabelle, wird aber für dieses Beispiel nicht zu den Quasi-Identifikatoren gezählt. Als sensitives Attribut ist hier die *Note* gewählt. Diese soll durch De-Anonymisierung keinem Individuum mehr zugewiesen werden können.

Matrikelnummer	Studiengang	Geburtsdatum	PLZ	Ort	Note
209202223	MA Informatik	22.06.1990	18057	Rostock	3,2
209204742	MA Chemie	14.04.1989	18209	Bad Doberan	1,7
212203190	BA Informatik	28.10.1995	18069	Rostock	1,9
212205002	BA Informatik	26.03.1993	18057	Rostock	3,7
212205999	MA Chemie	08.06.1991	18202	Bad Doberan	2,0
214200123	MA Informatik	01.01.1990	18057	Rostock	2,3
214201234	MA Informatik	06.07.1992	18059	Rostock	1,9
214202356	BA Physik	07.10.1995	18107	Rostock	3,0
214202644	MA Informatik	01.02.1993	18069	Rostock	3,2
214203141	BA Physik	17.07.1993	18147	Rostock	2,9
215202718	BA Chemie	22.02.1992	18209	Bad Doberan	2,5
215209159	MA Physik	25.12.1993	18119	Rostock	3,0

Tabelle 2.1: Studenten, QI= {{Geburtsdatum}, {Studiengang, Ort}, {Studiengang, PLZ}

k-Anonymität

Die k-Anonymität ist ein Anonymitätsmaß, welches sicherstellen soll, dass durch Kombination von Attributen keine Tupel in einer Relation eindeutig identifizierbar sind. In Tabelle 2.1 lässt sich zum Beispiel beobachten, dass das Attribut *Geburtsdatum* alle Tupel eindeutig identifiziert. Weiterhin lassen sich Tupel auch durch die Kombination von *Studiengang* und *Ort* eindeutig identifizieren. So existiert beispielsweise nur ein Tupel mit den Werten *Studiengang* = *„BA Chemie“* und *Ort* = *„Bad Doberan“*. Dies würde bereits die k-Anonymität für alle $k > 1$ verletzen.

Der Begriff „k-Anonymität“ ist folgendermaßen definiert: Ein Datensatz besitzt k-Anonymität, wenn für jede Kombination von Quasi-Identifikatoren mindestens k Einträge in einer Relation vorhanden sind [Sam01]. Daraus lässt sich schließen, dass Tabelle 2.1 nur anonym für $k = 1$ ist.

Um eine höhere k-Anonymität zu erreichen gilt es, die Quasi-Identifikatoren zu generalisieren. Die Generalisierungen können wie folgt angewendet werden: Das Attribut *Studiengang* wird um den Titel, sprich *MA* oder *BA*, gekürzt, sodass nur die Disziplin übrig bleibt. *Geburtsdatum* wird generalisiert, sodass nur 3-Jahre-Intervalle angezeigt werden, zum Beispiel wird *22.06.1990* auf *1989 - 1992* umgeschrieben. In der *PLZ* werden die letzten zwei Ziffern entfernt und in *Ort* müssen keine Generalisierungen vorgenommen werden. Das Ergebnis ist Tabelle 2.2, mit einer k-Anonymität von $k = 3$, in welcher durch Kombination von Attributen Tupel nur Äquivalenzklassen vorkommen, welche eine Größe von mindestens 3 haben. Die vorkommenden Äquivalenzklassen sind in der Tabelle farbig gekennzeichnet.

Studiengang	Geburtsdatum	PLZ	Ort	Note
Informatik	1989 - 1992	180xx	Rostock	3,2
Chemie	1989 - 1992	182xx	Bad Doberan	1,2
Informatik	1993 - 1996	180xx	Rostock	2,3
Informatik	1993 - 1996	180xx	Rostock	1,9
Chemie	1989 - 1992	182xx	Bad Doberan	2,5
Informatik	1989 - 1992	180xx	Rostock	2,3
Informatik	1989 - 1992	180xx	Rostock	1,9
Physik	1993 - 1996	181xx	Rostock	3,0
Informatik	1993 - 1996	180xx	Rostock	3,2
Physik	1993 - 1996	181xx	Rostock	2,9
Chemie	1989 - 1992	182xx	Bad Doberan	3,7
Physik	1993 - 1996	181xx	Rostock	3,0

Tabelle 2.2: Studenten anonymisiert, k-Anonymität mit $k = 3$

l-Diversity

Obwohl die k-Anonymität bereits ein gewisses Maß an Schutz bietet, lassen sich Attacken auf Relationen durchführen, welche nur in Hinsicht auf diese anonymisiert wurden. Zwei bekannte Probleme sind die folgenden: Wenn in den anonymisierten Datensätzen wenig Vielfalt in den sensitiven Attributen existiert, dann kann dies dazu führen, dass diese weiterhin wenig oder gar nicht geschützt sind [MKGV07]. Zu beobachten ist dies auch in der anonymisierten Tabelle 2.2. Soll zum Beispiel die Note eines Physikstudenten ermittelt werden, so sind trotz k-Anonymität von $k = 3$ nur zwei verschiedene Werte unter den sensitiven Attributen. In diesem Fall kommt der Wert 2,9 ein einziges Mal, und der Wert 3,0 sogar zweimal vor. Mit dieser Erkenntnis kann ein Angreifer sich bereits ziemlich sicher über die Leistung des gesuchten Studenten sein.

Das zweite Problem, welches sich im Hinblick auf eine k-anonymisierte Relation ergibt ist, dass Hintergrundwissen eines Angreifers nicht beachtet wird [MKGV07]. Auch dies lässt sich in Tabelle 2.2 zeigen: Ist dem Angreifer beispielsweise ein Chemiestudent bekannt, welcher zum Lernen außergewöhnlich viel Zeit in der Bibliothek verbringt, so kann die Annahme getroffen treffen, dass sein Eintrag in der Tabelle nicht der mit der Note 3,7 oder 2,5 ist, sondern wahrscheinlich die 1,2 zum gesuchten Eintrag gehört.
Durch eine geringe Vielfalt der sensitiven Attribute kann es einem Angreifer in beiden Fällen gelingen, zusätzliche Informationen aus einer anonymisierten Tabelle zu gewinnen. Um diesem Problem entgegenzuwirken, wurde das Konzept der l-Diversity entwickelt.

Der Begriff „l-Diversity“ ist folgendermaßen definiert: Eine Tabelle mit k-Anonymität besitzt l-Diversity, wenn jedes sensitive Attribut in den durch Kombination der Quasi-Identifikatoren entstandenen Äquivalenzklassen mindestens l Mal vertreten ist [MKGV07].
Nach dieser Definition hat die k-anonymisierte Studenten-Tabelle 2.2 nur eine l-Diversity von $l = 1$, da die sensitiven Werte nicht in allen vorhandenen Äquivalenzklassen mehr als ein Mal vertreten sind. Werden jedoch nur die Tupel mit dem *Studiengang* Informatik betrachtet, so wird bereits eine l-Diversity von $l = 2$ erreicht. Dies ist in Tabelle 2.3 zu sehen. Hier existieren für alle vorkommenden sensitiven Werte mindestens zwei Einträge.

Studiengang	Geburtsdatum	PLZ	Ort	Note
Informatik	1989 - 1992	180xx	Rostock	3,2
Informatik	1993 - 1996	180xx	Rostock	2,3
Informatik	1993 - 1996	180xx	Rostock	1,9
Informatik	1989 - 1992	180xx	Rostock	2,3
Informatik	1989 - 1992	180xx	Rostock	1,9
Informatik	1993 - 1996	180xx	Rostock	3,2

Tabelle 2.3: Studenten anonymisiert, $k = 3$, $l = 2$

t-Closeness

Auch nach guter Einschätzung der Anonymität durch k-Anonymität und l-Diversity existieren weitere Möglichkeiten, um mehr Informationen aus einem Datenbestand zu extrahieren, als durch die Anonymisierung erreicht werden sollte. Weicht die Verteilung der sensitiven Attributwerte einer Äquivalenzklasse beispielsweise zu sehr von der Verteilung aller sensitiven Attributwerte ab, können durch Vergleich damit weitere Informationen aus dieser Äquivalenzklasse abgeleitet werden. Mit t-Closeness ist es möglich, diesen Umstand zu erkennen, und bei entsprechender Wahl von t kann ein Angreifer keine zusätzlichen Informationen über ein Individuum erfahren, selbst wenn er weiß, in welcher Äquivalenzklasse es sich befindet [LLV07].

Der Begriff „t-Closeness“ ist folgendermaßen definiert: Eine Äquivalenzklasse besitzt t-Closeness, wenn die Distanz zwischen den zu veröffentlichenden sensitiven Attributen dieser Klasse zur gesamten Tabelle nicht mehr als ein Grenzwert von t beträgt. Eine Tabelle besitzt t-Closeness, wenn alle ihre Äquivalenzklasse t-Closeness besitzen [Hau07].

In der hier als Beispiel dienenden Tabelle 2.2 kann auch dieses Problem gezeigt werden. Dafür ist die Verteilung der Noten in dem gesamten Datensatz zu betrachten, in welchem die Werte von 1,2 bis 3,7 verhältnismäßig gut repräsentiert sind. Vergleichen wir diese Verteilung mit der der Äquivalenzklasse, die in der Tabelle rot gekennzeichnet ist – diejenige, welche die Physiker beinhaltet – so kann erkannt werden,

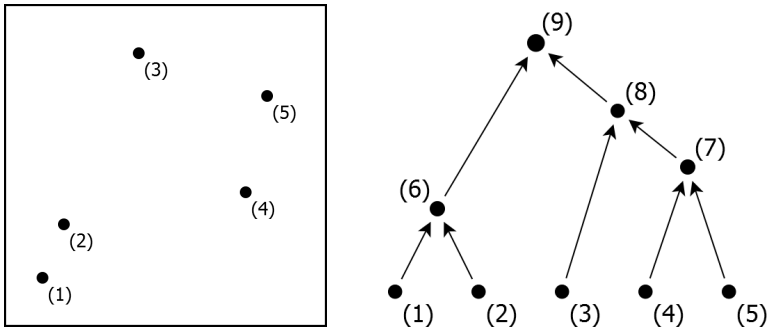
dass diese ihre sensitiven Werte im Vergleich zum gesamten Datensatz viel unausgewogener repräsentiert. Mit Hilfe der t-Closeness kann dieser Umstand erkannt und dementsprechend auch verhindert werden.

2.3 Hierarchisches Clustering

Unter dem Begriff „Clustering“ werden Techniken verstanden, welche eine Menge von Elementen aufgrund ihrer Eigenschaften in Gruppen einteilen. Diese finden insbesondere im Bereich des Data-Mining Anwendung [CL14]. Nach der Erläuterung des hierarchischen Clusterings wird das Thema des Clusterings in Absatz 3.3 von Kapitel 3 erneut aufgegriffen und auch auf einen der beliebtesten Clustering-Algorithmen, k-Means, eingegangen.

Bei dieser Art des Clusterings wird aus einer Menge von Objekten eine Hierarchie von Clustern aufgebaut. In jedem Schritt werden dabei die beiden Cluster mit der kleinsten Distanz voneinander verschmolzen und damit eine Baumstruktur von Clustern erzeugt. In dieser ist die Wurzel des Baumes der gesamte Datensatz, die Knoten sind alle Cluster die während des Prozesses erzeugt worden sind und die Blätter sind einzelne Objekte aus der Ausgangsmenge [CL14]. Es wird bei hierarchischen Clusterverfahren zwischen zwei verschiedenen Arten unterschieden: Die *agglomerative Clusterbildung* beginnt mit der Verschmelzung einzelner Objekte, oder möglichst kleiner Cluster, zu größeren Clustern und führt diesen Prozess weiter bis nur noch ein großer Cluster vorhanden ist. Im Gegenzug dazu unterteilt *divisive Clusterbildung* den gesamten Datensatz iterativ in immer kleiner werdende Cluster, bis nur noch einzelne Objekte übrig sind [CL14].

Bei *agglomerativer Clusterbildung* können für die Distanzberechnung verschiedene Techniken verwendet werden. *Single-Linkage* definiert die Distanz zwischen zwei Clustern als die Distanz zwischen den zwei sich am nahestehendsten Objekten beider Clustern. *Complete-Linkage* verhält sich ähnlich, jedoch wird hier die Distanz der sich am weitesten voneinander entfernten Objekte als Maß genommen und *Average-Linkage* nutzt den durchschnittlichen Abstand von Objekten aus beiden Clustern. Die Distanz kann jedoch auch, ähnlich wie bei k-Means, durch die *Centroide* der Cluster berechnet werden [CL14]. In Abbildung 2.2 ist die *agglomerative Clusterbildung* mit *Single-Linkage* beispielhaft aufgezeigt. Hier sollen aus den Punkten in (a) Cluster gebildet werden.



(a) Ungeclusterte Punktmenge (b) Geclusterte Punkte

Abbildung 2.2: Hierarchisches Clustering

Wie bereits angedeutet, werden Punkte als Cluster angesehen und in jedem Schritt die sich am nahestehendsten Cluster verschmolzen. Alle Schritte und die daraus resultierenden Cluster sind in Abbildung 2.2 (b) aufgezeigt: Als erstes werden die Punkte (1) und (2) verschmolzen. Der nächste Schritt wird

zur Folge haben, dass das eben gebildete Cluster (6) außer Acht gelassen wird, und die Punkte (4) und (5) zu (7) verschmolzen werden. Dieses Cluster wird danach um (3) zu (8) erweitert und verschmilzt schlussendlich mit dem zuerst erzeugten Cluster (6) zu (9). Im Gegensatz zu anderen Clusterverfahren wie zum Beispiel k-Means, erlaubt das hier verwendete Verfahren, eine beliebige Clusterzahl aus der Clusterhierarchie auszuwählen: Sollen drei Cluster verwendet werden, dann kann in dieser der Zustand von vor drei Schritten verwendet werden. Dies sind in diesem Beispiel die Cluster (3), (6) und (7). Sollen nur zwei Cluster aus dem Verfahren gewonnen werden, wird der vorletzte Zustand verwendet: Die Cluster (6) und (8).

Hierarchische Clusteringverfahren eignen sich dazu, Informationen über die Verbindungen zwischen den Clustern zu erhalten. Auch muss keine gewünschte Clusteranzahl vorgegeben werden, wenn auch sie durch die nötige paarweise Distanzbildungen, welche in jedem Schritt nötig sind, nur schlecht auf großen Datenmengen skalieren [CL14].

2.4 Fazit

Datafly, in Absatz 2.1 erklärt, bot ein Grundgerüst für spätere Verfahren, welche die Anonymität von Personen in Datensätzen einschätzen. Da sich aus diesem System heraus das Konzept der k-Anonymität entwickelte [Sam01], ist es für die Praxis nur noch eingeschränkt relevant. Deshalb wird im Folgenden auch nur noch auf die k-Anonymität Rücksicht genommen.

Die in Absatz 2.2 behandelten Anonymitätsmaße bieten qualitative Bewertungen der Anonymität von Datensätzen. Diese zählen derzeit zu den prominentesten Ansätzen ihres Bereiches und sind im PArADISE-Projekt der Universität Rostock [GH16] bereits implementiert. Insbesondere sind sie für diese Arbeit relevant, da die Domänengeneralisierungshierarchien, welche durch die in dieser Arbeit implementierte Software automatisch generiert werden, im PArADISE-Projekt indirekt zusammen mit den Anonymitätsmaßen arbeiten.

Das hierarchische Clustering aus Absatz 2.3 ist ein einfaches Verfahren und ein starker Kandidat für die Implementierung in dieser Arbeit. Besondere Vorteile sind unter Anderem, dass die Clusterung in einer Baumstruktur entsteht und keine Clustergröße vorgegeben seien muss.

Kapitel 3

Stand der Technik

Verfahren und Techniken, welche in der Praxis bereits Anwendung finden und für die automatische Generierung von Domänengeneralisierungshierarchien unter Umständen von Nutzen seien könnten, finden in diesem Kapitel Erwähnung. Insbesondere eine der wichtigsten Grundlagen, die Domänengeneralisierungshierarchie, wird hier erklärt. Zusätzlich wird das Konzept der Ontologien erläutert, welche für die semantische Wissensgewinnung von großer Bedeutung sind.

3.1 Domänengeneralisierungshierarchien

Anonymität kann mit Hilfe von k-Anonymität, l-Diversity und t-Closeness überprüft werden. Für die Anonymisierung von Datensätzen existieren verschiedene Ansätze. Verfahren wie Datafly sind dazu in der Lage, spezielle Datensätze durch Generalisierung von numerischen, und Unterdrückung von nicht-generalisierbaren Werten anonym zu machen. Diese Unterdrückung, praktisch die Löschung eines Tupels, ist jedoch ein sehr grobes Vorgehen und führt zu hohem Informationsverlust. Domänengeneralisierungshierarchien sind Strukturen, die eine schrittweise Generalisierung auch für nicht-numerische Attribute ermöglichen.

Auch die Tabelle 2.2 wurde mit Hilfe von Domänengeneralisierungshierarchien generalisiert. Diese hat alle Einträge, welche ein Bachelor-, beziehungsweise Master-Kürzel vor dem Fach angegeben haben so anonymisiert, dass nur noch das Fach alleine angegeben wurde. In Abbildung 3.1 ist die in der Studenten-Tabelle vorgenommene sowie eine weitere Generalisierung beispielhaft in einem Generalisierungsbaum gezeigt.

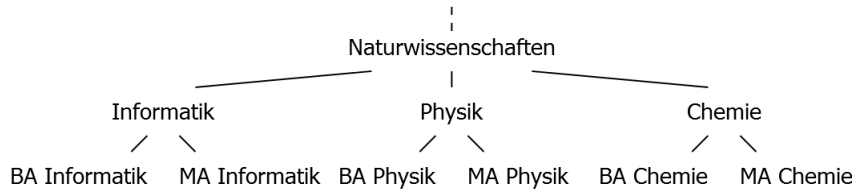


Abbildung 3.1: Generalisierung des Attributes *Studiengang* aus Tabelle 2.2

3.2 Ontologien

Da sie in der Praxis meist per Hand erzeugt werden [Mül16], sind Domänengeneralisierungshierarchien meist sehr starre Strukturen, die sich nicht auf Veränderungen der zu Grunde liegenden Daten anpassen. Ein automatisches Verfahren zur Erzeugung von diesen Hierarchien soll in dieser Arbeit ermittelt werden. Hierzu müssen jedoch prinzipiell zusätzliche Informationen über die in Datensätze vorhandenen Informationen vorliegen, da sonst keine Basis zur Generalisierung existiert. Für die Lösung dieses Problems sollen Ontologien genutzt werden. Diese basieren auf Beschreibungslogiken und können die Semantik von Objekten sowie deren Beziehungen untereinander modellieren [Fen03].

Ontologien werden in ihrer einfachsten Form aus Relationen und Axiomen aufgebaut [Vos03]. Dabei ist es – im Gegensatz zu den aus Programmiersprachen oder der Softwaretechnik bekannten objektorientierten Formalismen – auch möglich, Klassen mit Unterklassen zu definieren, welche sich gegenseitig überlappen [DHI12]. Dies hat auch zur Folge, dass aus Ontologien generierten Domänengeneralisierungshierarchien nicht immer die gleiche Struktur haben. Die Generalisierungshierarchie in Abbildung 3.1 beispielsweise verfügt über mehrere Klassen, wobei Unterklassen hier strikt getrennt sind. Stehen die Konzepte *BA Informatik* und *MA Informatik* in einer „gehört-zu“-Relation zu *Informatik*, ist die Ontologie für diese Werte hierarchisch. Anhand einer solchen hierarchischen Ontologie wie in Abbildung 3.2 sollte es wiederum möglich sein, eine gültige Domänengeneralisierungshierarchie abzuleiten.

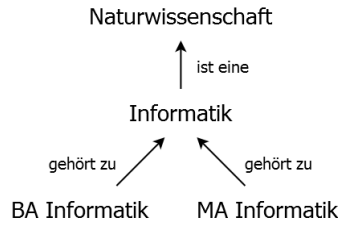


Abbildung 3.2: Ontologie für die Generalisierung in Abbildung 3.1

Seit Tim Berners-Lees Version des Semantic Web [BLCL+94] genießen Ontologien besondere Aufmerksamkeit in der Informatik und seither wurden verschiedene Wissensrepräsentationsformalismen dafür entwickelt. In diesem werden Ontologien dann mit Hilfe von formalen Sprachen wie OWL spezifiziert, da Semantiken in dieser Darstellungsform auch maschinenlesbar sind [SSB15]. OWL selbst basiert auf den im Folgenden erklärten Technologien RDF und RDF-Schema.

Resource Description Framework

Die vom World Wide Web Consortium (W3C) spezifizierte Sprache RDF [RDF15] wurde nach dem Vorschlag von Tim Berners-Lee entwickelt und ist mittlerweile fester Bestandteil verschiedener Semantic-Web-Technologien [SH13]. Im Grunde werden mit RDF Aussagen über Ressourcen gemacht, wobei jede Ressource ein Tripel bestehend aus *Subjekt*, *Prädikat* und *Objekt* ist. *Subjekt* sowie *Prädikat* sind jeweils Verweise auf Web-Identifikatoren (URIs) und das *Objekt* kann sowohl URI als auch Literal sein.

Soll beispielsweise eine Aussage über einen Studenten gemacht werden, so muss auf diesen sowie auf die Relation in der dieser mit dem Objekt steht mit einer Web-Ressource referenziert werden. Allgemein besitzen RDF-Aussagen die folgende Form:

SUBJEKT PRÄDIKAT OBJEKT .

Das *Subjekt* wird von dem *Prädikat* in eine Relation mit dem *Objekt* gestellt. Der abschließende Punkt beendet die Aussage. Als Beispiel sei hier formuliert, dass ein bestimmter Student Informatik studiert:

<http://uni-rostock.de/studenten/richard.dabels> <http://uni-rostock.de/ontol/studiert> "Informatik" .

Da eine Beschreibung durch vollständige URIs jedoch umständlich und unübersichtlich ist, können diese auch durch Präfixe ersetzt werden. In den folgenden Beispielen soll deshalb mit dem Präfix *wer* `http://uni-rostock.de/studenten/` und mit *was* `http://uni-rostock.de/ontol/` abgekürzt werden. Hier nochmal das obere Beispiel in abgekürzter Form:

```
wer:richard.dabels was:studiert "Informatik"
```

RDF-Schema

RDF-Schema erweitert RDF durch ein Typsystem, durch welches es ermöglicht wird, Klassen, Klassenhierarchien sowie die Zugehörigkeit von Ressourcen in Klassen zu definieren [Fen03]. Diese wurde selbst in RDF definiert und ist unter `http://www.w3.org/2000/01/rdf-schema` aufrufbar. Die für RDF-Schema-Objekte verwendete Abkürzung ist *rdfs*. Angelehnt an die Beispiele von [Fen03, SH13] lassen sich Klassen wie folgt definieren:

```
Person rdfs:type rdfs:Class .
Student rdfs:subClassOf Person .
```

Hier wurde eine Klasse mit dem Namen *Person* sowie eine zu ihr gehörende Subklasse *Student* erzeugt. Anschließend ist es möglich, Eigenschaften zu definieren...

```
Matrikelnummer rdfs:type rdfs:Property .
Geburtsdatum rdfs:type rdfs:Property .
```

... und die Klassen um diese zu erweitern:

```
Matrikelnummer rdfs:domain rdfs:Student .
Matrikelnummer rdfs:range xsd:integer .
Geburtsdatum rdfs:domain rdfs:Student .
Geburtsdatum rdfs:range xsd:date .
```

Die Bezeichnung `rdfs:domain` legt dabei den Typ des Subjekts, und `rdfs:range` den des Objekts fest. Einem mit RDF-Schema entwickelten *Vokabular* ist es möglich, domänenspezifische Klassen zu erzeugen und diese Informationen zwischen verschiedenen Systemen auszutauschen [SH13].

Web Ontology Language

Die Web Ontology Language (OWL) [OWL13] ist eine formale Sprache zur Darstellung von Ontologien [SSB15]. OWL basiert auf Grundlagen der Beschreibungslogiken. Die damit implizierte Ausdruckskraft und die Open World Assumption ist einerseits ein enormer Vorteil der Sprache, kann andererseits bei unendlichen Suchräumen zum Problem werden. Aus diesem Grund und aus der Annahme heraus, dass die Sprache in sehr unterschiedlichen Umgebungen genutzt werden würde, wurden drei verschiedene Versionen von OWL vorgeschlagen, von welchen jede eine andere Teilmenge von Beschreibungslogiken umfasst. Die folgenden Erklärungen der Versionen basieren auf [DHI12].

OWL-Lite ist eine effiziente Implementierung der Sprache, welche nur eine kleine Untermenge der Beschreibungslogik umfasst. Sie verzichtet auf die aus manchen Beschreibungslogiken bekannte Unique Name Assumption, bietet dafür jedoch Methoden um festzulegen, dass zwei Objekte gleich sind (*sameAs*) oder sich voneinander unterscheiden (*differentFrom*). Des Weiteren werden Operationen wie die logische Konjunktion, Allquantoren, Existenzquantoren sowie numerische Beschränkungen unterstützt, wobei die letzteren auf 0 und 1 limitiert sind.

OWL-DL bietet weitere Funktionalitäten, die aus Gründen der Komplexität nicht in OWL-Lite enthalten sind. Mitunter werden die Beschränkungen der Kardinalitäten aufgehoben, weshalb diese nun beliebig spezifizierbar sind. Die bereits in OWL-Lite enthaltenen Mengenoperationen werden um die Vereinigung, Komplemente und Schnittmengen auf Konzeptbeschreibungen erweitert. Weitere Funktionen sind *oneOf*, die Aussage, dass ein Konzept zu einer Menge gehört sowie *hasValue* für die Beschreibung der Aussage, dass ein Konzept einen bestimmten Wert besitzen muss.

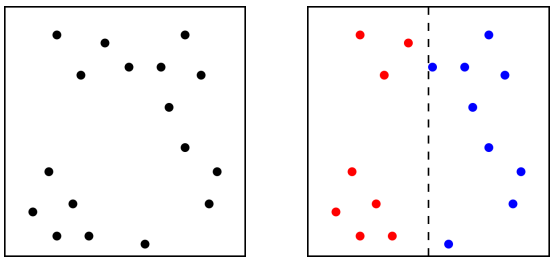
OWL-Full bietet alle in OWL-DL (und OWL-Lite) vorhandenen Funktionen. Zusätzlich dazu werden alle weiteren in OWL-DL existierenden Beschränkungen aufgehoben und die Reifikation von Aussagen zugelassen.

3.3 k-Means Clustering

Der zuerst in 1955 veröffentlichte k-Means-Algorithmus ist eines der meistgenutzten Verfahren in seinem Gebiet [Jai10]. Das Ziel ist hierbei, in Mengen von Objekten natürliche Muster zu finden, sodass diese in den dabei entstehenden Gruppen nur minimale Unterschiede aufweisen. k-Means gehört zu der Gruppe der partitionierenden Clustering-Algorithmen und wurde seither des Öfteren modifiziert, wie z.B. bisecting-k-Means oder anderweitig abgeändert [Jai10].

Um optimale Cluster zu ermitteln, werden im ersten Schritt k Cluster vorgegeben. Alle darauffolgenden Schritte verbessern das Ergebnis iterativ.
Nachdem die Initialcluster vorgegeben oder generiert wurden, werden für alle Cluster die Mittelpunkte berechnet. Durch die Bestimmung der Abstände aller einzelnen Punkte zu den Mittelpunkten erfolgt eine Neuzuordnung: Jeder Punkt wird dem Cluster zugeordnet, dessen Mittelpunkt er am nächsten steht. Es folgt ein iterativer Prozess, bestehend aus Berechnung der Abstände aller Punkte zu Mittelpunkten und einer anschließenden erneuten Einordnung in neue Cluster, bis keine Punkte mehr ausgetauscht werden. Ein Cluster ist optimal, wenn die Summe aller Abstände von Punkten zu ihren Clustermittelpunkten minimal ist [CL14].

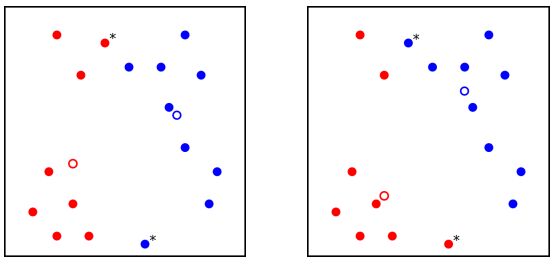
Sei als Beispiel für diesen Prozess die Punktmenge in Abbildung 3.3(a) gegeben. Um initiale Cluster zu erzeugen, soll hier ein Schnitt in der Mitte des Raumes vorgenommen werden. Das Ergebnis ist in Abbildung 3.3(b) zu sehen.



(a) Ungeclusterte Menge (b) Initiale Clusterung

Abbildung 3.3: k-Means: Initialisierung

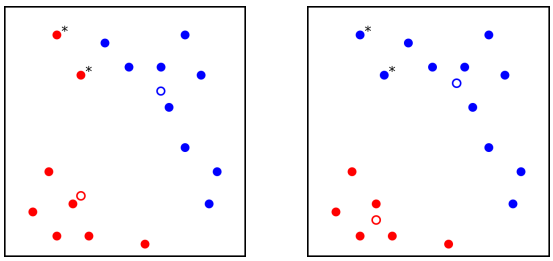
Als Nächstes, in Abbildung 3.4(a), werden die Mittelpunkte der Cluster berechnet. Anhand dieser gilt es nun, alle Punkte so in neue Cluster einzuordnen, dass sie ihren jeweiligen Mittelpunkten so nahe wie möglich sind. Die betroffenen Punkte sind mit (*) markiert. In Abbildung 3.4(b) sind diese neu zu Clustern zugeordnet zu sehen. Auch die neu berechneten Mittelpunkte sind hier bereits berechnet.



(a) Abstandsberechnung (b) Neuzuordnung

Abbildung 3.4: k-Means: Schritt 1

Anhand dieser neuen Mittelpunkte wird wiederum erkannt, welche Punkte die Cluster in dem folgenden Schritt wechseln müssen. Diese sind in Abbildung 3.5(a) wieder mit (*) markiert. In Abbildung 3.5(b) ist der Zustand nach der Neuzuordnung zu sehen.



(a) Abstandsberechnung (b) Neuzuordnung

Abbildung 3.5: k-Means: Schritt 2

Bei Betrachtung der nun entstandenen Cluster in Abbildung 3.6 kann erkannt werden, dass durch die Ermittlung der Mittelpunkte und die darauffolgende Neuzuordnung keine Veränderungen mehr entstehen

würden. Sobald dieser Punkt erreicht ist, endet der k-Means-Algorithmus und die finalen Cluster sind entstanden[1].

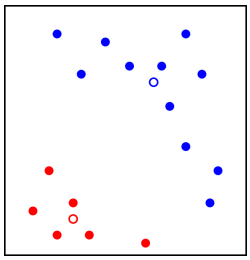
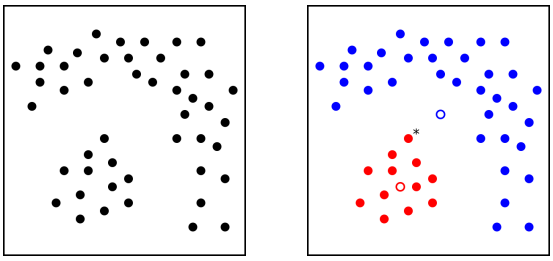


Abbildung 3.6: Ergebnis von k-Means

Das Verfahren erzeugt in der Praxis bereits nach relativ wenigen Iterationen stabile Cluster. Die notwendige Berechnung der Abstände jedes einzelnen Objektes zu jedem der Mittelpunkte erzeugt dabei die meisten Kosten. Ein weiterer Vorteil ist, dass der einfache Aufbau das Verfahren sehr anschaulich macht und somit auch leicht zu implementieren ist [CL14].

Zu bemerken ist jedoch, dass k-Means nur zur Erzeugung von konvexen Clustern verwendet werden kann. Für nicht-konvexe Clusterbildung ist es nötig, Algorithmen zu nutzen, die Cluster dichtebasiert erzeugen. Dies lässt sich besonders an der Abbildung 3.7 erkennen. So sollte bei der Betrachtung von Abbildung 3.7(a) bereits eine bestimmte Erwartung an die zu entstehenden Cluster bestehen – wahrscheinlich würde diese die Form der Cluster in Abbildung 3.7(b) annehmen. Das Problem ist aber, dass diese mit k-Means nicht erreicht werden kann. Der Punkt, welcher mit (*) markiert ist und im nächsten Schritt seine Farbe wechselt, zeigt bereits einen Trend auf, welcher gegen die zu erwartende Clusterbildung arbeiten würde.



(a) Konvexes Cluster (b) Clusterbildung

Abbildung 3.7: k-Means: Clusterbildung bei konvexen Clustern

Es besteht auch eine große Abhängigkeit von den vorgegebenen Clustern. Diese wirken sich nämlich direkt auf die Anzahl der Iterationen aus [CL14]. Weiterhin ist die Auswahl eines optimalen k's selbst nur heuristisch bestimmbar. Da k-Means ein Greedy-Algorithmus ist, kann auch nicht garantiert werden, dass das Ergebnis tatsächlich optimal ist. Aus diesen Gründen ist es oft ratsam, sich das Ergebnis mehrfach, mit verschiedenen Parametern, erzeugen zu lassen [Jai10].

[1] Die in Abbildungen 3.3, 3.4, 3.5 und 3.6 verwendeten Mittelpunkte und Abstände zur Veranschaulichung des Verfahrens sind lediglich geschätzt. Aus diesem Grund ist es möglich, dass die Genauigkeit der Angaben abweicht

3.4 k-Member-Clustering-Problem

Byun et al. [BKBL07] haben bei der Ermittlung einer optimalen k-Anonymisierung von Datensätzen eine alternative Vorgehensweise genutzt. Hier werden die für die Ermittlung der k-Anonymität notwendigen Äquivalenzklassen als Cluster und die Erzeugung einer gültigen Generalisierung als das k-Member Clustering Problem angesehen, welches wie folgt definiert wird: Bei n Einträgen ist ein Cluster-Schema $\varepsilon = \{e_1, ..., e_l\}$ gesucht, sodass

1. $|e_i| \geq k, 1 < k \leq n$: Die Größe der Cluster ist mindestens so groß wie der ganzzahlige Wert k (k-Anonymität), und
2. $\sum_{i=1,...,\ell} IL(e_i) < c, c > 0$: Der gesamte Informationsverlust überschreitet nicht den positiven Grenzwert c, wobei $IL(e_i)$ der Informationsverlust bei der Generalisierung mit e_i ist [BKBL07].

Bei Verfahren wie dem k-Means-Clustering-Algorithmus wird ein k vorgegeben, sodass k Cluster generiert werden. Ein k-Anonymitätproblem jedoch hat keine natürliche Beschränkung an die Anzahl seiner Cluster. Stattdessen wird vorausgesetzt, dass jedes Cluster eine Mindestanzahl an Einträgen besitzt, um die k-Anonymität zu erfüllen: $\forall e_i \in \epsilon, |e_i| \geq k$. Diese Anforderung wird durch das k-Member Clustering Problem erfüllt. Weitere Anforderungen an die entstehenden Cluster sind die folgenden:

1. $\forall i \neq j \in \{1, ..., m\}, e_i \cap e_j = \emptyset$: Paarweise Disjunktheit.
2. $\bigcup_{i=1,...,m} e_i = S$: Vollständigkeit.
3. $\sum_{\ell=1,...,m} |e_\ell| \cdot MAX_{i,j=1,...,|e_\ell|} \Delta(p_{(\ell,i)}, p_{(\ell,j)})$, wobei $\Delta(x, y)$ die Distanz zwischen den Punkten x und y ist: Die zu minimierende Kostenfunktion [BKBL07].

Bei der Distanzberechnung wird zwischen numerischen und kategorischen Wertebereichen unterschieden. Erstere können im Grunde genommen bereits direkt aus der Differenz des Ausgangswertes und des generalisierten Wertes gewonnen werden. Um sie jedoch später zusammen mit den kategorischen Distanzen zu verwenden, werden diese wie folgt normiert berechnet:

> Sei D ein endlicher numerischer Wertebereich. Die normalisierte Distanz zwischen zwei Werten $v_i, v_j \in D$ ist definiert als: $\delta_N(v_1, v_2) = \frac{|v_1 - v_2|}{|D|}$, wobei $|D|$ die Größe des Wertebereichs ist, berechnet durch die Differenz des größten und des kleinsten Wertes aus D [BKBL07].

Komplizierter hingegen ist die Berechnung der Distanz zwischen kategorischen Werten. Nach der gegebenen Definition ist diese wie folgendermaßen zu berechnen:

> Sei D ein kategorischer Wertebereich und T_D eine Domänengeneralisierungshierarchie für D. Die normalisierte Distanz zwischen zwei Werten $v_i, v_j \in D$ ist definiert als: $\delta_C(v_1, v_2) = H(\wedge(v_i, v_j))/H(T_D)$, wobei $\wedge(x, y)$ der Teilbaum der Domänengeneralisierungshierarchie für D ist, der seine Wurzel am nächsten gemeinsamen Vorgänger von x und y hat. $H(R)$ ist dabei die Höhe der Hierarchie T [BKBL07].

Als Beispiel für die Berechnung der kategorischen Distanz zweier Werte soll erneut die Abbildung 3.1 dienen. Hierfür soll der Abstand zwischen den Einträgen *BA Informatik* und *MA Physik* bestimmt werden. Der nächste gemeinsame Vorgänger der Einträge ist *Naturwissenschaften*. Der daraus resultierende Teilbaum hat eine Höhe von 3, weshalb der Abstand der beiden Einträge $2/2 = 1$ beträgt. Als weiteres Beispiel sollen die Einträge *BA Chemie* und *MA Chemie* betrachtet werden. Hier ist der nächste gemeinsame Vorgänger bereits *Chemie*, der daraus zu berechnende Abstand beträgt $1/2 = 0{,}5$.

Durch die Aufsummierung aller einzelnen Abstände lässt sich der gesamte Datenverlust berechnen, den es niedrig zu halten gilt. Byun et al. zeigen, dass das k-Member-Clustering-Problem NP-vollständig ist. Die in [BKBL07] vorgeschlagene Heuristik, welche in dieser Arbeit nicht weiter ausgeführt wird, besitzt jedoch eine Zeitkomplexität von $O(n^2)$.

3.5 Fazit

Die in diesem Kapitel behandelten Konzepte sind alle von besonderer Relevanz für diese Arbeit. Insbesondere gilt dies natürlich für die in Absatz 3.1 behandelten Domänengeneralisierungshierachien, aber auch für die Ontologien und den Clustering-Kandidaten k-Means für die Implementierung in Absatz 3.2 und 3.3.

Absatz 3.4 bietet mit dem k-Member-Clustering-Problem einen interessanten Ansatz zur k-Anonymität-konformen Clusterung von Elementen. Da das in dieser Arbeit zu ermittelnde Verfahren zur automatischen Generierung von Domänengeneralisierungshierarchien jedoch nur eine Grundlage für die Anonymisierung schafft und nicht selbst anonymisiert, wird diese Idee hier auch keine Anwendung finden. Ein Teilaspekt des k-Member-Clustering-Problems, die genormte Distanzberechnung, wird jedoch trotzdem verwendet werden.

Kapitel 4

Konzept

In diesem Kapitel soll ein mögliches Konzept zur automatischen Generierung von Domänengeneralisierungshierarchien erarbeitet werden. Hierfür wird erneut die für diese Arbeit gegebene Aufgabenstellung erläutert sowie passende Lösungen aus den vorhergehenden Kapiteln vorgeschlagen. Die durch die Implementierung generierten Domänengeneralisierungshierarchien sollen zusätzlich mit dem PArADISE-Projekt des Lehrstuhls Datenbanken der Universität Rostock [GH16] kompatibel sein.

4.1 Anforderungen

Das bekannte Problem mit derzeit verwendeten Generalisierungshierarchien ist, dass diese manuell erzeugt werden müssen und sich nicht an verändernde Datenbestände anpassen. Ein vergleichbares Problem wurde bereits im PArADISE-Projekt der Universität Rostock bearbeitet: Auch die zur teilweisen Identifikation von Personen nutzbaren Quasi-Identifikatoren sind von diesen Änderungen abhängig und können bei sich stark ändernden Datensätzen zu jeweils höherem oder niedrigerem Informationsgewinn beitragen [GH14]. Aus diesem Grund werden Quasi-Identifikatoren der Datenbestände des PArADISE-Projektes nach vorgegebenen Zeitabständen neu berechnet, um weiterhin nur die Anonymisierung der relevantesten Attribute einer Tabelle zu garantieren. Da die Generalisierung von Daten ausschließlich auf diesen speziellen Attributen geschieht, müssen Domänengeneralisierungshierarchien nur für diese erzeugt werden. Beide Prozesse können aufgrund ihrer Abhängigkeit voneinander direkt nacheinander ausgeführt werden.

Die aktuell zur Anonymisierung verwendetet Generalisierunginformationen befinden sich bereits auf den Servern der Universität Rostock. Das für deren Speicherung verwendete Datenbankschema in Abbildung 4.1 wurde von Martin Müller entworfen [Mül16] und wird auch die Schnittstelle für automatisch generierte Domäneneneralisierungshierarchien im PArADISE-Projekt werden. Die mit dieser Spezifikation entstehenden Einträge sollten bereits für die vollständige Einbindung der hier erarbeiteten Software in die bestehende Systemumgebung genügen.

Da nicht zu erwarten ist, dass ein einheitliches Konzept, welches mit allen verfügbaren und verwendbaren Ontologien kompatibel ist, erarbeitet werden kann, muss eine hohe Erweiterbarkeit und Modifizierbarkeit für die hier zu erarbeitende Software gegeben sein. Die Implementierbarkeit von neuen Ontologien und Verfahren muss mit Hilfe von passenden Schnittstellen unbedingt möglich sein, damit die Anwendbarkeit des Verfahrens sich nicht nur auf wenige, ausgewählte Einsatzfälle beschränkt.

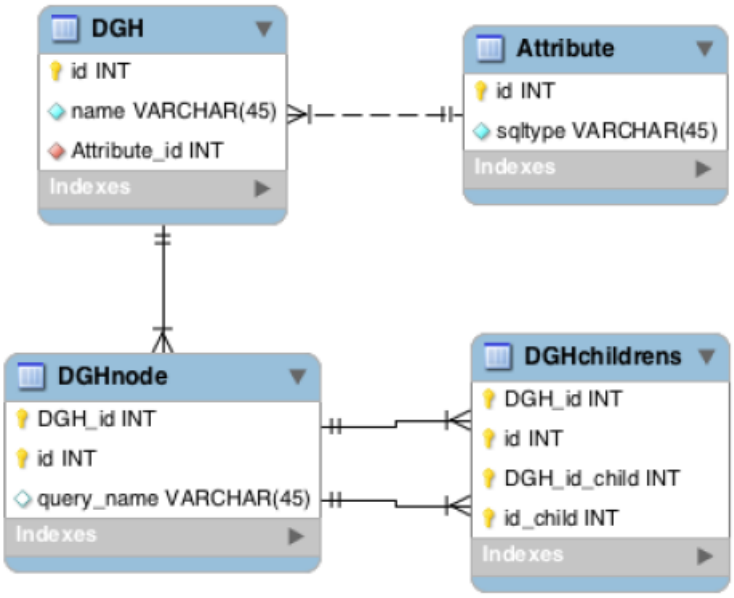


Abbildung 4.1: Datenbankschema für Domänengeneralisierunghierarchien nach [Mül16]

Für die Generierung von Domänengeneralisierunghierarchien sollen verschiedene Clusteringverfahren betrachtet sowie mindestens eines dieser Verfahren implementiert werden. Des Weiteren soll abhängig von dem Wertebereich des zu generalisierenden Attributes die Entscheidung getroffen werden, ob hierfür Ontologien zu nutzen sind, oder lediglich einer der implementierten Clusteringalgorithmen zum Einsatz kommt.

Die Ontologien, welche für die automatische Generierung von Domänengeneralisierunghierarchien genutzt werden, sind für die Präservierung und garantierte Zugänglichkeit auf mindestens einem der Datenbanksysteme, die von der Universität Rostock bereitgestellt werden, zu speichern. Die Auswertung der dort gespeicherten Daten erfolgt über die Software mit Hilfe von JDBC sowie natürlich SQL.

Die im Zuge der fortschreitenden Implementierung erzeugten Testfälle bleiben zusätzlich im Projekt enthalten und dienen als konkrete Anwendungsbeispiele. Darüber hinaus wird die Software so in das bestehende PArADISE-Projekt eingebunden, dass Generalisierunghierarchien zusammen mit den Quasi-Identifikatoren generiert und in einer Datenbank abgelegt werden.

4.2 Entwurf

Aus den oben genannten Anforderungen wird folgend ein Softwareentwurf erstellt, der das vorliegende Problem löst. Zusammenfassend lassen sich aus den Anforderungen drei verschiedene Komponenten ableiten:

1. Für sich unterscheidende Anwendungsfälle sind unter Umständen andere Clustering-Algorithmen zu verwenden. Aus diesem Grund sollte die Software leicht um zusätzliche Verfahren zu erweitern sein.

2. Die Ontologien spielen, neben dem Clustering, die größte Rolle in dem zu entwickelnden Ansatz. Diese sind jedoch meist sehr anwendungsspezifisch und können sich untereinander stark unter-

scheiden, was die Entwicklung eines einheitlichen Verfahrens zur Abfrage dieser erschwert. Auf für zusätzliche Ontologien muss deshalb eine hohe Erweiterbarkeit gegeben sein.

3. Mit Hilfe des Clusterings und / oder den Ontologien muss eine gültige Domänengeneralisierunghierarchie aufgebaut, angezeigt und in einer Datenbank gespeichert werden.

Aus diesen Punkten heraus wurde der erste Grobentwurf der Software entwickelt, zu sehen in Abbildung 4.2. Hier gezeigt sind die drei Komponenten in zentraler Lage, der *Clusterer*, welcher die Schnittstelle zu verschiedenen Clustering-Algorithmen bieten sollte, eine *Ontologie-API*, um möglicherweise sehr komplexe Ontologien vereinfacht von anderen Komponenten abfragbar zu machen sowie der DGH-Builder, welcher Clustering und Ontologien nutzt, um gültige Domänengeneralisierunghierarchien aufzubauen und Datenbankschnittstellen verwaltet.

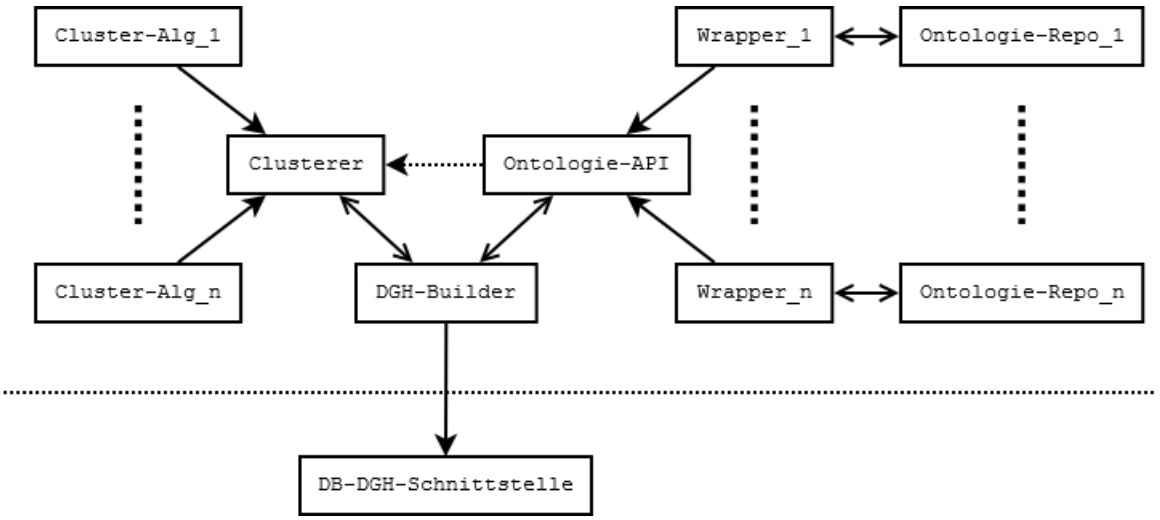


Abbildung 4.2: Erster Grobentwurf der Software

Bei Betrachtung verschiedener Clustering-Algorithmen sowie verfügbaren Ontologien wurde schnell offensichtlich, dass eine solche Trennung der Komponenten unter Umständen nicht durchführbar ist. Bereits die Spezifikation eines einheitlichen Interfaces für Clustering-Verfahren erweist sich als umständlich. Sei zum Beispiel ein hierarchischer und ein gleichverteilender Clustering-Algorithmus zu implementieren: Während der gleichverteilende Algorithmus lediglich einen Parameter erwartet, welcher die Größe der zu generierenden Cluster bestimmt, muss bei hierarchischem Clustering angegeben werden, welches Distanzmaß (Single-Linkage, Complete-Linkage, Average-Linkage) verwendet wird. Obwohl es möglich ist dieses Problem zu lösen, indem als Parameter übergebene Strings zu anderen Werte geparsed werden, wurden die zwei gewählten Clustering-Implementierungen ohne Interface implementiert. Daraus resultierende Einbuße der Erweiterbarkeit sind zum Entwurf des Konzeptes durchaus bewusst; die vorhandene Version entsprechend anzupassen sollte sich jedoch als einfach herausstellen, falls dies noch vorgesehen ist.

Der überarbeitete Entwurf in Abbildung 4.3 implementiert die neu gewonnenen Erkenntnisse. Die Domänengeneralisierunghierarchie wird nun als eigenes Objekt aufgefasst, welches sich aus vielen einzelnen Einträgen zusammensetzt, an welchen auch die neue Ontologie-Schnittstelle spezifiziert wird. Clustering-Algorithmen werden fortan direkt an der Domänengeneralisierunghierarchie ausgeführt, die nach Bedarf ihre Struktur ändert, neue Einträge erzeugt und diese auch verschmilzt. Auch die Distanzberechnung für das Clustering soll ab nun direkt über die Implementierungen der Einträge passieren.

Dies bringt den Vorteil mit sich, dass verschiedenen Arten von Einträgen mit allen neu implementierten Clustering-Algorithmen kompatibel sind, so lange wie diese die spezifizierten Distanzberechnungsfunktionen der abstrakten Oberklasse DGHEntry nutzen.

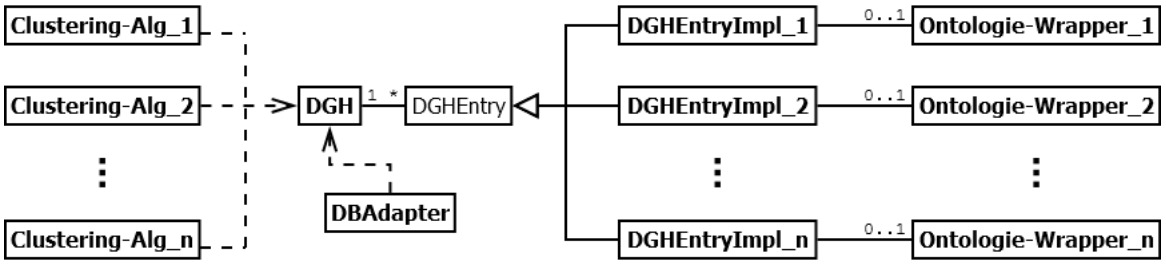


Abbildung 4.3: Überarbeiteter Entwurf

Weiterhin wird das Problem der Ontologie-Schnittstelle in die Implementierung der Einträge verschoben. Da einzelne Ontologien jeweils nur für sehr spezielle Anwendungsfälle nützlich sind, sollte es kaum nötig sein diese wiederzuverwenden. Aus diesem Grund wurde entschieden, die Implementierung der Ontologieschnittstellen der Subklasse von DGHEntry zu überlassen.

Da Domänengeneralisierunghierarchien unabhängig von der Art ihrer Einträge immer die gleiche Baumstruktur tragen, können Funktionen für den Datenbankzugriff sowie für eine mögliche grafische Ausgabe, zentral über die DGH-Klasse geschehen. Für die Anbindung an Projekte wie PArADISE können zusätzlich Klassen geschrieben werden, welche die automatische Generierung von Domänengeneralisierunghierarchien schrittweise ausführen.

4.3 Datentypen

Die größte Funktionalität dieses Konzeptes befindet sich in den Implementierungen der einzelnen Einträge, den DGHEntry's. Diese repräsentieren verschiedene Objekte, durch dessen Eigenschaften eine Domänengeneralisierunghierarchie generierbar seien muss. Um dieses Vorgehen beispielhaft zu zeigen, wird eine Auswahl an Einträgen implementiert. Diese beinhalten DGHEntry's für Objekte wie Integer, Doubles, Daten (von Datum) und zwei verschiedene Auswertungen von Postleitzahlen.

4.3.1 Numerische Werte

Zu den numerischen Werten gehören natürlich Objekte wie Integer und Doubles. Die Distanzberechnung zwischen diesen ist mit dem Betrag der Differenz zweier Werte auch denkbar einfach. Da Java eine umfangreiche Bibliothek zu Berechnung und Auswertung von Datums-Datentypen mit sich bringt, ist auch der Umgang mit diesen nicht kompliziert. Auch Postleitzahlen lassen sich numerisch und abhängig davon, ob und wie Distanzen bei ihrer Berechnung genormt werden, sogar als Integer darstellen. Dies kann jedoch unerwünschte Folgen nach sich ziehen.

Anders als natürliche Zahlenwerte und Daten müssen Postleitzahlen keiner logischen Ordnung unterliegen. Damit kann es passieren, dass die Clusterung der Werte unter einfacher, numerischer Auswertung an Bedeutung verliert. Als Beispiel soll eine Karte von Postleitzahlen aus Mönchengladbach in Abbildung 4.4 betrachtet werden, welche hierarchisch mit Complete-Linkage geclustert wird. Die Reihenfolge, in der die Postleitzahlen 41061, 41063, 41065, 41066, 41068 und 41069 zusammengefasst werden würden, wäre die Folgende: Als Erstes verschmelzen die Werte 41068 und 41069. Danach 41065 und 41066, worauf das

Paar 41061 und 41063 folgt. Da Complete-Linkage angewendet wird, ist nun die Distanz zwischen den Clustern 41065, 41066 und 41068, 41069 mit einem Wert von 4 kleiner als die von {41061, 41063} nach {41065, 41066} mit 5. Aus diesem Grund werden im kommenden Schritt {41065, 41066} und {41068, 41069} verschmolzen, was aus geographischer Sicht keinen Sinn ergibt, da zwischen beiden Clustern noch das Cluster {41061, 41063} steht. Die hier entstehende Clusterung und daraus resultierende Domänengeneralisierunghierarchie ist somit für einige Anwendungsgebiete unter Umständen wertlos.

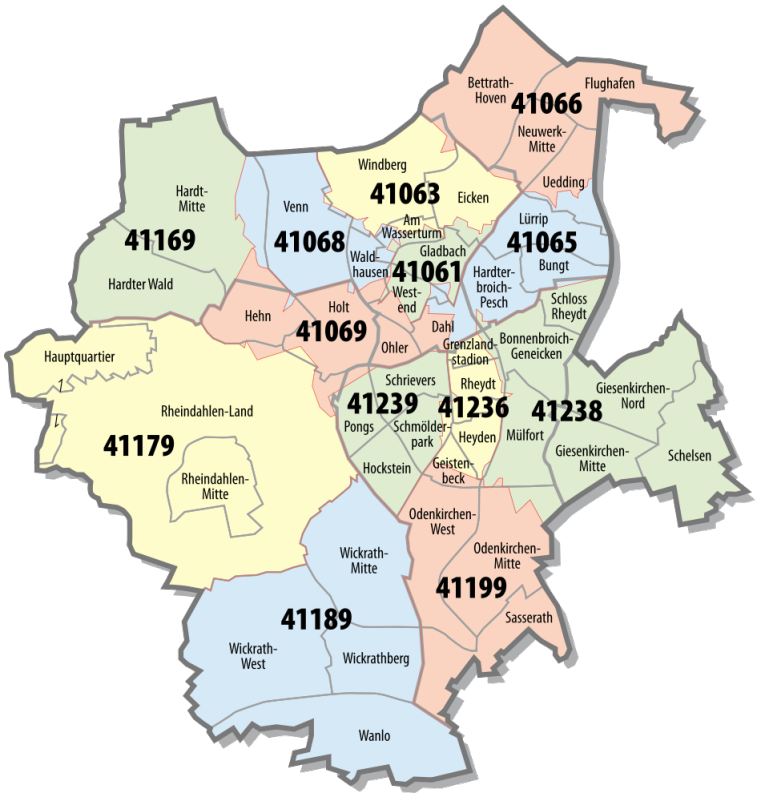


Abbildung 4.4: Postleitzahlen in Mönchengladbach nach [Nor]

4.3.2 Kategorische Werte

Aufgrund des eben genannten Beispiels ist es nötig, eine zusätzliche, von der numerischen Version unabhängige, kategorische Implementierung des Postleitzahl-Datentyps zu erzeugen. Hierfür soll die Ontologie von OpenGeoDB [Ope15] zum Einsatz kommen. Durch diese lassen sich den Postleitzahlen GPS-Daten zuordnen, mit deren Hilfe eine geografisch korrekte Clusterung der Einträge ermöglicht wird. Zusätzlich ist es mit Hilfe der Ontologie auch möglich, einem Cluster von Postleitzahlen einen Ortsnamen zuzuordnen, um die Übersichtlichkeit der Gruppierungen zu verbessern. Mit diesem müsste, sobald alle Postleitzahlen Mönchengladbachs zu einem Cluster zusammengefügt worden sind, nicht die unhandliche Bezeichnung „{41061, 41063, 41065, 41066, 41068, 41069, ..., 41239}“ verwendet, sondern die gesamte Menge könnte unter dem Namen „Mönchengladbach“ generalisiert werden. Dieses Beispiel zeigt, wie Ontologien zusammen mit Clustering-Algorithmen dazu in der Lage sind, Domänengeneralisierunghierarchien aufzubauen.

4.4 Clustering

Für die Clusterung numerischer Werte stehen verschiedene Verfahren zur Auswahl, von denen einige bereits in den vorherigen Kapiteln vorgestellt wurden. Ein mögliches Verfahren wäre k-Means, welches aus einer Anzahl von Objekten eine Menge von k Cluster erzeugt. Dessen Vorteile wären die einfache Implementierung und die möglicherweise kleine Anzahl an Iterationen bis zum Ergebnis, was jedoch von einer geeigneten Auswahl der Initialcluster abhängt [CL14]. Zusätzlich besteht die Gefahr, dass die Wahl eines geeigneten k's ineffizient und nur durch mehrere Durchführungen zu lösen ist [Jai10]. Dieses Problem verstärkt sich nur dadurch, dass eine Domänengeneralisierunghierarchie aus mehreren Ebenen aufgebaut ist und die genannten Schwierigkeiten erneut bewältigt werden müssen. Aufgrund dieser Problemstellung wurde gegen die Implementierung von k-Means entschieden.

Die Wahl des zu implementierenden Clusterings fiel hingegen auf das hierarchische Clustering. Grund dafür ist vor allem, dass die in hierarchischen Verfahren entstehenden Baumstrukturen bei guter Generierung direkt als Domänengeneralisierunghierarchie verwendet werden können. Zusätzlich zur einfachen Implementierung sind die verschiedenen Cluster-Modi Single-Linkage, Complete-Linkage und Average-Linkage von besonderem Interesse, da durch diese die Möglichkeit besteht, das Verfahren in einem breiten Spektrum an Fällen anzuwenden und die optimale Ausführung bereits durch eine kleine Anzahl an Versuchen festgestellt werden kann. Dabei ist ein optimales Ergebnis dadurch gekennzeichnet, dass der entstehende Cluster-Baum nahezu balanciert ist. Ein weiterer Vorteil ist, dass keine Initialcluster für die Ausführung nötig sind.

Neben dem hierarchischen Clustering wird jedoch ein weiteres Verfahren, das eindimensionale, gleichverteilte Clustering, in der Implementierung der Software Anwendung finden. Mehr als alle anderen Verfahren verspricht dieses, Cluster mit optimalen Größen und Verteilungen zu finden. Auch sind hierzu keine komplizierten Algorithmen notwendig, da in jedem Schritt die Menge aller vorhandenen Cluster in Gruppen fester Größe unterteilt werden muss. Bei dessen korrekter Wahl ist bereits nach einmaliger Generalisierung mit der entstehenden Domänengeneralisierunghierarchie die k-Anonymität für ein Attribut garantiert. Der Nachteil dieser Art des Clusterings ist jedoch, dass sie grundsätzlich inkompatibel mit höherdimensionalen Wertebereichen ist. So können Punkte auf einer Karte, wie in Abbildung 4.4, nicht korrekt in nur einer Dimension dargestellt werden.

Kapitel 5

Implementierung

Die Implementierung erfolgt, wie im Vorhinein erwähnt, in Java mit Hilfe von JDBC und SQL. Dieses Kapitel dient zur ausführlichen Erklärung der Implementierung einzelner Klassen. Zur Übersicht wird des Öfteren auf Abbildung 5.1 verwiesen, in welcher das vollständige Klassendiagramm der Software zu sehen ist. Hier lässt sich zuerst die Aufteilung in ihre Packages erkennen. So bilden nun, wie in dem überarbeiteten Entwurf in Abbildung 4.3, alle Komponenten die zur Struktur der Domänengeneralisierungshierarchie gehören, zusammen mit der beispielhaften Implementierung der Ontologie von OpenGeoDB [Ope15], die *LocationEntry*, in `agvdgh.dgh` eine Sinneinheit. Die implementierten Clustering-Algorithmen befinden sich in ihrem eigenen Package, `agvdgh.clustering` und haben Zugriff auf die Klassen in `agvdgh.dgh` um die dort definierten Domänengeneralisierungshierarchien umzustrukturieren. Um die eben genannten Komponenten einfach zu halten, werden zusätzliche Funktionalitäten und Klassen in `agvdgh.util` ausgegliedert.

5.1 Clustering

Die Erläuterung der Implementierung wird bei der dem Package `agvdgh.clustering` beginnen. Die hier gewählten Verfahren sind auch ohne Vorkenntnisse der Klassen *DGH* und *DGHEntry* sehr anschaulich und die verwendeten Methoden geben bereits einen Einblick auf die kommenden Absätze.

5.1.1 Hierarchisches Clustering

Die Klasse *H_ Clustering* implementiert die Funktionalität des hierarchischen Clusterings in das Projekt. Sie besteht aus vier öffentlichen Methoden, die eine saubere Schnittstelle für die Anwendung durch einen Nutzer bereitstellen sowie zwei private Methoden, in welchen die eigentlichen Berechnungen stattfinden.

Das hierarchische Clustering kann mit verschiedenen Parametern initiiert werden. Grundlegend sind drei verschiedene Informationen für die Durchführung nötig:

1. Eine Menge von Elementen, welche zu clustern sind.
2. Der Modus, in welchem das hierarchische Verfahren durchgeführt werden soll (Single-Linkage, ...).
3. Der Name der Klasse, welche den der Menge zugehörigen Datentyp implementiert (*IntegerEntry* für Integer, ...).

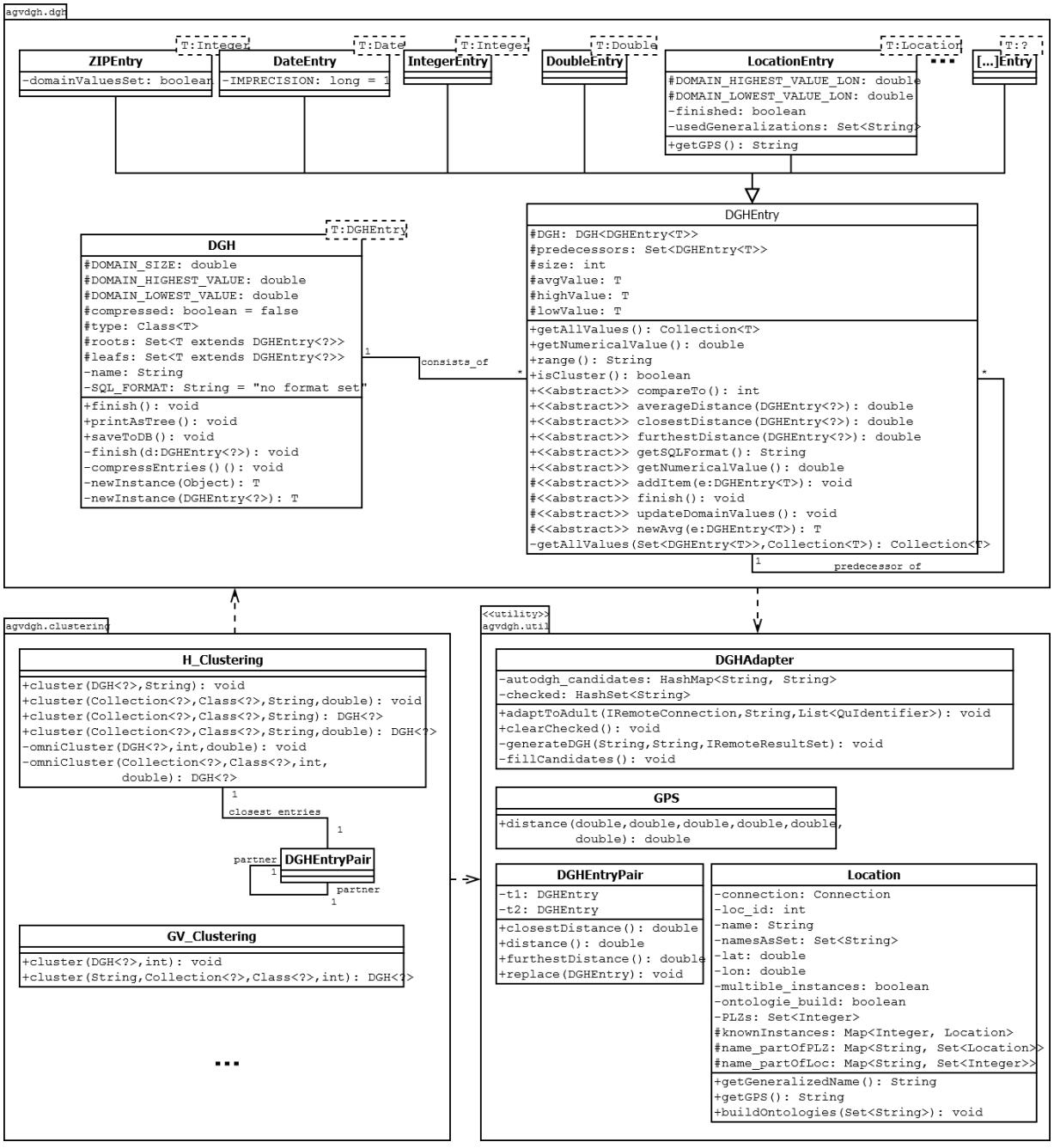


Abbildung 5.1: Klassendiagramm

Die Methoden *cluster(DGH<?>, String)* und *cluster(Collection<?>, Class<?>, String)* implementieren diese Voraussetzungen indirekt und direkt. Zu bemerken ist, dass unabhängig davon, ob ein *DGH*-Objekt explizit als Parameter übergeben wird, die Verfahren trotzdem immer auf solchen stattfinden. Sollte nur eine *Collection* angegeben werden, wird diese mit Hilfe der im anderen Parameter spezifizierten Klasse

entsprechend umgewandelt. In *cluster(DGH<?>, String)* sind durch den ersten Parameter, die *DGH*, Bedingungen 1 und 3 gegeben, da es beide Informationen enthält. Nur die Voraussetzung 2 muss noch im zweiten Parameter angegeben werden. Die Funktion *cluster(Collection<?>, Class<?>, String)* hingegen benötigt die Angabe der Informationen 1, 2 und 3 explizit. Auf die übrigen zwei öffentlichen Funktionen soll später eingegangen werden.

Die Vorgehensweise in Zusammenhang mit der DGH-Klasse wird im Folgenden beschrieben. Einige Details sind dabei noch außer Acht gelassen, oder werden erst später ausführlich erklärt. Da alle Funktionen der Klasse letztendlich auf die Funktion *omniCluster(DGH<?>, int, double)* zurückgeführt werden, ist es auch diese, welche mit Algorithmus 1 vorgestellt wird.

Algorithmus 1 : Hierarchisches Clustering mit Hilfe der *DGH*-Struktur

```
   Input : DGH dgh, int modeNum
   Output : DGH dgh
 1 roots ← getRoots(dgh);
 2 minDist ← 0.0;
 3 while |roots| > 1 do
 4 |   for e1, e2 in roots do
 5 |   |   if e1 != e2 then
 6 |   |   |   dist ← distance(dgh, e1, e2, modeNum);
 7 |   |   |   if dist < minDist ∨ minDist == 0 then
 8 |   |   |   |   pair ← DGHEntryPair(e1, e2);
 9 |   |   |   end
10 |   |   end
11 |   end
12 |   e1 ← first(pair);
13 |   e2 ← second(pair);
14 |   merge(dgh, e1, e2);
15 |   roots ← getRoots(dgh);
16 end
```

Aus einem *DGH*-Objekt lassen sich, wie in Zeile 1 und 15 zu sehen ist, mit einer Funktion immer die aktuellen Wurzelknoten gewinnen. Das hierarchische Verfahren sucht in jedem Schritt die *DGHEntry*-Objekte, deren Distanz zueinander am kleinsten ist. Diese werden als Paar gespeichert, bis alle Wurzelknoten überprüft worden sind, und ihre Inhalte nach jedem vollständigen Durchlauf in Zeile 14 miteinander über die *DGH* verschmolzen. Hiermit verkleinert sich die Anzahl der Wurzelknoten mit jedem Schritt um 1. Sobald nur noch ein Objekt in den Wurzelknoten vorhanden ist, bricht das Verfahren ab – die Domänengeneralisierungshierarchie ist aufgebaut.

Ergebnis

In Abbildung 5.2 ist eine durch hierarchisches Clustering erzeugte Domänengeneralisierungshierarchie für Daten zu sehen. Diese wurde aus einer kleinen Menge von 13 Testeinträgen erzeugt. Durch die hierarchische Vorgehensweise werden in jedem Schritt immer nur zwei Cluster verschmolzen. Das bedeutet, dass jedes Cluster auch nur zwei Einträge umfasst. Zu erkennen sind zwei Eigenschaften des Baumes:

1. Der Baum ist nicht balanciert. Dies kann, abhängig davon, ob für jeden Knoten im Baum oder in ganzen Baumebenen generalisiert wird, ein Problem für einige Generalisierungsalgorithmen sein. Das hierarchische Verfahren bietet für diesen Umstand keine Abhilfe. Es ließe sich, falls die Gleich-

verteilung im Baum ein Qualitätskriterium ist, jedoch die Qualität des Baumes als durchschnittliche Tiefe der Zweige beschreiben.

2. Ausreißer-Einträge sind vorhanden. Da das Cluster des Wertes „11.11.1964“ zu weit von den anderen Werten entfernt ist, bieten keine der Modi Single-Linkage, Complete-Linkage oder Average-Linkage Abhilfe um dieses Problem zu lösen. Eine andere Möglichkeit wurde deshalb implementiert.

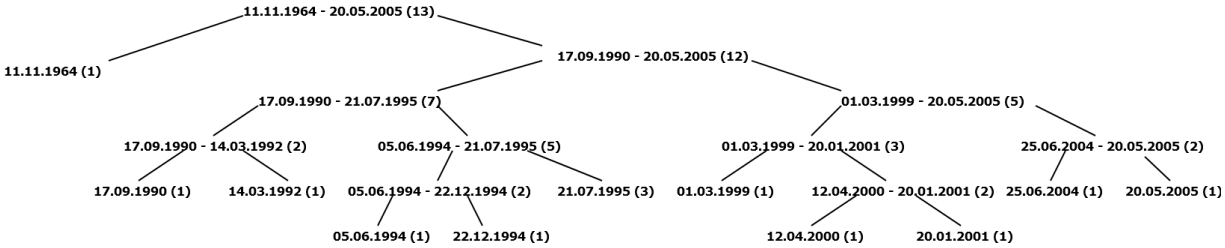


Abbildung 5.2: Durch Complete-Linakge-Clustering erzeute DGH mit Ausreißer

Das in Punkt 2 beschriebene Problem äußert sich insbesondere dadurch, dass die genannten Modi keinen Einfluss auf Ausreißer haben. Vorgeschlagen wird deshalb, dass ein *Penalty*-Wert für die Größe der Cluster eingeführt wird. Implementiert werden kann dieser in der Distanzberechnung zwischen den Clustern.

Dies kann wie in der Implementierung 5.1 geschehen. Der *Penalty*-Wert darf in dieser Implementierung die Größe des Wertebereichs nicht überschreiten, da Cluster andernfalls nicht mehr korrekt gebildet werden würden. Aus diesem Grund ist in einem *DGH*-Objekt die jeweilige Größe des Wertebereichs mitgeführt. Auch ist es so möglich, angelehnt an [BKBL07], genormte Werte für Distanzberechnungen zu verwenden. Auf mehrdimensionalen Wertebereichen ist die genormte Distanzberechnung jedoch nicht korrekt anwendbar. Deshalb wurde auch der in den vorherigen Erklärungen der *cluster*-Methoden außer Acht gelassene Parameter *penalty* vom Datentyp *double* implementiert, welcher als Faktor auf die Distanzberechnung einwirkt.

Listing 5.1: Implementierung der Penalty

```
1 private static void omniCluster(DGH<?> dgh, int modeNum, double penalty)
2   throws ClusteringException {
3 ...
4 while (dghRoots.size() > 1) {
5   ...
6   for (DGHEntry<?> current : dghRoots) { ...
7     for (DGHEntry<?> comparator : dghRoots) { ...
8     // Complete-Linkage mit Penalty
9     distance = current.fruthestDistance(comparator);
10    sizePenalty =
11      (current.getSize()+comparator.getSize() / (dgh.getDomainSize()*2));
12    distance += sizePenalty * penalty;
13    ...
14  }...}
15  dgh.merge(closestEntries.getFirst(), closestEntries.getSecond());
16  dghRoots = dgh.getRoots();
17 }...}
```

Die mit Hilfe eines *penalty*-Wertes erzeugte Domänengeneralisierungshierarchie in Abbildung 5.3 zeigt bereits eine relative Verbesserung des in Punkt 2 genannten Problems: Es ist gelungen, den Ausreißer-Wert um eine Ebene in den Baum zu verschieben. Größere Datenmengen würden diesen Effekt wahrscheinlich verstärken, was für ein besseres Ergebnis spricht.

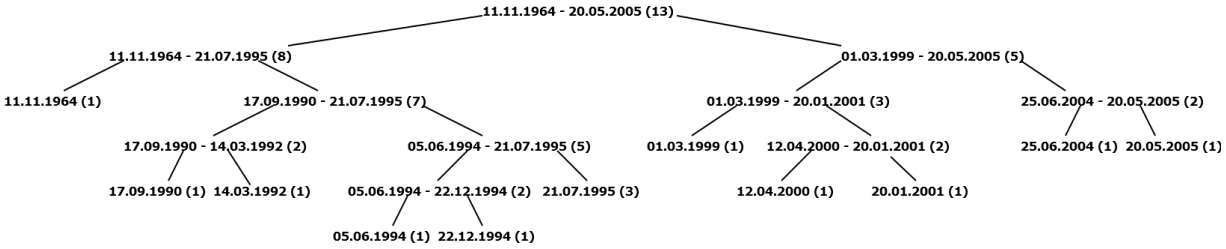


Abbildung 5.3: Anwendung von Complete-Linkag mit Hilfe von *Penalty*-Wert

5.1.2 Gleichverteiltes Clustering

Um eine Alternative für die problembehafteten Ergebnisse von Abschnitt 5.1.1 für numerische Wertebereiche zu ermitteln, wird ein weiteres Verfahren, das gleichverteilte Clustering, implementiert. Hier wird versucht, eine Menge von sortierten Einträgen in eine Menge von Clustern einzuteilen, sodass diese nach Möglichkeit immer die gleiche Größe besitzen. Die Implementierung dieses Verfahrens erfolgt im Klassendiagramm der Abbildung 5.1 durch die Klasse *GV_Clustering*, welche sich im Package `agvdgh.clustering` befindet.

Auch hier existieren mehrere Möglichkeiten das Verfahren zu parametrisieren. Dabei gelten die gleichen Bedingungen, die bereits in Aufzählung 5.1.1 formuliert wurden. Der zusätzliche ganzzahlige Parameter beschreibt die Größe der zu bildenden Cluster.

In Algorithmus 2 wird das implementierte Verfahren gezeigt. Sollte das verwendete *DGH*-Objekt nicht mehr als eine Wurzel besitzen, wird dies in Zeile 2 erkannt und es wird abgebrochen. Eine weitere Abbruchbedingung wird in Zeile 5 gezeigt. Sollte die der Funktion als Parameter übergebene Clustergröße 1 betragen, muss nur die Wurzel verschmolzen werden. Als nächstes, in Zeile 10, wird die Größe der zu bildenden Cluster angepasst. Dies passiert unter der Bedingung, dass bei der Einteilung in gleichgroße Cluster noch Einträge übrig bleiben würden. Sollen beispielsweise elf Einträge in vier Cluster eingeteilt werden, ist dies nicht ohne Weiteres möglich, da nach der Clusterbildung ein Überschuss von drei Einträgen bleiben würde. In diesem Beispiel ist es auch nicht möglich, diese drei Einträge nach und nach während der Clusterbildung einzeln in andere Cluster einzustreuen, da aus einer Menge von elf Einträgen mit einer Größe von 4 nur zwei Cluster gebildet werden würden, und so trotzdem ein Eintrag kein zugehöriges Cluster findet. Aus diesem Grund wird die Clustergröße in Zeile 11 in jedem Schritt um 1 erhöht und mit dem neuen Wert geprüft, ob eine gültige Clusterbildung möglich ist. Wird währenddessen erkannt, dass diese *nicht* existieren kann, werden in Zeile 13 einfach die aktuellen Wurzeln zu einer Ebene in der Domänengeneralisierungshierarchie verschmolzen.

Nachdem die eben genannten "vorbereitenden" Schritte vollzogen wurden, beginnt das eigentliche, schrittweise Clustering in der Schleife ab Zeile 20. Die Werte, welche in den Variablen *from* und *to* gespeichert werden, beschreiben dabei den von dem Array *sorted*, den sortierten Wurzelknoten, zu verschmelzenden Bereich. Die Fallunterscheidung in Zeile 21 beschreibt, wie sonst übrig bleibende Werte in die zu bildenden Cluster eingestreut werden. So würde, bei einer Wurzelmenge von elf Einträgen und

einer *gewünschten* Clustergröße von 4, der ermittelte Werte für *tmpSize* bei 5 liegen. Statt Cluster der Größe 4, würden nun also Cluster der Größe 5 gebildet werden. Dies würde jedoch auch bedeuten, dass noch ein Eintrag auf die Cluster verteilt werden muss, was nach der Anweisung in Zeile 21 passiert. Zu beachten ist auch, dass die übrigen Werte gleichmäßig auf die entstehenden Cluster verteilt werden sollte, da diese sich sonst bei mehrfacher Durchführung des Verfahrens am niedrigwertigen Bereich der Domänengeneralisierungshierarchie häufen.

Nach jedem Durchlauf, werden in den Zeilen 25 und 26 die Grenzen der darauffolgenden Verschmelzung um die Größe der zu erzeugenden Cluster verschoben, bis die Bedingung in Zeile 20 verletzt wird. Sobald dies passiert, wird das Verfahren erneut mit den gerade entstandenen Wurzelknoten durchgeführt.

Algorithmus 2 : Gleichverteiltes Clustering mit Hilfe der *DGH*-Struktur

```
   Input : DGH dgh, int clusterSize
   Output : DGH dgh
 1 roots ← getRoots(dgh);
 2 if |roots| == 1 then
 3 |   return;
 4 end
 5 if clusterSize == 1 then
 6 |   merge(dgh, roots);
 7 |   return;
 8 end
 9 tmpSize ← clusterSize;
10 while tmpSize nicht groß genug do
11 |   tmpSize ← tmpSize + 1;
12 |   if keine korrekte Größe möglich then
13 |   |   merge(dgh, sorted);
14 |   |   return;
15 |   end
16 end
17 sorted ← sort(roots);
18 from ← 0;
19 to ← tmpSize;
20 while to <= |sorted| do
21 |   if Zeit zum Einstreuen übriger Einträge then
22 |   |   to ← to + 1;
23 |   end
24 |   merge(dgh, sorted, from, to);
25 |   from ← to;
26 |   to ← to + tmpSize;
27 end
28 cluster(dgh, clusterSize);
```

Ergebnis

Ein Ausschnitt einer bei der Anwendung des Verfahrens erzeugte Domänengeneralisierunghierarchie ist in Abbildung 5.4 zu sehen. Hier sollte aus einer Menge von 229 Daten eine Hierarchie aufgebaut werden, in welcher die Cluster eine Größe von 8 besitzen. Da bei der hohen Anzahl an Einträgen eine vollständige

Angabe des Baumes nicht möglich gewesen wäre, werden die meisten Cluster nur durch Verbindungen angedeutet. Einige Parameter wurden durch die oben genannten Verfahren angepasst:

1. In der obersten Ebene existieren nur drei Cluster. Dieses Verhalten wird durch die Fallunterscheidung in Zeile 12 des Verfahrens 2 beschrieben: Durch die vorhergehende Clusterung war es nicht möglich ein Cluster der Größe 8 aufzubauen.

2. In der zweiten Ebene lässt sich erkennen, dass die präferierte Clustergröße nicht mehr 8, sondern 9 ist. Dies zeigt sich dadurch, dass zwei der drei vorhandenen Cluster der zweiten Ebene eine Größe von 9, und das übrige eine Größe von 10 besitzt. Die Größe wurde durch die Schleife in Zeile 10 geändert. Trotzdem blieb ein Eintrag übrig, welcher durch die Fallunterscheidung in Zeile 21 einem der Cluster zugeordnet wurde.

3. Die gezeigte dritte Ebene besteht wieder aus Clustern der Größe 8. Nach dem gleichen Prinzip, das im letzten Punkt genannt wurde, musste hier ein Element einem der Cluster zugeordnet werden, was die Größe von 9 in einem der Fälle erklärt.

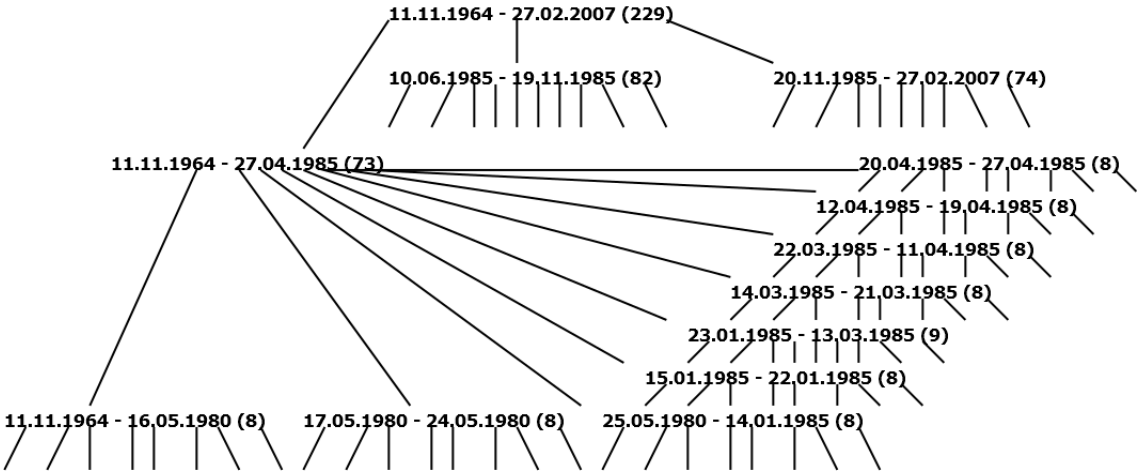


Abbildung 5.4: Durch gleichverteiltes Clustering erzeuge Domänengeneralisierunghierarchie

Die durch gleichverteiltes Clustering erzeugte Domänengeneralisierunghierarchie ist, entgegen dem durch das hierarchische Clustering erzeugten Ergebnis, ein balancierter Baum, was sie kompatibel zu den meisten Generalisierungsverfahren machen sollte. Auch ist es nicht durch verschiedene Modi und *Penalty*-Werte modifizierbar, was es im Vergleich zum hierarchischen Clustering einfacher macht. Der Nachteil ist jedoch, dass nur Attribute mit eindimensionalem Wertebereich korrekt geclustert werden können. Das bedeutet, dass es beispielsweise für Clusterung anhand von GPS-Daten, welches später noch durchgeführt wird, nicht einsetzbar ist.

5.2 DGH

Die Klasse *DGH* stellt ist die Implementierung einer Domänengeneralisierungshierarchie dar. Sie bietet zu jeder Zeit Zugriff auf die *DGHEntry*-Einträge ihrer Blätter sowie ihrer Wurzeln, welche bei korrekten Aufbau auch implizit auf die Blätter verweisen sollten. Beide werden in globalen HashSets des Objektes gespeichert, auf die mit entsprechenden Methoden zugegriffen werden können.

Des Weiteren existieren in einem *DGH*-Objekt standardmäßig auch Einträge für die höchsten und niedrigsten Werte ihres Wertebereichs sowie dessen Größe, welche für genormte Distanzberechnung benötigt wird. Diese werden jeweils in den globalen Variablen *DOMAIN_HIGHEST_VALUE*, *DOMAIN_LOWEST_VALUE* und *DOMAIN_SIZE* gespeichert. Auch hier wird Zugriff durch entsprechende Methoden gewährleistet.

Initialisiert wird ein *DGH*-Objekt durch dessen Konstruktor *DGH(String, Collection, Class)*. Der erste Parameter wird als Name aufgefasst. Dies ist nötig für die spätere Sicherung einer generierten Domänengeneralisierunghierarchie in eine Datenbank. Die aufgelistete Menge sind Blatteinträge des Generalisierungsbaumes, aus denen eine Hierarchie aufzubauen ist. Der letzte Parameter, ein Klassen-Objekt, ist die Implementierung einer *DGHEntry*, welche Funktionen für die Clusterung bereitstellt. Während der Initiierung einer *DGH* wird für jedes in der *Collection* übergebene Element ein Objekt der entsprechenden *DGHEntry*-Implementierung erzeugt. Folgend werden die globalen Mengen *roots* und *leafs* mit diesen befüllt.

5.2.1 Verschmelzung von Einträgen

Die Blatt- und Wurzelmengen des *DGH*-Objektes sind anfangs noch identisch. Eine gültigen Domänengeneralisierunghierarchie wird dadurch erzeugt, dass Einträge der Wurzelknoten nacheinander verschmolzen werden. Dies geschieht durch die in Abbildung 5.1 gezeigten Methoden *merge(DGHEntry, DGHEntry)* sowie *merge(DGHEntry[])*. Die Anwendung sollte sofort klar sein: Die letztgenannte Funktion verbindet beliebig viele Einträge miteinander, während die erste lediglich mit zwei arbeitet.

Die Implementierung der mächtigeren Version der *merge*-Funktion ist in Beispiel 5.2 zu sehen. Auch wird hier gezeigt, wie die alleinige Anwendung dieser Funktion die Domänengeneralisierunghierarchie in jedem Aufruf aufbaut: Die als Parameter übergebenen Objekte werden aus den aktuellen Blattknoten entfernt, und die Vereinigung all dieser Knoten wird zum Schluss zurück in die Blattknoten eingefügt. Der Typ T ist hierbei die Implementierung der verwendeten *DGHEntry*, die auch bei der Initiierung des zugehörigen *DGH*-Objektes angegeben wurde.

Listing 5.2: Implementierung von merge(DGHEntry, DGHEntry)

```
1 public T merge(DGHEntry<?>[] entries) {
2   for (DGHEntry<?> e : entries)
3     roots.remove(e);
4
5   T newInst = newInstance(entries);
6   roots.add(newInst);
7
8   return newInst;
9 }
```

5.2.2 Erzeugung neuer *DGHEntry*'s

Aus dem genannten Typ kann mit Hilfe der Java-Bibliothek und dem DGH-Objekt übergebenen Typ auch leicht ein neues *DGHEntry*-Objekt erzeugt werden. Das verwendete Verfahren dafür ist seinerseits in Beispiel 5.3 zu sehen.

Listing 5.3: Erzeugen einer neuen *DGHEntry*-Instanz

```
1 private T newInstance(DGHEntry<?> e1, DGHEntry<?> e2) {
2   try {
3     Constructor<T> ctor = (Constructor<T>)
4       type.getConstructor(DGH.class , DGHEntry.class, DGHEntry.class);
5     return ctor.newInstance(new Object[] {this, e1, e2});
6   } catch ...
7 }
```

5.2.3 Komprimierung von Einträgen

Wenn für einen großen Datensatz eine Domänengeneralisierunghierarchie erzeugt werden soll, kann es natürlich vorkommen, dass unter den entstehenden Einträgen Duplikate existieren. Für Postleitzahlen beispielsweise kann bei genügend Datenvielfalt fast garantiert werden, dass Werte mehrfach vorkommen. Ein daraus entstehendes Problem ist, dass in Verfahren wie dem hierarchischen Clustering auch diese Werte geclustert werden müssen und damit Duplikate in der Domänengeneralisierunghierarchie auftauchen. Sollte die verwendete *DGHEntry*-Implementierung Ontologien nutzen verschlimmert sich dieses Verhalten nur, da möglicherweise jeder Eintrag in einer Datenbank abgefragt werden muss. Aus diesem Grund wurde eine Methode in der *DGH*-Klasse implementiert, welche Duplikate zu einer einzelnen *DGHEntry* zusammenfasst. Diese wird *compressEntries()* genannt und ist auch in Abbildung 5.1 zu sehen.

5.2.4 Speicherung in einer Datenbank

Die letztendliche Speicherung wird von der Methode *saveToDB()* bereitgestellt, welche ihrerseits auch in Abbildung 5.1 zu sehen ist. Wie in Kapitel 4 bereits erwähnt, wird das von Martin Müller [Mül16] entworfene SQL-Schema aus Abbildung 4.1 verwendet. Dieses wird jedoch um die Angabe eines aktuellen Datums erweitert. Dies ermöglicht, dass in die Datenbank eingetragene Domänengeneralisierungshierarchien nur in bestimmten zeitlichen Abständen erneuert werden. Da das einzutragende Datum automatisch generiert wird, ist das Schema weiterhin mit Müllers Entwurf kompatibel. Der Eintrag für den zur Domänengeneralisierungshierarchie gehörenden SQL-Typ des Attributes, welcher laut Schema 4.1 vorausgesetzt wird, befindet sich abrufbar in den Implementierungen der jeweiligen *DGHEntry*. Aktuell als Datenbanksystem unterstützt wird nur *MySQL*, wobei die Unterstützung für weitere Datenbanksysteme leicht zu implementieren ist.

5.3 DGHEntry

DGHEntry wird als abstrakte, generische Klasse implementiert, dessen Subklassen mit dem Typ parametrisiert werden, der durch die Klasse dargestellt werden soll. Die für diese Software implementierte Version eines Eintrags für Integer, die Klasse *IntegerEntry*, ist also vom Typ *DGHEntry<Integer>*.

In der Klasse werden verschiedene Informationen gespeichert. So besitzt jede *DGHEntry* einen maximalen, minimalen und durchschnittlichen Wert ihres Typs, der das Objekt beschreibt. Eine Referenz auf das zugehörige *DGH*-Objekt ermöglicht das Überprüfen und Erneuern der Eigenschaften des dort eingetragenen Wertebereichs. Dies geschieht über die Funktion *updateDomainValues()*, welche zur Initiierung der *DGH* auf jedem ihrer Einträge aufgerufen wird und von den Subklassen der *DGHEntry* implementiert wird. Dort werden lediglich der aktuell höchste und niedrigste eingetragene Wert des Wertebereichs abgerufen und, falls die Werte des neu erzeugten Objekts größer, bzw. kleiner sind, einer dieser übernommen sowie der Wert *DOMAIN_SIZE* der *DGH* neu berechnet.

Weiterhin enthält jede Instanz Verweise auf weitere *DGHEntry*'s, die lokal in einem *HashSet* gespeichert werden. Über diese kann von den Wurzeln des *DGH*-Objektes aus eine Domänengeneralisierungshierarchie ausgelesen werden. Zusätzlich wird die Größe der in ihr repräsentierten Attribute in jeder Instanz der *DGHEntry* mitgeführt. Im Folgenden sollen noch die wichtigsten Funktionen der Klasse erklärt werden.

5.3.1 Distanzberechnung

Die Berechnung der Distanzen der von *DGHEntry*'s dargestellten Clustern befindet sich in deren Implementierungen. Die dafür bereitgestellten Funktionen sind mit den Namen *averageDistance(DGHEntry)*, *closestDistance(DGHEntry)* und *furthestDistance(DGHEntry)* in Abbildung 5.1 gezeigt. Durch die in jeder Instanz vorhandenen Werte *avgValue*, *highValue* und *lowValue* ist dies sehr effizient möglich. In Beispiel 5.4 ist die Funktion *closestDistance(DGHEntry)* der Implementierung *IntegerEntry* zur Erklärung der Funktionsweise aufgezeigt.

Listing 5.4: Funktion *closestDistance()* in *IntegerEntry*

```
1 public double closestDistance(DGHEntry<?> e) {
2   IntegerEntry ie = (IntegerEntry) e;
3   if (ie.getAverageValue() > avgValue)
4     return ie.getLowValue() - highValue;
5   else
6     return lowValue - ie.getHighValue();
7 }
```

5.3.2 Umfang eines Eintrags

Da sie unter Umständen viele Einträge umfassen, werden Cluster in Abbildungen wie 5.2, 5.3 und 5.4 nur durch ihrer Umfang dargestellt. Dieser lässt sich leicht durch die in ihnen gespeicherten Werten ermitteln. Die Generierung des Umfangs findet in der Funktion *range()* statt. Dieser einfache Vorgang ist in Beispiel 5.5 zu sehen.

Listing 5.5: Funktion *range()* in *DGHEntry*

```
1 public String range() {
2   if (isCluster())
3     return lowValue.toString() + " - " + highValue.toString();
4   else
5     return avgValue.toString();
6 }
```

Diese Funktion wird von den meisten der Implementierungen der *DGHEntry* verwendet. Sobald jedoch mehrdimensionale Datentypen implementiert werden, ist diese möglicherweise nicht mehr ausreichend. Zu sehen ist dieses Problem in der Klasse *LocationEntry*, in Beispiel 5.6, welche die Clusterung von Postleitzahlen anhand der von OpenGeoDB bereitgestellten Ontologie mit GPS-Daten ermöglicht. Da die dort entstehenden Cluster nicht mehr untereinander zu ordnen sind, müssen ihre Einträge entweder nacheinander, oder durch von der Ontologie bereitgestellten Werte ausgegeben werden.

Listing 5.6: Funktion *range()* in *LocationEntry*

```
1 public String range() {
2   String range = "";
3   if (DGH.compressed)
4     range = avgValue.getName();
5   else {
6     Iterator<Integer> it = getPLZs().iterator();
7     range = "" + it.next();
8     while (it.hasNext())
9       range = range + "," + it.next();
10  }
11  return range;
12 }
```

5.4 LocationEntry

Die Implementierung der Klasse *LocationEntry*, welche vom Typ *Location*, zu sehen in Abbildung 5.1, ist, stellt insofern einen Sonderfall dar, da sie aktuell die einzige der Implementierungen von *DGHEntry* ist, welche eine Ontologie nutzt. Durch diese werden Postleitzahlen mit Hilfe der von OpenGeoDB [Ope15] bereitgestellten GPS-Daten geclustert. Durch die zweidimensionale Eigenschaften der Längen- und Breitengrade muss das Framework um zusätzliche Wertebereich-Angaben erweitert werden, da die derzeit implementierten Felder *DOMAIN_HIGHEST_VALUE* und *DOMAIN_LOWEST_VALUE* nicht für beide dieser Werte genügen. Dies ist leicht möglich, indem in der Klasse *LocationEntry* statische Variablen hinzugefügt werden, welche in diesem Fall *DOMAIN_HIGHEST_VALUE_LON* und *DOMAIN_LOWEST_VALUE_LON* genannt wurden. Die Funktion *updateDomainValues()* wird entsprechend ergänzt. Zusätzlich ist es nötig, dass weitere Funktionen der *DGHEntry* an die Verwendung einer Ontologie angepasst werden. Diese sind folgend erwähnt. Da im Kontext der Verwaltung eines zweidimensionalen Wertebereichs die in den *DGHEntry*-Instanzen gespeicherten Werte *highValue* und *lowValue* an Bedeutung verlieren, arbeitet die Klasse *LocationEntry* lediglich mit *avgValue* um ihren Zustand zu speichern.

5.4.1 Distanzberechnung

Die Distanzberechnung, welche in anderen Implementierungen der *DGHEntry* sehr einfach durch Nutzung der Werte *avgValue*, *highValue* und *lowValue* geschieht, kann in *LocationEntry* aufgrund des zweidimensionalen Wertebereichs nicht mehr mit dem gleichen Verfahren geschehen. Stattdessen müssen hier alle Einträge, dessen Distanz zueinander bestimmt werden soll, miteinander verglichen werden. Dies ist zwar nicht viel komplizierter als das vorher verwendetet Verfahren, jedoch entsprechend komplexer. Da die Distanz nach wie vor genormt zurückgegeben werden soll, gilt es folgend auch *DOMAIN_SIZE* so zu implementieren, dass die Größe eines zweidimensionalen Wertebereichs dargestellt wird.

5.4.2 *updateDomainValues()*

Wie vorher bereits erwähnt, wird das Framework durch die Klasse *LocationEntry* so erweitert, dass die höchsten und niedrigsten Werte für Längen- und Breitengrad jeweils aufgezeichnet werden. Dies passiert nach wie vor durch Vergleiche und Neuzuweisung von Werten in der Funktion *updateDomainValues()*, welche beim Erzeugen von *DGHEntry*'s aufgerufen wird. Die Berechnung des genormter Wertes *DOMAIN_SIZE* erfolgt dabei wie in Beispiel 5.7.

Listing 5.7: Berechnung von DOMAIN_SIZE in *LocationEntry*

```
protected void updateDomainValues() {
  if (domainValuesSet) {
    ...
    double latLength = DGH.DOMAIN_HIGHEST_VALUE - DGH.DOMAIN_LOWEST_VALUE;
    double lonLength = DOMAIN_HIGHEST_VALUE_LON - DOMAIN_LOWEST_VALUE_LON;
    DGH.DOMAIN_SIZE = Math.sqrt(latLength*latLength + lonLength*lonLength);
  } else {
    ...
    domainValuesSet = true;
  }
}
```

5.4.3 *isCluster()*

In der abstrakten Klasse *DGHEntry* ist diese Funktion bereits so implementiert, dass die lokalen Werte *highValue* und *lowValue* miteinander verglichen werden. Unterscheiden sich beide Werte voneinander, handelt es sich nach dieser Implementierung um ein Cluster, da scheinbar mehr als ein Wert enthalten ist. Weil *LocationEntry* jedoch vom Typ *Location*, einer Hilfsklasse zur Anbindung an die Ontologie und Darstellung von Orten, ist, reicht dieses Verfahren nicht mehr aus. Da die Klasse *Location* ihre enthaltenen Postleitzahlen in einer Menge speichert und diese Menge leicht abrufbar ist, kann auf Basis ihrer Größe bestimmt werden, ob es sich bei einer Instanz von *LocationEntry* um ein Cluster handelt, oder nicht: Existiert nur ein Wert in ihr, dann ist es kein Cluster.

5.4.4 *range()*

Auch *range()* wird bereits von *DGHEntry* implementiert, reicht jedoch nicht für den Datentyp *Location*. Bei eindimensionalen Wertebereichen reicht die Darstellung des Umfangs eines Clusters durch Angabe des höchsten und niedrigsten enthaltenen Wertes, bei mehrdimensionalen Wertebereichen jedoch nicht. Die Ausgabe eines Namens wird durch die Klasse *Location* mit der Funktion *getName()* bereitgestellt. Auch die enthaltenen Postleitzahlen können mit Hilfe der Funktion *getPLZs()* ausgegeben werden. Hieraus lässt sich ein einfaches Verfahren ableiten – zu sehen in Algorithmus 3 – um die Darstellung des Umfangs des Clusters nach Möglichkeit übersichtlich zu halten.

Algorithmus 3 : Ausgabe des Umfangs eines *LocationEntry*-Clusters

```
   Input : DGH dgh, int clusterSize
   Output : String range
 1 if DGH ist komprimiert then
 2 |  range ← getName(avgValue);
 3 end
 4 else
 5 |  PLZs ← getPLZs(avgValue);
 6 |  range ← next(PLZs);
 7 |  while weitere Postleitzahlen enthalten do
 8 |  |  range ← range + ',' + next(PLZs);
 9 |  end
10 end
11 return range;
```

In Zeile 2 ist zu sehen, wie ein von der Ontologie bereitgestellter Name verwendet wird. Aufgrund des zur Namenszuweisung implementierten Verfahrens ist dies jedoch erst bei vollständig generierten Domänengeneralisierungshierarchien möglich. Ein Indikator dafür ist, dass die Einträge bereits komprimiert wurden, weswegen die Funktion *getName()* nur in dieser Fallunterscheidung aufgerufen wird. Ist dies jedoch nicht der Fall, werden stattdessen die Postleitzahlen nacheinander und durch ein Komma getrennt ausgegeben.

5.4.5 *finish()*

Diese Funktion wurde extra für *LocationEntry* in *DGHEntry* implementiert und dient für die Nachbearbeitung von Werten in der *DGH*-Struktur. Sie ist zwar in jeder Implementierung von *DGHEntry* enthalten, wird jedoch nur von *LocationEntry* genutzt.

Die Notwendigkeit für eine Nachbearbeitung einzelner Einträge hat sich daraus ergeben, dass während der Clusterung von *LocationEntry*-Einträge Postleitzahlen noch nicht zuverlässig durch Ortsnamen substituiert werden konnten. Das Problem wird durch ein kurzes Beispiel erläutert: Eine Menge von vier Postleitzahlen {18055, 18057, 18069, 12487} soll hierarchisch geclustert werden. Als erstes verschmelzen die Einträge 18055 und 18057, dann {18055, 18057} mit 18069 und schließlich das entstandene Cluster mit 12487. Da das Verfahren nur lokal arbeitet, kann nicht zu Zeitpunkt der Clusterung ein Name für die Substitution der Postleitzahlen verwendet werden. Grund dafür ist, dass es weiß nicht, ob dieser Name bereits in der ungeclusterten Menge eines Schrittes eingesetzt werden kann. Bei dem ersten Schritt, der die Menge {18055, 18057, 18069, 12487} clustert, ist beispielsweise nur für den Eintrag 12487 die Substitution zu „Berlin“ in Ordnung. Für die restlichen Postleitzahlen ist dies erst fehlerfrei möglich, sobald das Cluster {18055, 18057, 18069} entstanden ist, welches sich zu „Rostock“ generalisieren lässt. Zu vermeiden ist der Fall, dass diese Bezeichner in der Domänengeneralisierungshierarchie mehrdeutig sind. Deshalb muss das *DGH*-Objekt erst mit Hilfe des Clusterings aufgebaut und danach die Werte teilweise durch Ortsnamen ersetzt werden.

Dies wird mit Hilfe der hier erklärten Funktion ermöglicht. So ist *finish()* nach jeder Durchführung des Clusterings in der entsprechenden Klasse (in Abbildung 5.1 die Klassen *H_Clustering* sowie *GV_Clustering*) auf dem zugehörigen *DGH*-Objekt aufgerufen. Dieses ruft seinerseits *finsish()* auf jedem seiner Cluster auf. Dieser Vorgang ist in Beispiel 4 zu sehen. In Zeile 2 wird für alle in dem *DGH*-Objekt vorkommenden Ortsnamen eine Hierarchie von Verallgemeinerungen aufgebaut, mit denen ein entsprechender Name generalisiert werden kann. Dies geschieht über die Klasse *Location* und wird später weiter erläutert. Falls der derzeitige Eintrag kein Cluster ist, also nur eine Postleitzahl beschreibt, wird keine Generalisierung ausgeführt, was in Zeile 4 durch die Fallunterscheidung gezeigt wird. Falls jedoch mehrere Postleitzahlen in dem Eintrag vorhanden sind, muss aus diesen ein generalisierter Name ermittelt werden, der alle enthaltenden Postleitzahlen am genauesten beschreibt. Sollen zum Beispiel Cluster von Rostocker und Schweriner Postleitzahlen miteinander verschmolzen werden, dann sollte das entstehende Cluster nicht durch den allgemeineren Begriff „*Bundesrepublik Deutschland*“, sondern viel eher durch „*Mecklenburg-Vorpommern*“ substituiert werden. Um diesen Namen zu ermitteln, wird ab Zeile 9 aus dem zur ersten Postleitzahl gehörendem Ortsnamen eine Liste generiert, welche eine Hierarchie bis zum allgemeinsten in der Ontologie vorkommenden Namen repräsentiert. Im Falle einer Rostocker Postleitzahl wäre dies {„*Rostock*“ → „*Mecklenburg-Vorpommern*“ → „*Bundesrepublik Deutschland*“}. Bei dem Vergleich dieser Liste mit der Hierarchie einer Schweriner Postleitzahl, {„*Schwerin*“ → „*Mecklenburg-Vorpommern*“ → „*Bundesrepublik Deutschland*“}, wird „*Mecklenburg-Vorpommern*“ als geeignetes Element ermittelt. Der zugehörige Index wird dann gespeichert und für den Fall, dass eine allgemeinere Generalisierung benötigt wird, ab Zeile 17 erhöht. Falls der durch dieses Verfahren ermittelte Name noch nicht in dem *DGH*-Objekt vorkommt, wird er für den aktuellen Eintrag festgelegt, was in

der Fallunterscheidung in Zeile 25 zu sehen ist. Schlussendlich wird die Funktion in Zeile 31 auch auf allen Vorgängern des aktuellen Eintrags aufgerufen.

Algorithmus 4 : Implementierung von *finish()* in *LocationEntry*

```
   Input : Location avgValue
 1 if Ontologie noch nicht aufgebaut then
 2 |   buildOntologie(Alle zu den Einträgen der DGH gehörende Ortsnamen);
 3 end
 4 if Eintrag ist kein Cluster von Einträgen then
 5 |   name ← getPLZ(avgValue);
 6 end
 7 else
 8 |   for name in getNames(avgValue) do
 9 |   |   if listComparator ist leer then
10 |   |   |   listComparator ← erzeuge aus 'name' Hierarchie für Generalisierung;
11 |   |   |   genIndex ← 0;
12 |   |   |   continue();
13 |   |   end
14 |   |   else
15 |   |   |   listGen ← erzeuge aus 'name' Hierarchie für Generalisierung;
16 |   |   end
17 |   |   for name in listGen do
18 |   |   |   if name in listComparator enthalten ∧ Index < genIndex then
19 |   |   |   |   genIndex ← Index von name in listComparator;
20 |   |   |   |   break();
21 |   |   |   end
22 |   |   end
23 |   end
24 |   name ← get(listComparator, genIndex);
25 |   if name kommt in DGH noch nicht vor then
26 |   |   thisname ← name;
27 |   end
28 |   else
29 |   |   thisname ← String aus enthaltenen Postleitzahlen;
30 |   end
31 |   for entry in predecessors do
32 |   |   finish(entry);
33 |   end
34 end
```

5.4.6 Ergebnis

In Abbildung 5.5 ist eine mit Hilfe der *DGHEntry*-Implementierung *LocationEntry* erzeugte Domänengeneralisierungshierarchie zu sehen. Verwendet wurden dafür die in den Blatteinträgen gezeigten Postleitzahlen und das in Kapitel 5.1.1 beschriebene hierarchische Clustering mit Single-Linkage und einem *penalty*-Wert von 0.05. Es hat sich gezeigt, dass der *penalty*-Wert in etwa so groß seien sollte, wie die kleinste durch die Postleitzahlen repräsentierte Stadt auf der Karte, sprich, wenn Rostock auf dem durch die Postleitzahlen spezifizierten Bereich ca. 5% der Karte einnimmt, dann sollte dieser auch bei ca. 0.05

liegen. Bei zu großen *penalty*-Werten verschieben sich die Clusterbildung, sodass zuerst Postleitzahlen aus verschiedenen Städten miteinander verschmolzen werden. Dies ist unbedingt zu vermeiden.

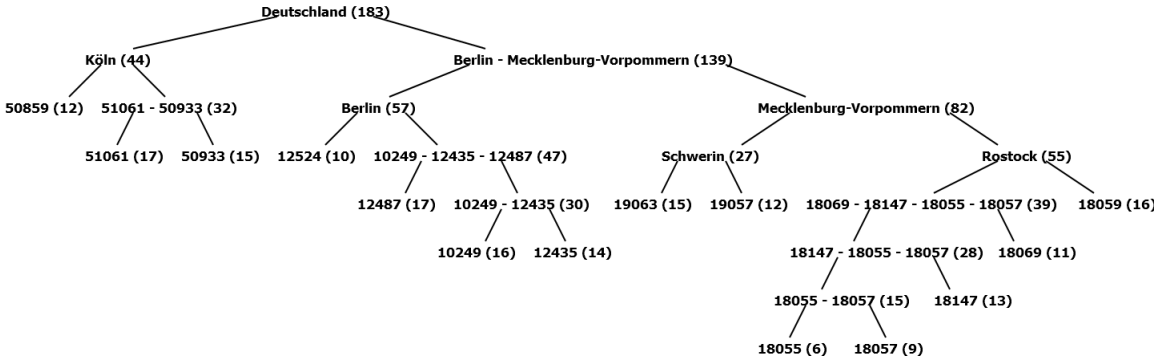


Abbildung 5.5: Domänengeneralisierungshierarchie, erzeugt mit *LocationEntry*

5.5 Utility

Die hier enthaltenen Klassen bieten weitere Funktionen, welche aus verschiedenen Gründen nicht in anderen Packages implementiert wurden. Darunter ist die Klasse *Location*, welche in Kapitel 5.4 bereits erwähnt wurde, die Adapterklasse *DGHAdapter*, welche für die Einbindung in das PArADISE-Projekt [GH16] genutzt wird, *GPS* zur Auswertung von GPS-Daten, und das im hierarchisch Clustering verwendete *DGHEntryPair*. Zwei dieser Implementierungen, *Location* und *DGHAdapter* werden folgend erläutert.

5.5.1 *Location*

In dieser Klasse werden Postleitzahlen der *LocationEntry*'s verarbeitet. So verfügt jede Instanz über Angaben des Längen- und Breitengrades, enthaltene Postleitzahlen und dazugehörige Namen. Auch wird statisch in Tabellen festgehalten, zu welchen Ortsnamen welche Postleitzahlen gehören und wie diese in hierarchischen Relationen zu anderen Ortsnamen stehen. All diese Informationen stammen dabei aus der Ontologie von OpenGeoDB [Ope15].

Ein *Location*-Objekt wird mit Hilfe der Angabe einer Postleitzahlen initiiert. Für diese wird eine SQL-Anfrage generiert, welche GPS-Informationen und den zugehörigen Ortsnamen abfragt. Für den Fall, dass für eine Postleitzahl mehrere Längen- und Breitengrade im Datenbestand eingetragen sind, wie beispielsweise für den Wert 18069, welcher in Rostock sowohl für *„Reutershagen“* als auch für *„Lambrechtshagen“* stehen kann, wird aus allen vorkommenden Einträgen ein Durchschnitt gebildet. Weiterhin können Instanzen auch durch direkte Angabe der GPS-Daten, Postleitzahlen und Ortsnamen erzeugt werden.

Die Funktion *buildOntologies(Set<String>)*, welche auch in Beispiel 4 angewendet wurde, nutzt die in der Ontologie vorkommenden Hierarchieinformationen über Ortsnamen um eine Tabelle für deren Generalisierung aufzubauen. Dabei muss darauf geachtet werden, dass alle Einträge eindeutig sind. So verfügt die Ontologie von OpenGeoDB über mehrere Einträge für den Ort *„Rostock“*: Zum Beispiel *„ROSTOCK“* und *„Hansestadt Rostock“*. Dies wird gelöst, indem nur bekannte Begriffe einer Hierarchieebene genutzt werden – existiert bereits ein Eintrag für *„Rostock“*, wird keiner der anderen

Einträge mehr gespeichert. Zum Abfragen der Generalisierung eines Ortsnamens kann die Funktion *getGeneralizedName(String)* genutzt werden.

5.5.2 *DGHAdapter*

Die Klasse *DGHAdapter* wird aktuell direkt im PArADISE-Projekt eingebunden, wo sie aus den Attributen, für die eine Domänengeneralisierungshierarchie erzeugt werden soll, ein *DGH*-Objekt der zugehörigen *DGHEntry* initiiert und schlussendlich einen passenden Clustering-Algorithmus anwendet, um die Struktur aufzubauen. So muss der Funktion *adaptToAdult(...)* als Parameter eine Datenbankverbindung, der aktuelle Tabellenname und eine Liste von Quasi-Identifikatoren übergeben werden, damit die Generierung der Domänengeneralisierungshierarchie beginnt. Dabei wird darauf geachtet, dass automatische Generierung pro Attribut nur ein Mal stattfindet.

Derzeit werden zu Attributen gehörende *DGHEntry*-Implementatierungen statisch in der Klasse *DGHAdapter* durch eine Tabelle festgehalten. Einige Einträge für die Attribute, welche in der adult-Datenbank des PArADISE-Projekts vorkommen, sind angelegt:

{{*„fnlwgt“*, *„DoubleEntry“*},
{*„capitalGain“*, *„DoubleEntry“*},
{*„capitalLoss“*, *„DoubleEntry“*},
{*„age“*, *„IntegerEntry“*}}

Soll für ein Attribut eine Domänengeneralisierungshierarchie erzeugt werden, wird als erstes die zugehörige *DGHEntry*-Implementierung ermittelt, falls eine solche existiert. Durch eine Fallunterscheidung wird dann ein *DGH*-Objekt initiiert, ein passendes Clustering-Verfahren darauf angewendet und das Ergebnis in einer Datenbank gespeichert. Ausschnitte dieses einfachen Adapters sind in Beispiel 5.8 zu sehen.

Listing 5.8: Adapter für die automatische Generierung von DGHs

```
1 private static void generateDGH(String dghname, String type,
  IRemoteResultSet rset)
2     throws RemoteException, SQLException {
3   ...
4   switch (type) {
5     case "IntegerEntry":
6       HashSet<Integer> iset = new HashSet<Integer>();
7       // Fülle 'iset' mit Attributen aus der Datenbank
8       ...
9       dgh = new DGH<IntegerEntry>(dghname, iset, IntegerEntry.class);
10      GV_Clustering.cluster(dgh, 10);
11      break;
12
13    case "DoubleEntry":
14      ...
15      break;
16
17    ...
18
19    default:
20      System.err.println("No auto-DGH implementation for '" + type +"'");
21      break;
22  }
23  if (dgh != null) dgh.saveToDB();
24 }
```

Kapitel 6

Zusammenfassung und Ausblick

Die Erhaltung der Privatsphäre von Personen ist nach wie vor ein großes Problem der sich stetig fortentwickelnden Informationsgesellschaft – wo mit Daten wie mit Gold gehandelt wird, geraten Betroffene immer häufiger unter die Räder. Vorliegende Fälle des Datenmissbrauchs, in Absatz 1.2 beispielhaft aufgezeigt, sind längst kein Phänomen von gestern. Tagtäglich findet dieser statt, auch wenn Fortschritte im Bereich des Datenschutzes gemacht werden.

Von diesen wurde eine Auswahl vorgestellt: Kapitel 2 und 3 zeigen verschiedene Techniken zur Erhaltung der Privatsphäre. Darunter befanden sich Systeme wie zum Beispiel Datafly und die sich daraus entwickelten Anonymitätsmaße, vorgestellt in Absatz 2.1 und 2.2. Um Datensätze zu anonymisieren werden Werte in weniger konkrete Wertebereiche eingeteilt. Ein gemeinsames Problem im Bereich der Generalisierung von Informationen war bisher der Umgang mit kategorischen Werten: Diese können, entgegen der numerischen Werte, nicht natürlich zusammengefasst werden. Dieser Umstand wird in existierenden Systemen, die solche Daten verarbeiten, mit sogenannten Domänengeneralisierungshierarchien, erklärt in Absatz 3.1, gelöst. In den meisten Fällen werden diese statisch eingetragen, was unter Umständen zu Fehlern, wie ungleichmäßig generalisierten Datensätzen, führen kann. Ein solcher Umgang mit Wertebereichen kann zur Folge haben, dass bei sich verändernden Datenbeständen die Qualität der Generalisierung stark abnimmt.

Diese Fehlerquelle diente als Motivation für die vorliegende Arbeit. Es war Ziel herauszufinden, wie sich solche Generalisierungshierarchien automatisch generieren lassen. Dafür sollten verschiedene Clustering-Algorithmen und Ontologien für kategorische Wertebereiche betrachtet werden. Hierarchisches Clustering sowie das k-Means-Verfahren, beschrieben in den Absätzen 2.3 und 3.3, standen dabei zur engeren Auswahl. Zusätzlich zum hierarchischen Clustering, auf dessen Implementierung die Wahl gefallen ist, wurde ein weiteres Verfahren, das gleichverteilte Clustering, implementiert. Im Zuge der Erkenntnisgewinnung zum Thema der Ontologien, wurden in Absatz 3.2 mehrere Semantic-Web-Technologien erarbeitet. Beispielhaft implementiert wurde die Ontologie von OpenGeoDB [Ope15], welche hier zur Clusterung von Postleitzahlen anhand dazugehöriger GPS-Daten dient.

Der in Kapitel 4 beschriebene Entwurf sowie die daraus entstandene Implementierung in Kapitel 5 ermöglichen eine hohe Erweiterbarkeit des gewählten Ansatzes. So wird die Generalisierungshierarchie über eine Struktur aufgebaut, auf der sich verschiedene Clustering-Algorithmen ausführen lassen. Des Weiteren wird diese aus Implementierungen einer abstrakten Klasse aufgebaut, sodass sich das Verfahren leicht um zusätzliche Datentypen ergänzen lässt. Erzeugte Domänengeneralisierungshierarchien lassen sich auch einfach auslesen und für spätere Verwendung in Datenbanken abspeichern. Durch die Software erzeugte Ergebnisse sind in Abbildung 5.2, 5.3, 5.4 und 5.5 zu sehen.

Die hohe Erweiterbarkeit bietet auch Einstiegsmöglichkeiten für künftige Problemstellungen. Da sie durch Implementierungen der Domänengeneralisierungshierarchieeinträge realisiert werden, ist es leicht möglich weitere, wie beispielsweise die durch [Gyr17] zum Thema Internet-of-Things bereitgestellten, Ontologien in das Projekt einzubinden. Auch ist das Hinzufügen neuer Clustering-Techniken einfach, da diese nur die bestehende Struktur der Generalisierungshierarchie Schritt für Schritt durch bereits implementierte Funktionen anpassen.

Auch das vor allem in Abbildung 5.5 zu sehende Problem, das der unbalancierten Bäume, ließe sich durch weitere Modifikationen am hierarchischen Clustering, oder durch vollkommen andere Algorithmen, wahrscheinlich lösen. Alternativ sollte es auch möglich sein, Blatteinträge einer solchen Domänengeneralisierungshierarchie durch das Hinzufügen weiterer Knoten künstlich zu balancieren. Insbesondere das Problem der Kompatibilität mit verschiedenen Generalisierungsverfahren würde so gelöst werden können.

Letztendlich sollte es aber auch möglich sein Domänengeneralisierungshierarchien unabhängig davon, ob sie balanciert sind oder nicht, für die Anonymisierung von Daten zu nutzen. Inwiefern dies jedoch in der Praxis möglich ist, wurde durch diese Arbeit nicht behandelt. Abbildung 5.5 zeigt jedoch, dass abhängig von der Art und Menge der zu generalisierenden Einträge mit der Möglichkeit gerechnet werden sollte, dass auf natürlichem Wege keine balancierten Bäume zu Stande kommen. Aus diesem Grund gilt es zu untersuchen, ob und wie es möglich ist Datenbestände so zu generalisieren, dass dafür nicht ganze Baumebenen in jedem Schritt genutzt werden.

Neben offensichtlichen Erweiterungsmöglichkeiten der Software sollte diese noch vollständig in das bestehende PArADISE-Projekt der Universität Rostock [GH16] eingebunden werden. Dieses nutzt die Software zwar bereits zur automatischen Generierung und Speicherung der Domänengeneralisierungshierarchien, verwendet sie aber noch nicht zur Generalisierung von Datenbankanfragen. Zusätzlich dazu ist die implementierbare Unterstützung weiterer auf den Server der Universität installierten Datenbanksysteme zu betrachten.

Literaturverzeichnis

[BKBL07] Byun, Ji-Won, Ashish Kamra, Elisa Bertino und Ninghui Li: *Efficient* k *- Anonymization Using Clustering Techniques.* In: Ramamohanarao, Kotagiri, P. Radha Krishna, Mukesh K. Mohania und Ekawit Nantajeewarawat (Herausgeber): *Advances in Databases: Concepts, Systems and Applications, 12th International Conference on Database Systems for Advanced Applications, DASFAA 2007, Bangkok, Thailand, April 9-12, 2007, Proceedings*, Band 4443 der Reihe *Lecture Notes in Computer Science*, Seiten 188–200. Springer, 2007.

[BLCL+94] Berners-Lee, Tim, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen und Arthur Secret: *The World-Wide Web.* Commun. ACM, 37(8):76–82, August 1994.

[Bun15] Bundesrepublik Deutschland: *Bundesdatenschutzgesetz*, 2015.

[BW90] Brandeis, Louis D. und Samuel D. Warren: *The Right to Privacy*, 1890. `http://faculty.uml.edu/sgallagher/Brandeisprivacy.htm`, zuetzt aufgerufen am 21.05.2017.

[CL14] Cleve, Jürgen und Uwe Lämmel: *Data Mining.* Walter de Gruyter GmbH, Berlin, 2014.

[Dal86] Dalenius, Tore: *Finding a needle in a haystack or identifying anonymous census records.* Journal of Official Statistics, 2(3):329–336, 1986.

[Deu17] Deutscher Bundestag: *Grundgesetz für die Bundesrepublik Deutschland*, 2017.

[DHI12] Doan, AnHai, Alon Y. Halevy und Zachary G. Ives: *Principles of Data Integration.* Morgan Kaufmann, 2012.

[Eur00] Europäisches Parlament: *Charta der Grundrechte der Europäischen Union*, 2000.

[Fen03] Fensel, Dieter: *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce.* Springer Science & Business Media, 2003.

[GH14] Grunert, Hannes und Andreas Heuer: *Big Data und der Fluch der Dimensionalität: Die effiziente Suche nach Quasi-Identifikatoren in hochdimensionalen Daten.* In: *Grundlagen von Datenbanken*, Band 1313 der Reihe *CEUR Workshop Proceedings*, Seiten 29–34. CEUR-WS.org, 2014.

[GH16] Grunert, Hannes und Andreas Heuer: *Datenschutz im PArADISE.* Datenbank-Spektrum, 16(2):107–117, 2016.

[Gol17] Goltz, Johannes: *De-Anonymisierungsverfahren: Kategorisierung und deren Anwendung für Datenbankanfragen*, 2017.

[Gyr17] Gyrard, Dr. Amélie: *Machine-to-Machine Measurement (M3).* `http://sensormeasurement.appspot.com/?p=ontologies`, 2017. zuletzt abgerufen am 3. September 2017.

[Hau07] Hauf, Dietmar: *Allgemeine Konzepte - K-Anonymity, l-diversity and T-Closeness*, 2007. https://dbis.ipd.kit.edu/img/content/SS07Hauf_kAnonym.pdf, zuletzt aufgerufen am 28.05.2017.

[Jai10] Jain, Anil K.: *Data clustering: 50 years beyond K-means.* Pattern Recognition Letters, 31(8):651–666, 2010.

[LLV07] Li, Ninghui, Tiancheng Li und Suresh Venkatasubramanian: *t-closeness: Privacy beyond k-anonymity and l-diversity.* In: *IEEE 23rd International Conference on Data Engineering, 2007, ICDE 2007*, Seiten 106–115. IEEE, 2007.

[MKGV07] Machanavajjhala, Ashwin, Daniel Kifer, Johannes Gehrke und Muthuramakrishnan Venkitasubramaniam: *l-diversity: Privacy beyond k-anonymity.* ACM Transactions on Knowledge Discovery from Data (TKDD), 1(1):3, 2007.

[Mül16] Müller, Martin: *Technischer Bericht, CS-02-16, Universität Rostock, Institut für Informatik, Generalisierung von Attributen in Relationalen Datenbanken*, 2016. www.ls-dbis.de/digbib/dbis-tr-cs-02-16.pdf, zuletzt aufgerufen am 29.05.2017.

[Nor] NordNordWest/Wikipedia. https://upload.wikimedia.org/wikipedia/commons/8/89/MG_Postleitzahlen.svg, zuetzt aufgerufen am 18.08.2017, Lizenz: http://creativecommons.org/licenses/by-sa/3.0/de/legalcode.

[Ope15] OpenGeoDb: *OpenGeoDB — OpenGeoDb, Die freie Geoinformatik-Wissensdatenbank.* http://opengeodb.giswiki.org/index.php?title=OpenGeoDB&oldid=13813, 2015. zuletzt abgerufen am 18. August 2017.

[OWL13] OWL Working Group: *Owl Web Ontology Language Current Status*, 2013. https://www.w3.org/standards/techs/owl#w3c_all, zuletzt aufgerufen am 07.06.2017,.

[RDF15] RDF Core Working Group: *RDF Current Status*, 2015. https://www.w3.org/standards/techs/rdf#w3c_all, zuletzt aufgerufen am 07.06.2017.

[Sam01] Samarati, Pierangela: *Protecting Respondents' Identities in Microdata Release.* IEEE Transactions on Knowledge and Data Engineering, 13(6):1010–1027, 2001.

[SH13] Saake, Gunter, Sattler Kai-Uwe und Andreas Heuer: *Datenbanken - Konzepte und Sprachen.* mitp Verlags GmbH, Frechen, 5. Auflage, 2013.

[SSB15] Sazonau, Viachaslau, Uli Sattler und Gavin Brown: *General Terminology Induction in OWL.* In: *OWLED*, Band 9557 der Reihe *Lecture Notes in Computer Science*, Seiten 1–13. Springer, 2015.

[Swe97] Sweeney, Latanya: *Guaranteeing anonymity when sharing medical data, the Datafly System.* In: *Proceedings of the AMIA Annual Fall Symposium*, Seiten 51–55. American Medical Informatics Association, 1997.

[Vos03] Voss, Jakob: *Grundlegende Aspekte des Semantik Web: Modellierung von Ontologien*, 2003.

Anhang A

Datenträger

Anhang B

Aufbau des Datenträgers

Auf dem beiliegenden Datenträger befinden sich ergänzende Materialien. Die Struktur und der Inhalt werden im Folgenden kurz erläutert. Die Überschriften entsprechen den Ordnern im Wurzelverzeichnis des Datenträgers.

Software

- Verwendete Technologien (siehe Kapitel 5)
- Java Runtime Environment (JRE) und Java Development Kit (JDK) in der zum Prototyp passenden Version
- Die von OpenGeoDB bereitgestellte Datenbank

Quellen Die im Literaturverzeichnis aufgelisteten Quellen, mit Ausnahme digital nicht verfügbarer Werke, werden hier aufgelistet. Es wurden ebenso alle Webseiten, welche innerhalb der Arbeit zitiert wurden, als Portable Dokument Format (*.pdf) exportiert und bereitgestellt. Der Dateiname der Quellen entspricht dem Kürzel im Literaturverzeichnis. Hinweis: Die beigefügte Literatur dient der Nachvollziehbarkeit von Aussagen und darf nicht öffentlich bereitgestellt werden. Dies gilt speziell für Werke, welche durch Lizenzverträge der Universität Rostock durch die Universitätsbibliothek zur Verfügung gestellt wurden.

PArADISE

- Vollständiger Source-Code der in Kapitel 5 gezeigten Implementierung
- Das PArADISE-Projekt der Universität Rostock, als .jar exportiert

Selbständigkeitserklärung

Ich, Richard Dabels, erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Vorlage der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 7. September 2017