Institut für Formale Methoden der Informatik

Abteilung Algorithmik

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Bachelorarbeit

Organisation schwach strukturierter Textdokumente

Benjamin Kopf

Studiengang: Informatik

Prüfer: Prof. Dr. Stefan Funke

Betreuer: Prof. Dr. Stefan Funke

begonnen am: 02.06.2017

beendet am: 04.12.2017

CR-Klassifikation: H.3.3

Erklärung

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

B. Kopf

Stuttgart, 4.12.2017

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

B. Kopf

Stuttgart, 4.12.2017

Zusammenfassung

Suchmaschinen wie OSCAR liefern als Ergebnis auf eine Suchanfrage oft große Mengen unstrukturierter Daten, die für den Nutzer sehr unübersichtlich sind. In dieser Arbeit werden Algorithmen erarbeitet, welche die Organisation und interaktive Exploration solcher Ergebnismengen ermöglichen. Dabei wird anhand von anschaulichen Beispielen auf die Stärken und Schwächen der jeweiligen Vorgehensweisen eingegangen. Außerdem wird auf Probleme und Grenzen der Verfahren hingewiesen.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufgabenstellung 1

1.2 Verwandte Arbeiten 2

1.3 Gliederung 2

2 Semistrukturierte Daten **3**

2.1 Die Suchmaschine OSCAR 4

2.2 Die OSCAR-Query 6

3 Vorüberlegungen **6**

3.1 Das Vorbild DBLP 7

3.2 Die Testfälle 7

4 Naiver Ansatz **11**

4.1 Auswertung der Ergebnisse 13

5 Einbeziehung der Parents **15**

5.1 Erster Ansatz 16

5.2 Verbesserter Algorithmus 17

5.3 Laufzeiteffizienz 19

5.4 Auswertung 21

6 Übertragung des Parent-Algorithmus **22**

6.1 Strukturierung nach Key-Value-Paaren 23

6.2 Strukturierung nach Keys 24

6.3 Interaktive Ergebnisse 24

6.4 Verbesserungen 26

7 Beispielhafte Ergebnisse **26**

7.1 Beispiel 1: Supermärkte in Stuttgart 27

7.2 Beispiel 2: Fastfoodrestaurants in Berlin 29

8 Zusammenfassung und Ausblick **30**

8.1 Zusammenfassung 30

8.2 Ausblick 31

1 Einleitung

In der Zeit der Digitalisierung, welche immer größere Datenmengen mit sich bringt, nimmt effizientes Suchen eine stets bedeutender werdende Stellung ein. Genauso wichtig ist aber auch eine sinnvolle und übersichtliche Anordnung der Ergebnismenge für den Benutzer. Dafür ist es nötig, dieser eine anschauliche Struktur zu geben. Wird dabei in einer bereits strukturierten Umgebung gearbeitet, ist es vergleichsweise einfach, sinnvolle Parameter zu finden, nach denen die Ergebnisse organisiert werden können. So kann beispielsweise eine Suchmaschine wie DBLP[1] (Ley, 2002), welche eine Datenbank aus Journalen und Protokollen durchsucht, die Ergebnisse nach festen Parametern wie beispielsweise Autor, Publikationsjahr und Veranstaltung ordnen, da beinahe jedes Element der Datenbank diese Parameter besitzt.

Seit Mitte der 90er Jahre, vor allem durch die Verbreitung des Datenformats XML, gewinnt der Ansatz der schwach strukturierten Daten zunehmend an Beliebtheit. Im Gegensatz zu strukturierten Daten ist diesen schwach strukturierten Daten keine äußere Struktur anzusehen. Diese ist vielmehr in den Daten an sich enthalten. In einer solchen Umgebung kann man nur schwer Parameter festlegen, welche für die Strukturierung der Menge sinnvoll sind. Stattdessen müssen diese für jede Ergebnismenge neu ermittelt werden.

Ein Beispiel dafür bietet die Suchmaschine OSCAR[2] (Bahrdt und Funke, 2015), welche auf OpenStreetMap (OSM[3]) Daten basiert. OSCAR gibt auf eine Query eine meistens große Menge an Informationen zurück, welche ohne Struktur nur schwer erfassbar sind.

1.1 Aufgabenstellung

Das Ziel dieser Arbeit ist die Entwicklung eines Algorithmus, welcher Ergebnismengen von OSCAR strukturiert, organisiert und schließlich übersichtlich

[1] http://dblp.uni-trier.de/
[2] http://oscardev.fmi.uni-stuttgart.de/
[3] http://openstreetmap.org/

präsentiert. Als Vorbild eines guten Ergebnisses dient dabei die Suchmaschine DBLP.

Das Problem bei der Organisation dieser Ergebnismengen liegt darin, dass die verschiedenen Entitäten, jeweils bestehend aus Key-Value-Paaren, oft komplett unterschiedliche Keys besitzen, auch wenn sie ähnliche Objekte beschreiben. Das macht es schwierig zu entscheiden, welche Keys und Values besonders interessant und aussagekräftig für den Benutzer sind.

1.2 Verwandte Arbeiten

Da die Algorithmen in dieser Arbeit für Ergebnismengen der Suchmaschine OSCAR konzipiert sind und an solchen getestet werden, ist die Arbeit von Bahrdt und Funke (2015) die Grundlage für diese Thesis.

Das Vorbild für diese Arbeit bildet die Suchmaschine der Sammlung für wissenschaftliche Publikationen DBLP (Ley, 2002). Die Suchmaschine sucht nicht nur nach Textdokumenten, sondern stellt die Ergebnisse auch, nach mehreren Kriterien strukturiert, übersichtlich dar. Besonders der Complete-Search Algorithmus von Bast und Weber (2007), auf welchem die Suchmaschine von DBLP basiert, ist dabei sehr interessant. Die Effektivität dieses Suchalgorithmus basiert vor allem auf der Datenstruktur *HYB*, welche eine verbesserte Form des invertierten Index darstellt (Bast und Weber, 2006).

Eine vergleichende Studie zwischen den semistrukturierten Datenformaten JSON und XML führten Nurseitov et al. (2009) durch. In dieser Arbeit werden Stärken und Schwächen der jeweiligen Datenformate herausgestellt.

1.3 Gliederung

In Kapitel 2 wird zunächst eine Einführung in die semistrukturierten Daten gegeben, gefolgt von der Betrachtung der explizit vorhandenen Datensätze, mit denen gearbeitet wird.

Als nächstes beschreibt Kapitel 3 die Ziele dieser Arbeit und stellt erste Vermutungen über mögliche Ergebnisse auf. Außerdem werden die gewählten Testsätze vorgestellt.

In Kapitel 4 wird eine erste Idee für einen Algorithmus vorgestellt und analysiert.

Kapitel 5 befasst sich damit, wie die Parents der OSCAR-Daten sinnvoll genutzt werden können. Dazu wird ein weiterer Algorithmus entwickelt und vorgestellt.

Dieser Algorithmus wird in Kapitel 6 auch auf die Keys und Values übertragen und den anderen Umständen angepasst und erweitert.

In Kapitel 7 findet eine Darstellung der Funktionsweise des Algorithmus anhand von mehreren Beispielen, sowie die Bewertung der verschiedenen Ergebnisse statt.

Zuletzt liefert Kapitel 8 eine Zusammenfassung der Arbeit und spricht Möglichkeiten an, die Arbeit fortzuführen.

2 Semistrukturierte Daten

Die immer größer werdenden Mengen an Daten machten es in der Vergangenheit immer schwerer diesen eine strukturierte Form zu geben. Ein festgelegtes Datenschema scheint oft zu restriktiv, weshalb der Ansatz der semistrukturierten Daten seit Mitte der 90er Jahre immer populärer wird (Bry et al., 2001). Anstatt die Daten einer Struktur anzupassen, ist die Struktur laut Bry et al. (2001) in den semistrukturierten Datensätzen enthalten, was sie selbsterklärend machen soll. Ein großes Anwendungsgebiet finden die semistrukturierten Daten beispielsweise durch XML (eXtensible Markup Language) (Goldman et al., 1999). Die XML Daten bestehen laut Goldman et al. (1999) aus verschiedenen Elementen, welche jeweils aus Attribut-Wert-Paaren bestehen, sowie gegebenenfalls aus einer Menge an Subelementen.

Eine andere Datenaustauschsprache ist JSON (JavaScript Object Notation). Diese legt nach Nurseitov et al. (2009) besonderen Wert darauf, sowohl für Menschen als auch Computer, leicht lesbar zu sein. Wie bei XML liegen auch bei JSON die Daten in Form von Objekten vor, welche aus Key-Value-Paaren bestehen.

```
{
    "firstname" : "John",
    "lastname" : "Smith"
}
```

```
<name>
    <first>John</first>
    <last>Smith</last>
</name>
```

Abbildung 1: Vergleich der Darstellung eines einfachen Objektes in JSON (links) und XML (rechts). (Nurseitov et al., 2009)

Abbildung 1 zeigt beispielhaft wie das Objekt bestehend aus dem Vornamen *John* und dem Nachnamen *Smith* in JSON und XML dargestellt werden könnte.

2.1 Die Suchmaschine OSCAR

Die für diese Arbeit zu strukturierenden Daten kommen von der Suchmaschine OSCAR von Bahrdt und Funke (2015). Diese beruht auf Daten von dem Geodatenprojekt OpenStreetMap (OSM), bei der alle Objekte aus Key-Value-Paaren bestehen. Die OSCAR-Suchmaschine gibt als Antwort auf eine Query die gefundenen OSM-Objekte, mit passenden Tags, als JSON-Datei zurück. Die einzelnen Objekte sind dabei in einem Array aufgelistet und können wie folgt aussehen:

”id”:35367421

”osmid”:1202732923

”type”:”node”

"score":20

"bbox":[48.816382,48.816382,9.4200821,9.4200821]

"shape":"t":-1

"k":["addr:city","addr:housenumber","addr:postcode","addr:street", "amenity","name","opening_hours","phone","phone_1","wheelchair"]

"v":["Remshalden","7","73630","Schorndorfer Straße","fast_food","Babylon Döner","Mo-Sa 11:00-22:00; Su 12:00-22:00","+49 7151 9753794","+49 7151 2711421","yes"]

"p":[188490,22072,5734,538,64]

Dabei steht links immer der Key und rechts von dem Doppelpunkt der dazugehrige Value. Die eigentlichen Key-Value-Paare von OSM sind dabei in zwei Arrays gespeichert, wobei beispielsweise zu dem ersten Key im "k" Array der erste Value des "v" Arrays gehört. In diesem Beispiel also "addr:city":"Remshalden". Von jedem Objekt benötigt man außer den Keys mit den dazugehörigen Values noch eine der beiden ids, mit denen die Objekte später leicht von einander unterschieden werden können, sowie das Array der Parents "p". Da sowohl die von OSM vergebene "osmid" als auch die von OSCAR neu vergebene "id" eindeutig sind, ist es irrelevant welche der beiden verwendet wird. Für diese Arbeit wurde deswegen immer die von OSCAR vergebene "id" verwendet, da diese auch für die Parentbeziehungen genutzt werden. In dem Array der Parents sind die ids der Regionen in welchen sich das aktuelle Objekt befindet, hinterlegt. Laut Bahrdt und Funke (2015) ist diese Parentstruktur als DAG (directed acyclic graph) aufgebaut. So befindet sich der Babylon Döner aus dem obigen Beispiel in Deutschland, welches die id 64 besitzt und deswegen im Parrent Array aufgeführt ist. Damit können relativ einfach die lokalen Beziehungen zwischen den verschiedenen Objekten festgestellt werden.

2.2 Die OSCAR-Query

Bahrdt und Funke (2015) entwarfen die Query Sprache für OSCAR mengenbasiert. Während ein einzelnes Suchwort natürlich alle Objekte mit diesem Wort als String oder Substring in dessen Keys oder Values findet, erhält der Nutzer bei zwei durch Leerzeichen voneinander getrennten Worten in der Query, die Schnittmenge der beiden einzelnen Querys als Ergebnis. Eine Vereinigung der Ergebnismengen erreicht man mit einem '+' zwischen den beiden Worten, mit einem '-' die Differenz. Nach einem bestimmten Key in den Objekten kann mit '@' gesucht werden und mit '@key:value' nach einem expliziten Key-Value-Paar. Möchte man nun zum Beispiel mit der Query

Stuttgart @amenity:fast_food

nach Fastfoodrestaurants in Stuttgart suchen fällt auf, dass auch einige Treffer außerhalb von Stuttgart und sogar ein Treffer in Leipzig gefunden werden. Das liegt laut Bahrdt und Funke (2015) daran, dass die Ergebnisse in einem größeren administrativen Bereich, dem *Regierungsbezirk Stuttgart* liegen, welcher natürlich den Substring *Stuttgart* enthält. Ebenso verhält es sich mit dem Treffer in Leipzig, welcher in der *Stuttgarter Allee* liegt. Um das zu verhindern kann der Nutzer mit Anführungszeichen einen exakten, nicht Substring Treffer forcieren:

"Stuttgart" @amenity:fast_food

3 Vorüberlegungen

Angestrebt wird ein Algorithmus, der in möglichst kurzer Zeit alle von einer OSCAR-Query gefundenen Objekte in eine sinnvolle Struktur bringt. Als Vorbild, wie eine sinnvolle Struktur aussehen kann, dient für diese Arbeit wie bereits erwähnt die Datenbank für wissenschaftliche Publikationen DBLP (Ley, 2002).

3.1 Das Vorbild DBLP

Das Digital Bibliography and Library Project kurz DBLP startete laut Ley (2009) im Jahre 1993 und hat sich seitdem zu einem beliebten Service für wissenschaftliche Publikationen entwickelt. Die einzelnen Objekte sind in Form einer XML-Datenbank gespeichert, wobei als Vorbild BIBTEX dient (Ley, 2009). Die Objekte sollen deswegen falls möglich Felder wie *Autor, Titel, Seiten* und *Jahr* enthalten, doch nur das Element *Titel* muss tatsächlich in jedem DBLP Eintrag enthalten sein (Ley, 2009).

Die Suchmaschine von DBLP basiert auf dem CompleteSearch Algorithmus von Bast und Weber (2007). Neben den ersten Ergebnissen auf die Suchanfrage stellt die Suchmaschine auch eine strukturierte Übersicht dar, mit der die Ergebnismenge verfeinert werden kann. In Abbildung 2 ist diese Struktur für die Suchanfrage *Stefan Funke* abgebildet.

Da die Publikationen nach denen dabei gesucht wird sehr ähnlich aufgebaut sind, kann man gewisse Attribute nach denen das Ergebnis strukturiert werden soll, im Voraus festlegen. So werden die Publikationen nach jeder Query in Kategorien wie *Autor, Erscheinungsdatum* und so weiter aufgelistet. Bei den Ergebnissen von Suchanfragen an OSCAR ist das jedoch nicht ohne weiteres möglich, weil verschiedene Objekte auch komplett verschiedene Keys haben können. Deswegen muss der entstehende Algorithmus während der Laufzeit geeignete Attribute zur Strukturierung finden. Somit ist das Feststellen geeigneter strukturgebender Elemente ein wichtiger Bestandteil den der Algorithmus bewältigen muss.

3.2 Die Testfälle

Als erste Testfälle, auf welchen die Funktionalität des entwickelten Algorithmus geprüft werden soll, dienen die OSCAR-Anfragen *Döner* und *Waiblingen*. Diese beiden Querys eignen sich aufgrund der Unterschiede in den jeweiligen Ergebnismengen. Die Ergebnisse auf die Suchanfrage *Döner* sind,

[–] Refine list

refine by author
Stefan Funke (89)
Sabine Storandt (27)
Domagoj Matijevic (9)
Kurt Mehlhorn (9)
Stefan A. Funken (8)
Sören Laue (7)
Friedrich Eisenbrand (7)
André Nusser (7)
Carsten Carstensen (6)
Nikola Milosavljevic (6)
114 more options

refine by venue
ALENEX (7)
SODA (7)
Symposium on Computational Geometry (6)
AAAI (5)
Int. J. Comput. Geometry Appl. (5)
DCOSS (4)
Comput. Geom. (4)
ESA (3)
SIGSPATIAL/GIS (3)
Computing (3)
52 more options

refine by type
Conference and Workshop Papers (72)
Journal Articles (33)
Informal Publications (1)
Books and Theses (1)
Editorship (1)
Reference Works (1)

refine by year
2017 (6)
2016 (7)
2015 (8)
2014 (5)
2013 (5)
2012 (3)
2011 (9)
2010 (2)
2009 (6)
2008 (6)
10 more options

Abbildung 2: Strukturierte Ergebnisse der DBLP Suchmaschine bei der Query *Stefan Funke*. (09.2017)

wie in Abbildung 3 gezeigt, in (und vereinzelt auch außerhalb von) Europa verteilt. Die meisten Ergebnisse befinden sich zwar in Deutschland, sind aber auch dort relativ gleichmäßig auf die verschiedenen Bundesländer aufgeteilt. Außerdem ist zu erwarten, dass die Objekte, die von dieser Query zurückgegeben werden, alle sehr ähnlich sind, da es sich zum größten Teil um Fastfoodresaurants handeln sollten. Auf der anderen Seite befinden sich die Ergebnisse der Query *Waiblingen*, wie in Abbildung 4 gezeigt, nahezu ausschließlich

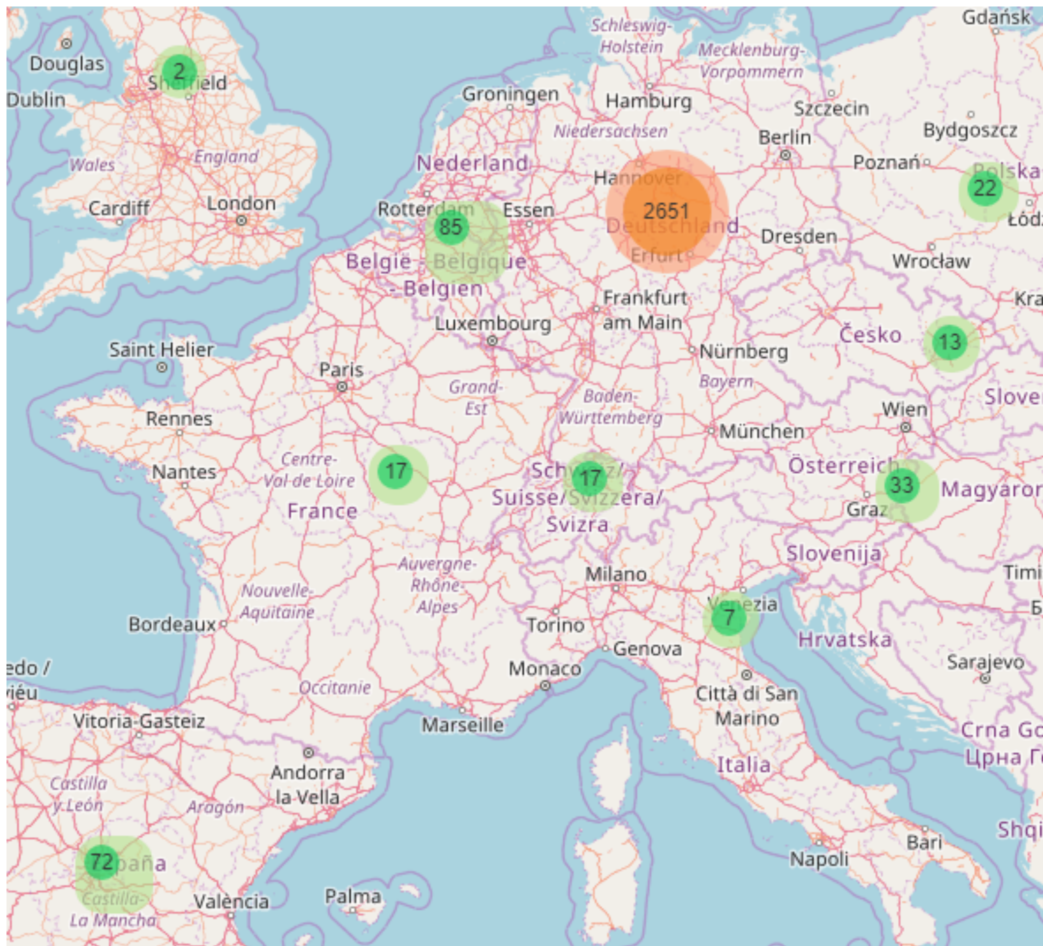

Abbildung 3: Ausschnitt der Ausgabe von OSCAR bei Suchanfrage *Döner*. (09.2017)

in Deutschland im Rems-Murr-Kreis, in welchem sich die Stadt Waiblingen befindet. Des Weiteren ist zu erwarten, dass die Treffer der Anfrage aus sehr unterschiedlichen Objekten bestehen, da nicht nach einer bestimmten Art Objekt, sondern nach allen Objekten in einer bestimmten Region gesucht wird. Den Vergleich der beiden Querys zeigt Tabelle 1 im Detail. Dabei sieht man deutlich die größere räumliche Verteilung der Ergebnismenge der Query *Döner* anhand der deutlich größeren Menge an vorkommenden Parents, sowie die höhere Vielfalt in der Ergebnismenge der Query *Waiblingen*, durch die größere Anzahl verschiedener Keys und Values.

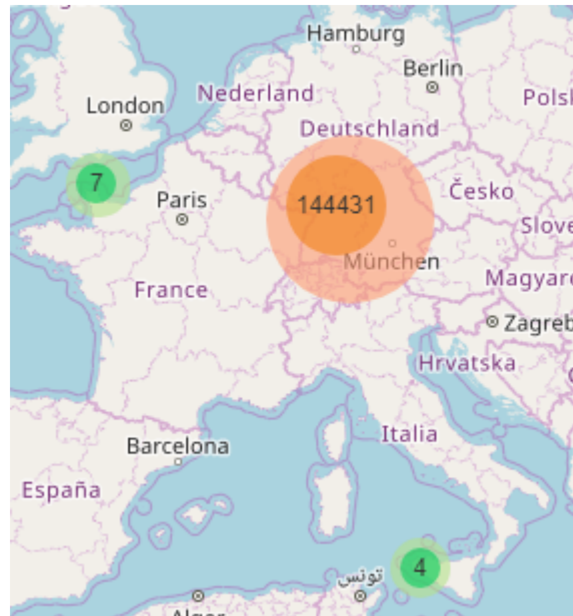

Abbildung 4: Ausschnitt der Ausgabe von OSCAR bei Suchanfrage *Waiblingen.* (09.2017)

Query	**Döner**	**Waiblingen**
Objekte	3.161	748.451
verschiedene Parents	3.795	52
verschiedene Keys	219	827
verschiedene Values	6.221	19.495
Lage der Objekte	verteilt	konzentriert
Art der Objekte	ähnlich	verschieden

Tabelle 1: Vergleich der OSCAR-Querys *Döner* und *Waiblingen.*

Die Ähnlichkeit der Objekte der Query *Döner* lässt vermuten, dass die Strukturierung dieser Ergebnisse schwierig ausfallen wird, da es, wenn überhaupt, nur wenige Merkmale geben wird, die Objekte zu unterscheiden. Die Ergebnisse der Query *Waiblingen* sollten dagegen deutlich vielseitigere und interessantere Lösungen bieten, da die sehr unterschiedlichen Objekte anhand von einigen Kriterien strukturierbar sein sollten.

4 Naiver Ansatz

Als erste Idee sollten möglichst sinnvolle Keys in der Ergebnismenge gefunden werden, nach welchen die Objekte strukturiert werden können. Nach diesen Keys soll das Ergebnis dann anschaulich, wie bei der Suchmaschine DBLP in Abbildung 2, dargestellt werden. Dabei wurde das Problem wie folgt angegangen:

1. Bestimmung aller in der Ergebnismenge vorkommenden Key-Value-Paare.

2. Evaluation der Relevanz der Keys.

3. Strukturieren nach den relevanten Keys.

1. Bestimmung aller in der Ergebnismenge vorkommenden Key-Value-Paare

Dazu werden die Key-Value-Paare der von der Query gefundenen Objekte einmal durchgegangen. Dabei wird, wie in Abbildung 5 dargestellt, eine zweidimensionale Hashmap erzeugt, wobei die äußere Map alle vorkommenden Keys als Keys bekommt und eine innere Hashmap als Value. Die innere Map bekommt als Keys alle Values, die zu dem Key in der Ergebnismenge vorkommen, und als Value die Anzahl, wie oft dieses Key-Value-Paar in der Menge erscheint als Integer.

2. Evaluation der Relevanz der Keys

Zunächst sollen nur Keys zugelassen werden, welche auch bei einer möglichst großen Menge an Objekten vorkommen. Deswegen werden nur Keys in Betracht gezogen, die in mindestens zehn Prozent der Objekte erscheinen. Genauso ist es auch nicht sinnvoll alle Values des Keys aufzuführen, sondern lediglich solche, die oft genug erscheinen. Deswegen werden alle Values, die weniger als ein Prozent des Keys ausmachen als *others* zusammengefasst. In Abbildung 6 kann man sehen, wie dann zum Beispiel die Strukturierung nach

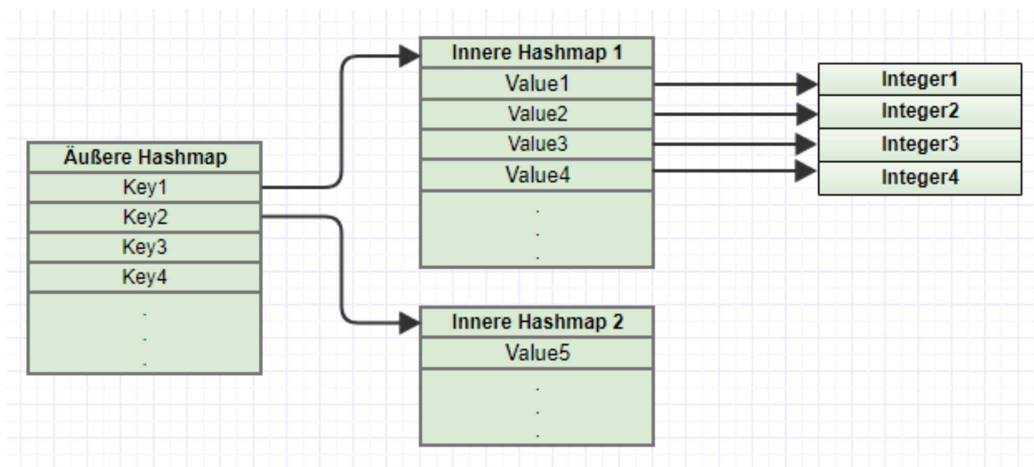

Abbildung 5: Aufbau der zweidimensionalen Hashmap.

refine by author
Stefan Funke (89)
Sabine Storandt (27)
Domagoj Matijevic (9)
Kurt Mehlhorn (9)
Stefan A. Funken (8)
Sören Laue (7)
Friedrich Eisenbrand (7)
André Nusser (7)
Carsten Carstensen (6)
Nikola Milosavljevic (6)
114 more options

refine by addr:city (914)
Berlin (25)
Dresden (12)
Karlsruhe (11)
Hamburg (11)
Düsseldorf (10)
Essen (10)
Halle (Saale) (10)
others (825)

Abbildung 6: Ausschnitt der Ergebnisse der Query *Döner* nach Vorbild von DBLP.

dem Key *addr:city* bei Query *Döner* aussehen würde. Dieser Key kommt insgesamt bei 914 der 3161 betrachteten Objekten vor, 25 mal davon mit dem Value *Berlin*. Es werden insgesamt nur sieben der 561 Values angegeben, da nur diese mindestens zehn mal vorkommen. Die meisten der Values erscheinen sogar nur ein mal.

Von den so gefundenen Keys sind jedoch immer noch nicht alle sinnvoll. Es werden zuletzt noch alle ausgeschlossen, bei denen alle Objekte den gleichen Value haben oder aber jeder Value so selten ist, dass alle Values zu *others*

zusammengefasst werden würden.

3. Strukturieren nach den relevanten Keys
Die Keys, die dann noch übrig sind, werden nach Häufigkeit geordnet und danach wie in Abbildung 6 gezeigt dargestellt.

4.1 Auswertung der Ergebnisse

name (3119)
City Döner (40)
Döner Kebab (68)
Berlin Döner (39)
Döner (306)
Döner King (34)
others (2632)

amenity (3065)
restaurant (344)
fast_food (2695)
others (26)

cuisine (2293)
pizza (134)
turkish (438)
kebab (1504)
others (217)

wheelchair (1289)
no (478)
limited (389)
yes (421)
others (1)

addr:street (1047)
Bahnhofstraße (31)
Hauptstraße (42)
others (974)

addr:housenumber (955)
22 (17)
24 (11)
26 (12)
27 (10)
28 (11)
29 (10)
11 (23)
12 (13)
13 (15)
35 (13)
14 (19)
15 (15)
16 (13)
17 (13)
18 (15)
19 (14)
1 (65)
2 (36)
3 (35)
4 (28)
5 (27)
6 (19)
7 (20)
8 (15)
9 (26)
20 (11)
43 (10)
others (439)

addr:city (914)
Düsseldorf (10)
Dresden (12)
Berlin (25)
Halle (Saale) (10)
Karlsruhe (11)
Hamburg (11)
Essen (10)
others (825)

addr:country (509)
DE (490)
AT (7)
others (12)

opening_hours (499)
Mo-Su 11:00-23:00 (7)
Mo-Sa 11:00-22:00; Su 12:00-22:00 (5)
10:00-23:00 (5)
Mo-Su 11:00-22:00 (16)
11:00-23:00 (6)
11:00-22:00 (6)
24/7 (10)
Mo-Su 10:00-23:00 (7)
Mo-Su 10:00-22:00 (8)
Mo-Sa 10:00-20:00 (5)
others (424)

Abbildung 7: Anschaulich dargestellte Ergebnisse der Query *Döner*.

In den Abbildungen 7 und 8 sieht man eine anschauliche Anordnung der Ergebnisse des naiven Algorithmus bei Eingabe der beiden Testquerys.

building (80306)
residential (3823)
yes (61807)
garage (3725)
house (7937)
others (3014)

highway (33337)
unclassified (846)
tertiary (1150)
steps (908)
crossing (733)
secondary (992)
path (2106)
residential (6170)
turning_circle (469)
service (3232)
footway (4510)
bus_stop (629)
track (9970)
others (1622)

addr:street (33236)
Bahnhofstraße (361)
others (32875)

addr:housenumber (32656)
22 (528)
23 (469)
24 (470)
25 (439)
26 (441)
27 (407)
28 (420)
29 (396)
30 (395)
31 (366)
10 (1002)
32 (347)
11 (842)
12 (874)
13 (623)
14 (767)
15 (741)
16 (715)
17 (636)
18 (634)
19 (577)
1 (989)
2 (904)
3 (1113)
4 (1104)
5 (1132)
6 (1081)
7 (1024)
8 (1038)
9 (966)
20 (604)
21 (536)
others (10076)

addr:city (29997)
Winterbach (2022)
Schwaikheim (1942)
Remshalden (419)
Schorndorf (2199)
Waiblingen (2033)
Kernen im Remstal (593)
Neustadt (620)
Winnenden (5256)
Fellbach (4072)
Welzheim (1420)
Hohenacker (1050)
Berglen (434)
Alfdorf (1031)
Weinstadt (5066)
Kernen (675)
others (1165)

addr:postcode (28980)
70734 (1521)
71336 (1992)
73614 (2391)
71334 (1181)
71332 (533)
73642 (1421)
73663 (475)
73630 (474)
73553 (1327)
71364 (5213)
71394 (1268)
73650 (2023)
71384 (5074)
70736 (1438)
71409 (1944)
others (705)

source (22458)
Maps4BW, LGL, www.lgl-bw.de (3102)
Maps4BW, LGL, www.lgl-bw.de, Bing (496)
Bing Sat (2012) + own video (558)
Bing, LGL, lgl-bw.de (1361)
Yahoo imagery (1129)
survey (882)
Bing, Maps4BW, LGL, www.lgl-bw.de (1340)
Yahoo imagery and surveying (376)
bing (3204)
Bing (7316)
others (2694)

Abbildung 8: Anschaulich dargestellte Ergebnisse der Query *Waiblingen*.

Visuell ähnelt die Darstellung schon sehr der von DBLP, die Qualität der Ergebnisse muss aber noch geprüft werden.

Es wurden für die Query *Waiblingen* sieben und für die Query *Döner* neun Keys als relevant eingestuft. Diese Zahlen sind in Ordnung, bewegen sich aber an der Obergrenze dessen was erwünscht ist. Um eine möglichst große Übersicht zu gewährleisten werden fünf bis sieben Attribute zur Strukturierung angestrebt. Von den gefundenen Keys sieht vor allem der *addr:city* Key sinnvoll aus, da hier die Treffer nach Ortschaften getrennt werden. Der Key *addr:postcode* hat quasi die selbe Funktion, weshalb nicht unbedingt

beide nötig wären. Es erscheint jedoch nicht besonders sinnvoll den Key *addr:country*, wie in Abbildung 7 gezeigt, separat darzustellen. Sinnvoller wäre es die Lokalitätseigenschaften der Objekte in einem Punkt darzustellen, was in Kapitel 5 weiter verfolgt wird. Sehr wenig Aussagekraft hat der Key *addr:housenumber*. Es ist aber durchaus offensichtlich, warum der Key erscheint. Das liegt daran, dass er einige Values mit relativ ähnlicher Häufigkeit besitzt, was eigentlich genau das ist, wonach gesucht wurde. Bei manchen Keys wie beispielsweise *building* ist die Unterscheidung in die einzelnen Values nicht besonders sinnvoll, insbesondere da der am Öftesten vorkommende Value *yes* ist. Hier würde vermutlich die Unterteilung in Objekte, die diesen Key haben und Objekte, die ihn nicht haben, genügen. Dies ist aber schwer dynamisch umzusetzen, da die Unterscheidung in die einzelnen Values zum Beispiel bei Key *wheelchair* sehr wichtig ist.

Alles in allem ist der erste Eindruck aber doch zufriedenstellend und zeigt, an welchen Stellen noch verbessert werden muss.

5 Einbeziehung der Parents

In diesem Kapitel wird, wie schon in Kapitel 4.1 angedeutet, nach einer guten Möglichkeit gesucht, die lokale Organisation der Objekte mit Hilfe der Parents zu erreichen. Das bietet sich an, da die OSCAR Objekte durch die Parent Beziehungen in einem DAG (Directed Acyclic Graph) angeordnet sind (Bahrdt und Funke, 2015). Durch diese Struktur können beispielsweise leicht alle Objekte die in Deutschland liegen aus der Ergebnismenge gefiltert werden, indem man alle Objekte wählt, welche die Id von dem Objekt *Deutschland* (also 64) in ihrem Parent Array haben.

Auch hier stellt sich wieder die Frage, wie die Objekte organisiert werden sollen. Betrachtet man die Query *Döner* scheint es sinnvoll die Objekte nach den verschiedenen Ländern in denen sie liegen zu trennen. Eine Ausnahme dazu bilden die Objekte in Deutschland. Da dort mit Abstand der

größte Teil liegt, sollten diese noch in die Bundesländer unterteilt werden. An diesem Beispiel sieht man, dass auch die Wahl der Parents, nach denen später geordnet werden soll, dynamisch stattfinden muss, um ein sinnvolles Ergebnis zu erhalten.

5.1 Erster Ansatz

Zunächst einmal sollen zwei besonders geeignete Parents gefunden werden. Dazu soll jedem Paar aus zwei verschiedenen Parents P_i und P_j ein Wert $w_{i,j}$ zugewiesen werden, der angibt wie geeignet dieses Paar als erste Parents sind. Das Paar mit dem größten Wert wird danach gewählt. Dafür betrachten wir die Parents als Mengen, wobei jedes Objekt, welches einen bestimmten Parent besitzt, ein Element der Menge darstellt.

Da die Parents möglichst groß und verschieden sein sollen ist die naheliegende Idee:

$$w_{i,j} = |P_i \cup P_j| - |P_i \cap P_j|$$

Also die Differenz zwischen Kardinalität der Vereinigung und Kardinalität des Schnitts der beiden Mengen. Ein offensichtliches Problem bei diesem Ansatz ist, dass für die beiden Parents P_i und P_j mit dem größten Wert $w_{i,j}$ $P_i \subset P_j$ möglich ist, wenn alle Objekte in P_j liegen und $|P_i|$ minimal ist. Da man grundsätzlich disjunkte Mengen als Ergebnis möchte, ist es sinnvoll die Bedingung $|P_i \cap P_j| = 0$ hinzuzufügen. Bei näherem Betrachten ist es auch nicht gut, die Vereinigungen zu bewerten, da so eine extrem große und eine kleine Menge gewählt werden können. Es sollen aber beide Mengen möglichst groß sein. Aus diesen Überlegungen folgt:

$$w_{i,j} = \begin{cases} min\,\{|P_i|, |P_j|\} & \text{falls } |P_i \cap P_j| = 0 \\ 0 & \text{sonst} \end{cases}$$

Anschließend werden die Paare nach den Werten sortiert und die beiden Parents des Paares mit dem höchsten Wert werden als Startparents gewählt.

Dieses Verfahren wählt für die Query *Döner Nordrhein-Westfalen* und *Bayern* sowie *Vereinbarte Verwaltungsgemeinschaft der Stadt Schorndorf* und *Gemeindeverwaltungsverband Winnenden* für *Waiblingen*. Dabei sieht man, dass sich die Ergebnisse von *Döner* auf Bundesländerebene und die von *Waiblingen* auf Städteebene befinden, was den oben beschriebenen, gewünschten Resultaten entspricht.

Als nächstes müssen die restlichen relevanten Parents gefunden werden. Dazu werden in absteigender Reihenfolge die verbliebenen Paare betrachtet. Ist ein Partner bereits in der Liste der gewählten Parents, wird der andere Partner der Liste hinzugefügt. Sind beide Partner nicht in der Liste, wird der Algorithmus beendet. Dieser Ansatz ist nicht besonders gut, da dadurch wieder Überschneidungen in der Ergebnismenge vorkommen können. Für die Query *Döner* wählt der Algorithmus beispielsweise die *Türkei*, da sie mit *Bayern* keinen Schnitt hat und *Bayern* bereits bei den Startparents enthalten ist. Da aber *Deutschland* ebenfalls keinen Schnitt mit der *Türkei* hat, werden schließlich sowohl *Deutschland* als auch *Bayern* für die Ergebnisliste gewählt. Da genau das vermieden werden sollte, da $P_{Bayern} \subset P_{Deutschland}$ ist, muss das Verfahren noch verbessert werden.

5.2 Verbesserter Algorithmus

Die Auswahl der ersten beiden Parents hat im ersten Ansatz zwar gut funktioniert, ist aber sehr aufwendig. Bei n Parents werden dafür $\binom{n}{2}$ Schnittmengen gebildet. Deutlich effizienter ist es, zuerst die Parents nach Häufigkeit zu sortieren. Somit werden nur n Entitäten sortiert, anstelle von $\binom{n}{2}$ im ersten Algorithmus. Danach wird der Schnitt der beiden häufigsten Parents gebildet. Ist er disjunkt werden diese Parents als Startparents gewählt. Ansonsten werden die nächsten Parents betrachtet, und zwar so lange bis ein Paar mit disjunktem Schnitt gefunden wird, welches dann gewählt wird. In dem folgenden Pseudocode wird dieses Verfahren nochmal übersichtlich dargestellt:

```
for i = 2 to n do
  for j = 1 to i − 1 do
    if |P_i ∩ P_j| = 0 then
      add P_i, P_j to resultlist
      end algorithm
    end if
  end for
end for
```

Dieser Pseudocode berechnet im worst case wiederum $\binom{n}{2}$ Schnittmengen. Dieser tritt aber nur ein, wenn die beiden kleinsten Parents die einzigen mit disjunktem Schnitt sind, oder wenn alle Parentmengen mindestens ein Element mit den anderen Mengen gemeinsam haben. Diese beiden Fälle treten jedoch nur auf, wenn in OSCAR nach einer bestimmten Region gesucht wird, oder nur ganz wenige Suchergebnisse erzielt werden. In beiden Fällen ist die Ergebnisorganisation nach Lokalität nicht besonders sinnvoll. In jedem anderen Fall sind deutlich weniger Schnittmengen nötig als bei dem ersten Ansatz.

Bei dem zweiten Teil des Algorithmus ist das Problem, dass durch das Auswahlverfahren Parents in die Ergebnisliste kommen können, welche Teilmengen von anderen Parents in dieser Liste sind, was natürlich unerwünscht ist. Um das zu verhindern muss, bei der Abwägung ob ein neuer Parent zur Liste hinzugefügt wird, dieser mit allen anderen Parents verglichen werden. Da durch den ersten Teil bereits eine sortierte Liste der Parents vorliegt ist es sinnvoll, diese weiter zu nutzen. Beginnen kann man dabei mit dem Parent dessen Index $i + 1$ ist, wobei das i aus dem vorigen Teil des Algorithmus ist, da die anderen Parents bereits ausgeschlossen wurden. Nun wird von diesem Parent an geprüft, ob der Parent mit jedem Parent der Liste einen disjunkten Schnitt hat. Ist dies der Fall wird der Parent gewählt und der Ergebnisliste hinzugefügt. Ansonsten wird er verworfen.

for $k = i + 1$ to n **do**

  for $l : P_l \in$ resultlist **do**

    if $|P_k \cap P_l| \neq 0$ **then**
      discard P_k
    end if
  end for
  if P_k is not discarded **then**
    add P_k to resultlist
  end if
end for

Da Parents, welche nur bei sehr wenigen Objekten vorkommen nicht interessant sind, kann der Algorithmus vorzeitig abgebrochen werden, wenn eine bestimmte Anzahl an Vorkommen nicht mehr erreicht wird. Die Parents, die bei weniger als einem Prozent der Objekte vorkommen werden deswegen nicht berücksichtigt um unnötige Rechenzeit zu sparen.

5.3 Laufzeiteffizienz

Eine Obergrenze für sowohl den ersten, als auch den verbesserten Ansatz des Parents-Algorithmus sind n^2 Schnittmengen, die bei n Parents gebildet werden müssen. Wegen dieser großen Zahl an Schnittmengen ist es notwendig, dass diese so wenig Rechenzeit beanspruchen wie möglich. Dazu stellt Java die Methode *set1.retainAll(set2)* im Interface *Set* zur Verfügung. Diese prüft der Reihe nach für jedes Element der Menge set1, ob dieses auch in set2 enthalten ist. Falls nicht, wird es aus set1 entfernt. Für $n = |\text{set1}|$ und $m = |\text{set2}|$ liegt die Laufzeit dieses Verfahrens in $\mathcal{O}(n \cdot m)$. Es benötigt also quadratische Laufzeit für $n \approx m$.

Um eine schnellere Schnittmengenberechnung implementieren zu können ist eine andere Datenstruktur sinnvoll, wie die von Bast und Weber (2007)

vorgestellte Datenstruktur HYB. Dazu wird ein Array aus Tupeln verwendet, welches alle Objekt-Parent Paare enthält. Das erste Element des Tupels beinhaltet dabei die ID des Parent und das zweite Element die ID des dazu gehörigen Objekts. Dieses Array wird dann aufsteigend nach dem ersten Element sortiert, wobei bei identischen Parents als zweites Kriterium nach dem zweiten Element sortiert wird. Als Beispiel werden die Objekte A, B, C und D mit Parents 0 bis 9 und folgender Verteilung betrachtet:

$$\begin{aligned} A &: 0, 3, 4, 7, 8 \\ B &: 1, 2, 3, 4, 5 \\ C &: 6, 9 \\ D &: 0, 2, 7, 9 \end{aligned}$$

Dann sähen die sortierten Tupel folgendermaßen aus:

0	0	1	2	2	3	3	4	4	5	6	7	7	8	9	9
A	D	B	B	D	A	B	A	B	B	C	A	D	A	C	D

Um effizient auf bestimmte Parents zugreifen zu können wird außerdem ein Offset-Array benötigt, welches an Stelle i den Index des ursprünglichen Arrays hat, an welchem die Einträge mit Parent i beginnen. Für das obige Beispiel sähe das Offset-Array wie folgt aus:

$$[0, 2, 3, 5, 7, 9, 10, 11, 13, 14]$$

Mit Hilfe dieser Arrays können nun relativ schnell Schnittmengen zweier Parents bestimmt werden, da die Einträge sortiert vorliegen. Dafür wird zunächst von beiden Parents das erste Objekt (also das mit der kleinsten ID) betrachtet. Sind die Objekte identisch, wird ein Zähler, der die Schnittgröße zählt um eins erhöht und die nächsten beiden Objekte werden verglichen. Sind die Objekte verschieden, wird von dem Parent mit dem kleineren Objekt das nächst größere als nächstes betrachtet. Sobald bei einem der beiden Parents das Ende erreicht ist und das nächste Element betrachtet werden soll, wird der Algorithmus beendet.

Zum besseren Verständnis dieses Algorithmus wird in Tabelle 2 der Schnitt zwischen zwei Parents gezeigt. Dabei sind die aktuell betrachteten Zahlen markiert.

	Schritt 1	Schritt 2	Schritt 3	Schritt 4	Schritt 5
Parent 1	[**1**, 3, 6, 8]	[1, **3**, 6, 8]	[1, 3, **6**, 8]	[1, 3, **6**, 8]	[1, 3, **6**, 8]
Parent 2	[**3**, 4, 5, 6]	[**3**, 4, 5, 6]	[3, **4**, 5, 6]	[3, 4, **5**, 6]	[3, 4, 5, **6**]
Zähler	0	1	1	1	2

Tabelle 2: Algorithmus zur Bestimmung der Schnittmengen beispielhaft dargestellt.

Da jede der beiden Listen bei diesem Algorithmus nur einmal durchlaufen wird, hat dieser die Laufzeitschranke $\mathcal{O}(n+m)$. Er läuft also in linearer Zeit, was eine deutliche Verbesserung zur quadratischen Laufzeit darstellt.

5.4 Auswertung

In Abbildung 9 sind die Ergebnisse des oben beschriebenen Algorithmus dargestellt. Wie in Kapitel 5 bereits genannt, war das Ziel einige repräsentative Parents zu finden, welche untereinander disjunkt sind und eine ähnliche Menge an Objekten beinhalten. Für die Query *Döner* war dabei erwünscht, dass in Deutschland nach Bundesländern getrennt wird. Andere Länder sollten nicht weiter aufgeteilt werden, was von dem Algorithmus auch umgesetzt wurde. Bei der Query *Waiblingen* ist das nicht sinnvoll, da so gut wie alle Objekte in *Baden-Württemberg* liegen. Wie man sieht wurde hier nach verschiedenen Städten getrennt.

Die Ergebnisse, die der Algorithmus liefert sind also wie erwartet und gewünscht. Er ist deswegen für die Einteilung der Ergebnismenge einer OSCAR-Query nach geographischer Lage sehr sinnvoll.

Query Döner:
Türkiye - 168
Nederland - 85
Hessen - 258
Baden-Württemberg - 317
Thüringen - 64
Sachsen - 225
Berlin - 112
Niedersachsen - 211
Brandenburg - 100
Schleswig-Holstein - 134
Hamburg - 53
Nordrhein-Westfalen - 413
Espana - 72
Sachsen-Anhalt - 96
Mecklenburg-Vorpommern - 68
Bayern - 411
Österreich - 33
Rheinland-Pfalz - 138

Query Waiblingen:
Waiblingen - 19241
Korb - 3956
Gemeindeverwaltungsverband
Winnenden - 21667
Kernen im Remstal - 6187
Remshalden - 5440
Rudersberg - 6421
Berglen - 3716
Vereinbarte
Verwaltungsgemeinschaft der Stadt
Welzheim - 8499
Gemeindeverwaltungsverband
Plüderhausen-Urbach - 7339
Fellbach - 12799
Vereinbarte
Verwaltungsgemeinschaft der Stadt
Schorndorf - 22125
Weinstadt - 12613

Abbildung 9: Ergebnisse des Parent-Algorithmus für die OSCAR-Querys *Döner* und *Waiblingen*.

6 Übertragung des Parent-Algorithmus

Da der Parent-Algorithmus aus Kapitel 5 erstaunlich gute Ergebnisse liefert wird er in diesem Kapitel auf die Keys und Values übertragen, um abzuwägen ob ähnlich gute Ergebnisse erzielt werden können. Dabei wird in Kapitel 6.1 untersucht, wie gut sich der Algorithmus eignet, um relevante Key-Value-Paare zu finden. In dem Kapitel 6.2 werden dagegen lediglich relevante Keys gesucht.

Da die Keys und Values aber im Gegensatz zu den Parents komplett semistrukturiert vorliegen, wird die Restriktion bei der Bildung der Schnittmengen wesentlich gelockert. Anstelle eines komplett disjunkten Schnittes wird lediglich gefordert, dass $|P_i \cap P_j| \leq (|P_i| + |P_j|)/200$ gilt. Das ist sinnvoll, da Objekte einen bestimmten Key besitzen können, der gar nicht zu ihnen passt, zum Beispiel mit Value *no*. Außerdem soll dadurch erschwert

werden, dass ein einziger Key bei der Key-Value Strukturierung die anderen komplett verdrängt, da der Schnitt der verschiedenen Values des Keys fast immer disjunkt ist. Dieses Problem kann dadurch dennoch nicht komplett ausgeschlossen werden, wie man in Abbilung 10 sieht.

6.1 Strukturierung nach Key-Value-Paaren

Query Döner:
wheelchair: no - 478
wheelchair: limited - 389
wheelchair: yes - 421

Query Waiblingen:
building: residential - 3823
building: garage - 3725
power: pole - 1439
amenity: parking - 1431
waterway: stream - 1444
highway: residential - 6170
highway: service - 3232
highway: path - 2106
highway: footway - 4510
highway: track - 9970
building: yes - 61807
building: house - 7937

Abbildung 10: Ergebnisse des Algorithmus für die OSCAR-Querys *Döner* und *Waiblingen* bei Betrachtung der Key-Value-Paare.

In Abbildung 10 sind die Ergebnisse des Algorithmus bei der Suche nach geeigneten Key-Value-Paaren dargestellt. Bei der Query *Döner* wurde dabei lediglich nach den verschiedenen Values des Keys *wheelchair* getrennt. Da die Ergebnisse, wie in Kapitel 3.2 beschrieben, sehr ähnlich sind, da es sich immer um Döner Imbisse handelt, ist es jedoch auch schwierig bestimmte Trennungsmerkmale zu finden. Als Resultat wird ein Key herausgesucht, der zwei möglichst häufige Values hat, was eigentlich dem Vorgehen des ursprünglichem Algorithmus aus Kapitel 4 ähnelt.

Durch die Heterogenität der Ergebnismenge der Query *Waiblingen* findet der Algorithmus deutlich mehr geeignete Key-Value-Paare. Einen großen Teil nehmen die Keys *highway* und *building* ein, da beide sehr oft vorkommen und quasi keine Überschneidungen haben. Trotzdem konnten noch drei

andere Key-Value-Paare dazu gefunden werden, was das Ergebnis sehr zufriedenstellend macht.

6.2 Strukturierung nach Keys

Query Döner:
building - 310
highway - 32
level - 68

Query Waiblingen:
highway - 33337
landuse - 2555
power - 2255
waterway - 1742
building - 80306

Abbildung 11: Ergebnisse des Algorithmus für die OSCAR-Querys *Döner* und *Waiblingen* bei Betrachtung der Keys.

Abbildung 11 zeigt, wie bereits in Kapitel 6.1 angesprochen, wie schwierig es ist Keys mit kleinen Schnittmengen für die Query *Döner* zu finden, da die gefundenen Objekte so ähnlich sind. Deswegen wurden durch den Algorithmus lediglich drei Keys, die selten vorkommen, gefunden. Dass es durchaus häufigere Keys gibt sieht man beispielsweise in Abbildung 7.

Für eine Query, die mehrere verschiedene Objekte liefert, wie beispielsweise *Waiblingen*, können dagegen mehrere große Keys gefunden werden. Abbildung 8 zeigt, dass mit *building* und *highway* direkt die beiden größten, vom ersten Algorithmus als relevant betrachteten, Keys gewählt werden.

6.3 Interaktive Ergebnisse

Obwohl die Ergebnisse des Algorithmus bei der Suche nach interessanten Keys beziehungsweise Key-Value-Paaren stellenweise gut gelungen sind, sind noch Verbesserungsmöglichkeiten vorhanden. Deswegen wird in diesem Kapitel eine interaktive Ergebnisfindung vorgestellt. Dies ist sinnvoll, da manche Ergebnisse ausgeschlossen werden sollen, beispielsweise weil sie für den Nutzer nicht relevant oder interessant sind. Außerdem sind Keys, die Lokalitäten

angeben (wie *addr:city* oder *addr:postcode*) nicht erwünscht, da diese deutlich präziser mit dem Parent-Algorithmus von Kapitel 5 gefunden werden können. Der Algorithmus soll jedoch möglichst dynamisch und vielseitig anwendbar gehalten werden und deswegen werden keine Keys explizit ausgeschlossen. Aus diesem Grund wird dem Nutzer die Möglichkeit zur Verfügung gestellt, nach Abschluss des Algorithmus eine oder mehrere der gefundenen Lösungen auszuschließen und ihn anschließend neu zu starten. Dieser Vorgang kann mehrmals wiederholt werden, bis das Ergebnis zufriedenstellend ist.

Schritt 1
0 wheelchair: no - 478
1 wheelchair: limited - 389
2 wheelchair: yes - 421

Schritt 2
0 amenity: fast_food - 2695
1 amenity: restaurant - 344

Schritt 3
0 name: Berlin Döner - 39
1 name: Döner Kebab - 68
2 name: Döner - 306
3 cuisine: pizza - 134
4 name: City Döner - 40

Abbildung 12: Interaktive Exploration der Ergebnisse der Query *Döner*.

Schritt 1
0 building: residential - 3823
1 building: garage - 3725
2 power: pole - 1439
3 amenity: parking - 1431
4 waterway: stream - 1444
5 highway: residential - 6170
6 highway: service - 3232
7 highway: path - 2106
8 highway: footway - 4510
9 highway: track - 9970
10 building: yes - 61807
11 building: house - 7937

Schritt 2
0 tracktype: grade1 - 3824
1 amenity: parking - 1431
2 highway: residential - 6170
3 highway: footway - 4510
4 building: house - 7937
5 power: pole - 1439
6 addr:postcode: 71336 - 1992
7 addr:postcode: 70736 - 1438
8 source: Maps4BW, LGL, www.lgl-bw.de - 3102
9 addr:postcode: 71409 - 1944
10 highway: path - 2106
11 tracktype: grade2 - 1923
12 source: Bing - 7316
13 addr:postcode: 73650 - 2023
14 addr:postcode: 73642 - 1421

Schritt 3
0 power: pole - 1439
1 source: Maps4BW, LGL, www.lgl-bw.de - 3102
2 addr:city: Schwaikheim - 1942
3 tracktype: grade2 - 1923
4 surface: asphalt - 4026
5 addr:city: Winterbach - 2022
6 addr:city: Welzheim - 1420
7 source: Bing - 7316
8 building: house - 7937

Abbildung 13: Interaktive Exploration der Ergebnisse der Query *Waiblingen*.

Die Abbildungen 12 und 13 zeigen beispielhaft den interaktiven Umgang mit den Ergebnissen. In beiden Fällen werden nach repräsentativen Key-Value-Paaren gesucht. Dabei sind immer die Paare, die nach dem jeweiligen Schritt ausgeschlossen werden rot markiert. Auffallend ist dabei, dass es dennoch schwierig ist für die Query *Döner* ein Ergebnis zu finden, welches nicht

von einem bestimmten Key dominiert wird. Bei der Query *Waiblingen* erscheinen dagegen immer neue Keys, wie zum Beispiel *addr:postcode, tracktype* und *source* in Schritt 2.

6.4 Verbesserungen

Zum Ende der Arbeit wurden noch einige kleine Veränderungen an dem Quellcode des Algorithmus vorgenommen, um das Arbeiten damit angenehmer und intuitiver zu gestalten:

Sortieren der Ergebnisse nach Häufigkeit
Zur besseren Übersicht über die erzielten Ergebnisse werden diese nach Häufigkeit geordnet. Das ist intuitiver und entspricht dem Vorbild von DBLP, wie man beispielsweise an Abbildung 2 sehen kann.

Abfangen von OSCAR-Querys ohne Treffer
Das Eingeben einer OSCAR-Query, welche keine Treffer in der OSCAR-Suchmaschine erzielt, da sie zum Beispiel einen Schreibfehler enthält, wirft nicht länger eine Nullpointer Exception. Stattdessen wird der Benutzer darauf hingewiesen und erhält die Möglichkeit die Query erneut einzugeben.

Anzeigen der Ergebnisse in OSCAR
Ist der Benutzer mit den gefundenen Ergebnissen zufrieden, hat er nun die Möglichkeit diese mit dem Befehl *show* gefolgt von der gewünschten Zeilennummer anzeigen zu lassen. Daraufhin öffnet das Programm OSCAR in einem Browser und zeigt die gewählten Ergebnisse an.

7 Beispielhafte Ergebnisse

In diesem Kapitel wird die Arbeitsweise des Algorithmus, welcher in den Kapiteln 5 und 6 erarbeitet wurde anhand von zwei anschaulichen Beispielen gezeigt. Dabei werden die von der Query erhaltenen Daten nach Key-

Value-Paaren strukturiert, da diese vermutlich für die meisten Nutzer am interessantesten ist.

7.1 Beispiel 1: Supermärkte in Stuttgart

Dieses Beispiel betrachtet die Suche nach Supermärkten in Stuttgart, mit der Query *Stuttgart @shop:supermarket*. OSCAR liefert also Ergebnisse, die den Key *shop* mit dem Value *supermarket* besitzen und den Substring *Stuttgart* enthalten, was bedeutet, dass die Objekte auch außerhalb von Stuttgart, zum Beispiel im deutlich größeren *Regierungsbezirk Stuttgart* liegen können (Bahrdt und Funke, 2015). Nach dem ersten Durchlauf des Algorithmus werden folgende Ergebnisse gefunden:

0 wheelchair: yes - 849
1 wheelchair: limited - 149
2 wheelchair: no - 29

Da in diesem Beispiel nach Supermärkten gesucht wird und die Rollstuhltauglichkeit für den Benutzer nicht relevant ist, werden die drei Key-Value-Paare ausgeschlossen und der Algorithmus wird neu gestartet. Nach dem zweiten Durchlauf liefert er folgende Ergebnisse:

0 name: Lidl - 155
1 name: Netto - 109
2 name: Edeka - 77
3 name: REWE - 71
4 name: Penny - 65
5 name: Rewe - 56
6 name: Norma - 56
7 name: Kaufland - 54
8 name: Aldi Süd - 51
9 organic: only - 46
10 name: Aldi - 46
11 name: ALDI SÜD - 26

12 name: LIDL - 17
13 name: Netto Marken-Discount - 17
14 organic: yes - 16
15 name: Penny Markt - 15
16 name: EDEKA - 14

Nach dem zweiten Durchlauf liefert der Algorithmus eine Strukturierung in die verschiedenen Supermarktketten, was ein gewünschtes Ergebnis für die Query darstellt. Außerdem werden durch den Key *organic* Läden gezeigt, welche auch oder ausschließlich Bioprodukte anbieten. Nun kann der Benutzer zum Beispiel mit *show 2* die Filialen der Kette *Edeka* anzeigen lassen. Dadurch wird ein Browserfenster geöffnet, in welchem die Ergebnisse in OSCAR angezeigt werden. Diese entsprechen der Query *Stuttgart @shop:supermarket "@name:Edeka"* und zeigt sowohl die Objekte mit Value *Edeka* als auch *EDEKA* an. Die Ergebnisse werden in Abbildung 14 dargestellt, wobei von OSCAR immer nur 20 Ergebnisse auf einmal angezeigt werden.

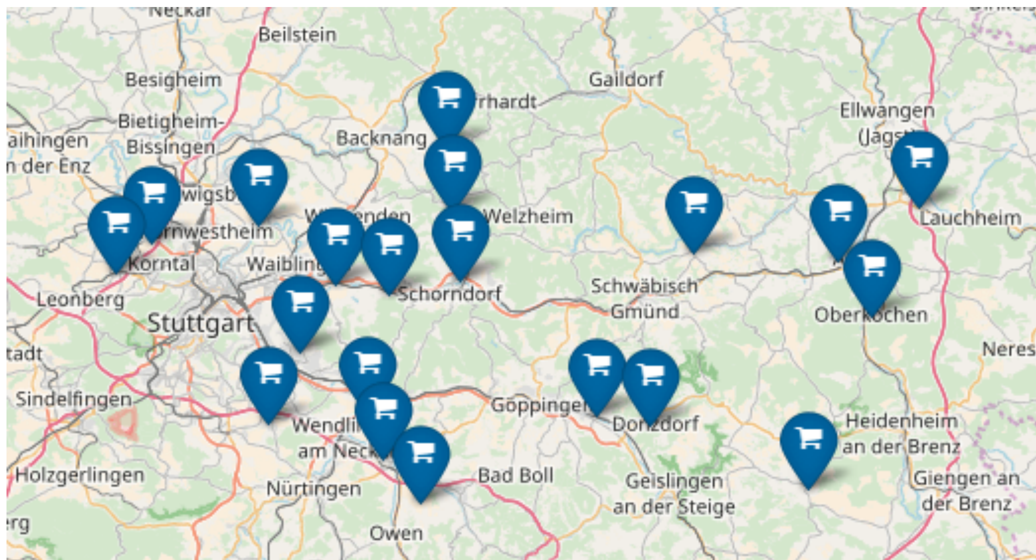

Abbildung 14: Ergebnisse bei Suche nach Filialen der Kette *Edeka* in Stuttgart. (11.2017)

7.2 Beispiel 2: Fastfoodrestaurants in Berlin

Als zweites Beispiel soll mit der Query *Berlin @amenity:fast_food* nach Fastfoodrestaurants in Berlin gesucht werden. Wie im ersten Beispiel dominiert auch hier der Key *wheelchair* die Ergebnisse beim ersten Durchlauf:

0 wheelchair: yes - 857
1 wheelchair: no - 513
2 wheelchair: limited - 493

Auch hier werden diese wieder für den zweiten Durchlauf ausgeschlossen, was folgende Ergebnisse liefert:

0 cuisine: kebab - 642
1 cuisine: burger - 321
2 cuisine: asian - 255
3 cuisine: german - 175
4 cuisine: turkish - 164
5 cuisine: regional - 91
6 cuisine: italian - 82
7 name: Subway - 65

Wie schon im ersten Beispiel liefert der Algorithmus auch hier im zweiten Durchlauf gute Ergebnisse. Auffallend ist, dass die Kette *Subway* separat gelistet ist. Das liegt daran, dass die meisten mit dem Key-Value-Paar *cuisine:sandwich* getaggt sind, aber nicht alle. Nun können mit dem Befehl *show 2* die asiatischen Restaurants angezeigt werden, was in Abbildung 15 dargestellt ist. Die gezeigten Ergebnisse entsprechen der OSCAR-Query *Berlin @amenity:fast_food "@cuisine:asian"*. Wegen der großen Menge an Ergebnissen werden nah beieinanderliegende Ergebnisse von OSCAR zusammen gefasst.

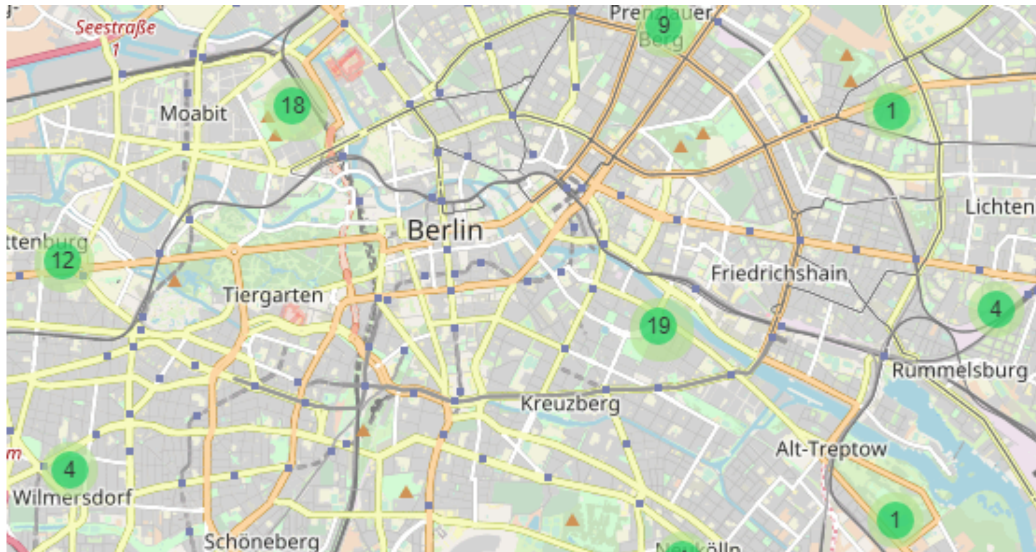

Abbildung 15: Ergebnisse bei Suche nach asiatischen Fastfoodrestaurants in Berlin. (11.2017)

8 Zusammenfassung und Ausblick

Ziel dieser Arbeit war das Entwickeln eines Algorithmus für die anschauliche Organisation von semistrukturierten Daten, welche von der Suchmaschine OSCAR geliefert werden. Im folgenden letzten Kapitel sollen die Ergebnisse davon zusammengefasst werden und es wird evaluiert, in wie weit diese den Erwartungen entsprechen. Zuletzt werden noch mögliche Anknüpfungspunkte für zukünftige Arbeiten vorgestellt.

8.1 Zusammenfassung

Zu Beginn der Arbeit wurde ein erster Ansatz zu der Strukturierung der OSCAR-Daten vorgestellt, welcher nach Vorbild von DBLP geeignete Keys sucht, deren Values als Unterscheidungskriterien für die verschiedenen Objekte dienen. Als nächstes ging es um die lokale Strukturierung der Ergebnisse durch die Betrachtung der Parents. Dieser Algorithmus wurde aufgrund der überraschend guten Ergebnisse auf die Keys, sowie die Key-Value-Paare

übertragen. Wie erwartet tut sich der Algorithmus jedoch bei Mengen aus sehr ähnlichen Objekten schwer, kann jedoch bei heterogenen Mengen interessante Unterscheidungskriterien herausstellen. Zuletzt wurde die Möglichkeit der interaktiven Ergebnisfindung mit diesem Algorithmus vorgestellt.

Es ist also durchaus möglich, Struktur in eine semistrukturierte Umgebung zu bringen, auch wenn es natürlich deutlich aufwändiger als für einen vollständig strukturierten Datensatz ist. Je nachdem wie die semistrukturierten Daten gestaltet sind, fällt das Ergebnis dabei unterschiedlich gut aus.

8.2 Ausblick

Für das Finden geeigneter Keys beziehungsweise Key-Value-Paare für die Strukturierung können aufwändigere Verfahren als das Bilden der Schnittmenge vermutlich etwas bessere Ergebnisse erzielen. Beispielsweise könnten Beziehungen zwischen mehreren verschiedenen Keys hergestellt werden, anstatt immer nur einen bestimmten Key mit einem anderen zu vergleichen.

Um die interaktive Ergebnisfindung für den Nutzer intuitiver zu gestalten ist sicherlich der Aufbau einer grafischen Benutzeroberfläche sinnvoll, auf der beispielsweise die Elemente durch Klicken ausgeschlossen werden können. Außerdem sollte man mit dieser angenehm durch die Ergebnisse navigieren können.

Interessant wären auch Untersuchungen, wie gut sich die hier erarbeiteten Algorithmen auf anderen semistrukturierten Datensätzen, als die von OSCAR bereitgestellten, verhalten.

Literatur

Daniel Bahrdt und Stefan Funke. Oscar: Openstreetmap planet at your fingertips via osm cell arrangements. In *International Conference on Web Information Systems Engineering*, pages 153–168. Springer, 2015.

Holger Bast und Ingmar Weber. Type less, find more: fast autocompletion search with a succinct index. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 364–371. ACM, 2006.

Holger Bast und Ingmar Weber. The completesearch engine: Interactive, efficient, and towards ir & db integration. In *Third Biennial Conference on Innovative Data Systems*, pages 88–95, 2007.

François Bry, Michael Kraus, Dan Olteanu, und Sebastian Schaffert. Semistrukturierte Daten. *Informatik-Spektrum*, 24(4):230–233, 2001.

Roy Goldman, Jason McHugh, und Jennifer Widom. From semistructured data to xml: Migrating the lore data model and query language. 1999.

Michael Ley. The dblp computer science bibliography: Evolution, research issues, perspectives. In *String processing and information retrieval*, pages 481–486. Springer, 2002.

Michael Ley. Dblp: some lessons learned. *Proceedings of the VLDB Endowment*, 2(2):1493–1500, 2009.

Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, und Clemente Izurieta. Comparison of json and xml data interchange formats: a case study. *Caine*, 2009:157–162, 2009.