

# ALGORITHMIK UND EVALUATION DES MUSIKEMPFEHLUNGSSYSTEMS LIBMUNIN

BACHELORARBEIT  
AN DER HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HOF

VORGELEGT BEI:

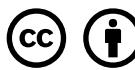
PROF. DR. GÜNTHER KÖHLER  
ALFONS-GOPPEL-PLATZ 1  
95028 HOF

VORGELEGT VON:

CHRISTOPHER PAHL  
LORENZ-SUMMA-STR. 4  
95126 SCHWARZENBACH A. D. SAALE

HOF, 11. MAI 2014

© Copyleft by Christopher Pahl, 2014.  
Some rights reserved.



Diese Arbeit ist unter den Bedingungen der  
*Creative Commons Attribution-3.0* lizenziert.  
<http://creativecommons.org/licenses/by/3.0/de/>

## Abstract

### English:

This paper shows the algorithms of the previously developed music recommendation system *libmunin*. Apart from the construction of the internal graph data structure, some selected, so called providers and distance functions are explained. Additionally, *libmunin*'s mechanism for learning from the user, in an implicit or explicit way, is shown in detail. Users of the library are supposed to read this paper in order to maximize their understanding of *libmunin*. For this purpose, some helpful tips for developing with and on *libmunin* are given in the first chapters. At the end of the paper, some graphs and other visualizations are shown which were developed during the project. In order to understand the context, it is advisable to read the project report before reading this paper. To be useful for interested developers, the topics in every chapter are reflected and potential improvements are listed.

### Deutsch:

Diese Arbeit zeigt die Algorithmik hinter der vorgestellten Musikempfehlungsbibliothek *libmunin*. Neben dem Aufbau des intern genutzten Graphen, werden auch einige ausgewählte Provider und Distanzfunktionen erklärt. Zusätzlich wird detailliert auf *libmunin*'s Mechanismus zum impliziten und expliziten Lernen vom Nutzer eingegangen. Anwendungsentwickler sollten diese Arbeit lesen, um ihr Verständnis von *libmunin* zu vertiefen. Zu diesem Zwecke werden anfangs einige hilfreiche Tipps für die Entwicklung von und mit *libmunin* gegeben. Am Ende der Arbeit werden einige Graphen und Visualisierungen gezeigt, die während der Arbeit an dem Projekt entstanden sind. Es wird dringend empfohlen, vor der Lektüre die Projektarbeit zu lesen, um eine grobe Übersicht zu bekommen und den Kontext zu verstehen. Damit die Arbeit nützlich für interessierte Entwickler ist, werden in jedem Kapitel die vorgestellten Themen noch einmal reflektiert und potenzielle Verbesserungen aufgezeigt.

## Danksagung

Dank sei an folgende Personen und Dinge gerichtet: Die Gattungen Felis silvestris und Ursus arctos, sowie der Obergattung der Aves, Espressohersteller aller Länder, Herrn Prof. Dr. Jörg Scheidt (für die Zelle), Herrn Prof. Dr. Günther Köhler (für die Freiheit) und meiner Familie.

Ihr wisst schon warum.

# Inhaltsverzeichnis

|   |           |
|---|-----------|
| Abstract . . . . .  | iii       |
| Danksagung . . . . .  | iii       |
| Abbildungsverzeichnis . . . . .   | vi        |
| Abkürzungsverzeichnis . . . . .   | vii       |
| <b>1. Einleitung</b>  | <b>1</b>  |
| 1.1. Das allgemeine Problem . . . . .   | 1         |
| 1.2. Das konkrete Problem . . . . .   | 1         |
| <b>2. Allgemeine Entwicklerhinweise</b>   | <b>2</b>  |
| 2.1. Zusammenfassung des aktuellen Stands . . . . .                                   | 2         |
| 2.2. Der Begriff der Distanzfunktion . . . . .  | 3         |
| 2.3. Zur Nutzung von <i>libmunin</i> . . . . .  | 4         |
| 2.4. Zur Erweiterung von <i>libmunin</i> . . . . .                                    | 5         |
| 2.4.1. Hinweise zum Schreiben von Distanzfunktionen . . . . .                         | 5         |
| 2.4.2. Hinweise zum Schreiben von neuen Providern . . . . .                           | 6         |
| 2.5. Vergleich verschiedener Playlisten . . . . .                                     | 7         |
| 2.6. Ressourcenverbrauch . . . . .  | 8         |
| <b>3. Graphenoperationen</b>  | <b>10</b> |
| 3.1. Einleitung . . . . .   | 10        |
| 3.1.1. <code>rebuild</code> : Aufbau des Graphen . . . . .                            | 10        |
| 3.1.2. <code>fixing</code> : Umbauen von Einbahnstraßen . . . . .                     | 13        |
| 3.1.3. <code>add</code> : Hinzufügen von Songs vor dem <code>rebuild</code> . . . . . | 14        |
| 3.1.4. <code>remove</code> : Löschen von Songs zur Laufzeit . . . . .                 | 14        |
| 3.1.5. <code>insert</code> : Hinzufügen von Songs zur Laufzeit . . . . .              | 16        |
| 3.1.6. <code>modify</code> : Verändern der Songattribute zur Laufzeit . . . . .       | 16        |
| 3.1.7. Ablauf beim Hinzufügen einer Distanz . . . . .                                 | 17        |
| 3.2. Graphentraversierung . . . . .   | 19        |
| 3.2.1. Empfehlungsiteratoren . . . . .  | 19        |
| 3.2.2. Anwendung von Regeln . . . . .   | 19        |
| 3.2.3. Filtern der Iteratoren . . . . .   | 20        |
| <b>4. Algorithmen bei Providern</b>   | <b>21</b> |
| 4.1. Einleitung . . . . .   | 21        |
| 4.2. Genrenormalisierung und Vergleich von Genres . . . . .                           | 21        |
| 4.2.1. Zusammenstellung der Genredatenbank . . . . .                                  | 21        |
| 4.2.2. Überführung der Genreliste in einem Genrebaum . . . . .                        | 23        |
| 4.2.3. Zuordnung von Genres . . . . .   | 25        |

|   |           |
|---|-----------|
| 4.2.4. Vergleichen der unterschiedlichen Genrepfad-Mengen . . . . . | 27        |
| 4.2.5. Probleme . . . . .   | 27        |
| 4.3. Schlüsselwortextraktion . . . . .                              | 28        |
| 4.3.1. Der RAKE-Algorithmus . . . . .                               | 28        |
| 4.3.2. Ergebnisse . . . . .   | 30        |
| 4.3.3. Probleme . . . . .   | 31        |
| 4.4. Moodbar-Analyse . . . . .                                      | 32        |
| 4.4.1. Vergleich von Moodbars . . . . .                             | 32        |
| 4.4.2. Probleme . . . . .   | 34        |
| <b>5. Implizites Lernen vom Nutzer</b>                              | <b>36</b> |
| 5.1. Generierung von Regeln . . . . .                               | 36        |
| 5.1.1. Finden von wiederkehrenden Mustern . . . . .                 | 36        |
| 5.1.2. Der RELIM-Algorithmus . . . . .                              | 37        |
| 5.1.3. Ableitung von Regeln aus Mustern . . . . .                   | 37        |
| 5.2. Anwendung von Regeln . . . . .                                 | 40        |
| 5.3. Lernerfolg . . . . .   | 41        |
| 5.4. Explizites Lernen . . . . .                                    | 41        |
| <b>6. Ausblick</b>  | <b>43</b> |
| 6.1. Verbesserung der Algorithmik . . . . .                         | 43        |
| 6.1.1. Audioanalyse . . . . .                                       | 43        |
| 6.1.2. Andere Provider . . . . .                                    | 44        |
| 6.1.3. Empfehlungen . . . . .                                       | 44        |
| 6.2. Erweiterungen . . . . .  | 45        |
| 6.3. Fazit . . . . .  | 46        |
| <b>A. Bilder des Song-Graphen</b>                                   | <b>47</b> |
| <b>B. Bilder des Genregraphen</b>                                   | <b>54</b> |
| <b>C. Medien</b>  | <b>58</b> |
| <b>D. Literaturverzeichnis</b>                                      | <b>60</b> |

# Abbildungsverzeichnis

|   |    |
|---|----|
| 2.1. Darstellung der Dreiecksungleichung . . . . .                                  | 3  |
| 2.2. Skalierungsfunktion der Distanzfunktion . . . . .                              | 5  |
| 2.3. Auflistung des Ressourcenverbrauchs verschiedener Operationen . . . . .        | 8  |
| 2.4. Vergleich verschiedener Playlisten . . . . .                                   | 9  |
| 3.1. Schematische Darstellungen der einzelnen Basisiterationen . . . . .            | 11 |
| 3.2. Vergleich der Distanzberechnungen für rebuild_stupid und rebuild . . . . .     | 13 |
| 3.3. Vor und nach der remove-Operation . . . . .                                    | 14 |
| 3.4. Vor und nach der insert-Operation . . . . .                                    | 15 |
| 3.5. Vor und nach der modify-Operation . . . . .                                    | 15 |
| 3.6. Traversierung durch verschachtelte Iteratoren . . . . .                        | 20 |
| 4.1. Aufbau des Genrebaums in 4 Schritten . . . . .                                 | 24 |
| 4.2. Beispielablauf des Zuordnungs-Algorithmus . . . . .                            | 26 |
| 4.3. Extrahierte Schlüsselwörter aus verschiedenen Liedern . . . . .                | 30 |
| 4.4. Liedtext des Volksliedes „Das Wandern ist des Müllers Lust“ . . . . .          | 31 |
| 4.5. Beispiel-Moodbar von „Avril Lavigne – Knockin’ on Heaven’s Door“ . . . . .     | 32 |
| 4.6. Auflistung der einzelnen Moodbar-Merkmale . . . . .                            | 34 |
| 4.7. Dieselbe Moodbar bei unterschiedlichen Encoding der Audiodaten . . . . .       | 34 |
| 4.8. Moodbar einer Live und einer Studioversion von „Rammstein – Tier“ . . . . .    | 35 |
| 5.1. Die Muster für einige einfache Warenkörbe . . . . .                            | 37 |
| 5.2. Mögliche Regeln, die aus den drei warenkörben erstellt werden können . . . . . | 38 |
| 5.3. Vierfeldertafel mit Beispieldaten . . . . .                                    | 38 |
| 5.4. Graph vor und nach Vergeben eines hohen Ratings . . . . .                      | 42 |
| 6.1. Schematische Darstellung der idealen Traversierungsreihenfolge . . . . .       | 45 |
| A.1. Abbildungen des linearen Testgraphen . . . . .                                 | 48 |
| A.2. Graph aus Zufallsdaten nach erster Basisiteration . . . . .                    | 49 |
| A.3. Graph aus Zufallsdaten nach allen Basisiterationen . . . . .                   | 50 |
| A.4. Graph aus Zufallsdaten nach einem Verfeinerungsschritt . . . . .               | 51 |
| A.5. Der „korrekte“, mittels rebuild_stupid erstellte Graph . . . . .               | 52 |
| A.6. Vollständiger Graph aus 666 Knoten (aus der Demonstration) . . . . .           | 53 |
| B.1. Übersicht über alle 1876 Musikgenres. (Detailstufe: 0,0) . . . . .             | 55 |
| B.2. Übersicht über die gebräuchlichsten Musikgenres. (Detailstufe: 0,1) . . . . .  | 56 |
| B.3. Übersicht über die wichtigsten Musikgenres. (Detailstufe: 0,5) . . . . .       | 57 |
| C.1. Mögliche logos für libmunin . . . . .  | 58 |

# Abkürzungsverzeichnis

| Abkürzung | Bedeutung                                |
|-----------|--|
| API       | <i>Application Programming Interface</i> |
| BPM       | <i>Beats per Minute</i>                  |
| FAQ       | <i>Frequently Asked Questions</i>        |
| URL       | <i>Uniform Resource Locator</i>          |
| MP3       | <i>MPEG-2 Audio Layer III</i>            |
| FLAC      | <i>Free Lossless Audio Codec</i>         |

# 1 | Einleitung

## 1.1 Das allgemeine Problem

IN der, zu dieser Arbeit vorangegangenen, Projektarbeit [1] wurde das Musikempfehlungssystems *libmunin* implementiert. Um Musikempfehlungen auszusprechen, muss ein solches System die Ähnlichkeit zwischen zwei Liedern feststellen können. Dies ist das grundsätzliche Problem eines solchen Systems: Musik ist nur schwer vergleichbar. Fragt man mehrere Menschen, wie *ähnlich* ein Musikstück zu einem anderem ist, so erhält man genauso viele Antworten, wie man Fragen gestellt hat. Die Einschätzung von Musik ist eine sehr subjektive Angelegenheit, die auch häufig zwischen Menschen Diskussionen auslöst. Stuft man den Künstler *Status Quo* als *Rock* ein? Oder doch eher als *Pop*? Was zählt überhaupt noch als *Rock*? Gibt es eine, für den Computer verständliche, Definition von *Rock*?

Wenn man jetzt noch versucht, einem Computer den Begriff der *Musikähnlichkeit* beizubringen, so wird es noch weitaus komplizierter. Dieser kann nur objektiv nach bestimmten Metriken entscheiden. Diese Metriken zu definieren, muss dann wiederum die Aufgabe eines Menschen sein — also sind auch diese wiederum subjektiv, da sie die Vorlieben des Autors widerspiegeln. Auch können diese Metriken nie für alle Fälle funktionieren. Ein gutes Stück „*Kaffeesatzleserei*“ lässt sich leider nie ganz vermeiden. Daher werden in dieser Arbeit einige Annahmen getroffen, die sich aufgrund ihrer Natur nur schwer empirisch nachweisen lassen. An den entsprechenden Stellen wird auf die gemachten Annahmen hingewiesen.

## 1.2 Das konkrete Problem

Erschwerend kommt hinzu, dass jeder Nutzer andere Vorlieben und Gewohnheiten hat. So gesehen, ist *libmunin* in der „*Standardeinstellung*“ ein Musikempfehlungssystem, das genau auf einen Nutzer und dessen Vorlieben zugeschnitten ist: Seinem Entwickler. Bibliotheksanwender können jedoch die Bibliothek an ihre Präferenzen anpassen oder ihren Endnutzern eine Möglichkeit geben, selbst Einstellungen vorzunehmen. Damit die Bibliotheksanwender diese Anpassungen vornehmen können, sollten sie verstehen was intern vor sich geht — genau darum soll es in dieser Arbeit gehen. Hauptsächlich wird diskutiert, wie *libmunin* die Ähnlichkeit zwischen den Attributen eines Liedes berechnet und wie aus diesen Ähnlichkeiten ein Graph aufgebaut wird. Auch auf *libmunin*'s Möglichkeit, vom Nutzer zu lernen wird eingegangen. Zu jedem vorgestellten Thema werden auch Überlegungen angestellt, welche Verbesserungen in zukünftigen Versionen gemacht werden können.

## 2 | Allgemeine Entwicklerhinweise

In diesem Kapitel werden einleitend einige allgemeine Hinweise gegeben, die man bei der Entwicklung mit und von *libmunin* beachten sollte. Statt wie in der API-Referenz [Link-1] auf die einzelnen Methoden und Klassen von *libmunin* einzugehen, sollen hier „Best Practices“ vermittelt werden. Zuvor wird noch eine kurze Zusammenfassung gegeben, um den Leser an die in [1] eingeführten Begriffe heranzuführen.

### 2.1 Zusammenfassung des aktuellen Stands

*Libmunin* ist eine in der Programmiersprache Python geschriebene Bibliothek und implementiert ein Musikempfehlungssystems auf Graphen-Basis. Der dahinterstehende Graph bildet die Nachbarschaftsbeziehungen zwischen den einzelnen Musikstücken (als Knoten) mittels bidirektonaler Kanten ab. Eine Kante verbindet zwei ähnliche Songs miteinander. Um den Graphen aufzubauen zu können, müssen, während des sogenannten Kaltstartes, vom Nutzer der Bibliothek alle Songs mit allen relevanten Attributen eingegeben werden. Damit *libmunin* weiß, wie die einzelnen Werte zu behandeln sind, wird jedem Attribut (Künstler, Titel, Genre...) ein sogenannter *Provider* und eine *Distanzfunktion* zugeordnet.

Ein Provider normalisiert einen Wert anhand verschiedener Charakteristiken. Sein Ziel ist es, für die Distanzfunktion einfache und effizient vergleichbare Werte zu erzeugen, da die Distanzfunktion sehr viel öfters aufgerufen wird als der Provider. Die Distanzfunktion erzeugt dann aus diesen normalisierten Werten eine Distanz, also ein Maß dafür, wie ähnlich diese Werte sind. Die Distanzen gehen dabei von 0 (vollkommen ähnlich) bis 1 (absolut unähnlich). *Libmunin* implementiert eine große Anzahl von vorgefertigten Providern und Distanzfunktionen für gängige Attribute, wie dem Genre, den Audiodaten oder den aus den Liedtext extrahierten Schlüsselwörtern.

Aus den einzelnen Distanzen wird dann der obige Graph aufgebaut. Um aus diesen Graphen dann Empfehlungen abzuleiten, werden einzelne Knoten, sogenannte Seedsongs, nach bestimmten Kriterien ausgewählt. Beginnend von diesen wird eine Breitensuche gestartet, also eine sich kreisförmig vom Seedsong ausbreitende Traversierungstrategie. Bereits besuchte Knoten werden dabei nicht nochmal besucht. Die einzelnen besuchten Knoten werden, nach dem Filtern von doppelten Künstlern und Alben, dann als Empfehlungen angenommen

Zudem lernt *libmunin* während einer Sitzung vom Nutzer, indem es die Gewohnheiten des Nutzers beobachtet und daraus Regeln ableitet. Diese Regeln verbinden zwei Mengen von Songs , die oft



**Abbildung 2.1.:** Die Beziehung dreier Songs untereinander. Die Dreiecksungleichung besagt, dass der direkte Weg von A nach B kürzer oder gleich lang sein sollte als der Umweg über C. Die einzelnen Attribute „a“ und „b“ sind gleich stark gewichtet. Wenn keine Straftwertung für leere Werte gegeben wird, so sind die Umwege manchmal kürzer.

miteinander gehört werden, mit einer Wahrscheinlichkeit miteinander. Alle gehörten Songs werden in einer Historie abgespeichert. Im Vergleich zu bestehenden Systemen ist *libmunin* nicht von den Audiodaten abhängig, sondern kann durch seine flexible Schnittstelle auch alleine auf den Metadaten eines Stücks, wie den Tags eines Musikstückes, operieren. Ein Tag ist eine direkt in der Audiodatei hinterlegte Information um bestimmte Werte wie den Künstler des Stücks zu beschreiben. Das erklärte Ziel der Bibliothek ist es, eine freie Bibliothek zu schaffen, die sowohl offline (in Musicplayern) als auch online (in Streamingdiensten) funktioniert und mit großen Datenmengen umgehen kann. Durch die GPLv3-Lizenz [Link-2] ist ein libertärer, weitläufiger Einsatz möglich.

## 2.2 Der Begriff der Distanzfunktion

Eine Distanzfunktion ist im Kontext von *libmunin* eine Funktion, die zwei Songs als Eingabe nimmt und die Distanz zwischen diesen berechnet. Dabei wird jedes Attribut betracht, welches in beiden Songs vorkommt. Für diese wird von der Maske eine spezialisierte Distanzfunktion festgelegt, die weiß wie diese zwei bestimmten Werte sinnvoll verglichen werden können. Die so errechneten Werte werden, gemäß der Gewichtung in der Maske, zu einem Wert verschmolzen. Fehlen Attribute in einen der beiden Songs, wird für diese jeweils eine „*Straf*“-Distanz von 1 angenommen. Diese wird dann ebenfalls in die gewichtete Oberdistanz eingerechnet. Die folgenden Bedingungen müssen sowohl für die allgemeine Distanzfunktion als auch für die speziellen Distanzfunktionen gelten.  $D$  ist dabei die Menge aller Songs,  $d$  eine Distanzfunktion. Beim Schreiben von Distanzfunktionen sollte versucht werden, alle dieser Eigenschaften zu erfüllen. Technisch nötig sind dabei nur die Bedingungen 1–3. Bedingung 4 sollte aber aus unten genannten Gründen ebenfalls eingehalten werden.

1. *Uniformität*:  $0 \leq d(i, j) \leq 1 \quad \forall i, j \in D$

*Aussage:* Die errechneten Werte sollten sich immer zwischen und einschließlich 0 und 1 befinden. *Libmunin* schneidet die Werte nötigenfalls auf diesen Bereich zu.

2. *Symmetrie*:  $d(i, j) = d(j, i) \quad \forall i, j \in D$

*Aussage:* Die Reihenfolge, in der die Songs der Distanzfunktion übergeben werden, darf keine Auswirkung auf das Ergebnis haben. Diese Eigenschaft wird von *libmunin* nicht überprüft — eine Nichteinhaltung würde zu falschen Kanten im Graphen führen.

3. *Identität:*  $d(i, i) = 0 \quad \forall i \in D$

*Aussage:* Wird zweimal der selbe Song übergeben, so muss die Distanz immer 0 betragen. Autoren von Distanzfunktionen sollten dies testen. Werte  $\neq 0$  deuten auf fehlerhafte Distanzfunktionen hin.

4. *Dreiecksungleichung:*  $d(i, j) \leq d(i, x) + d(x, j) \quad \forall i, j, x \in D, i \neq j \neq x$

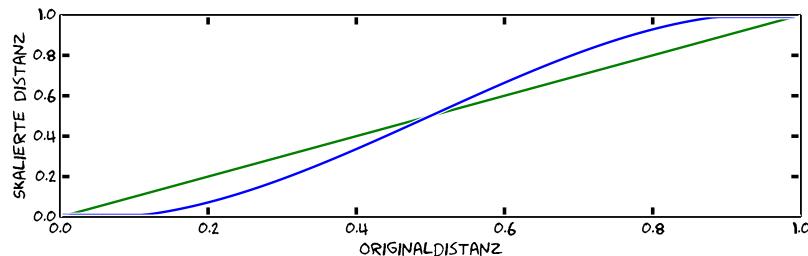
*Aussage:* In einer Dreiecksbeziehung zwischen drei Songs muss der direkte Weg zwischen zwei Songs immer kürzer oder gleich lang wie der Umweg über den dritten Song sein. Dies ist in Abbildung 2.1 gezeigt. Diese Eigenschaft ist nötig, damit man annehmen kann, dass direkte Nachbarn ähnlicher sind als indirekte Nachbarn.

## 2.3 Zur Nutzung von *libmunin*

Die Qualität der Empfehlungen kann nur so gut sein, wie die Qualität der Eingabedaten. Da in den meisten Fällen die Metadaten zu den einzelnen Liedern aus den Tags der Audiodateien kommen, empfiehlt es sich, diese vorher mit Musiktaggern einheitlich zu pflegen. Der Autor empfiehlt hierfür *Picard* [Link-3], welches im Hintergrund auf *Musicbrainz* [Link-4] zugreift. Für schwerer zu besorgende Daten, wie Liedtexte, kann unter anderem auf *libglyr* [Link-5], *beets* [Link-6] oder dem eingebauten *PlyrLyricsProvider* (sucht im Web nach Liedtexten) und *DiscogsGenreProvider* (sucht bei Discogs [Link-7] nach der Genreberechnung) zurückgegriffen werden.

Welche Lieder man zu *libmunin's Historie* hinzufügt, sollte abgewogen werden. Fügt man auch Lieder ein, welche vom Nutzer einfach übersprungen worden sind, so sind die erstellten Regeln nicht repräsentativ. Es sollten nur Lieder hinzugefügt werden, welche mehr als 50% angehört worden sind.

Um das Format der Musiksammlung zu spezifizieren, muss der Nutzer der Bibliothek bei einer neuen Sitzung eine Maske angeben. In dieser werden die Provider und Distanzfunktionen für die einzelnen Attribute eines Songs festgelegt. Mit der *EasySession* bietet *libmunin* aber eine Sitzung mit vorgefertigter Maske. Anwendungsentwickler sollten aber nach Möglichkeit eine eigene, für ihre Zwecke konfigurierte, Session-Maske verwenden. Zwar ist der Einsatz der vorgefertigten *EasySession* deutlich einfacher, doch ist diese mehr für den schnellen Einsatz gedacht. Zudem sollte es dem Endanwender möglich gemacht werden, die Gewichtungen der einzelnen Attribute zu ändern.



**Abbildung 2.2.:** Die blaue Kurve zeigt die skalierten Werte der Distanzfunktion in Blau. Werte unter 0,5 werden etwas herabgesetzt, schlechtere Werte über 0,5 werden erhöht. Zur Referenz ist die ursprüngliche Gerade in Grün gegeben.

## 2.4 Zur Erweiterung von *libmunin*

Oft ist es von Interesse neue Distanzfunktionen und Provider für eigene Zwecke zu schreiben. Beispielsweise könnte man ein Paar aus Provider und Distanzfunktion verfassen um die einzelnen Mitglieder einer Band automatisch aus dem Netz zu besorgen und mit anderen Bands zu vergleichen, um Relationen zu finden. Im Folgenden werden einige Beispiele gegeben und Stolperfallen aufgelistet.

### 2.4.1 Hinweise zum Schreiben von Distanzfunktionen

Wenn eine Distanzfunktion eine Menge von Elementen vergleichen muss, so besteht dieselbe oft aus einem *Fusionierungsverfahren* und einer weiteren Metrik, die die einzelnen Elemente untereinander vergleicht. Ein Fusionierungsverfahren verschmilzt mehrere Teildistanzen auf definierte Weise zu einer Gesamtdistanz. Als Beispiel kann man hier den Vergleich von zwei Mengen von Wörtern nennen. Einzelne Wörter kann man relativ einfach auf Ähnlichkeit untersuchen<sup>1</sup>. Ein simples Fusionierungsverfahren wäre hier, jedes Wort aus der einen Menge mit jedem Wort aus der anderen Menge zu vergleichen und den Durchschnitt der Einzeldistanzen als Ergebnis anzunehmen. Ein anderes Fusionierungsverfahren nimmt statt dem Durchschnitt die kleinste gefundene Distanz. Hier gibt es kein richtig oder falsch. Je nach Einsatzzweck, muss ein passendes Verfahren gewählt werden. Der dazugehörige Wikipedia-Artikel bietet, unter dem Punkt Fusionierungsalgorithmen, einen guten Überblick über weitere Verfahren: [Link-8].

Distanzfunktionen sollten schlechte Werte abstrafen und gute belohnen. Während der Entwicklung hat sich gezeigt, dass simple Distanzfunktionen, die auch für gar nicht mehr ähnliche Werte eine Distanz errechnen, die  $\neq 1,0$  ist, zu qualitativ schlechten Verbindungen im Graphen führen. Man sollte daher den Bereich, in denen man eine Distanz  $< 1,0$  vergibt, einschränken.

Im folgendem Beispiel wird dies nicht getan und in der nachfolgenden Version verbessert:

<sup>1</sup> Etwa mit der Levenshtein-Distanzfunktion [3] und der Python-Bibliothek pyxDamerauLevenshtein [Link-11].

```

from munin.distance import DistanceFunction

# Eine Distanzfunktion, die beispielsweise ein Rating von 1-5 vergleicht.
# Leite von der Distanzfunktions-Oberklasse ab:
class WrongDistanceFuntion(DistanceFunction):
    def do_compute(self, A, B):
        # A und B sind, der Konsistenz halber auch bei einzelnen Werten immer Tupel
        # Daher müssen wir diese erst "entpacken".
        a, b = A[0], B[0]
        return abs(a - b) / max(a, b) # Teile Differenz durch Maximum aus beiden:

class CorrectDistanceFuntion(DistanceFunction):
    def do_compute(self, A, B):
        diff = abs(A[0] - B[0])
        if diff > 3:
            return 1.0 # Zu unterschiedlich.
        return diff / 4 # Verteile auf [0, 0.25, 0.5, 0.75]

```

Manchmal ist eine Eingrenzung des Bereichs nicht so einfach möglich, vor allem wenn komplexere Daten im Spiel sind. Dann empfiehlt es sich, die Verteilung der Distanz auf den Bereich zwischen 0,0 und 1,0 zu untersuchen. Sollte sich die Distanz beispielsweise gehäuft im Bereich zwischen 0,3 und 0,7 bewegen, so ist es empfehlenswert diesen Bereich zu dehnen. In Abbildung 2.2 werden mit der Funktion<sup>2</sup>  $f(x) = -2\frac{2}{3}x^3 + 4x^2 - \frac{1}{3}x$  Distanzen unter 0,5 verbessert und darüber verschlechtert.

## 2.4.2 Hinweise zum Schreiben von neuen Providern

Provider laufen im Gegensatz zu Distanzfunktionen nur einmal. Sie sind als Präprozessor zu verstehen, der die vom Nutzer eingegebenen Daten auf möglichst einfache und effiziente Vergleichbarkeit optimiert. Die Laufzeit, die er dafür braucht, ist daher im Vergleich zur Distanzfunktion vernachlässigbar. Daher sollte gut abgewogen werden, wieviele Daten man dem Provider produzieren lässt. Im Zweifelsfall, empfiehlt es sich, Unnötiges wegzulassen. Ist zu erwarten, dass stark redundante Daten eingepflegt werden, dann sollte die provider-interne Kompression genutzt werden. Ein typisches Beispiel dafür ist der Künstlername. Dieser ist für sehr viele Songs gleich. Daher wäre eine separate Speicherung desselben nicht sinnvoll. Intern bildet eine bidirektionale Hashtabelle<sup>3</sup> (mittels des Python-Pakets `bidict` [Link-9]) gleiche Werte auf einen Integer-Schlüssel ab. Dies wird im folgenden Python-Beispiel gezeigt:

```

from munin.provider import Provider

```

<sup>2</sup> Die Werte der Funktion können leicht unter 0 und über 1 gehen. Um den Begriff der Distanz einzuhalten, werden die Werte auf den Bereich [0, 1] zugeschnitten.

<sup>3</sup> Eine Hashtabelle ist eine Datenstruktur, die eine effiziente Abbildung von eindeutigen Schlüsselwerten auf beliebige Werte möglich macht. Der Aufwand für den Zugriff auf einzelne Werte ist dabei konstant.

```
class DoublingProvider(Provider): # Leite von der Provider-Oberklasse ab.
    def __init__(self):
        # Kompression anschalten, ansonsten muss auf nichts geachtet werden.
        Provider.__init__(self, compress=True)

    def do_compute(self, input_value): # Wird bei jeder Eingabe aufgerufen.
        return input_value * 2 # Verdoppele den Input.
```

## 2.5 Vergleich verschiedener Playlisten

Eine *Playlist*, zu deutsch *Wiedergabeliste*, ist eine Liste einzelner Lieder, die nacheinander abgespielt werden. Die Zusammstellung einer Playlist erfüllt oft einen gewissen Zweck. So stellt man für gewöhnlich Lieder in einer *Playlist* zusammen, die eine gemeinsame Stimmung oder eine andere Gemeinsamkeit („*Favoriten*“) besitzen. Im Folgenden wird die subjektive Qualität der Playlisten bezüglich der Ähnlichkeit der einzelnen Stücke beurteilt.

In Abbildung 2.4 wird eine Auflistung verschiedener, mit unterschiedlichen Methoden erstellter Playlisten gegeben. Dies ist interessant, da die Struktur der von *libmunin* gegebenen Empfehlungen gewissen Regeln unterliegt, die man als Anwendungsentwickler kennen sollte. Zudem ist der subjektive Vergleich mit anderen Systemen interessant.

Der ursprüngliche Plan, hier auch eine von `last.fm` [Link-10] erstellte Playlist zu zeigen, wurde eingestellt, da man dort die Empfehlungen nicht auf die hier verwendete Testmusiksammlung aus 666 Songs einschränken konnte. Stattdessen wurde eine Alternative zu *libmunin* getestet: *Mirage* [2]. Da *Mirage* momentan nur als Plugin für Banshee vorhanden ist und nicht als allgemeine Bibliothek verfügbar, wurde die Testmusikdatenbank auch in Banshee importiert. Die Testmusikdatenbank selbst besteht aus einigen ausgewählten Alben des Autors. Viele allgemein gebräuchliche Genres werden dabei abgedeckt, obwohl der Schwerpunkt beim Genre *Rock* und *Metal* liegt. Die einzelnen Playlisten wurden auf jeweils 15 Songs begrenzt. Darin enthalten ist an erster Stelle der willkürlich ausgewählte Seedsong, der zum Generieren der Playlist genutzt wurde (*Knorkator – Böse*). Die zufällig erstellte Playlist wurde als Referenz abgedruckt, damit man die dort fehlende Struktur sehen kann.

### Auffälligkeiten:

Bei *libmunin* wiederholt sich der Künstler *Knorkator* alle 3–5 Stücke, da der *Filter* entsprechend eingestellt ist. Daher ist eine Wiederholung des Künstlers nur alle drei und eine Wiederholung des Albums nur alle fünf Stücke erlaubt. Bei *Mirage* scheint lediglich eine direkte Wiederholung des Künstlers ausgeschlossen zu sein. Ansonsten wiederholen sich die Künstler beliebig. Die zufällige Playlist hat zwar auch keinerlei Wiederholungen, aber entbehrt dafür auch jeglicher Struktur.

*Mirage* leistet gute Arbeit dabei, ähnlich klingende Stücke auszuwählen. Der tempomäßig vergleichsweise langsame Seedsong (*Mirage* besitzt hier tatsächlich ein ähnliches Konzept) besitzt eine dunkle Stimmung und harte E-Gitarren. Die von *Mirage* vorgeschlagenen Songs sind hier tatsächlich sehr passend zu dieser Stimmung. Die von *libmunin* vorgeschlagenen Songs sind in Punkt Audiodaten, bei weitem nicht so übereinstimmend. Was aber auffällig ist, ist dass größtenteils deutsche Titel (wie der Seedsong) vorgeschlagen werden. Auch führt das *Parody* in der Genre-Beschreibung dazu, dass ebenfalls lustig oder ironisch gemeinte Lieder vorgeschlagen werden. Zwar ist die Stimmung im Seedsong düster, doch wird textlich ein Thema ironisch behandelt – was *Mirage* an den Audiodaten natürlich nicht erkennen kann. Hier zeigt sich *libmunin*'s (momentaner) Fokus auf Metadaten. Bei der zufälligen Playlist stimmen die Genres einigermaßen überein, doch liegt das eher an dem sehr dehbaren Begriff *Rock*, der bei Discogs [Link-7] für sehr viele Lieder eingepflegt ist.

Der Kaltstart bei *Mirage* verlief in wenigen Minuten, während der Kaltstart bei *libmunin* beim ersten Mal für die 666 Songs im Vergleich dazu sehr lange (etwa 53 Minuten) benötigte. Größtenteils liegt das daran, dass für jedes Lied ein Liedtext sequentiell automatisch besorgt wird. Siehe dazu auch Tabelle 2.3. Bei der Ausgabe der Empfehlungen selbst, war bei allen Methoden keinerlei Verzögerung zu beobachten.

## 2.6 Ressourcenverbrauch

Damit Anwendungsentwickler die Aufwändigkeit einzelner Operation einschätzen können, wird in Tabelle 2.3 eine kurze Übersicht über den Ressourcenverbrauch einzelner Aspekte gegeben. Die gemessenen Werte beziehen sich stets auf die Testumgebung mit 666 Songs.

| Operation                                       | Ressourcenverbrauch                               |
|---|---|
| <i>Speicherverbrauch</i>                        | 77,5 MB   |
| <i>Speicherplatz der Session (gzip-gepackt)</i> | 0,9 MB  |
| <i>Speicherplatz der Session (ungepackt)</i>    | 2,5 MB  |
| <i>Zeit für den Kaltstart</i>                   | 53 Minuten (63% Liedtextsuche + 37% Audioanalyse) |
| <i>rebuild</i>                                  | 44 Sekunden                                       |
| <i>add</i>                                      | 87ms  |
| <i>insert</i>                                   | 164ms   |
| <i>remove</i>                                   | 54ms  |
| <i>modify</i>                                   | 219ms   |

Abbildung 2.3.: Auflistung des Ressourcenverbrauchs verschiedener Operationen.

Wie man sieht, sollte noch unbedingt Zeit investiert werden um den *Kaltstart* zu beschleunigen. Auch die *modify*-Operation könnte durchaus noch optimiert werden. Wie allen anderen Geschwindigkeitsangaben in dieser Arbeit, beziehen sich diese auf den Rechner des Entwicklers und sind daher nur untereinander vergleichbar.

| Nummer           | Künstler            | Titel                 | Genre                        |
|------------------|---------------------|-----------------------|------------------------------|
| <b>libmunin:</b> |                     |                       |                              |
| 01               | Knorkator           | Böse                  | Rock/Parody, Heavy Metal     |
| 02               | Letzte Instanz      | Egotrip               | Rock/Folk Rock, Goth Rock    |
| 03               | Nachtgeschrei       | Lass mich raus        | Rock/Folk Rock               |
| 04               | Knorkator           | Ick wer zun Schwein   | Rock/Parody, Heavy Metal     |
| 05               | Finntröll           | Svart djup            | Rock/Folk Metal, Black Metal |
| 06               | Heaven Shall Burn   | Endzeit               | Rock/Hardcore, Death Metal   |
| 07               | In Extremo          | Liam                  | Rock/Medieval, Hard Rock     |
| 08               | Knorkator           | Konflikt              | Rock/Parody, Heavy Metal     |
| 09               | Letzte Instanz      | Schlangentanz         | Rock/Folk Rock, Goth Rock    |
| 10               | Marc-Uwe Kling      | Scheißverein          | Folk/Parody                  |
| 11               | Johnny Cash         | Heart of Gold         | Folk/Country, Rockabilly     |
| 12               | Knorkator           | Geh zu ihr            | Rock/Parody, Heavy Metal     |
| 13               | In Extremo          | Erdbeermund           | Rock/Medieval, Hard Rock     |
| 14               | The Rolling Stones  | Stealing My Heart     | Rock/Pop Rock, Rock & Roll   |
| 15               | Knorkator           | Klartext              | Rock/Parody, Heavy Metal     |
| <b>Mirage:</b>   |                     |                       |                              |
| 02               | Knorkator           | Ganz besond'rer Mann  | Rock/Parody, Heavy Metal     |
| 03               | Coppelius           | Operation             | Rock/Classic, Medieval Metal |
| 04               | Letzte Instanz      | Salve Te              | Rock/Folk Rock, Goth Rock    |
| 05               | Apocalyptica        | Fisheye               | Rock/Symphonic Rock          |
| 06               | Coppelius           | I Told You So!        | Rock/Classic, Medieval Metal |
| 07               | Apocalyptica        | Pray!                 | Rock/Symphonic Rock          |
| 08               | Knorkator           | Klartext              | Rock/Parody, Heavy Metal     |
| 09               | Devildriver         | Black Soul Choir      | Rock/Death Metal             |
| 10               | Finntröll           | Fiskarens Fiende      | Rock/Folk Metal, Black Metal |
| 11               | Devildriver         | Swinging the Dead     | Rock/Death Metal             |
| 12               | Knorkator           | Es kotzt mich an      | Rock/Parody, Heavy Metal     |
| 13               | Heaven Shall Burn   | Forlorn Skies         | Rock/Hardcore, Death Metal   |
| 14               | Knorkator           | Hardcore              | Rock/Parody, Heavy Metal     |
| 15               | Rammstein           | Roter Sand            | Rock/Industrial, Hard Rock   |
| <b>Zufall:</b>   |                     |                       |                              |
| 02               | Schandmaul          | Drei Lieder           | Rock/Folk Rock               |
| 03               | Tanzwut             | Götterfunken          | Electronic, Industrial       |
| 04               | Finntröll           | Suohengen sija        | Ambient                      |
| 05               | Biermösl Blosn      | Anno Domini           | Brass Band, Parody           |
| 06               | Finntröll           | Mordminnen            | Rock/Folk Metal, Black Metal |
| 07               | The Rolling Stones  | Stealing My Heart     | Rock/Pop Rock, Rock & Roll   |
| 08               | Die Ärzte           | Ein Mann              | Rock/Punk, Pop Rock          |
| 09               | Letzte Instanz      | Regenbogen            | Rock/Folk Rock, Goth Rock    |
| 10               | Billy Talent        | White Sparrows        | Rock/Punk, Alternative Rock  |
| 11               | Letzte Instanz      | Schlangentanz         | Rock/Folk Rock, Goth Rock    |
| 12               | Christopher Rhyne   | Shadows of the Forest | Classical, Ambient           |
| 13               | The Beatles         | Eight Days a Week     | Pop/Rock & Roll              |
| 14               | Of Monsters and Men | From Finner           | Pop/Folk, Indie Rock         |
| 15               | The Cranberries     | Dreaming My Dreams    | Rock/Alternative Rock        |

**Abbildung 2.4.:** Vergleich verschiedener, je 15 Lieder langen Playlisten. Die Playlist im oberen Drittel wurde mittels des Seedsongs (01) erstellt. Die im zweitem Drittel wurde mittels Mirage/Banshee erstellt, die letzte wurde komplett zufällig generiert.

# 3 | Graphenoperationen

## 3.1 Einleitung

EINE grobe Übersicht über die einzelnen Graphenoperationen und ihrer Zuständigkeiten wurde bereits in der Projektarbeit ([1], S.15f) gegeben. Im Folgenden wird detailliert auf ihre Funktionsweise und Internas eingegangen.

### 3.1.1 rebuild: Aufbau des Graphen

Bevor irgendeine andere Operation ausgeführt werden kann, muss mittels der `rebuild`-Operation der Graph aufgebaut werden. Aufgrund einer Komplexität von  $O(n^2)$  kann der Aufbau des Graphen nicht einfach durch das Vergleichen aller Songs untereinander erfolgen. Daher muss eine Lösung mit subquadratischen Aufwand gefunden werden. Vorzugsweise eine, bei der der Rechenaufwand gegen die Qualität der Approximation abgewägt werden kann. So kann der Nutzer entscheiden, wie lange er *libmunin* rechnen lassen will.

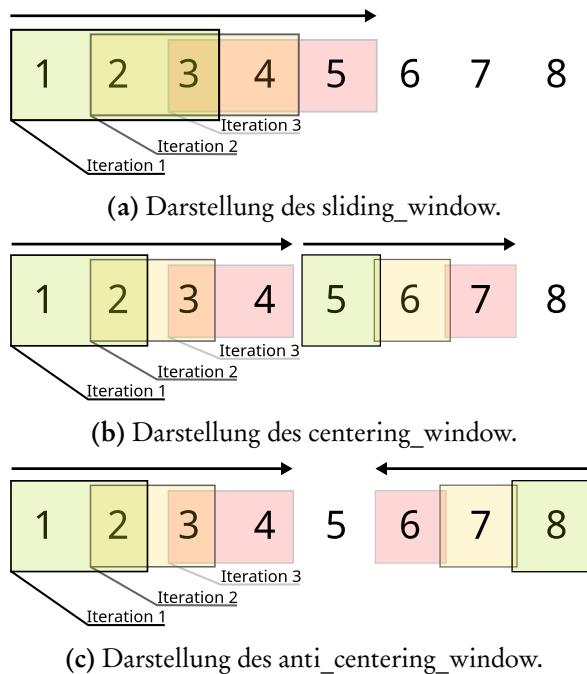
Der Ausgangszustand der `rebuild`-Operationen ist eine Liste von Songs, die vom Nutzer bereitgestellt wird. Jeder Song darin soll nun so im Graphen platziert werden, dass er im Bestfall die ähnlichsten Songs als Nachbarn hat.

Jeder Song speichert seine Nachbarn mit der dazugehörigen Distanz. Soll ein neuer Nachbar hinzugefügt werden, so wird geprüft ob die Distanz zu diesem neuen Song kleiner ist, als die zum schlechtesten vorhandenen Nachbar. Ist dies der Fall, so wird die Entfernung zu diesem schlechtesten Nachbarn *in eine Richtung* (die Gründe hierfür werden unter 3.1.7 betrachtet) gekappt. Als Ersatz wird zu dem neuen, besseren Song, eine bidirektionale Verbindung aufgebaut. Da die Verbindung zum schlechtesten Song nur unidirektional abgebaut wird, ist die Anzahl der Nachbarn eines Songs nicht auf ein Maximum begrenzt, da das Hinzufügen neuer Songs „*Einbahnstraßen*“ hinterlässt. Jedes neue Einfügen kann schließlich Einbahnstraßen hinterlassen.

Vielmehr handelt es sich dabei um einen Richtwert, um den sich die tatsächliche Anzahl der Songs einpendeln wird. Momentan ist dieser Richtwert standardmäßig auf 15 gesetzt — der durchschnittlichen Länge eines heutigen Albums plus eins<sup>1</sup>. Dieser Wert hat sich nach einigen Tests als passabel erwiesen.

---

<sup>1</sup> Bestimmt an der persönlichen Sammlung des Autors. Bei 1590 einzelnen Alben ist dieser Wert etwa 14,142.



**Abbildung 3.1.:** Schematische Darstellungen der einzelnen Basisiterationen. Es werden jeweils drei Iterationen in einem Bild dargestellt. Das Fenster in der ersten Iteration ist dabei jeweils grün, in der zweiten gelb und in der letzten rot dargestellt. Die Zahlen repräsentieren einzelne Songs.

sen. Bei zu niedrigen Werten verbinden sich die einzelnen Songs eines Albums nur untereinander, bei zu hohen entstehen zu viele qualitativ schlechte Verbindungen quer über den ganzen Graphen.

Wenn im folgenden vom „*Berechnen der Distanz*“ gesprochen wird, so ist damit auch das Hinzufügen der Distanz zwischen beiden Songs gemeint. Jeder Song verwaltet eine Hashtabelle mit seinen Nachbarn. Eine Kante ist, technisch gesehen, das gegenseitige Vorhandensein von zwei Songs in deren jeweiligen Tabellen.

Im Folgenden werden die drei Schritte der `rebuild`-Operation genauer beleuchtet:

- **Basisiteration:** Für jeden Song wird nach willkürlich festgelegten Prinzipien eine kleine Menge von möglicherweise ähnlichen Songs ausgewählt. Diese Menge von Songs wird untereinander mit quadratischen Aufwand verglichen. Diese Vorgehensweise wird mehrmals mit verschiedenen Methoden wiederholt. Das Ziel jeder dieser Iterationen ist es, für einen Song zumindest eine kleine Anzahl von ähnlichen Songs zu finden. Basierend auf diesen wird in den nächsten Schritten versucht, die Anzahl ähnlicher Songs zu vergrößern.

Momentan sind drei verschiedene Iterationsstrategien implementiert. Jede basiert auf gewissen heuristischen Annahmen, die über die Eingabemenge gemacht werden (siehe Abbildung 3.1):

- `sliding_window`: Schiebt ein „Fenster“ über die Liste der Songs. Alle Songs innerhalb des Fensters werden untereinander verglichen. Die Fenstergröße ist dabei konfigurierbar

und ist standardmäßig auf 60 eingestellt, da sich diese Größe nach einigen Tests als guter Kompromiss zwischen Qualität und Geschwindigkeit herausgestellt hat. Bei jeder Iteration wird das Fenster um ein Drittel der Fenstergröße weitergeschoben. Dadurch entsteht eine „*Kette*“ von zusammenhängenden Songs.

Die heuristische Annahme ist dabei, dass der Nutzer der Bibliothek seine Datenbank meist nach Alben sortiert eingibt. Durch diese Sortierung finden sich innerhalb eines Fensters oft Lieder desselben Albums — diese sind oft sehr ähnlich.

- `centering_window`: Basiert ebenfalls auf einem Fenster. Im Gegensatz zum obigen `sliding_window` besteht das Fenster allerdings aus zwei Hälften, wobei die eine vom Anfang an startet und die andere Hälfte von der Mitte aus bis zum Ende geschoben wird. Die Songs in beiden Hälften werden analog zu oben untereinander verglichen. Auch hier wird das Fenster immer zu einem Drittel der Fenstergröße weitergeschoben.

Die heuristische Annahme ist hier, dass in der bereits vorhandenen „*Kette*“ Querverbindungen hergestellt werden. Dies ist im folgenden Verfeinerungsschritt vorteilhaft um Iterationen einzusparen.

- `anti_centering_window`: Sehr ähnlich zum `centering_window`. Statt die zwei Hälften von der Mitte aus bis zum Ende weiter zu schieben, wird diese vom Ende zur Mitte geschoben. So werden die beiden Hälften solange weiter geschoben, bis sie sich in der Mitte treffen. Auch hier sollen weitere Querverbindungen hergestellt werden.
- **Verfeinerung:** Um den momentan sehr grob vernetzten Graphen benutzbar zu machen, sollten einige Iterationen zur „*Verfeinerung*“ durchgeführt werden (eine Gegenüberstellung von Vorher und Nachher wird im Anhang unter Abb. gezeigt). Dabei wird über jeden Song im Graphen iteriert und dessen *indirekte Nachbarn* (also die Nachbarn der direkten Nachbarsongs) werden mit dem aktuellen Song verglichen. Kommen dabei Distanzen zustande, die niedriger sind als die der aktuellen Nachbarn, wird der indirekte Nachbar zum direkten Nachbarn. Auf diese Weise nähern sich ähnliche Songs immer weiter an. Diese Vorgehensweise wird solange wiederholt, bis nur noch eine geringe Anzahl von Songs „*bewegt*“ wird oder bis eine maximale Anzahl von Iterationen erreicht ist. Die Begrenzung der Iterationen ist nötig, da es Fälle geben kann, in denen einzelne Songs immer wieder zwischen zwei gleich guten Zuständen hin- und herspringen können.

Als zusätzliche Optimierung werden nicht alle indirekten Nachbarn betrachtet, sondern nur diese, zu denen der Weg eine *Mindestdistanz* nicht überschreitet. Diese Mindestdistanz wird beim Start auf 2,0 (da ja die Distanz über zwei Kanten gemessen wird) gesetzt und während der folgenden Iterationen immer weiter abgesenkt.

Die Gesetzmäßigkeit, nach der die Mindestdistanz immer weiter abgesenkt wird, berechnet sich dabei aus dem arithmetischen Mittelwert der bis dahin berechneten Distanzen. Ist der Mittelwert hoch, so ist die Absenkung klein.

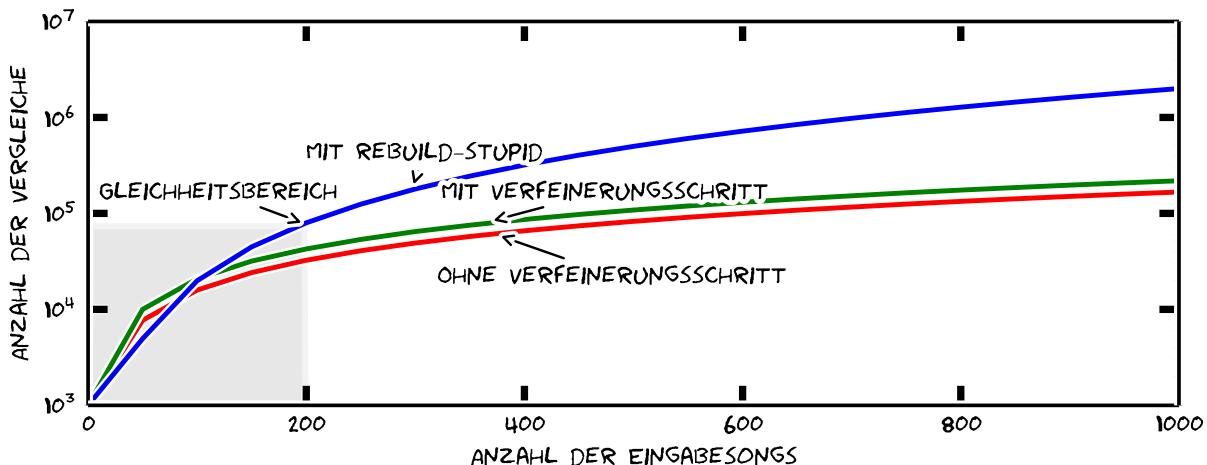


Abbildung 3.2.: Gegenüberstellung von verschiedenen Arten der rebuild-Operation. Auf der Y-Achse ist logarithmisch die Anzahl der Distanzberechnungen aufgetragen, auf der X-Achse die lineare Anzahl der Eingabesongs. Die blaue Kurve repräsentiert dabei die Vergleiche die für rebuild\_stupid notwendig sind. Wie man sieht, übersteigen diese, bis auf dem Gleichheitsbereich am Anfang, die anderen zwei Kurven deutlich.

- **Aufräumarbeiten:** Nach dem Verfeinerungsschritt wird der Graph von Einbahnstraßen durch einen fixing-Schritt bereinigt und auf Konsistenz geprüft.

Wie bereits erwähnt, gibt es eine `rebuild_stupid`-Operation, die für deutlich kleinere Mengen von Songs praktikabel einsetzbar ist. Die Algorithmik ist hierbei bedeutend einfacher: Es wird einfach jeder Song mit jedem anderen verglichen. Als Nachbarn erhält dabei jeder Song die Nachbarn, die global betrachtet, die kleinste Distanz zu diesem besitzen. Es handelt sich also um keine Approximation wie beim herkömmlichen `rebuild`.

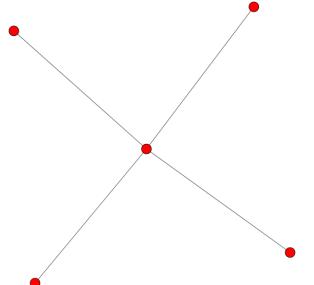
Auf die Betrachtung der Komplexität der `rebuild`-Operation, wird an dieser Stelle verzichtet. Keine der einzelnen Schritte erreicht dabei quadratische Komplexität. Die einzige Ausnahme ist dabei, das Vergleichen der Songs untereinander innerhalb eines Fensters, allerdings ist dabei die Fenstergröße stets auf ein verträgliches Limit begrenzt.

Unter Abbildung 3.2 findet sich eine Gegenüberstellung von den Aufrufen der Distanzfunktion, die bei `rebuild_stupid` und beim normalen `rebuild` (mit und ohne Verfeinerungsschritt) nötig sind.

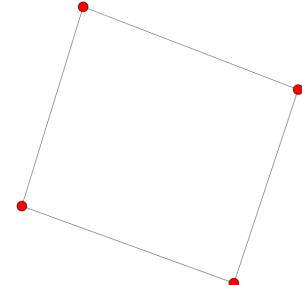
Die einzelnen Schritte des Graphenaufbaus lassen sich in Abbildung A.1, sowie bei den darauf folgenden Abbildungen, nachvollziehen.

### 3.1.2 fixing: Umbauen von Einbahnstraßen

Diese Operation dient dem Entfernen von Einbahnstraßen innerhalb des Graphen. Einbahnstraßen können, wie bereits erwähnt, beim Hinzufügen neuer Distanzen entstehen.



(a) Vor der remove-Operation.



(b) Nach der remove-Operation.

**Abbildung 3.3.:** Vor (3.3a) und nach (3.3b) der remove-Operation. Es wurde der mittlere Punkt in 3.3a gelöscht. Daher haben sich alle anderen Knoten einen anderen Nachbarn gesucht.

Beim Entfernen wird folgendermaßen vorgegangen: Im ersten Schritt werden alle unidirektionale Kanten gefunden und abgespeichert. Für jede dieser Kanten wird überprüft, ob die Songs an beiden Enden, den Richtwert für die Anzahl der Nachbarn überschreiten. Sollte das nicht der Fall sein, so wird die Kante in eine bidirektionale Kante umgebaut. Andernfalls wird die Kante gelöscht.

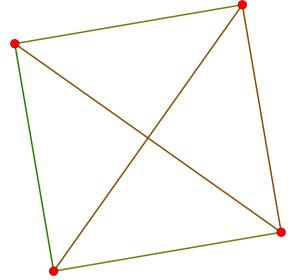
Dieses Vorgehen wurde gewählt, weil es nach einigen Versuchen schwierig erschien, den Graphen ohne Einbahnstraßen aufzubauen, ohne dass dieser zur Inselbildung neigt. Durch den nachgelagerten **fixing**-Schritt werden Songs, die nur wenige Nachbarn besitzen, durch die vorher als zu schlecht bewerteten Kanten verbunden. Als zusätzliche Konsistenzprüfung wird nach dem Bereinigen geprüft, ob alle Verbindungen im Graphen bidirektional sind. Sollten unidirektionale Kanten gefunden werden, so wird eine Warnung auf der Konsole ausgegeben. Eine weiterführende Fehlerbehandlung ist momentan noch nicht implementiert. Unidirektionale Kanten können bei der Traversierung zu Ausnahmefehlern führen.

### 3.1.3 add: Hinzufügen von Songs vor dem rebuild

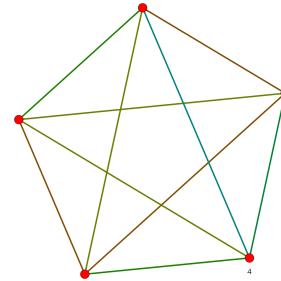
Diese Operation benötigt als Argument eine Hashtabelle mit einer Abbildung von Attributen auf Werte. Diese Werte werden dann durch verschiedene Provider normalisiert. Mit diesen normalisierten Informationen, wird dann eine neue Song-Instanz erzeugt, welcher beim Erzeugen, ein eindeutiger Identifier zugewiesen wird. Dieser Identifier dient dann als Index in der internen Songliste. Statt wie **insert**, bereits Verbindungen zu anderen Songs herzustellen, fügt diese Operation lediglich einen Song der internen Songliste hinzu. Die eigentlichen Verbindungen werden in einem Rutsch von **rebuild** aufgebaut.

### 3.1.4 remove: Löschen von Songs zur Laufzeit

Um nach einer **rebuild**-Operation einen Song aus dem Graphen zu löschen, müssen alle Verbindungen zu diesem entfernt werden. Um dabei eine Bildung von Inseln (durch das Entfernen von

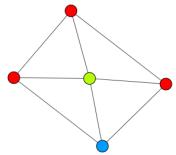
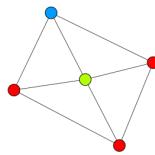


(a) Vor der insert-Operation.

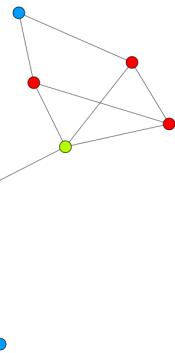


(b) Nach der insert-Operation.

**Abbildung 3.4.:** Vor (3.4a) und nach (3.4b) der insert-Operation. Es wurde einfach ein weiterer Punkt in den Graphen eingefügt. Dieser hat sich mit allen anderen verbunden.



(a) Vor der modify-Operation.



(b) Nach der modify-Operation.

**Abbildung 3.5.:** Vor (3.5a) und nach (3.5b) der modify-Operation. Es wurden jeweils die Mittelknoten der beiden Inseln mit einem höheren Rating „modifiziert“. Dadurch verbinden sich beide und verlieren dafür eine andere Verbindung jeweils.

Verbindungen) zu vermeiden, werden alle ursprünglichen Nachbarn des zu entfernenden Songs untereinander verbunden. Dabei wird nach folgendem Schema vorgegangen: Zuerst wird temporär für jeden Nachbarn, der Richtwert für die Anzahl der Nachbarn um eins erhöht. Im Anschluss wird die Menge aller Nachbarn untereinander mit quadratischem Aufwand verglichen. Dadurch bekommt jeder Nachbar, im besten Fall, eine neue Verbindung. Abschließend werden alle Verbindungen zum zu löschen Song entfernt und der Richtwert wird wieder um eins dekrementiert.

Da *libmunin* alle Songs in einer linearen Liste hält, muss auch dort der Song gelöscht werden. Da der Index des Songs in der Liste gleich des *Identifiers* des Songs ist, wird an dessen Stelle ein leerer Wert geschrieben. Damit dieser möglichst bald wieder besetzt werden kann, wird der gelöschte *Identifier-Index* in einer sogenannten *Revocation*-Liste gespeichert. Beim nächsten add oder insert wird dieser *Identifier* dann wiederverwendet. Dieses Verfahren soll eine Fragmentierung der Song-Liste nach vielen remove-Operation vermeiden.

### 3.1.5 insert: Hinzufügen von Songs zur Laufzeit

Diese Operation ist äquivalent zur `add`-Operation. Als Erweiterung fügt `insert` allerdings den, durch `add` erzeugten Song auch in den Graphen ein und verbindet ihn dort. Dazu muss zuerst ein Punkt gefunden werden, an dem der Song passend zu seinen Attributen *eingepasst* werden kann.

Diese Einpassung geschieht dabei folgendermaßen:

- **Basisiteration:** Es wird mit einer gewissen *Schrittweite* über die Songliste iteriert. Dabei werden die Distanzen vom momentan aktuellen Song zum einzufügenden Song berechnet. Dadurch wird der Song bereits mit einigen anderen Songs verknüpft. Die Größe der Schrittweite ist dabei abhängig von der Länge der Songliste. Je länger die Liste ist, desto größer ist die Schrittweite. Exakt ist sie dabei folgendermaßen definiert:

$$\text{Schrittweite} = \lceil \log_{10} \text{songlist\_length} \rceil$$

- **Verfeinerung:** Songs, zu denen im vorigen Schritt eine geringe Distanz gefunden wurde, werden nun detaillierter betrachtet. Dazu wird die Distanz zu den Nachbarn dieser *guten* Songs berechnet. Dies geschieht unter der Annahme, dass die indirekten Nachbarn des einzufügenden Songs auch als potenzielle direkte Nachbarn geeignet sind.

### 3.1.6 modify: Verändern der Songattribute zur Laufzeit

Diese Operation dient als Komfortfunktion. Sie ermöglicht das Verändern der Attribute, beziehungsweise deren zugeordneten Werte eines einzelnen Songs. Würde man die Werte eines Songs manuell verändern, so müsste man alle Distanzen zu diesem Song neu berechnen. Da dies wiederum Veränderungen im ganzen Graphen hervorrufen könnte, wurden die Song-Instanzen unveränderbar („*Immutable*“) gemacht.

Die `modify`-Operation umgeht dieses Problem, indem es den Song erst durch ein `remove` entfernt und eine Kopie des ursprünglichen Songs herstellt. In dieser werden die neuen Werte gesetzt. Dieser neue, noch unverbundene Song, wird dann mittels einer `insert`-Operation in den Graphen eingepasst.

Aufgrund dieser Abfolge unterschiedlicher Operation, ist `modify` um ein Vielfaches aufwendiger (siehe dazu auch Tabelle 2.3). Es wird empfohlen, diese Operation nur für einzelne Song jeweils einzusetzen. Sollte ein bestimmtes Attribut in allen Songs geändert werden, so ist eher eine `rebuild`-Operation zu empfehlen.

### 3.1.7 Ablauf beim Hinzufügen einer Distanz

Wie bereits erwähnt, speichert jeder Song eine Hashtabelle mit den jeweiligen Songs, zu denen er eine Verbindung hält, als Schlüssel und der Distanz als Wert. Um diese Hashtabelle zu füllen, ist eine Methodik nötig, die sich nach näherer Betrachtung als schwierig zu implementieren erwies. Tatsächlich wurden an die zwei Wochen mit unterschiedlichen Herangehensweisen verbracht.

Die Anzahl von Nachbarn pro Song sollte sich um einen gewissen *Richtwert* einpendeln, den man konfigurieren kann. Daraus folgt, dass bei zu vielen Nachbarn der schlechteste Nachbar entfernt werden muss. Der anfängliche Versuch, die Verbindung zwischen den beiden Songs komplett zu löschen hatte aber ein wichtiges Problem: Die Inseln im Graphen, die jeweils ein Album repräsentierten, haben sich nur untereinander verbunden. Verbindungen dazwischen wurden immer wieder als der *schlechteste Nachbar* erkannt und entfernt. Daher neigt der entstehende Graph stark zur Inselbildung und Bildung von starken Clustern.

Die momentane Lösung ist dabei, dass der schlechteste Nachbar eine unidirektionale Verbindung zu seinem ursprünglichen Partner aufrecht erhält. Die Verbindung wird nicht bidirektional gelöscht. Der Trick ist dabei: Bei der `rebuild`-Operation werden diese *Einbahnstraßen* immer noch von einer Seite als Nachbarn erkannt. So kann insbesondere der *Verfeinerungsschritt* gut zueinander passende Songs näher aneinander ziehen. Nach dem `rebuild` werden übrig gebliebene Einbahnstraßen in normale Verbindungen umgebaut oder, falls beide Enden der Verbindung bereits „voll“ sind, gelöscht. So bleiben Songs, zu denen kein passender Partner gefunden wurde, mit dem Rest des Graphen verbunden.

Dieses Vorgehen bringt aber einige algorithmische Probleme mit sich: Das Finden des schlechtesten Nachbarn würde jeweils linearen Aufwand zum Iterieren über die Hashtabelle erfordern. Zwar kann dann die schlechteste Distanz und der dazugehörige Song zwischengespeichert werden, doch nach einigen Tests stellte sich heraus, dass in den meisten Fällen ein neuer, schlechterer Song gesucht werden muss. Das ist damit zu erklären, dass gegen Ende der `rebuild`-Operation tendenziell immer niedrigere Distanzen gefunden werden — womit immer wieder der schlechteste Song herausgelöscht werden muss.

Der momentane Ansatz speichert pro Song, neben der Hashtabelle mit den Distanzen, auch einen Heap (vergleiche [4], S.144–155) als „*Lookup-Hilfe*“. In diesem werden, entgegen der prinzipbedingten Unordnung in einer Hastabelle, die zuletzt hinzugefügten Paare aus Distanzen und Songs partiell sortiert mit einem Aufwand von  $O(\log n)$  abgelegt. Bei einem Heaps ist dabei der Wurzelknoten immer das Element mit der größten Distanz. Ist es dann nötig, eine neue, schlechteste Distanz zu finden, so kann mit einem Aufwand von  $O(\log n)$  das oberste Paar herausgenommen werden. Der Heap wird dann so umgebaut, dass das nächstslechteste Paar zum neuen Wurzelknoten wird.

Die `distance_add()`-Funktion nimmt drei Parameter. Die ersten zwei sind die Songs (im Folgenden `self` und `other`), zwischen denen eine Verbindung hergestellt werden soll. Der Letzte, ist die Distanz

mit der diese Kante gewichtet wird. Im Folgenden ist der dazugehörige Python-Code in gekürzter, vereinfachter Form als Referenz gegeben:

```
def distance_add(self, other, distance):
    """Füge eine Kante zwischen zwei Songs mit einer Distanz hinzu.

    self, other: Die beiden Songs zwischen denen die Kante hergestellt werden soll.
    distance: Die Distanz dieser Kante.

    """
    if other is self:
        return # Selbe Referenz! Kann Endlosschleifen verursachen.

    if self.worst_cache < distance and song.is_full():
        return # worst_cache ist die gespeicherte schlechteste Distanz oder None.

    if other in self.dist_dict:
        if self.dist_dict[other] < distance:
            return # Distanz zu diesem Song war bereits vorhanden und besser.

        self.worst_cache = None
        self.dist_dict[other] = other.dist_dict[self] = distance
        return # Da other bereits enthalten: Einfach updaten.

    if self.is_full():
        while True: # Finde den schlechtesten Nachbarn der noch valide ist.
            worst_dist, worst_song = self.heap[0] # Wurzelknoten
            if worst_song in self.dist_dict:
                break
            heappop(self.heap) # Probiere nächstes Element.

        if worst_dist < distance.distance:
            self.worst_cache = worst_dist
            return

        del self.dist_dict[worst_song]
        heappop(self.heap)

    # Füge neue Kante in die Hashtabellen ein:
    self.dist_dict[other] = other.dist_dict[self] = distance

    # Speichere die Paare im Heap ab:
    heappush(self.heap, (distance, other))
    heappush(other.heap, (distance, self))
    self.worst_cache = None # Hat sich möglicherweise geändert.
```

## 3.2 Graphentraversierung

Um nun tatsächlich Empfehlungen abzuleiten, muss der Graph traversiert werden. Je nach Art der Anfrage werden ein oder mehrere Zentren für eine Breitensuche, sogenannte *Seedsongs*, ausgewählt. Bei einfachen Anfragen in der Art „*Gib 10 ähnliche zu Song X aus*“, kann einfach der Song *X* als Seed-song angenommen werden. Komplexere Anfragen benötigen allerdings mehr als einen Seedsong:

- „*Gib 10 Songs aus, die ein Genre ähnlich Y haben*“
- „*Empfiehl mir 10 Songs basierend auf dem Nutzerverhalten*“

### 3.2.1 Empfehlungsiteratoren

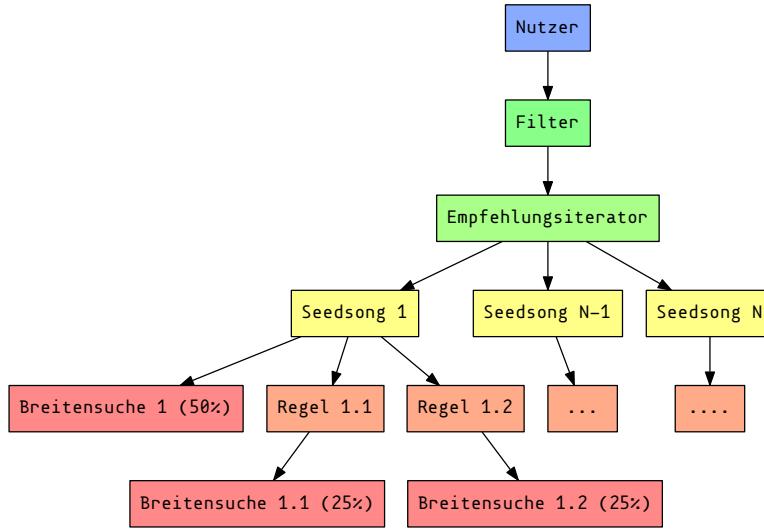
In allen Fällen wird jedoch von einem Seedsong aus eine Breitensuche gestartet. Statt diese Breitensuche *sofort* auszuführen, wird jeweils nur ein Iterator bereitgestellt, welcher immer nur eine Empfehlung generiert. Als Iterator wird bei *libmunin* ein Objekt bezeichnet, der einen internen Zustand besitzt und bei einer Anfrage immer einen neuen Wert liefert. Sind keine weiteren Werte mehr zu erwarten, so wird ein spezieller leerer Wert zurückgegeben. In der Softwareentwicklung wird er oft dazu genutzt um über die Elemente einer Menge zu traversieren. Hier werden Iteratoren dazu genutzt erst beim jeweiligen Aufruf jeweils eine Empfehlung dynamisch zu generieren.

Dieses Konzept ist sehr nützlich beim Filtern der generierten Empfehlungen. Denn man weiß im Vornherein nicht, wieviele Empfehlungen ausgefiltert werden. So kann der Iterator einfach so lange bemüht werden, bis die gewünschte Anzahl an Empfehlungen generiert worden ist.

Sollten mehrere Seedsongs vorhanden sein, so wird einfach für jeden ein Breitensuche-Iterator erstellt. Dieser liefert erst den Seedsong, dann den besten Nachbarn, dann den nächst schlechteren Nachbarn und später geht es mit den indirekten Nachbarn weiter. Diese Liste von Iteratoren wird dann im Round-Robin-Verfahren ineinander verwebt. Dabei wird erst der erste Iterator in der Liste genutzt, dann der nächste. Ist man am Ende der Liste, so wird von vorne begonnen. Der daraus entstehende Iterator wird dann dem Nutzer der Bibliothek bereitgestellt. Wird ein Element aus diesem obersten Iterator genommen, so hat das ein „*Nachrutschen*“ von Iteratoren zur Folge. Diese Hierarchie von Iteratoren ist in Abbildung 3.6 gezeigt.

### 3.2.2 Anwendung von Regeln

Die Assoziationsregeln, die beim impliziten Lernen entstehen, werden bei der Traversierung als „*Navigationshilfe*“ genutzt. In Abbildung 3.6 wird gezeigt, dass jedem Seedsong jeweils eine Breitensuche und eine Menge von *Regeliteratoren* unterstellt sind. *Libmunin* bietet einen Mechanismus, um alle Regeln abzufragen, die einen bestimmten Song betreffen. Für jeden Song, der auf der *anderen* Seite



**Abbildung 3.6.:** Traversierung durch verschachtelte Iteratoren. Jedes Kästchen ist ein Iterator. Zieht der Nutzer einen Song aus dem obersten Iterator, so löst das eine „Lawine“ von Iterationsschritten aus. Dabei werden die einzelnen Schritte „fair“ via einem Round-Robin-Verfahren auf die einzelnen Seed-Songs aufgeteilt.

der Regel vorkommt (also die Seite, in der nicht der Seedsong vorhanden ist), wird ein Breitensucheniteratator erstellt. Die einzelnen, den Regeln zugeordneten Iteratoren, werden wieder im Round-Robin-Verfahren abgewechselt. Der dadurch entstehende Iterator wird immer im Wechsel mit dem Breitensucheniteratator, der vom Seedsong ausgeht, abgefragt. Daher besteht der Iterator für einen Seedsong aus vielen Unteriteratoren.

### 3.2.3 Filtern der Iteratoren

Da Alben im Graphen eng beieinander gepackt sind, werden ohne zusätzliches Filtern natürlich auch Songs vom gleichen Album oder vom gleichen Künstler geliefert. Dies ist für gewöhnlich nicht erwünscht – man möchte ja neue Musik entdecken, die nicht immer vom selben Künstler kommt. Der optionale Filterschritt dient dazu diese unerwünschten Songs herauszufiltern.

Um dieses Ziel zu erfüllen, werden die 20 letzten Empfehlungen gespeichert, die von *libmunin* ausgegeben werden. War der Künstler einer zu überprüfenden Empfehlung in den, beispielsweise fünf letzten Empfehlungen bereits vorhanden, so wird er ausgesiebt. Ähnlich wird mit dem Album vorgegangen, nur hier ist die Schwelle standardmäßig bei drei. Die einzelnen Schwellen können vom Nutzer, pro Attribut, konfiguriert werden. Auch die Filterung ist als Iterator implementiert, welcher Songs von einem Empfehlungsiterator entgegennimmt, aber nicht alle an den Nutzer weitergibt. Die vom Iterator übergangenen Songs, werden für den nächsten Iterationsschritt zwischengespeichert, um sie vorzuschlagen, sobald sie wieder erlaubt sind.

# 4 | Algorithmen bei Providern

## 4.1 Einleitung

IM Folgenden werden einige ausgewählte Paare aus Providern und Distanzfunktionen näher betrachtet. Nicht alle vorhandenen und in der Projektarbeit ([1], S.28ff) vorgestellten Provider werden erläutert, dies würde auch den Umfang dieser Arbeit übersteigen. Zudem sind die meisten Provider eher einfacher Natur — die Lektüre des jeweiligen Quelltextes sagt oft mehr als ein separate Erklärung. Daher werden im Folgenden nur die stark erklärungsbedürftigen Paare näher betrachtet.

## 4.2 Genrenormalisierung und Vergleich von Genres

Der Vergleich einzelner Genres ist eine schwierige Angelegenheit, da es, zumindest im Bereich der Musik, keine standardisierte Einteilung von Genres gibt. Oft sind sich nicht mal Menschen untereinander einig, zu welchem Genre das Album eines Künstlers zuzuteilen ist. Manchmal sind sich nicht mal die Mitglieder einer Band untereinander einig. Ein Computer könnte höchstens erkennen, wie ähnlich zwei Genrebeschreibungen als Zeichenketten sind. Daher ist es nötig, dass die einzelnen Genre-Eingaben anhand einer Sammlung, von zusammengestellten geläufigen Genres normalisiert werden.

### 4.2.1 Zusammenstellung der Genredatenbank

Musikrichtungen können, wie in einem Baum, in Genres (*Rock, Pop*), Untergenres (*Country Rock, Japanese Pop*), Unteruntergenres (*Western Country Rock*) — und so weiter — aufgeteilt werden. So lassen sich alle Genres und ihre jeweiligen Untergenres als Baum darstellen. Als imaginären Wurzelknoten nimmt man das allumfassende Genre *Music* an — einfach weil sich *Music* hinter fast jedes Genre schreiben lässt, ohne den Sinn zu verändern. Dieser Baum kann dann genutzt werden, um beliebige Genres als *Pfad* durch den Baum normalisiert abzubilden.

Die eigentliche Schwierigkeit besteht nun darin, eine repräsentative Sammlung von Genres in diesen Baum einzupflegen. Bei der hohen Anzahl der existierenden Genres, kann man diese nur schwerlich manuell einpflegen.

Existierende Datenbanken wie, das sonst sehr vollständige *MusicBrainz*, liefern laut ihren *FAQ* keine Genredaten:

### Why Does Musicbrainz not support genre information?

*Because doing genres right is very hard. We have thought about how to implement genres, but we haven't completely settled on the right approach yet.*

—[https://musicbrainz.org/doc/General\\_FAQ](https://musicbrainz.org/doc/General_FAQ) [Link-12]

Also musste man sich nach anderen Quellen umschauen. Das vom DiscogsGenreProvider verwendete *Discogs* bietet zwar detaillierte Informationen, teilt aber die Genres hierarchisch in zwei Ebenen auf, dem Genre („Rock“) und dem Untergenre („Blackened Death Metal“) — eine zu grobe Einteilung zur Normalisierung.

Dafür fallen zwei andere Quellen ins Auge: *Wikipedia* — viele bekannte Künstler sind dort mit detaillierter Genreninformation vertreten, sowie *The Echonest* — einem Unternehmen, welches verschiedene Dienste rund um Musikmetadaten anbietet. Darunter eine Liste, von den ihnen bekannten Genres.

Mit diesen zwei Quellen sollte man einen repräsentativen Durchschnitt aller Genres bekommen. Zuerst muss man allerdings an die Daten herankommen. Bei *The Echonest* ist dies, nachdem man sich einen *API-Key*<sup>1</sup> registriert hat, relativ einfach:

```
http://developer.echonest.com/api/v4/artist/list_genres?api_key=ZSIUEIVVZGJVJWWIS
```

Die Liste enthält zum Zeitpunkt des Schreibens, 898 konkrete Genres und wird kontinuierlich vom Betreiber erweitert.

Die Suche bei Wikipedia gestaltet sich etwas schwieriger. Tatsächlich wurde diese Quelle erst nachträglich, nach einer Analyse des Quelltextes von *beets* (ein Musikmetadatenmanager, siehe [Link-13]) eingebaut. *Beets* hat ebenfalls das Problem, das Genre zu normalisieren. Daher muss dort ein entsprechender Mechanismus eingebaut sein. Dieser beruht, ähnlich wie hier, ebenfalls auf einem Baum<sup>2</sup>. Um diese Quelle in *libmunin* zu nutzen, wurde lediglich der relevante Code von *beets* (MIT-Lizenz) nach *Python3*<sup>3</sup> portiert. Von der englischen Wikipedia werden folgende Seiten *gescraped*, also der HTML-Seiteninhalt wird geparsst und die darin befindlichen Genres in eine Datei geschrieben:

- *List of popular music genres*: [Link-14]
- *List of styles of music: A–F, G–M, N–R, S–Z*: [Link-15]

<sup>1</sup> Ein *API-Key* ist zum nutzerabhängigen Zugriff auf den Webdienst nötig. Der in der URL gezeigte *API Key* ist auf *libmunin* registriert. Er sollte nicht für andere Zwecke verwendet werden.

<sup>2</sup> Anmerkung: Die Idee entstand allerdings ohne Kenntnis von *beets*.

<sup>3</sup> Sollte *beets* je nach Python  $\geq 3,0$  portiert werden, so wird der Autor den *beets*-Autoren gern einen Patch zusenden.

Von Wikipedia kommen daher zusätzliche 1527 Einträge. Diese werden mit den Einträgen von *Echonest* verschmolzen. Nach einer Deduplizierung ist die finale Genreliste 1876 Einträge lang.

### 4.2.2 Überführung der Genreliste in einem Genrebaum

Nachdem eine Liste von Genres nun vorhanden ist, muss diese noch in einem Baum, wie in 4.1d gezeigt, überführt werden. Unter 4.1a wird eine Genreliste gezeigt, die im Folgenden als Beispieleingabe benutzt wird.

Der Baum sollte dabei folgende Kriterien erfüllen:

- Der Pfad von einem Blattknoten („*Swedish*“) zum Wurzelknoten („*Music*“) sollte dabei das ursprüngliche Genre, mit dem optionalen Suffix *Music* ergeben („*Swedish-Pop-Music*“).
- Jeder Knoten erhält einen Index, der für jede Tiefenstufe von null wieder anfängt. So hat der Knoten *music* immer den Index null, bei der nächsten Ebene wird der Index nach alphabetischer Sortierung vergeben. *Pop* bekommt die Null, *Reggae* die Eins, *Rock* die Zwei und so weiter.

Das Umwandeln selbst geschieht folgendermaßen:

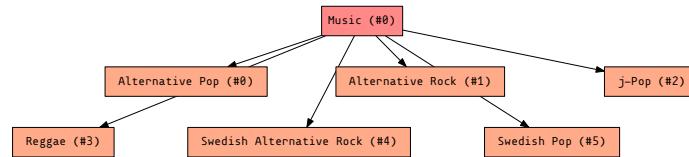
- Es wird manuell der Wurzelknoten *Music* angelegt.
- Alle Genres in der Genreliste werden diesem Wurzelknoten als Kind hinzugefügt. (siehe Abbildung 4.1b)

Nach dieser Vorarbeit wird rekursiv folgende Prozedur erledigt:

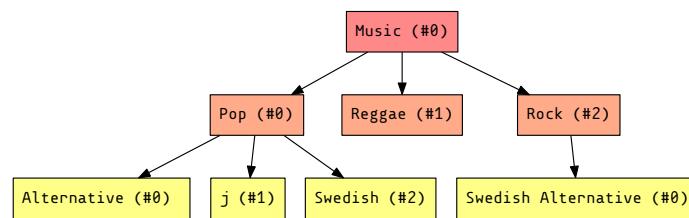
1. Gehe über alle Kinder des Wurzelknoten und breche dabei das *letzte Element* des *Genres* ab (*Western Country Rock* wird zu *Western Country* und *Rock*).
2. Der letzte Teil wird als Schlüssel in einer dem Knoten zugeordneten Hashtabelle gespeichert, mit dem Rest als dazugehörigen Wert. Aufgrund der Natur von Hashtabellen, entledigt sich dies eventueller Dupletten.
3. Die Liste der Kinder des Wurzelknotens, wird zu einer leeren Liste zurückgesetzt.
4. Die Schlüssel der Hashtabelle, werden als neue Kinder gesetzt. Die dazugehörigen Werte jeweils als deren Kinder. Dadurch vertieft sich der Baum.
5. Iteriere über die neuen Kinder, jedes Kind wird als neuer Wurzelknoten angenommen und es wird bei Schrit 1. weitergemacht. Der Rekursionsstopp ist erreicht, wenn keine Aufteilung des Genres in ein letztes Element und Rest mehr möglich ist.

|                          |                  |
|--------------------------|------------------|
| Reggae                   | Alternative Pop  |
| Swedish alternative Rock | Alternative Rock |
| j-Pop                    | Swedish Pop      |

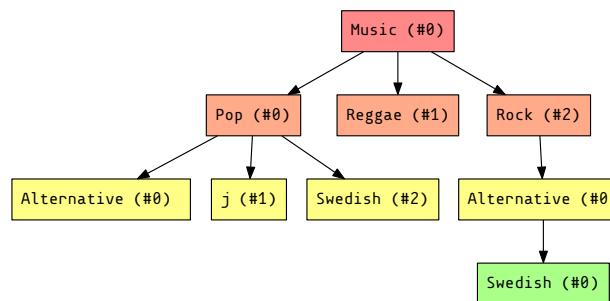
(a) Genreliste als Eingabe vor dem Prozessieren.



(b) Initialisierungsschritt: Vergabe von Indizes und Zuordnung zum Wurzelknoten.



(c) Der Genrebau nach der ersten Iteration, „Swedish Alternative“ wurde noch nicht aufgebrochen.



(d) Der nach zwei Iterationen fertige Genrebau.

**Abbildung 4.1.:** Der Baum wird aus der Eingabe unter 4.1a erzeugt indem erst alle Genres dem Wurzelknoten „Music“ unterstellt werden (4.1b). Danach wird der Baum rekursiv (hier in zwei Schritten, 4.1c und 4.1d) immer weiter vertieft.

In unserem Beispiel ist der Baum bereits nach zwei Iterationen fertig (siehe Abbildung 4.1d). In Abbildung 4.1c ist der Baum nach der ersten Iteration zu sehen. Bei der momentanen Datenquelle entstehen einige kleine Fehler im Baum. Daher werden nach dem manuellen Aufbau, noch einige halbautomatische Aufräumarbeiten erledigt.

1. Die fehlenden „*Musik*“-Genres „*Vocal*“ und „*Speech*“ werden manuell eingefügt.
2. Bei dem momentanen Vorgehen landen unter Umständen weitere „*Music*“-Knoten auf der ersten Ebene. Diese werden entfernt.
3. Alle Genres, die auf „*core*“ enden, werden aufgebrochen und dem Knoten „*core*“ auf erster Ebene hinzugefügt. Damit werden meist ähnliche Genres wie „*Metalcore*“ und „*Grindcore*“ zusammengefasst.

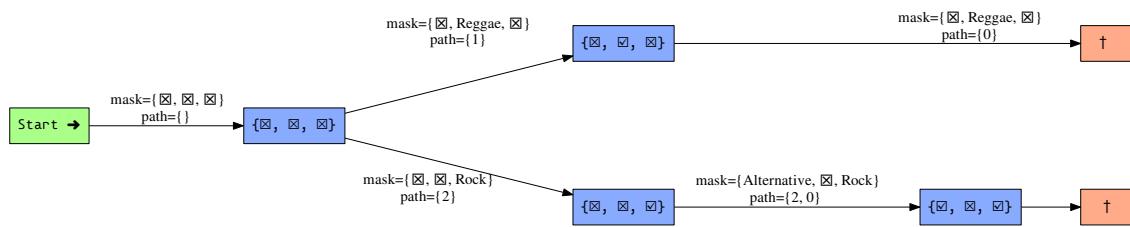
Der resultierende Baum ist im Anhang *Bilder des Genregraphen* in verschiedenen Detailstufen visualisiert. Er besitzt auf der ersten Ebene 1044 Untergenres. Die tiefste Verschachtelung erreicht das Genre „*New Wave of new Wave*“ mit einer Tiefe von fünf.

### 4.2.3 Zuordnung von Genres

Die Normalisierung des Genres ist nun mit dem aufgebauten Baum effizient möglich. Zuerst muss das Eingabegenre in Untergenres aufgeteilt werden, denn oft sind mehrere Genres in einem einzelnen String zusammengefasst, die durch bestimmte Zeichen getrennt sind. Als Beispiel: „*Rock, Reggae / Alternative Rock, Ska, Punk*“ Jedes dieser Untergenres wird dann mittels eines regulären Ausdrucks in einzelne Wörter aufgeteilt. Die Wörter werden noch in die kleingeschriebene Form gebracht: `{ {rock} , {reggae} , {alternative, rock} , {ska} , {punk} }`

Die einzelnen Wortlisten können in *Pfade* umgewandelt werden. Dazu werden zuerst folgende Variablen initialisiert:

| <i>Variable</i>          | <i>Beschreibung</i>   |
|--------------------------|---|
| <code>words</code>       | Eine Liste von Wörtern die im Genre vorkommen.<br>Beispiel: <code>{alternative, rock}</code>  |
| <code>root</code>        | Der momentane Wurzelknoten. Anfangs initialisiert auf „ <i>Music</i> “.   |
| <code>paths</code>       | Eine leere Liste mit Pfaden. Dient als Speicher für Resultate.  |
| <code>mask</code>        | Eine Liste mit Wahrheitswerten. Genauso lang wie <code>words</code><br>Die Wahrheitswerte werden mit <code>False</code> initialisiert.<br>Die Liste wird genutzt, um gefundene Wörter an dem entsprechenden Index „ <i>abzuhaken</i> “. |
| <code>path_result</code> | Eine Liste, die an die nächste Rekursionsstufe weitergegeben wird.<br>Sie speichert die Indizes des momentan aufgebauten Pfades.<br>Anfangs initialisiert auf ein leere Liste.  |



**Abbildung 4.2.:** Beispiel-Ablauf des Zuordnungs-Algorithmus an der Eingabe „Alternative Rock / Reggae“. In den Knoten ist die jeweils die momentane Maske eingetragen, an den Kanten jeweils die aktuelle mask und der bisher gebildete Pfad.

Nach diesen Vorbereitungen wird eine rekursive Backtracking-Suche gestartet:

1. Finde alle Kinder von `root`, deren Untergenres in `words` vorkommen. Wenn das entsprechende Untergenre noch nicht in `mask` abgehakt wurde, wird es in einer temporären Liste vermerkt.
2. Ist diese temporäre Liste dann leer und die `path_result`-Liste nicht leer, so wird die `path_result`-Liste zur `paths`-Liste hinzugefügt. Trifft dieser Fall ein, so ist in diesem Zweig der Rekursionsstopp erreicht.
3. Es wird über jedes Kindelement in der temporären Liste iteriert. Bei jeder Iteration wird folgendes durchgeführt:
  - a) Eine Kopie der `path_result`-Liste wird erstellt, bei der der Index des aktuellen Kindelements am Ende hinzugefügt wird.
  - b) Eine Kopie der `mask`-Liste wird erstellt, in der das vom Kind repräsentierte Wort „*abgehakt*“ wird (der entsprechende Index wird auf `True` gesetzt).
  - c) Das Kind wird als neuer Wurzelknoten angenommen und es wird wie bei Schritt 1) weitergemacht.
4. Nachdem alle Zweige der Rekursion beim Rekursionsstopp angekommen sind, stehen alle validen Pfade als Tupel von Indizes in `paths`.

In Abbildung 4.2 wird ein Beispiel dieses Verfahrens mit dem Genre „Alternative Rock / Reggae“ gegeben. Die passenden Pfade sind in diesem Fall also `Reggae` (`{0}`) und `Alternative Rock` (`{2, 0}`). Es ist zu bemerken, dass `Rock` (`{2}`) allein zwar ebenfalls ein valider Pfad ist, aber als eine Untermenge von `Alternative Rock` (`{2, 0}`) nicht in der Ergebnismenge enthalten ist.

#### 4.2.4 Vergleichen der unterschiedlichen Genrepfad-Mengen

Um zwei einzelne Pfade miteinander zu vergleichen, wird folgendermaßen vorgegangen:

- Zähle die Anzahl an Punkten, in denen sich der Pfad überdeckt. Beispiel: Für die Pfade  $\{2, 1, 0\}$  und  $\{2, 1, 2, 0\}$  wäre dies 2.
- Teile die Anzahl der Überdeckungen durch die Länge des längeren der beiden Pfade.
- Die daraus gewonnene Ähnlichkeit wird von 1,0 abgezogen um die Distanz zu erhalten.

In *libmunin* sind zwei Distanzfunktionen enthalten, welche diese Methode nutzt, um zwei Mengen mit Genrepfaden zu vergleichen.

**GenreTree:** Vergleicht jeden Genrepfad der Mengen  $A$  und  $B$ , mittels oben genannter Methode miteinander. Die minimale Distanz wird zurückgegeben. Als Optimierung wird frühzeitig abgebrochen, wenn eine Distanz von 0,0 erreicht wird.

Diese Distanzfunktion eignet sich für kurze Genre-Beschreibungen, wie sie in vielen Musiksammlungen vorkommen. Oft ist ein Lied als *Rock* oder *Metal* eingetragen, ohne Unterscheidung von Untergenres. Deshalb geht diese Distanzfunktion davon aus, wenige Übereinstimmungen zu finden — sollten welche vorkommen, so werden diese gut bewertet.

Setzt man voraus, dass  $d$  die unter 4.2.4 erwähnte Distanzfunktion ist, so berechnet sich die finale Distanz durch:

$$D_{min}(A, B) = \min\{d(a, b) : a, b \in A \times B, a \neq b\}$$

**GenreTreeAvg:** Seien  $A$  und  $B$  zwei Mengen mit Genrepfaden.  $A$  ist dabei die größere Menge und  $B$  die kleinere, falls die Mengen eine unterschiedliche Mächtigkeit besitzen. Dann gilt hier:

$$D_{avg}(A, B) = \frac{1}{|A|} \times \sum_{a \in A} \min\{d(a, b) : b \in B, a \neq b\}$$

Diese Distanzfunktion eignet sich für „reichhaltig“ gefüllte Genrebeschreibungen, bei denen auch ein oder mehrere Untergenres vorhanden sind. Ein Beispiel dafür wäre: „Country Rock/Folk/Rockabilly“. Die Distanzfunktion geht also davon aus, zumindest teilweise Überdeckungen in den Daten vorzufinden. Je nach Daten, die es zu verarbeiten gilt, kann der Nutzer der Bibliothek eine passende Distanzfunktion auswählen.

#### 4.2.5 Probleme

Insgesamt funktioniert dieser Ansatz gut. Die meisten Genres werden zufriedenstellend in Pfade normalisiert, die performant verglichen werden können.

Folgendes Problem wird allerdings noch nicht zufriedenstellend gelöst: Es wird davon ausgegangen, dass Genres die ähnlich sind auch ähnlich heißen. Eine Annahme, die zwar oft, aber nicht immer wahr ist. So sind die Genres *Alternative Rock* und *Grunge* sehr ähnlich — der obige Ansatz würde hier allerdings eine Distanz von 1 liefern. Auch Genres wie „*Rock'n'Roll*“ würde ähnlich schlechte Resultate liefern, da sie kaum sinnvoll aufgebrochen werden können.

Eine mögliche Lösung, wäre eine Liste von „*synonymen*“ Genres, die Querverbindungen im Baum erlauben würden. Allerdings wäre eine solche Liste von Synonymen schwer automatisch zu erstellen.

## 4.3 Schlüsselwortextraktion

Eine Idee bei *libmunin*, ist es auch die Liedtexte eines Liedes einzubeziehen, um Lieder mit ähnlicher *Thematik* näher beieinander im Graphen zu gruppieren. Sollten zwei Lieder nicht dieselben Themen behandeln, so soll sich zumindest die gleiche Sprache sich positiv auf die Distanz auswirken.

Um die Themen effizient zu vergleichen, extrahiert *libmunin* aus den Liedtexten die wichtigsten *Schlüsselwörter* mittels des *KeywordProviders*. Diese Phrasen sollen den eigentlichen Inhalt möglichst gut approximieren, ohne dabei schwer vergleichbar zu sein.

*Anmerkung:* Im Folgenden ist von *Schlüsselwörtern* die Rede. Ein einzelnes *Schlüsselwort*, wie „*dunkle Schwingen*“, kann aber aus mehreren Wörtern bestehen.

### 4.3.1 Der RAKE-Algorithmus

Zur Extraktion von Schlüsselwörtern aus Texten gibt es eine Vielzahl von Algorithmen [5]. Der verwendete Algorithmus zur Schlüsselwortextraktion ist bei *libmunin* der relativ einfach zu implementierende RAKE-Algorithmus (vorgestellt in [6]). Zwar könnte man mit anderen Algorithmen bessere Ergebnisse erreichen, diese sind aber schwerer zu implementieren (was die Anpassbarkeit verschlechtert) und sind in den meisten Fällen von sprachabhängigen Corpora (Wortdatenbanken) abhängig.

*Beschreibung des RAKE-Algorithmus:*

1. Aufteilung des Eingabetextes in Sätze, anhand von Interpunktions- und Zeilenumbrüchen.
2. Extraktion der *Phrasen* aus den Sätzen. Eine *Phrase* ist hier definiert als eine Sequenz von Nicht-stoppwörtern. Um Stoppwörter zu erkennen, muss eine von der Sprache abhängige Stopwortliste geladen werden. Zu diesem Zweck hat *libmunin* 17 Stopwortlisten in verschiedenen Sprachen eingebaut. Die Sprache selbst wird durch das Python-Modul `guess-language-spirit` [Link-16] anhand verschiedener Sprachcharakteristiken automatisch erraten. Zudem werden

lange Wörter mittels PyEnchant [Link-17] in einem Wörterbuch nachgeschlagen, um die Sprache herauszufinden, sofern die Enchant-Bibliothek samt Wörterbuch für die entsprechende Sprache [Link-18] installiert ist.

3. Berechnung eines *Scores* für jedes Wort in einer Phrase aus dem *Degree* und der *Frequenz* eines Wortes ( $P$  ist dabei die Menge aller Phrasen,  $|p|$  ist die Anzahl von Wörtern in einem Phrase):

$$\text{degree}(\text{word}) = \sum_{p \in P} \begin{cases} |p|, & \text{falls } \text{word} \in p \\ 0, & \text{sonst} \end{cases}$$

$$\text{freq}(\text{word}) = \sum_{p \in P} \begin{cases} 1, & \text{falls } \text{word} \in p \\ 0, & \text{sonst} \end{cases}$$

$$\text{score}(\text{word}) = \frac{\text{degree}(\text{word})}{\text{freq}(\text{word})}$$

4. Für jede Phrase wird nun ein *Score* berechnet. Dieser ist definiert als die Summe aller Wörter-Scores innerhalb einer Phrase. Die derart bewerteten Phrasen werden, absteigend sortiert, als Schlüsselwörter ausgegeben. Schlüsselwörter mit einem *Score* kleiner 2,0 werden ausgesiebt.

Es wurden zudem einige Änderungen, zum in [6] vorgestellten Algorithmus, vorgenommen, um diesen besser auf kleine Dokumente wie Liedtexte abzustimmen:

- Im Original werden Sätze nicht anhand von Zeilenumbrüchen aufgebrochen. Die meisten Liedtexte bestehen aber aus einzelnen Versen, die nicht durch Punkte getrennt sind, sondern durch eine neue Zeile abgegrenzt werden.
- Um die Ergebnisse leichter vergleichen zu können, werden die einzelnen Wörter nach dem Extrahieren auf ihren Wortstamm reduziert. Dabei wird der sprachsensitive *Snowball-Stemmer* [7] verwendet.
- Da sich viele Ausdrücke in einem Liedtext wiederholen, kamen während der Entwicklung viele Schlüsselwörter in verschiedenen Variationen mehrmals vor. Oft waren diese dann eine Untergruppe eines anderen Schlüsselwortes (Beispiel: *Yellow* und *Submarine* sind ein Teil von *Yellow Submarine*). Daher werden in einem nachgelagerten Schritt diese redundanten Phrasen entfernt.

#### Vergleich der einzelnen Schlüsselwortmengen:

Die einzelnen Mengen von Schlüsselwörtern werden unter der Prämisse verglichen, dass exakte Übereinstimmungen, durch den riesigen Wortschatz, selten sind.

- Zu einem Drittel geht der Vergleich der Sprache in die Distanz ein. Ist die Sprache gleich, so wird hier eine Teildistanz von 0 angenommen, andernfalls ist die Gesamtdistanz 1, da dann auch ein Vergleich der einzelnen Schlüsselwörter nicht mehr sinnvoll ist.
- Die restlichen zwei Drittel errechnen sich aus der Übereinstimmung der Schlüsselwörter. Für zwei Schlüsselwörter (eine Menge von Wörtern)  $A$  und  $B$  errechnet sich die Distanz folgendermaßen:

$$d_{kwd}(A, B) = 1 - \frac{|A \cap B|}{\max \{|A|, |B|\}}$$

Alle Schlüsselwörter werden damit untereinander verglichen. Die minimale dabei gefundene Distanz ist die finale Gesamtdistanz.

### 4.3.2 Ergebnisse

| Score | Schlüsselwörter ( <i>Wandern</i> ) | Score  | Schlüsselwörter ( <i>Yellow Submarine</i> ) |
|-------|------------------------------------|--------|---|
| 9,333 | <i>gerne stille stehn</i>          | 22,558 | <i>yellow submarin</i>                      |
| 5,778 | <i>wandern</i>                     | 20,835 | <i>full speed ahead mr</i>                  |
| 5,442 | <i>müllers lust</i>                | 8,343  | <i>live beneath</i>                         |
| 5,247 | <i>müde drehn</i>                  | 5,247  | <i>band begin</i>                           |
| 5,204 | <i>niemals fiel</i>                | 3,297  | <i>sea</i>                                  |
| 5,204 | <i>herr meister</i>                | 3,227  | <i>green</i>                                |
| 5,204 | <i>frau meisterin</i>              | 2,797  | <i>captain</i>                              |
| 5,074 | <i>muntern reihn</i>               | 2,551  | <i>sail</i>                                 |
| 5,031 | <i>schlechter müller</i>           | 2,551  | <i>blue</i>                                 |
| 5,031 | <i>wanderschaft bedacht</i>        | 2,551  | <i>cabl</i>                                 |
| 3,430 | <i>wasser</i>                      | 2,551  | <i>life</i>                                 |
| 3,430 | <i>steine</i>                      | 2,516  | <i>sky</i>                                  |
| 2,016 | <i>tanzen</i>                      | 2,516  | <i>aye</i>                                  |
| 2,016 | <i>frieden</i>                     | 2,016  | <i>friend</i>                               |
| 2,016 | <i>gelernt</i>                     | 2,016  | <i>aboard</i>                               |
| 2,016 | <i>schwer</i>                      | 2,016  | <i>boatswain</i>                            |

**Abbildung 4.3.:** Extrahierte Schlüsselwörter aus dem Volkslied „Das Wandern ist des Müllers Lust“ (links) und dem Beatles-Song „Yellow Submarine“ (rechts). Für jedes Schlüsselwort wird der Score angezeigt. Dieser hat keine Begrenzung nach oben. Rechts wurden die Schlüsselwörter zusätzlich auf den Wortstamm gebracht.

In Abbildung 4.3 sind die extrahierten Schlüsselwörter aus zwei Liedern aufgelistet.

Zur Referenz ist unter Abbildung 4.4 der Liedtextes des Volkliedes „Das Wandern ist des Müllers Lust“ abgedruckt. Der Text von „Yellow Submarine“ wird aus lizenzirechtlichen Gründen hier nicht abgedruckt.

|   |  |
|---|--|
| <p>Das Wandern ist des Müllers Lust,<br/>Das Wandern!<br/>Das muß ein schlechter Müller sein,<br/>Dem niemals fiel das Wandern ein,<br/>Das Wandern.</p> <p>Vom Wasser haben wir's gelernt,<br/>Vom Wasser!<br/>Das hat nicht Rast bei Tag und Nacht,<br/>Ist stets auf Wanderschaft bedacht,<br/>Das Wasser.</p> <p>Die sich mein Tag nicht müde drehn,<br/>Die Räder.<br/><i>(oben rechts weiter)</i></p> | <p>Das sehn wir auch den Rädern ab,<br/>Den Rädern!</p> <p>Die gar nicht gerne stille stehn,<br/>Die Steine selbst, so schwer sie sind,<br/>Die Steine!</p> <p>Sie tanzen mit den muntern Reihn<br/>Und wollen gar noch schneller sein,<br/>Die Steine.</p> <p>O Wandern, Wandern, meine Lust,<br/>O Wandern!</p> <p>Herr Meister und Frau Meisterin,<br/>Laßt mich in Frieden weiter ziehn<br/>Und wandern.</p> |
|---|--|

Abbildung 4.4.: Liedtext des Volksliedes „Das Wandern ist des Müllers Lust“.

Wie man in Abbildung 4.3 sieht, werden längere Phrasen automatisch besser bewertet — deren *Score* berechnet sich aus der Summe ihrer Wörter. Auch sieht man, dass viele unwichtige Wörter wie *aboard* trotz Stopwortlisten noch in das Ergebnis aufgenommen werden.

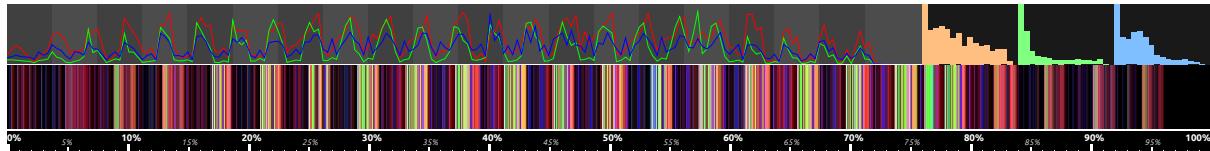
### 4.3.3 Probleme

Teilweise liefert diese Provider-Distanzfunktions-Kombination bereits interessante Ergebnisse. So werden die beiden staatskritischen deutschen Texte „Hey Staat“ von *Hans Söllner* und „Lieber Staat“ von *Farin Urlaub* mit einer relativ niedrigen Distanz von gerundet 0,4 bewertet.

Doch nicht bei allen Texten funktioniert die Extraktion so gut. Nimmt man den Ausdruck „*God save the Queen!*“, so wird *RAKE* diesen nicht als gesamtes Schlüsselwort erkennen. Stattdessen werden zwei einzelne Phrasen generiert: „*God save*“ und „*Queen*“, da „*the*“ ein englisches Stopwort ist.

Andererseits entstehen auch oft Schlüsselwörter, die entweder unwichtig („*mal echt*“), sinnentfremdet („*gerne still stehen*“ obwohl im Text oben „*nicht*“ davor steht) oder stark kontextspezifisch („*schlechter Müller*“) sind. Da ein Computer den Text nicht verstehen kann, lässt sich das kaum vermeiden.

Auch gemischtsprachige Liedtexte lassen sich nur schwer untersuchen, da immer nur eine Stopwortliste geladen werden kann. Für Liedtexte mit starkem Dialekt (wie von *Hans Söllner*) greift auch die normale hochdeutsche Stopwortliste nicht.



**Abbildung 4.5.:** Beispiel-Moodbar von „Avril Lavigne – Knockin’ on Heaven’s Door“. Ein Lied bei dem hauptsächlich eine Akustikgitarre (rot) und Gesang (grünlich) im Vordergrund steht. Der Gesang setzt etwa bei 10% ein. Die Grafik wurde durch ein eigens zu diesem Zwecke geschriebenes Script gerendert. Deutlich sichtbar sind die einzelnen Pausen zwischen den Akkorden.

## 4.4 Moodbar-Analyse

Die ursprünglich als Navigationshilfe in Audioplayern gedachte Moodbar (siehe [8] für genauere Informationen) wird in *libmumin* neben der BPM-Bestimmung als einfache Form der Audioanalyse eingesetzt. Kurz zusammengefasst, wird dabei ein beliebiges Audiotrack zeitlich in 1000 Blöcke unterteilt. Für jeden dieser Blöcke wird ein Farbwert (als RGB-Tripel) bestimmt. Der Rotanteil bestimmt dabei den Anteil niedriger Frequenzen, der Grünanteil den Anteil mittlerer Frequenzen und der Blauanteil den Anteil von hohen Frequenzen. Die Farbe Türkis deutet daher auf hohe und mittlere Frequenzen in einem Block hin – E-Gitarren haben häufig diese Farbe in der Moodbar. Akustikgitarren erscheinen dafür meist in einem dunklem Rot (siehe Abbildung 4.5).

Die Namensgebung des Verfahrens ist ein wenig irreführend. Man kann hier keineswegs die subjektive Stimmung in einem Lied herauslesen. Lediglich die Bestimmung einzelner Instrumente ist als Annäherung möglich. Nach Meinung des Autors sollte man das Verfahren daher eher „*frequencebar*“ oder Ähnliches nennen. Um aber auf die Einführung eines neuen Begriffes zu verzichten, wird die Namensgebung des Originalautors verwendet.

### 4.4.1 Vergleich von Moodbars

Das Vergleichen verschiedener Moodbars gestaltet sich aufgrund der hohen Länge der einzelnen RGB-Vektoren als schwierig. In einem vorgelagerten Analyseschritt wird daher versucht, die markanten Merkmale der einzelnen Vektoren zu extrahieren. Dieser Analyseschritt wird dabei durch den MoodbarProvider getätigter.

Vor der eigentlichen Verarbeitung wird jeder Farbkanal in einzelne Blöcke aufgeteilt (*Diskretisierung*), von der jeweils das arithmetische Mittel gebildet wird. So wird der ursprüngliche 1000 Werte lange Vektor in (momentan) 20 einzelne, handlichere Werte aufgeteilt. Bei einer durchschnittlichen Liedlänge von vier Minuten entspricht das immerhin zwölf Sekunden pro Block, was für gewöhnliche Lieder ausreichend sein sollte. Nach einigen subjektiven Tests haben sich folgende Merkmale als vergleichbar erwiesen:

- **Differenzsumme:** Für jeden Farbkanal wird die Summe der Differenzen zu den jeweiligen vorherigen Blockwert gebildet (C ist der jeweilige Farbkanal):

$$\sum_{i=1}^{|C|} |C_i - C_{i-1}|$$

Dieser Wert soll die grobe „*Sprunghäufigkeit*“ des Liedes beschreiben. Ändern sich die Werte für diesen Farbkanal kaum, so ist der Wert niedrig. Liegen hohe Änderungen zwischen jedem Block vor, so steigt dieser Wert bis zu seinem maximalen Wert von  $(20 - 1) \times 255 = 4845$ .

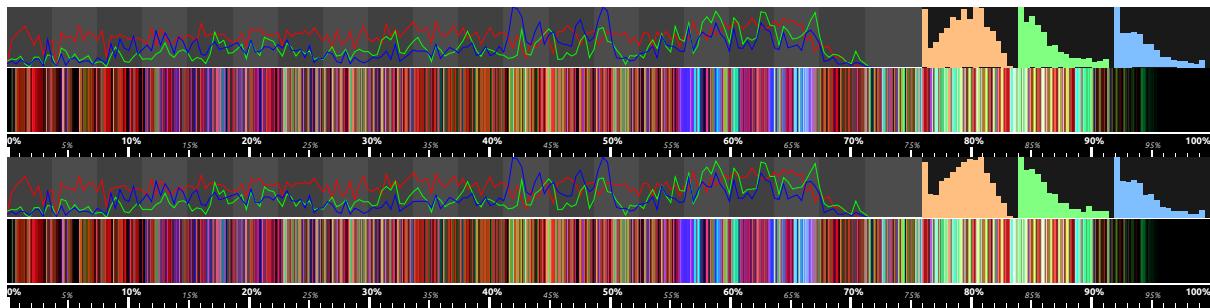
- **Histogramm:** Für jeden Farbkanal wird eine Häufigkeitsverteilung, also ein Histogramm, abgespeichert. Jeder einzelne Farbwert wird dabei auf einen von fünf möglichen Bereichen, die jeweils 51 Werte umfassen, aufgeteilt. So wird für jeden Farbkanal eine einfach zu vergleichende Verteilung der Frequenzen abgespeichert.
- **Dominante Farben:** Wie bereits erwähnt, ist es manchmal möglich, bestimmte Instrumente visuell, anhand deren charakteristischen Farbe, in der Moodbar zu erkennen. Das kann man sich beim Vergleichen zu Nutze machen, denn ähnliche Instrumente (ergo bestimmte, charakteristische Farben) deuten auf ähnliche Musikstile hin. Der MoodbarProvider teilt daher jeden Farbkanal in 15er-Schritten in einzelne Bereiche auf. Jede Farbkombination wird dann einem dieser Bereiche zugeordnet. Die 15 am häufigsten zusammen vorkommenden Tripel werden abgespeichert.
- **Schwarzanteil:** Gesondert werden sehr dunklen Farben behandelt. Haben alle Farbkanäle eines RGB-Tripels, einen Wert kleiner 30, so wird die Farbe nicht gezählt, sondern auf einen Schwarzanteil-Zähler aufaddiert. Geteilt durch 1000, ergibt sich daraus der Anteil des Liedes, der ganz oder beinahe still ist.
- **Durchschnittliches Minimum/Maximum:** Von jedem Block wird das Minimum/Maximum der drei Farbkanäle bestimmt. Die Summe über jeden so bestimmten Wert, geteilt durch die Anzahl der Blöcke, ergibt das durchschnittliche Minimum/Maximum. Für jeden Farbkanal ergibt sich so ein Wert, der zwischen 0 und 255 liegt. Dieser sagt aus, in welchem Bereich sich die „*Frequenzen*“ im jeweiligen Farbkanal bewegen.

In Tabelle 4.6 wird eine Auflistung der einzelnen Werte gegeben, die der Moodbar-Provider generiert. Daneben werden auch die entsprechenden Gewichtungen und Distanzfunktionen gegeben, mit dem die Moodbar-Distanzfunktion die einzelnen Werte verrechnet. Die entstehende, gewichtete Distanz, wird mittels der in Abbildung gezeigten Funktion noch skaliert, um hohe Werte anzuheben und niedrige weiter abzusenken.

Am subjektiv vergleichbarsten, erwiesen sich die dominanten Farben in einem Lied. Die zwischenzeitlich aufgekommene Idee, bestimmte markante Farbwertbereiche bestimmten Instrumenten automatisch zuzuordnen, erwies sich, mangels exakter Zuordnungstabellen, als unpraktikabel und unge nau.

| Name                               | Gewichtung   | ungewichtete Distanzfunktion $d(a, b)$                        |
|------------------------------------|--------------|---|
| Differenzsumme                     | 13,5%        | $1 - \sqrt{\frac{ a-b }{50}}$                                 |
| Histogramm                         | 13,5%        | $1 - \frac{\sum_{x \in \vec{a} - \vec{b}}  x }{5 \times 255}$ |
| Dominante Farben                   | 63,0%        | $\frac{ a \cap b }{\max\{ a ,  b \}}$                         |
| Schwarzanteil                      | 5,0%         | $1 - \sqrt{\frac{ a-b }{50}}$                                 |
| Durchschnittliches Minimum/Maximum | 5,0%         | $1 - \sqrt{\frac{ a-b }{255}}$                                |
|                                    | $\sum 100\%$ |   |

**Abbildung 4.6.:** Auflistung der einzelnen Werte, die der Moodbar-Provider ausliest und deren dazugehörige Distanzfunktion, sowie deren Gewichtung in der Gesamtdistanz. „a“ und „b“ sind Skalare, mit Ausnahme der Histogramm-Eingabewerte und der dominanten Farben. Dort sind „a“ und „b“ die einzelnen Farbkanäle als Vektor, bzw. eine Menge von Farben. Zur Bildung der Gesamtdistanz werden die einzelnen Werte über einen gewichteten Mittelwert verschmolzen. Die Werte im Nenner der meisten Formeln geben den maximalen Wert an, der für dieses Attribut erwartet wird.



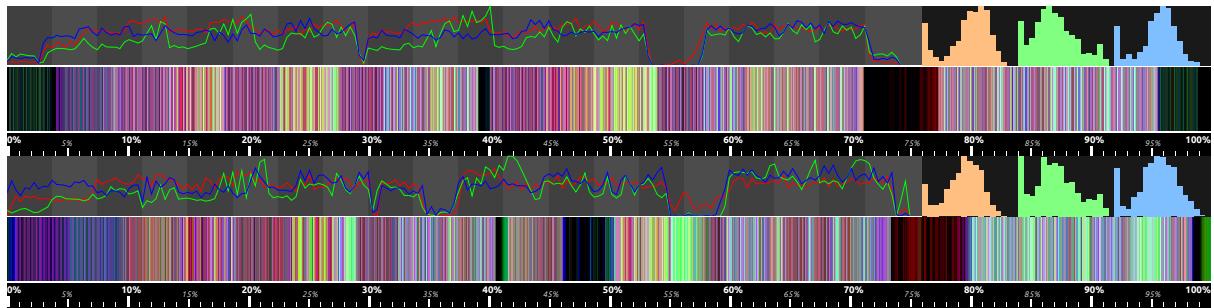
**Abbildung 4.7.:** Dieselbe Moodbar bei unterschiedlichen Encoding der Audiodaten. Oben das Beatles-Lied „Yellow Submarine“ als FLAC enkodiert, darunter dasselbe Lied mit stark komprimierter MP3-Enkodierung. Die von libmunin berechnete Moodbar-Distanz ist hier etwa 0,01.

#### 4.4.2 Probleme

Das Hauptproblem ist, dass das Verfahren ursprünglich nicht zum *Vergleichen* von Audiodaten auslegt war und vom Autor lediglich dafür „missbraucht“ wurde. Ursprünglich war das Verfahren dazu gedacht, um mittels der Farben eine Navigationshilfe für den Hörer des Liedes zu geben. So konnte dieser stille Bereiche schnell erkennen und zu bestimmten Stellen im Lied springen.

Wichtige Informationen, wie die eigentliche Stimmung in dem Lied (von *dunkel* bis *positiv*) bis hin zum Rhythmus des Liedes, lassen sich aber nicht davon ablesen. Lediglich die durchschnittliche Geschwindigkeit wird vom BPMProvider erfasst. Dieser muss aber die ganze Datei noch einmal zusätzlich dekodieren. Daher ist der MoodbarProvider momentan eher als *Notbehelf* zu sehen.

Zudem ist die Geschwindigkeit der Audioanalyse eher dürftig. Geht das Analysieren des RGB-Vektors an sich vergleichsweise schnell, so ist die Generierung desselben zeitlich aufwendig. Bei MP3-



**Abbildung 4.8.:** Moodbar einer Live und einer Studioversion von „Rammstein – Tier“. Oben die Studioversion, unten die Liveversion. Hier ist die von libmunin errechnete Moodbar-Distanz immerhin bei 0,32.

enkodierten Dateien dauert dies auf dem Entwicklungsrechners des Autors, je nach Größe, bis zu vier Sekunden. Die Dauer variiert dabei je nach Format. FLAC-enkodierte Dateien brauchen oft lediglich die Hälfte dieser Zeit. In beiden Fällen ist die Anwendung, bei einer mehreren zehntausend Lieder umfassenden Sammlung, sehr aufwendig. Neben der Liedtextsuche, ist dies der größte Posten beim *Kaltstart*.

Vorteile sind hingegen:

- **Robustheit:** Wie man in 4.7 sieht, ist das Verfahren unempfindlich gegen verschiedene Enkodierungen. Selbst Live- und Studioversionen zeigen gut vergleichbare Resultate (siehe dazu auch Abbildung 4.8).
- **Geringer Speicherverbrauch:** Obwohl für die Implementierung die relativ speicherhungrige Sprache Python benutzt wurde, nutzt der MoodbarProvider lediglich etwa 540 Bytes pro Analysedatensatz. Da Python die Zahlen –10 bis 255 im Speicher hält und der MoodbarProvider nur Zahlen in diesem Bereich erzeugt, reichen hier 8 Byte für eine Referenz auf ein Integer-Objekt aus.

# 5 | Implizites Lernen vom Nutzer

## 5.1 Generierung von Regeln

IN vorangegangenen Kapiteln wurde schon oft davon gesprochen, dass *libmunin* den Nutzer *beobachtet*. Dies geschieht, indem der Anwendungsentwickler, die vom Nutzer gehörten Titel, an *libmunin* zurückmeldet.

### 5.1.1 Finden von wiederkehrenden Mustern

Um eine „*Warenkorbanalyse*“ durchzuführen, braucht man erstmal *Warenkörbe*. Diese entstehen, indem man die einzelnen Songs in der Historie zeitlich gruppier. Wächst eine Gruppe über eine Grenze (momentan 5), so wird eine neue Gruppe begonnen.

Diese einzelnen Gruppen von Songs fungieren dann als *Warenkörbe*. Aus diesen gilt es zuerst eine Menge an Songs (im Folgenden *Muster*<sup>1</sup> genannt) zu finden, die jeweils oft zusammen in den einzelnen Warenkörben vorkommen. Der naive Ansatz wäre, für jede Kombination der Eingabesongs, das Vorkommen derselben, im Warenkorb zu zählen. Wie man sich bereits denken kann, ist hierfür der algorithmische Aufwand enorm, denn bereits bei einer Menge von 1000 unterschiedlichen Songs in der Historie, müssten bereits  $2^n - 1$  Kombinationen gebildet werden. Denn ein Warenkorb kann man als einer Menge von sich nicht wiederholender Songs sehen, bei der auch die Reihenfolge keine Rolle spielt.

Für die Lösung dieses Problems, gibt es einige etablierte Algorithmen. Der bekannteste ist vermutlich der *Apriori-Algorithmus* (vergleiche [9], S. 248–253). Statt alle Kombinationen zu betrachten, werden erst alle „*Einer-Kombinationen*“ gebildet und die ausgefiltert, welche einen zu niedrigen *Support-Count* besitzen. Die Grenze legt man vorher fest. Der *Support-Count*  $support(A)$  ist die Anzahl der *Warenkörbe*, in denen die Menge von Songs  $A$  vorkommt, geteilt durch die absolute Anzahl der Warenkörbe. Danach werden mit den verbliebenen Zweier-Kombination gebildet, wieder gefiltert, dann die noch relevanten Dreier-Kombinationen und so weiter. Dadurch wird eine große Menge von Kombinationen vermieden.

---

<sup>1</sup> In englischer Lektüre werden die *Wiederkehrenden Muster* als *Frequent Itemsets* bezeichnet.

Seit einiger Zeit haben sich jedoch eine Gruppe effizienterer Algorithmen etabliert. Dazu gehören der FP-Growth (siehe [9] S. 257–259, 272), Eclat (siehe [10]), sowie der hier verwendete RELIM-Algorithmus, der in [11] vorgestellt wurde.

| <i>Kombination (1er)</i> | <i>Kombination (2er)</i> | <i>Kombination (3er)</i> |
|--------------------------|--------------------------|--------------------------|
| <i>A</i>                 | <i>A, B</i>              | <i>A, B, C</i>           |
| <i>B</i>                 | <i>B, C</i>              |                          |
| <i>C</i>                 | <i>C, A</i>              |                          |

**Abbildung 5.1.:** Die Muster, welche aus den drei Warenkörben  $\{\{A, B, C\}, \{B, B, C\}, \{C, C, B\}\}$  generiert worden sind, mit der jeweiligen Anzahl von Vorkommnissen in den Warenkörben.

In Tabelle 5.1 sieht man ein Beispiel aus drei Warenkörben, aus denen per Hand mit der naiven Herangehensweise alle möglichen Kombinationen samt deren Support-Count, aufgelistet worden sind.

### 5.1.2 Der RELIM-Algorithmus

Generell gilt FP-Growth als der neue Standard-Algorithmus, der laut mehrerer Quellen andere Algorithmen wie Eclat und RELIM (*RE-cursive ELIM-ination*) aussticht [12] [13]. In diesem Fall wird trotzdem auf RELIM zurückgegriffen, da dieser für die Zwecke des Autors ausreichend schnell ist und die Datenmenge nie mehr als wenige tausend Songs übersteigen wird. Zudem gibt es mit dem Python-Paket *pymining* (siehe [Link-19]) bereits eine freie, qualitativ hochwertige Implementierung, während es für FP-Growth nur qualitativ schlechte Implementierungen zu geben scheint, oder welche, die nur für Python-Versionen  $\leq 2,7$  funktionieren.

### 5.1.3 Ableitung von Regeln aus Mustern

Hat man eine Gruppe von häufig zusammen auftretenden Song-Kombinationen gefunden, so können daraus Assoziationsregeln abgeleitet werden. Eine Assoziationsregel verbindet zwei Mengen  $A$  und  $B$  von Songs mit einer gewissen Wahrscheinlichkeit miteinander. Sie besagen, dass wenn eine der beiden Mengen miteinander gehört wird, dann ist es wahrscheinlich, dass auch die andere Menge daraufhin angehört wird. Regeln werden aus dem Verhalten des Nutzers abgeleitet. Dazu wird jedes Lied, das der Nutzer anhört, in einer *Historie* zwischengespeichert. Um die generelle Anwendbarkeit der Regel zu beschreiben, wird für jede Regel ein *Rating* berechnet.

*Anmerkung:* Im allgemeinen Gebrauch sind Assoziationsregeln nur in eine Richtung definiert. In *libminin* sind die Regeln aus Gründen der Einfachheit allerdings bidirektional. So gilt nicht nur, dass man wahrscheinlich die Menge  $B$  hört, wenn man  $A$  gehört hat ( $A \rightarrow B$ ), sondern auch umgekehrt ( $A \leftrightarrow B$ ). Ein natürlichsprachliches Beispiel hierfür:  $\frac{2}{3}$  der Basketballspieler essen Cornflakes

( $Basketball \Rightarrow Cornflakes$ ). Diese Regel besagt, dass der größere Teil der Basketballspieler Cornflakes isst, aber nicht, dass die meisten Cornflakes-Esser Basketballspieler sind. Da bei *libmunin* auf beiden Seiten der Regel immer der gleiche Typ (ein oder mehrere Songs) steht und die Beziehung immer „*werden miteinander gehört*“ ist, wird hier vereinfachend eine bidirektionale Assoziation angenommen. Dies erlaubt ein Anwenden der Regeln in beide Richtungen.

Um nun aus einem Muster Regeln abzuleiten, teilt man es in alle möglichen verschiedenen, disjunkten Teilmengen auf — allerdings in maximal zwei Teilmengen. Diese beiden Teilmengen nimmt man als die beiden Mengen einer Assoziationsregel an und testet, mittels verschiedener Metriken, wie zutreffend diese ist.

| Assoziationsregel                | Support      | Imbalance Ratio | Kulczynski   | Lift         |
|----------------------------------|--------------|-----------------|--------------|--------------|
| $\{A\} \leftrightarrow \{B\}$    | $0, \bar{3}$ | $0, \bar{6}$    | $0, \bar{6}$ | 0            |
| $\{B\} \leftrightarrow \{C\}$    | 1,0          | 0               | 1            | 0            |
| $\{C\} \leftrightarrow \{A\}$    | $0, \bar{3}$ | $0, \bar{6}$    | $0, \bar{6}$ | 0            |
| $\{A\} \leftrightarrow \{B, C\}$ | $0, \bar{3}$ | $0, \bar{6}$    | $0, \bar{6}$ | 0            |
| $\{B\} \leftrightarrow \{A, C\}$ | $0, \bar{3}$ | 0               | $0, \bar{3}$ | 0            |
| $\{C\} \leftrightarrow \{A, B\}$ | $0, \bar{3}$ | $0, \bar{6}$    | $0, \bar{6}$ | $0, \bar{8}$ |

**Abbildung 5.2.:** Mögliche Regeln, die aus den drei Warenkörben erstellt werden können. Zusätzlich wird der dazugehörige Gesamt-Support-Count, sowie der Metriken Imbalance-Ratio, Kulczynski und Lift abgebildet.

Als Beispiel kann man wieder die Warenkörbe aus Tabelle 5.1 nehmen. Muster mit nur einem Song können nicht weiter aufgeteilt werden, daher müssen diese nicht weiter betrachtet werden. Die Zweier-Kombination sind leicht in zwei disjunkte Teilmengen aufteilbar. Für die Dreier-Kombinationen können mehrere mögliche Teilmengen erstellt werden. Die einzelnen möglichen Regeln werden in Tabelle 5.2 aufgelistet.

| Eigenschaft       | Basketball | Basketball | $\Sigma$ |
|-------------------|------------|------------|----------|
| Cornflakes        | 400        | 350        | 750      |
| $\neg$ Cornflakes | 200        | 50         | 250      |
| $\Sigma$          | 600        | 400        | 1000     |

**Abbildung 5.3.:** Vierfeldertafel mit erfundenen Beispieldaten. Es werden 1000 Studenten untersucht, bei denen die Eigenschaften „Spielt Basketball“ und „Isst Cornflakes“ festgestellt worden sind.

Nicht jede Regel ist automatisch eine gute Regel. Ein gängiges Lehrbeispiel wäre hier die Regel  $Basketball \Rightarrow Cornflakes$ , also eine Regel, die laut Tabelle 5.3 besagt, dass  $\frac{2}{3}$  aller *Basketballspieler* zum Frühstück *Cornflakes* essen. Der Anteil der Menschen die aber insgesamt Cornflakes essen liegt aber bei 75% — daher ist die Eigenschaft „*Basketballspieler*“ sogar im Vergleich, zum durchschnittlichen Anteil von Cornflake-Essern, ein Gegenindiz für diese Eigenschaft.

Um solche kontraproduktiven Assoziationsregeln zu vermeiden, werden für jede Regel zwei Metriken errechnet. Die von *libmunin* genutzten Metriken wurden dem Buch *Datamining Concepts and Techniques* ([9], S. 268–271) entnommen: Die *Kulczynski-Metrik* und der *Imbalance-Ratio*.

Die *Kulczynski-Metrik* drückt die Güte der Regel als eine reelle Zahl im Bereich [0, 1] aus, wobei 1 die beste Wertung ist. Grob ausgedrückt besagt die Metrik, wie zutreffend die Regel im Durchschnitt ist. A und B sind im Folgenden die beiden nicht-leeren Teilmengen der Regel:

$$Kulczynski(A, B) = \frac{1}{2} \times (P(A | B) + P(B | A))$$

Diese Metrik ist der Durchschnitt aus zwei Variationen einer anderen Metrik: Dem *confidence-Measure* (vergleiche [9], S. 254f.):

$$\text{confidence}(A \rightarrow B) = P(A | B) = \frac{P(A \cap B)}{P(B)} = \frac{\text{support}(A \cap B)}{\text{support}(B)}$$

Diese Metrik gibt an, zu welchem Prozentsatz die Regel zutrifft. Ist der Quotient 1, so trifft die Regel bei jedem bekannten Warenkorb zu. Der Zähler  $\text{support}(A \cap B)$  beschreibt hier, wie oft sowohl A und B gleichzeitig in einem Warenkorb vorkommen. Bereits allein ist diese Metrik ein gutes Indiz für die Korrektheit einer Regel, die Kulczynski-Metrik prüft zusätzlich beide Seiten der Regel. Um zu zeigen wie sich die Kulczynski-Metrik berechnen lässt, können wir die obige Definition umstellen:

$$Kulczynski(A, B) = \frac{1}{2} \times \left( \frac{\text{support}(A \cap B)}{\text{support}(B)} + \frac{\text{support}(A \cap B)}{\text{support}(A)} \right)$$

Diese Metrik allein reicht allerdings nicht für eine qualitative Einschätzung einer Regel. Zwar kann die Regel oft zutreffen, doch kann sie, wie im obigen Beispiel mit den *Cornflakes*, trotzdem kontraproduktiv sein. Daher wird mit dem *Imbalance Ratio* eine weitere Metrik eingeführt. Der *Imbalance Ratio* gibt im Bereich [0, 1] an, wie unterschiedlich beide Seiten der Regel sind. Treten die Muster unterschiedlich oft auf, so steigt diese Metrik. Hier ist der beste Wert die 0, der Schlechteste eine 1. Er ist gegeben durch:

$$\text{ImbalanceRatio}(A, B) = \frac{|\text{support}(A) - \text{support}(B)|}{\text{support}(A) + \text{support}(B) - \text{support}(A \cap B)}$$

Sollte die *Kulczynski-Metrik* kleiner als 0,6 sein oder der *Imbalance-Ratio* größer als 0,35, so wird die Regel fallen gelassen. Diese Grenzwerte wurden, mehr oder minder willkürlich, nach einigen Tests festgelegt. Sollte die Regel akzeptabel sein, dann werden beide Metriken in eine einzelne, leichter zu handhabende *Rating-Metrik* verschmolzen:

$$\text{Rating}(A, B) = (1 - \text{ImbalanceRatio}(A, B)) \times \text{Kulczynski}(A, B)$$

Dieses *Rating* wird genutzt, um die einzelnen Assoziationsregeln zu sortieren. Das finale Rating bewegt sich im Bereich [0, 1], wobei 1 das höchste vergebene Rating ist.

Nach einigen Tests erwiesen sich beide Metriken aber nicht als ausreichend um schwache Regeln zu filtern. Daher wurde noch zusätzlich die *Lift-Metrik* eingeführt (vergleiche [9], S.266). Diese ist definiert als:

$$Lift(A, B) = P(A | B) - (P(A) \times P(B)) = support(A \cap B) - (support(A) \times support(B))$$

Sie vergleicht das erwartete gemeinsame Auftreten der Mengen A und B mit dem tatsächlichen Auftreten in den Warenkörben. Ist der berechnete Wert  $< 0$ , so korreliert das Auftreten von B negativ mit A. In diesem Fall wird die Regel ignoriert. Werte größer oder gleich 0 bedeuten eine positive/neutrale Korrelation. Das Auftreten von B impliziert das wahrscheinliche Auftreten von A. Für die unter Tabelle 5.3 gezeigten Werte können nun die einzelnen Metriken angewandt werden:

$$Kulczynski(Basketball, Cornflakes) = \frac{1}{2} \times \left( \frac{400}{600} + \frac{400}{750} \right) = 0,6$$

Dieses Ergebnis würde zum Ausschluss der Regel führen, da  $0,6 < 0,6$  ist. Allerdings ist dies, für diese kontraproduktive Regel, ein knappes Ergebnis, da die Grenze von  $\overline{0,6}$  willkürlich gewählt wurde.

$$ImbalanceRatio(Basketball, Cornflakes) = \frac{|750 - 600|}{750 + 600 - 400} \approx 0,16$$

Beim *ImbalanceRatio* war 0 der beste anzunehmende Wert. Laut dem Ergebnis von 0,16 wäre diese Regel also sogar gut balanciert.

$$Lift(Basketball, Cornflakes) = \frac{400}{1000} - \left( \frac{750}{1000} \times \frac{600}{1000} \right) = -0,05$$

Der *Lift* führt mit einem Ergebnis  $< 0$  zu einer definitiven Filterung der Regel.

## 5.2 Anwendung von Regeln

Wie bereits unter Kapitel 3.2.2 erklärt, werden Assoziationsregel als Navigationshilfe beim Traversieren genutzt. Zu diesem Zwecke müssen die entstandenen Regeln irgendwo sortiert abgelegt werden. Diese Ablage ist der `RuleIndex`. Beim Einfügen wird jeweils überprüft, ob die Maximalanzahl an Regeln (momentan maximal 1000) übertroffen wird. Sollte dem so sein, wird die älteste (ergo, zu erst eingefügte) Regel gelöscht, um Platz zu machen. Der Anwendungsentwickler kann mittels der `lookup(song)`-Methode eine Liste von Regeln abfragen, die diesen Song in irgendeiner Weise betreffen. Um diese Operation zu beschleunigen, wird intern eine Hashtabelle gehalten, mit dem Song als Schlüssel und der entsprechende Regel-Liste als zugehöriger Wert. Bei jeder Operation auf dem `RuleIndex` wird dieser automatisch bereinigt. Dabei werden Regeln entfernt, die Songs erwähnen, welche nicht mehr in der Historie vertreten sind.

## 5.3 Lernerfolg

Noch sind keine Aussagen darüber möglich, wie gut die momentane Lernstrategie funktioniert. Einerseits ist es schwer festzustellen was „gut“ bedeutet, andererseits wurde eine *libmunin*-Session noch nie lange genug benutzt, um Aussagen über die Langzeitfunktionalität zu geben.

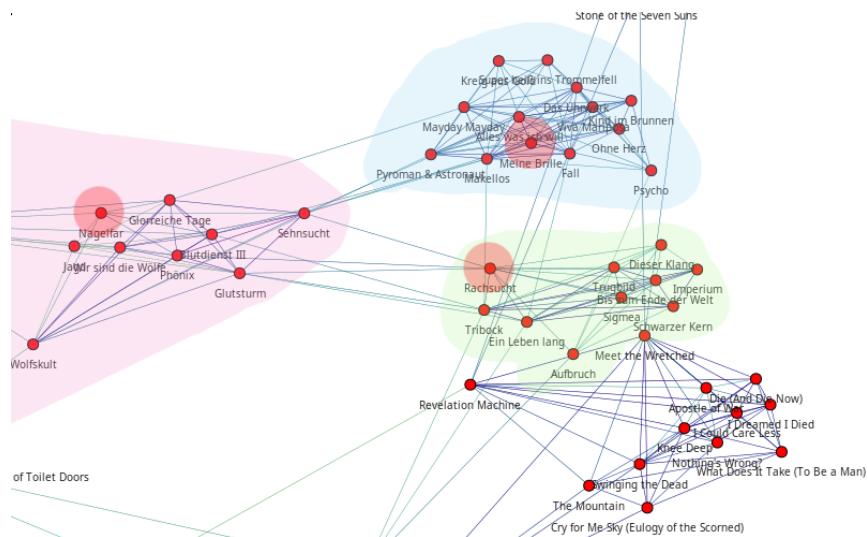
Daher ist die oben genannte Vorgehensweise als „Hypothese“ zu sehen, die sich erst noch in der Praxis bewähren muss. Änderungen sind wahrscheinlich. Zudem muss auch auf Seite der Implementierung noch ein Detail verbessert werden: Momentan wird nur die Historie aufgezeichnet, wenn die Demonanwendung läuft. Da die Anwendung lediglich eine Fernbedienung für den MPD ist, läuft diese nicht die ganze Zeit über. Abhilfe würde ein separater MPD-Client, der nur dafür dient, im Hintergrund die Historie-Daten mitzuloggen.

## 5.4 Explizites Lernen

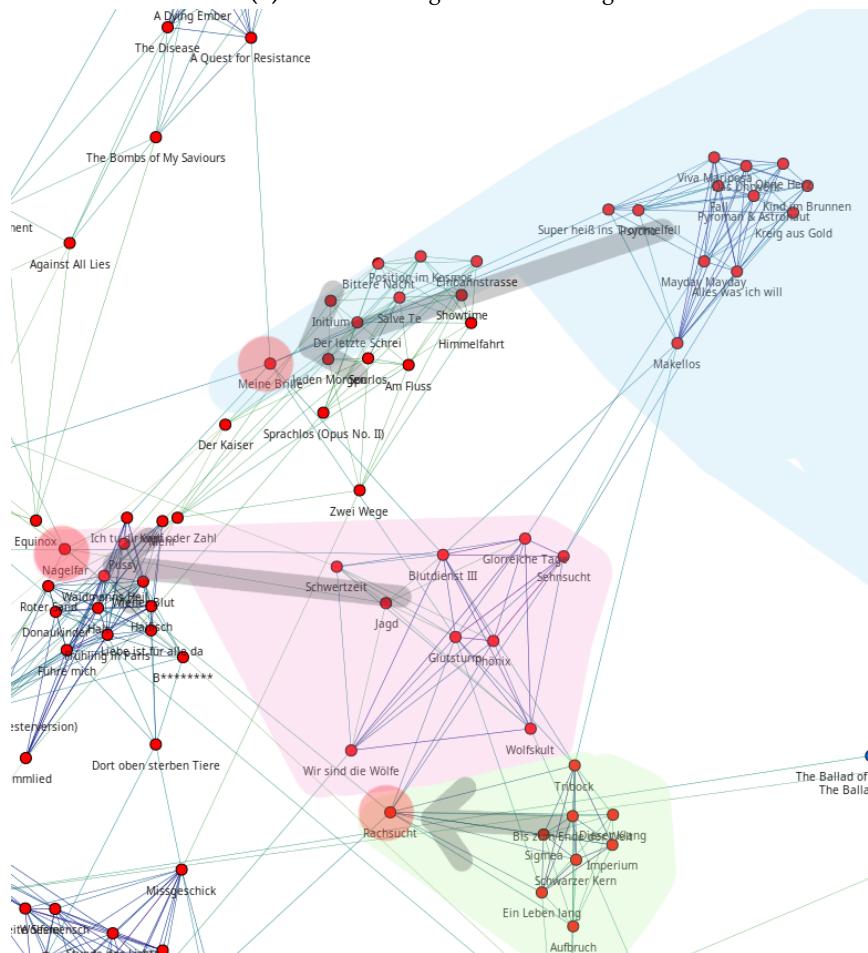
Bei einer `insert`-Operation lässt sich beobachten, dass die eingefügten Songs deutlich deutlich weitläufiger verbunden sind, als regulär per `add` hinzugefügte. Diese Eigenschaft macht sich die in der Projektarbeit ([1], S.37ff) gezeigte Demonanwendung zu Nutze: Ändert man das Rating eines Songs, so wird der Song mittels `remove` gelöscht und mittels `insert` an anderer Stelle wieder eingefügt. Meist verbindet sich dabei der Song, dann mit anderen ähnlich bewerteten Songs. Diese bilden ein *zusätzliches Netz* über dem Graphen, welches weitläufigere Sprünge ermöglicht. Dadurch hat der Nutzer eine Möglichkeit den Graphen seinen Vorstellungen nach umzubauen.

Unter Abbildung 5.4 soll dieses „explizite Lernen“ nochmal visualisiert werden. Die dort abgebildete Verschiebung ist dadurch zu erklären, dass die `insert`-Operation meist einen anderen Punkt zum Wiedereinfügen findet. Durch Ändern des Ratings in der Demonanwendung, können daher einzelne Knoten gezielt im Graphen bewegt werden. Knoten mit ähnlichem Rating wandern näher zusammen und stellen „Brücken“ zu anderen Alben-Clustern her. Man kann dieses *Feature* einerseits dazu nutzen, um seine Favoriten nahe im Graphen zusammenzupacken, andererseits, um unpassende Empfehlungen mit einem schlechten Rating abzustrafen. Letzteres hätte eine `insert`-Operation auf diesen Song zur Folge, wodurch er möglicherweise an anderer Stelle besser eingepasst wird.

Der „*Mechanismus*“ des *expliziten Lernens* ist war mehr ein Nebeneffekt der Entwicklung. Zukünftige Versionen könnten leichter steuerbar und intuitiver verständliche Mechanismen anbieten. Ein Ansatz wäre der Weg, den *Intelligente Playlisten* bei vielen Music-Playern gehen: Der Nutzer stellt Beziehungen zwischen Attributen und Werten her. Ein Attribut wäre beispielsweise `date`, ein Wert 2010 und eine Beziehung  $\geq$ . Weitere Beziehungen wären  $=$ ,  $\neq$ ,  $<$  oder  $\leq$ . Mit den unterschiedlichen Attributen, wären dann automatisch erstellte Playlisten wie „*Favouriten*“ ( $rating > 3$ ), „*Ungehörte*“ ( $Playcount = 0$ ) und „*Neu Hinzugefügtes*“ ( $date > (today - 7 \times days)$ ) möglich. Für Letztere könnten hilfreiche Konstanten wie `today` eingeführt werden.



(a) Vor dem Vergeben der Ratings.



(b) Nach dem Vergeben der Ratings. Die Bewegung wird durch Pfeile angedeutet.

**Abbildung 5.4.:** Vor und nach dem Vergeben von einem hohen Rating an drei Lieder („Rachsucht“, „Nagelfar“, „Meine Brille“, jeweils rot eingekreist). Die dazugehörigen Alben sind in röthlich, grünlich und bläulich hervorgehoben. Nach dem Vergeben sieht man, dass die entsprechenden Songs sich von den einzelnen Alben-Clustern räumlich entfernt haben und Verbindungen zu anderen Alben bekommen haben. Zudem haben sich die beiden erstgenannten Songs miteinander verbunden.

# 6 | Ausblick

## 6.1 Verbesserung der Algorithmik

IM Folgenden werden einige Ideen für mögliche Weiterentwicklungen an den vorgestellten Algorithmen gegeben. Einige davon sind vergleichsweise einfach umsetzbar. Andere könnten die Grundlage für fortführende Arbeiten sein.

### 6.1.1 Audioanalyse

Wie bereits erwähnt, ist *libmunin*'s momentane „*Audioanalyse*“ eher simpler Natur. Als konkrete Vorlage für eine verbesserte Audioanalyse könnte *Mirage* dienen. In seiner Arbeit stellt der Mirage-Autor [2] Dominik Schnitzer, einige Herangehensweisen zum performanten Vergleich von Audiodaten vor.

Angesichts der hohen Entwicklungsgeschwindigkeit in der Informatik und dem „hohem“ Alter der Arbeit (2007), sollte allerdings beachtet werden, dass es bereits neuere Methoden geben könnte. Beispielsweise arbeitet Schnitzer *nur* mit MP3-Audiodaten<sup>1</sup>. Eine Abhilfe wäre die relativ neue Bibliothek *libaubio*, die von *Paul Brossier* [Link-20] entwickelt wird. Zum Dekodieren der Audiodaten (*libaubio* erwartet bereits dekodierte PCM-Daten), könnte das weit verbreitete Audio-Framework *GStreamer* [Link-21] verwendet werden.

*Aubio* könnte folgendes leisten:

- Exaktere Bestimmung des *BPM-Wertes*. Beziehungsweise könnte man auch einen Verlauf des *BPM-Wertes*, über das Musikstück aufzeichnen, um exaktere Vergleiche ziehen zu können.
- *Onset-Detection*, also das Erkennen einzelner Noten, beziehungsweise *Sounds*, innerhalb eines Musikstücks. Die Bestimmung der Tonart wäre so in Ansätzen möglich.
- Eine direkte Möglichkeit, die Stimmung in einem Lied zu analysieren, wird momentan zwar noch nicht geboten, aber die dazu benötigten Informationen, wie die Erkennung der Tonlage zu einem bestimmten Zeitpunkt werden angeboten. Die technischen Details dazu werden in [2] diskutiert.

---

<sup>1</sup> *Mirage* verlässt sich dabei auf bestimmte Eigenschaften von MP3, um die Daten schneller in seine interne Datenrepräsentation zu konvertieren.

Die Bibliothek selber ist in C geschrieben, bietet aber eine komfortable Python-Schnittstelle.

Eine weitere Idee, wäre der Versuch, möglichst intelligent reine Sprachdateien (wie *Hörbücher*), Instrumental-Lieder ohne Stimme (wie *Intros*) und normale Musik zu unterscheiden. Oft werden zu bestimmten Titeln unpassenderweise *Intros* vorgeschlagen, die man für gewöhnlich nur hören möchte, wenn man das gesamte Album von vorn bis hinten anhört. Auch hier wäre ein Einsatz von *libaubio* denkbar.

### 6.1.2 Andere Provider

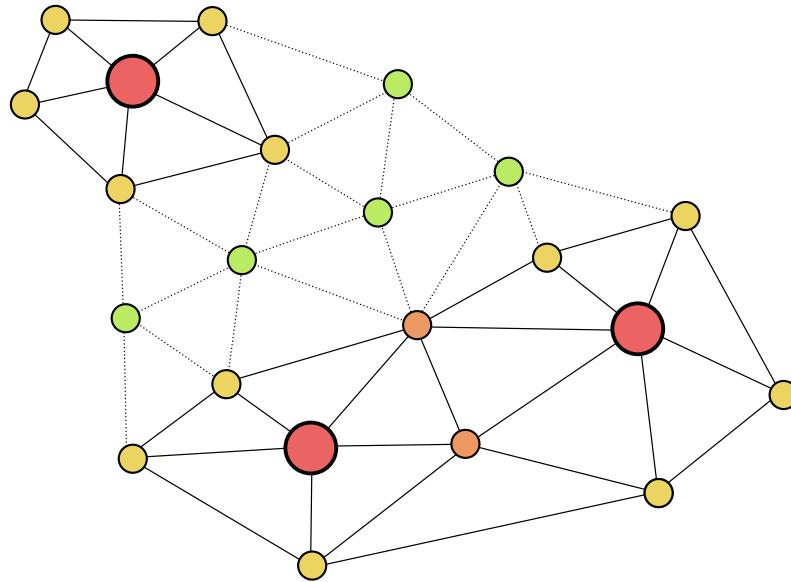
Wie man im Playlistenvergleich unter Kapitel 2.5 gesehen hat, ist momentan der Vergleich der Metadaten, die Stärke von *libmunin*. Diese Fähigkeit könnte noch weiter ausgebaut werden, indem die Sprache der einzelnen Titel (denn nicht immer sind Liedtexte vorhanden) erkannt wird. Dann könnte man mittels eines Thesaurus synonyme Titel finden. Für Python existiert mit *TextBlob* [Link-22] hierfür eine passende Bibliothek. Kommt beispielsweise in einem Liedtitel das Wort „*Sofa*“ vor, so könnte ein Titel mit dem synonymen Wort „*Couch*“ darin vorgeschlagen werden. Auch Taxonomien, also ähnliche Klassifikationen, sind denkbar. Man denke hier an einem Lied, welches das Wort „*Katze*“ enthält und ein anderes das „*Tier*“ beinhaltet. In der momentanen Implementierung wird jedes Wort im Titel, auf seinen Wortstamm gebracht und mittels der Levenshtein-Distanzfunktion verglichen. Diese Lösung ist zwar leicht zu implementieren, ist aber algorithmisch teuer und verhindert lediglich Tippfehler oder leicht divergente Schreibweisen.

Auch interessant zu sehen wäre es, ob die Länge der einzelnen Stücke in irgendeiner Form mit der Ähnlichkeit korrelieren. Hier müssten statistische Auswertungen gemacht werden, um diesen Zusammenhang zu überprüfen. Falls sich ein Zusammenhang zeigen sollte, ließe sich eine einfache DurationDistanceFunction schreiben, welche ähnlich lange Stücke gut bewertet.

### 6.1.3 Empfehlungen

Oft kommt es vor, dass es mehr als einen *Seedsong* gibt. Die momentane, simple Herangehensweise, ist für jeden einen Iterator zu erstellen und die einzelnen Iteratoren, im Reißverschlussverfahren zu verweben. Das ist durchaus valide, wenn man annimmt, dass die *Seedsongs* im Graphen verteilt und alle gleich wichtig sind. Oft ballen sich *Seedsongs* aber auf einem bestimmten Gebiet. Schematisch ist das in Abbildung 6.1 dargestellt. Besitzen zwei *Seedsongs* gemeinsame Nachbarn, dann sollten diese zuerst besucht werden.

Auch ist das Ausgabeformat von *libmunin* noch auf einzelne Songs als Empfehlung beschränkt. Nicht selten möchte man jedoch eine allgemeinere Auskunft wie „*Gib mir einen ähnlichen Künstler/Album/Genre*“. Momentan wäre dies nur durch Auslesen der jeweiligen Attribute aus den einzel-



**Abbildung 6.1.:** Schematische Darstellung der idealen Traversierungsreihenfolge. Die roten Knoten stellen die Seedsongs dar, die gelben und orangen Knoten sind direkte Nachbarn. Die grünen Knoten sind „irgendwo“ dazwischen. Die Traversierungsreihenfolge sollte hier sein: Orange, Gelb, Grün.

nen Empfehlungen möglich. Allerdings könnten hier von *libmunin* optimierte Traversierungsstrategien implementiert werden.

## 6.2 Erweiterungen

Die verwendeten Metadaten könnten ebenfalls erweitert werden. Für die Ähnlichkeit sind unter Umständen auch Attribute wie der *Producer*, die *Band-Mitglieder* oder die *Herkunft der Band* relevant. Einfache Beispiele wären hier: „Wer Songs von den Ärzten hört, der hört vermutlich auch gern Farin Urlaub Racing Team“ — natürlich unter der Annahme, dass derselbe Künstler auch immer ähnliche Musik produziert.

Was das Lernen von *libmunin* angeht, so sollten auch „*negative Impulse*“ behandelt werden. Wird ein bestimmtes Lied oder gar ein Künstler sehr oft übersprungen, könnte *libmunin* dies berücksichtigen, indem es bei der Traversierung diesen Knoten umgeht. Alternativ wäre auch ein nachträgliches Filtern der entsprechenden Lieder möglich.

Allgemein wäre auch eine Erweiterung von Assoziationsregeln denkbar. Momentan verbindet eine Regel immer zwei Mengen von Songs miteinander. Alternativ könnten aber auch verschiedene Genres, Künstler oder auch Alben in einer Regel miteinander verbunden werden. Das Erstellen solcher Regeln wäre relativ einfach mit der existierenden Implementierung. Was problematisch ist, ist diese neuen Regeln als *Traversierungshilfe* zu nutzen.

Ein weiterer Punkt, den man beim Lernen verbessern könnte, sind die Gewichtungen, die manuell für jedes Attribut festgelegt werden. Man könnte den Nutzer detaillierter beobachten und sehen, nach welchem Attribut er bevorzugt seine Lieder auswählt (beispielsweise nach Genre). Das entsprechende Attribut könnte dann höher gewertet werden.

Auch wäre ein zusätzliches Modul möglich, das *libmunin* nutzt, um Suchanfragen basierend auf natürlicher Sprache zu ermöglichen. So könnten Anfragen wie „*Happy Indie Pop*“ aufgelöst werden. Im Beispiel würde sich *Happy* auf die Stimmung beziehen, *Pop* auf das Genre und *Indie* auf einen Independent-Künstler. Letztere Information könnte man aus der Künstlerbiografie extrahieren. Die Biografie kann automatisch von Tools wie *libglyr* besorgt werden oder man greift alternativ auf Amazon-Rezensionen zurück. So gesehen, bietet sich hier ein Erweiterungspotenzial in Richtung „*Social-based-Recommendations*“. Also nutzt man das Wissen von vielen Menschen, um bestimmte Attribute zu bestimmen, anstatt diese mithilfe von Metriken zu errechnen. Die eigentliche Schwierigkeit bestünde aber darin, die einzelnen Wörter bestimmten Attributen zuzuordnen. Diese Idee basiert auf der Musiksuchmaschine von *Peter Knees* [14].

## 6.3 Fazit

Momentan ist *libmunin* vor allem eine Spielwiese für verschiedene Ideen, rund um die Frage, wie man einem Computer die Ähnlichkeit von zwei Musikstücken feststellen lässt. Trotzdem erstellt *libmunin* selbst als Prototyp in seiner Standardeinstellung bereits durchaus nützliche Playlisten. Aufgrund der kurzen Implementierungszeit für ein solches System, von etwas mehr als 3 Monaten, ist dies nach Meinung des Autors durchaus als Erfolg zu werten.

Die größte Schwäche ist aus Sicht des Autors der langsame Kaltstart, der einen produktiven Einsatz der Bibliothek verhindert. In der Weiterentwicklung, sollte dies die höchspriorisierte Aufgabe sein.

Die Neuerung dieser Arbeit ist weniger die vorgestellte Algorithmik. Der allergrößte Teil dieser, existiert natürlich bereits in ähnlicher Form, verstreut über viele verschiedene Softwarepakete. Die tatsächliche Neuerung ist, dass diese Funktionalität erstmals in einer allgemein nutzbaren und freien Bibliothek vorhanden ist.

# A | Bilder des Song-Graphen

## Allgemeine Hinweise

Folgende visuelle Aspekte sind mit Informationen besetzt:

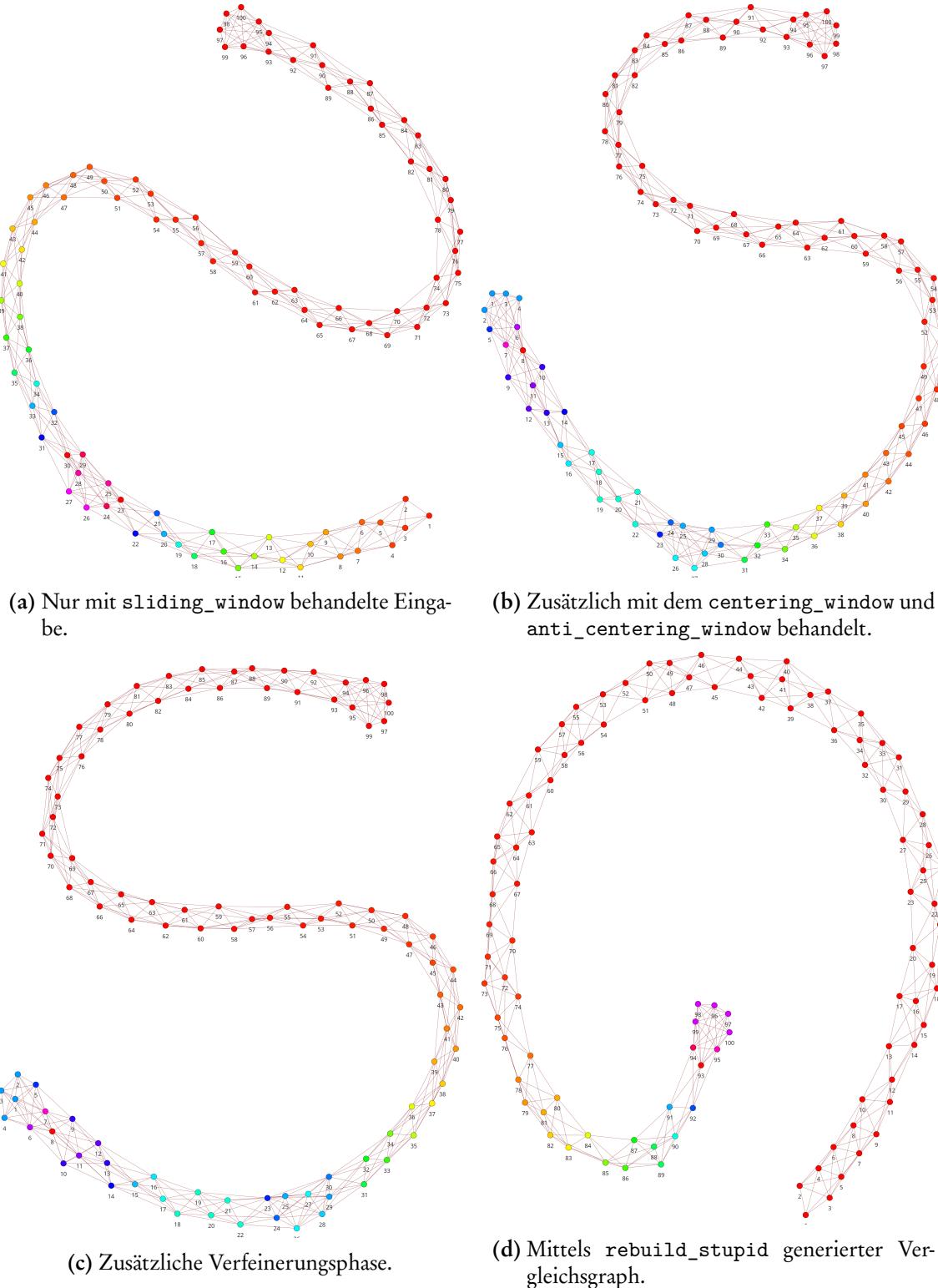
- *Kantenfarbe*: Rote Farbe indiziert eine hohe Distanz. Grün eine mittlere Distanz (etwa 0,5). Blau bis violett eine gute bis sehr gute (0,1 bis 0,0). Rote Kanten sieht man in der Praxis selten, da in den meisten Fällen immer ein guter Nachbar gefunden werden kann.
- *Kantendicke*: Zur optischen Untermalung variiert die Dicke der Kanten je nach Distanz. Hohe Distanzen bekommen eine dünnere Kante.
- *Knotenfarbe*: Zeigt grob die „Zentralität“ des Knotens an. Bietet lediglich eine optische Navigationshilfe um auf Cluster hinzuweisen.
- Der visuelle Abstand zwischen zwei Knoten, hat *keine Aussagekraft*, auch wenn ähnliche Knoten oft nah beieinander liegen.

Die Abbildungen A.1a bis A.1d und A.2 bis A.5 zeigen den Aufbau des Graphen, anhand von linearen und pseudozufälligen generierten Testdaten mit jeweils 100, beziehungsweise 200 Knoten. Abbildung A.6 zeigt einen realen, praktischen Graphen, mittlerer Größe mit 666 Knoten. Die Liedtitel wurden jeweils an die Knoten geschrieben.

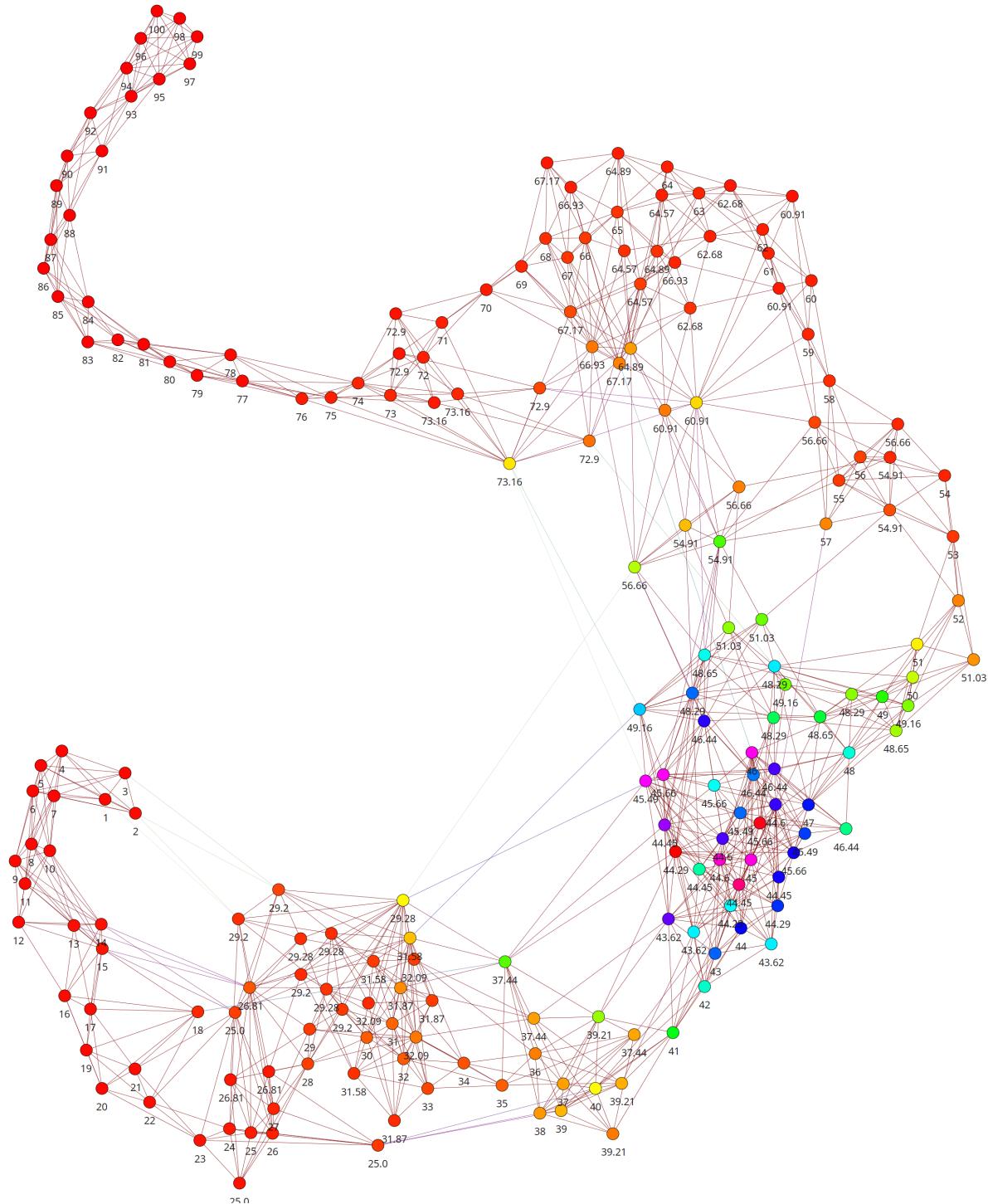
Bei den linearen Testdaten werden die Zahlen von 100 bis 1 absteigend jedem neuen Knoten gegeben. Das Resultat sollte im optimalen Fall daher eine Kette von Knoten zeigen, deren Knotennummer von einem Ende zum anderen immer weiter steigt. Die pseudozufälligen Testdaten mischen die erwähnten linearen Daten, mit zusätzlichen, zufällig erscheinenden Daten gleicher Anzahl. Diese sind zum größten Teil auf den Bereich 25–75 aufgeteilt. Es sollte ein Graph entstehen, der ähnlich linear ist, aber in der Mitte eine „Verdichtung“ aufweist. Beides sind Testdaten, die häufig zum Testen von *libmunin* genutzt worden sind, da sie spezielle Extremfälle gut testen. Die linearen Testdaten sind bereits nach der Basisiteration einsetzbar. Die linearen Enden der pseudozufälligen Daten, werden mit jeder Iteration klarer. Im Vergleich zu `rebuild_stupid` fällt hier auf, dass `rebuild` zur Bildung von mehr Kanten neigt.

## Plotting der Graphen

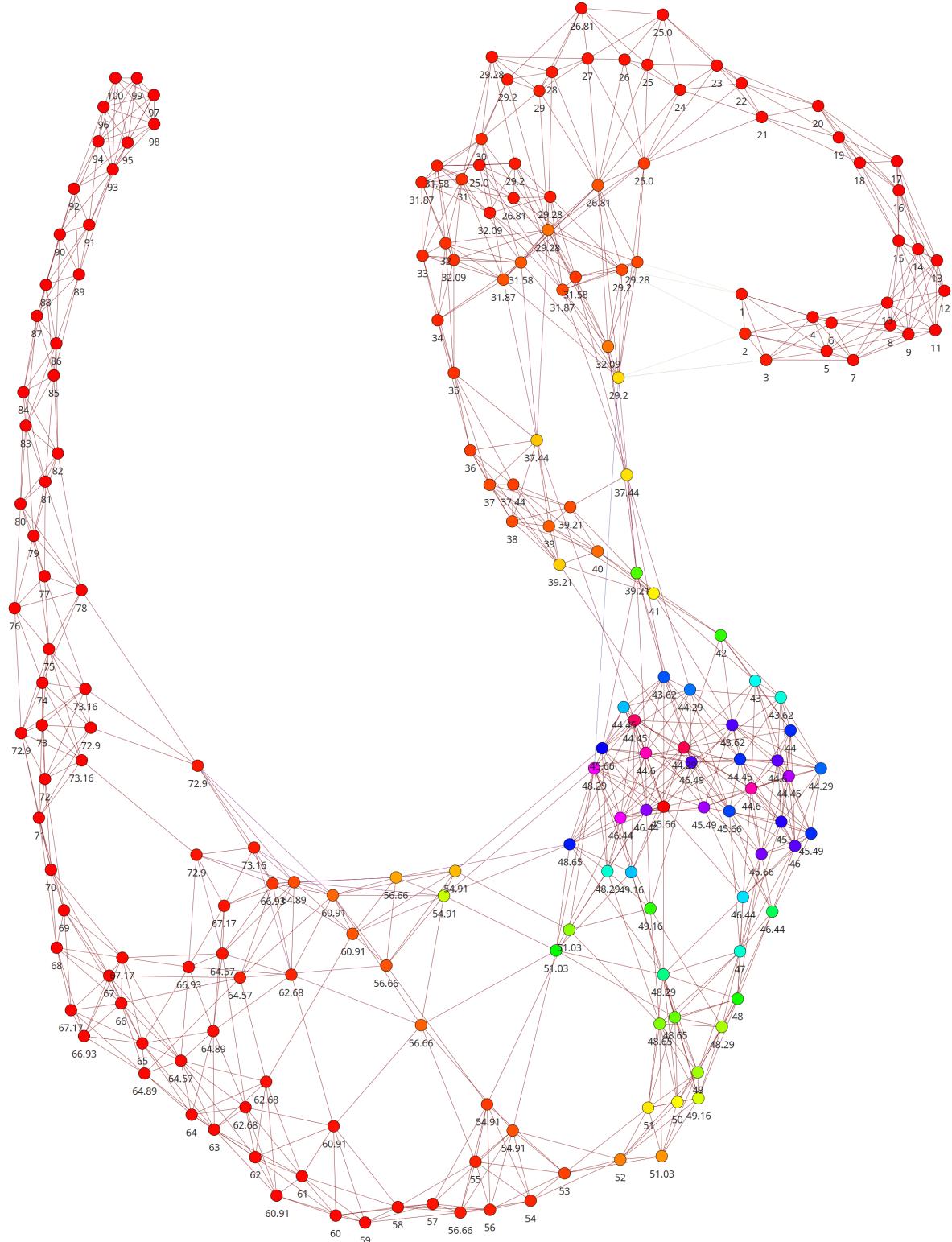
Alle Graphen in diesem Teil des Anhangs, sind mithilfe des freien Python-Graphenframeworks `igraph` [Link-23] entstanden. Als Zeichenbibliothek nutzt `igraph` die freie 2D-Zeichenbibliothek `cairo` [Link-24]. Der verwendete Layouting-Algorithmus ist „*Fruchtermann-Reingold*“.



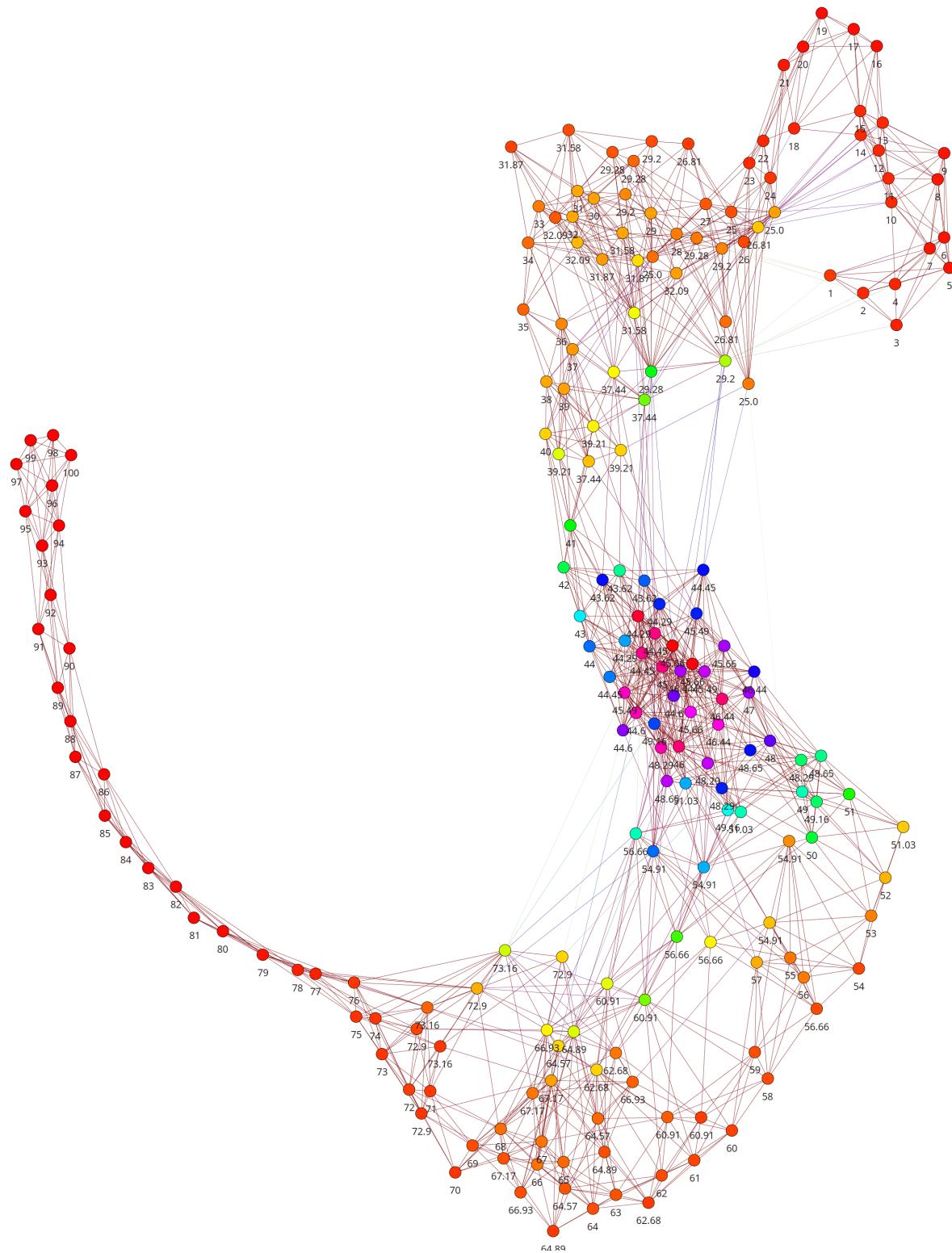
**Abbildung A.1.:** Verschiedene Stufen beim Aufbau eines Graphen aus linearen Testdaten. Die Testdaten bestehen aus den Integern 1 bis 100. Erwartet wird dabei als Ausgabe eine lineare Kette von Knoten, wobei jeder Knoten ca. 7 Nachbarn haben sollte. Bei rebuild\_stupid ist dies fast immer der Fall, beim normalen rebuild liegt der Durchschnitt etwa bei 9. Ansonsten sind bei diesem Test kaum Unterschiede zwischen den Varianten festzustellen.



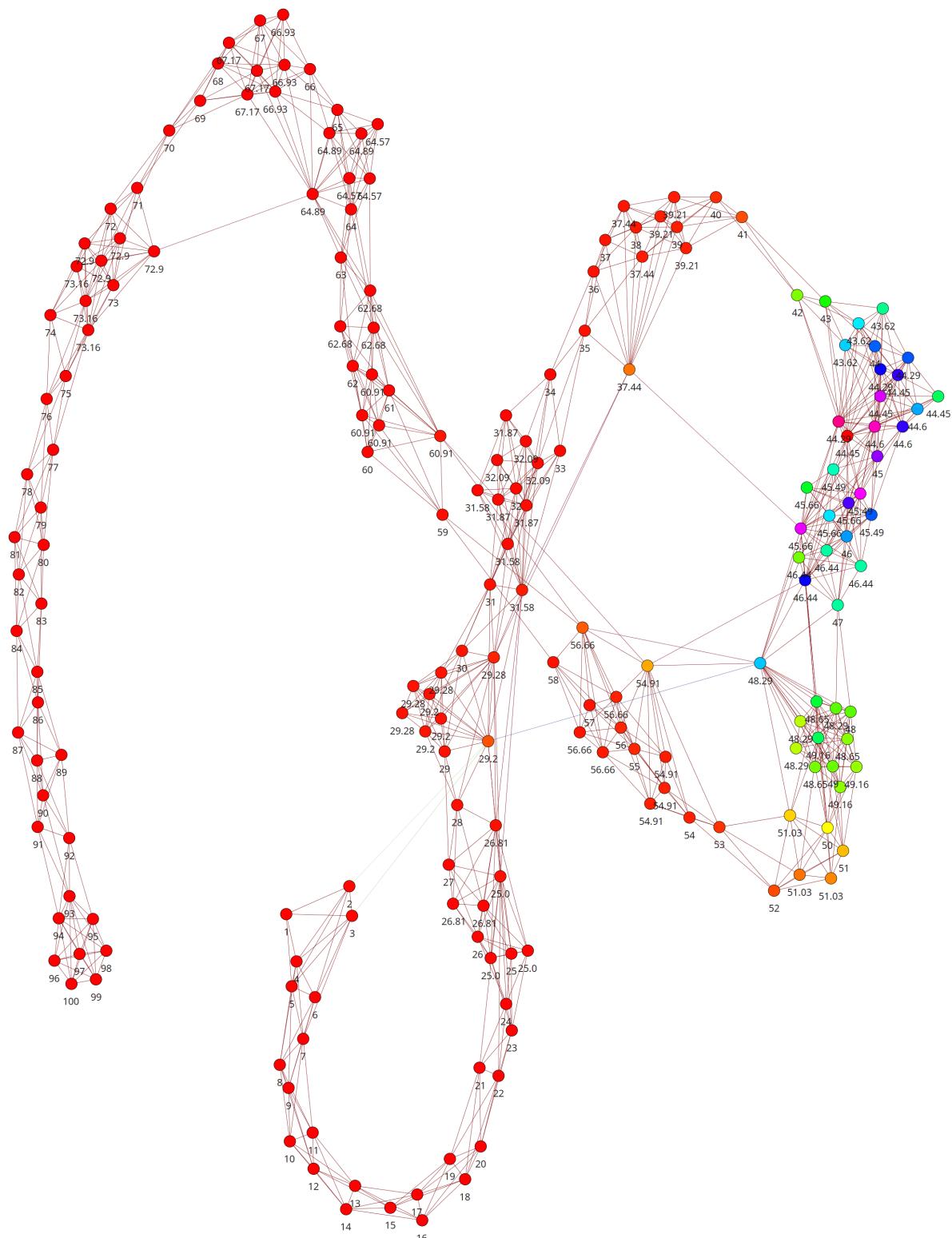
**Abbildung A.2.:** Graph aus Zufallsdaten nach erster Basisiteration. Die Zahlen 75–100 bilden einen „Schwanz“, während die Zahlen 0–25 noch nicht stark abgrenzen. Die restlichen Zahlen dazwischen bilden die erwartete Verdickung.



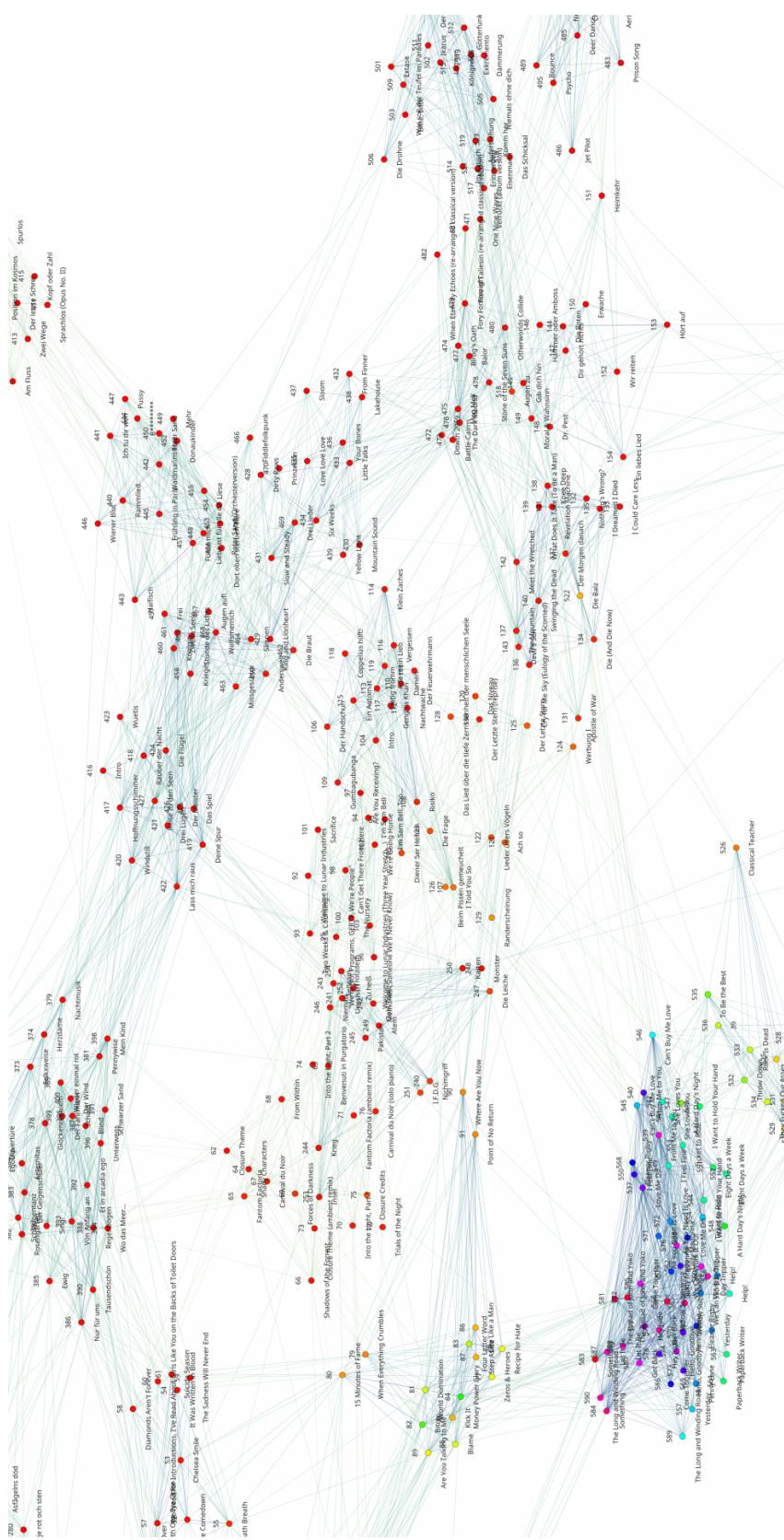
**Abbildung A.3.:** Graph aus Zufallsdaten nach allen Basisiterationen. Die Zahlen 0–25 sind nun stärker abgegrenzt.



**Abbildung A.4.:** Graph aus Zufallsdaten nach einem Verfeinerungsschritt. Im Vergleich zum vorigen Graph hat die Verkettungsdichte deutlich zugenommen. Die Zahlen 75–100 sind nun sehr deutlich abgegrenzt. Der Mittelteil ist insgesamt stärker zusammen gewachsen.



**Abbildung A.5.:** Der „korrekte“, mittels rebuild\_stupid erstellte, Graph. Er bildet eine vergleichsweise dünne Kette, die in der Mitte nur kleine Ausbuchtungen hat. Zudem gibt es nur einige wenige Querverbindungen zu weiter entfernten Knoten. Anmerkung: Durch das Layout bedingt, überlagert sich die Kette in der Bildmitte.



**Abbildung A.6.:** Auschnitt aus dem vollständigen Graph, der hinter den Empfehlungen der Demomanwendungen steckt. Auf der beiliegenden CD, ist der vollständige Graph, in A0-Größe enthalten. Klar erkennbar sind die einzelnen Alben im Graphen.

# B | Bilder des Genregraphen

Der Vollständigkeit halber, wird hier auch eine Visualisierung des Genrebaums gezeigt. Daraus kann man zwar keine „*neuen Erkenntnisse*“ ziehen, doch kann er in der weniger detaillierten Version zumindest als Überblick über die einzelnen Genres dienen. Die detailliertere Version führt einem vor Auge, wieviel einzelne Musikgenres mit der Zeit entstanden sind. Die Graphen wurden mit dem freien Graphenvisualisierungspaket **Graphviz** [Link-25] generiert. Das Vorgehen soll hier dokumentiert werden.

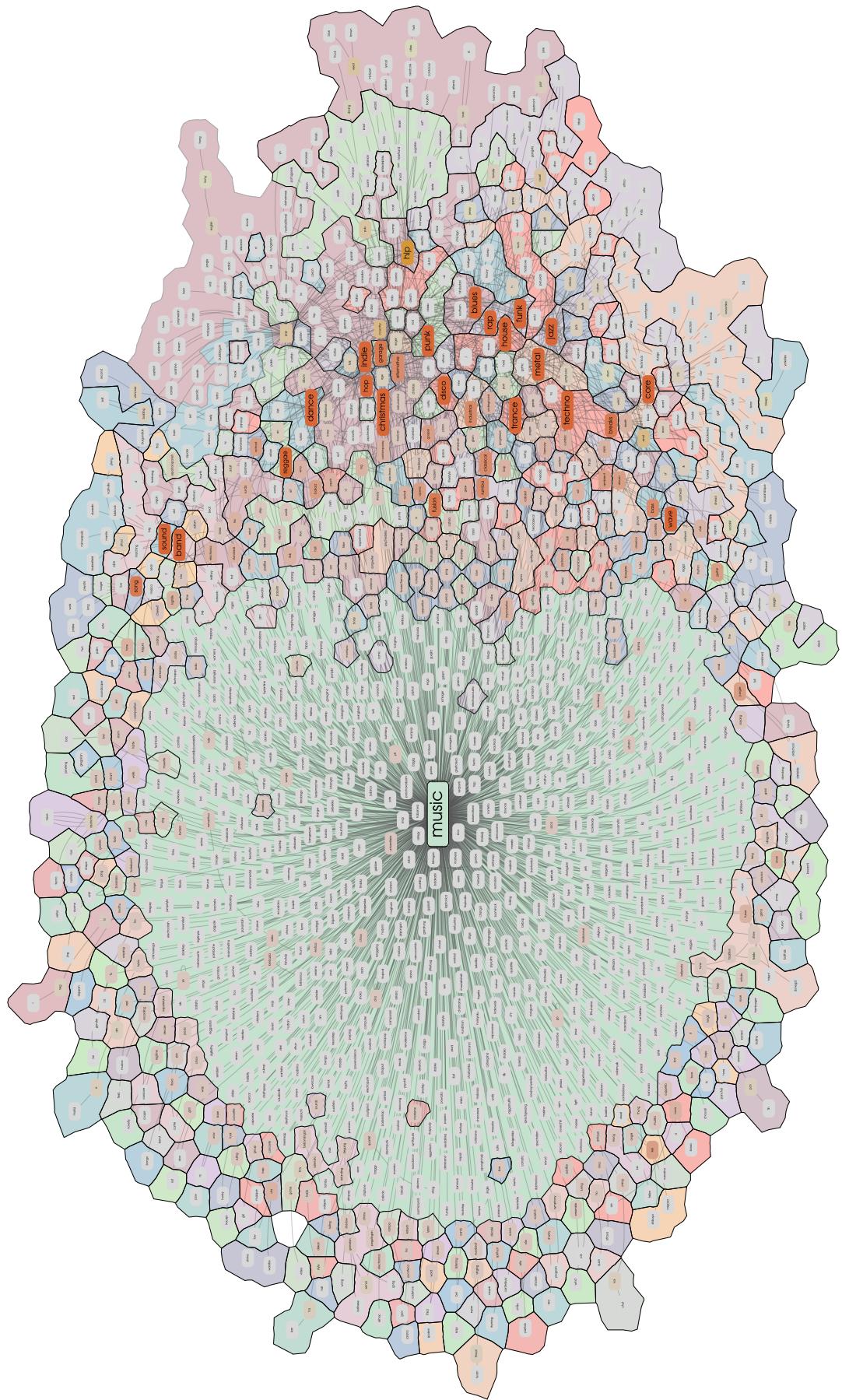
Folgendes erstellt die Graphviz-Eingabedatei.

```
$ pwd
~/dev/libmunin
$ export PYTHONPATH=$PYTHONPATH:..
$ python "munin/provider/genre.py" --cli --plot 0,0 # Detailstufe: 0,0 = max.
$ head -n8 "/tmp/genre.graph"
graph {
    overlap=prism3000; overlap_scale=-7; splines=curved
    edge [color="#666666"]
    node [shape="none", style="rounded, filled", fillcolor="..."]
    "music" -- "rock"
    "music" -- "pop"
    // ...
}
```

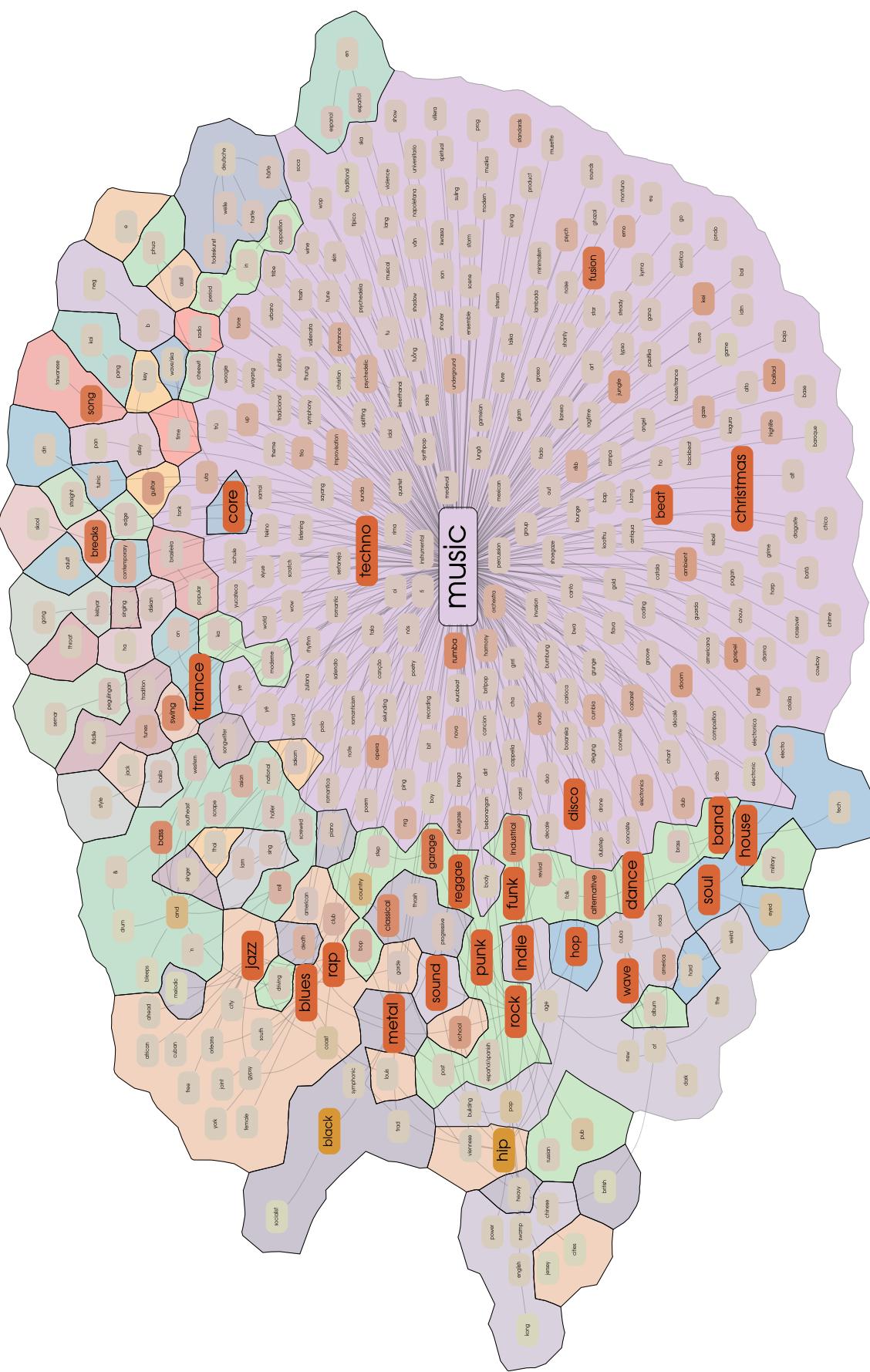
Daraus kann man dann, mittels Graphviz-Bordmitteln, den Graphen in verschiedenen Formaten rendern. Hier die Prozedur für ein *PDF*:

```
$ sfdp /tmp/genre.graph
      | \
      | gvmap -e          # Layoutting der einzelnen Nodes.
      | neato             # Einzeichnen der "Länderflächen".
      |                   \# Eigentliches Zeichnen.
      | -Ecolor="#55555555" \# Farbe der Kanten.
      | -Nfontname="TeX Gyre Adventor" \# Font für die Nodelabels.
      | -Tpdf              \# Ausgabeformat.
      > graph.pdf          \# Ausgabedatei.
$ pdf-viewer graph.pdf           # Fertiges Bild.
```

*Anmerkung:* Die einzelnen „*Länder*“ beziehungsweise Cluster im Graphen dienen lediglich der optischen Trennung. Die Farbsättigung der Knoten und die verwendete Größe der Schrift weist auf die Menge der Kinder hin, die der Knoten hat. Je weiter weg der Knoten vom Wurzelknoten entfernt ist, desto oranger wird er (von rot beginnend).



**Abbildung B.1.:** Übersicht über alle 1876 Musikgenres. (Detailstufe: 0,0)



**Abbildung B.2.:** Übersicht über die gebräuchlichsten Musikgenres. (Detailstufe: 0,1)

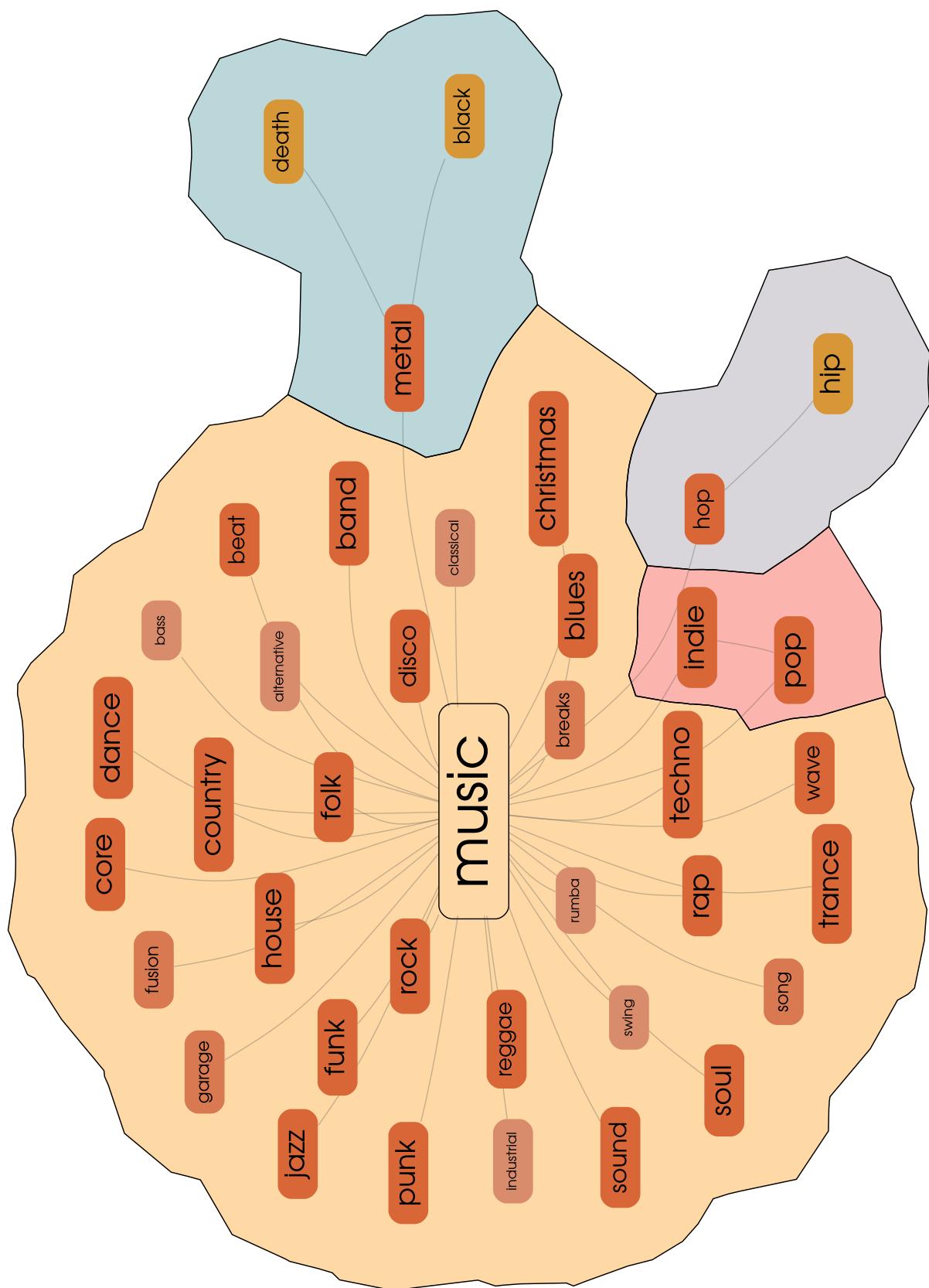


Abbildung B.3.: Übersicht über die wichtigsten Musikgenres. (Detailstufe: 0,5) Als „wichtig“ gelten dabei Genres, die viele Untergenres hervorgebracht haben.

# C | Medien

Der Vollständigkeit halber werden hier noch die für *libmunin* erstellten Logos abgedruckt. Abbildung C.1a dient dabei eher als Logo für eigene Zwecke, während das *Emblem* unter C.1b von Anwendungsentwicklern genutzt werden kann, um auf den Gebrauch von *libmunin* in ihren Anwendungen hinzuweisen.

Die Grafiken wurden mit dem freien Vektorzeichenprogramm *Inkscape* [Link-26] erstellt und können auf der API-Dokumentation heruntergeladen werden: [Link-27].

Dieses Dokument existiert neben dem vorliegenden *PDF* auch, wie schon bei der Projektarbeit, als HTML-Version: [Link-28].



(a) Logo im quadratischen Format.



(b) Logo im breiten Format.

Abbildung C.1.: Logos für libmunin. Der dargestellte Vogel stellt Odin's Rabe „Munin“ dar.

# D | Literaturverzeichnis

- [1] Christopher Pahl. *Design und Implementierung eines Musikempfehlungssystems basierend auf Datamining—Algorithmen*. Hof University of Applied Sciences, 2014.
- [2] Dominik Schnitzer. High-performance music similarity computation and automatic playlist generation. 2007.
- [3] Eric Brill and Robert C Moore. An improved error model for noisy channel spelling correction. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, 286–293. Association for Computational Linguistics, 2000.
- [4] Donald E Knuth. *The Art of Computer Programming: Volume 3 Sorting and Searching*. Addison-Wesley, 1997.
- [5] Benno Stein. *Automatische Extraktion von Schlüsselwörtern aus Text*. Bauhaus–Universität Weimar, 2006.
- [6] Michael W Berry and Jacob Kogan. *Text mining: applications and theory*. John Wiley & Sons, 2010.
- [7] Martin F Porter. Snowball stemmer. <http://snowball.tartarus.org/>, 2001.
- [8] Gavin Wood and Simon O’Keefe. On techniques for content-based visual annotation to aid intra-track music navigation. In *ISMIR*, 58–65. 2005.
- [9] Jian Pei Jiawei Han, Micheline Kamber. Datamining — concepts and techniques, (3rd edition). In *Datamining - Concepts and Techniques*. 2012.
- [10] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Ogihsara, and others. New algorithms for fast discovery of association rules. In *KDD*, volume 97, 283–286. 1997.
- [11] Christian Borgelt. Keeping things simple: finding frequent item sets by recursive elimination. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, 66–70. ACM, 2005.
- [12] Cornelia Győrödi, Robert Győrödi, and Stefan Holban. A comparative study of association rules mining algorithms. In *SACI 2004, 1st Romanian-Hungarian Joint Symposium on Applied Computational Intelligence*, 213–222. 2004.
- [13] B Santhosh Kumar and KV Rukmani. Implementation of web usage mining using apriori and fp growth algorithms. *International Journal of Advanced Networking*, 2010.
- [14] Peter Knees, Tim Pohle, Markus Schedl, and Gerhard Widmer. A music search engine built upon audio-based and web-based similarity measures. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, 447–454. ACM, 2007.

- [Link-1] <https://libmunin.rtfd.org>. [Stand: 11.2.2014].
- [Link-2] <http://www.gnu.org/copyleft/gpl.html>. [Stand: 5.5.2014].
- [Link-3] [http://musicbrainz.org/doc/MusicBrainz\\_Picard](http://musicbrainz.org/doc/MusicBrainz_Picard). [Stand: 15.4.2014].
- [Link-4] <http://musicbrainz.org/>. [Stand: 11.2.2014].
- [Link-5] <https://github.com/sahib/glyr>. [Stand: 11.2.2014].
- [Link-6] <http://beets.radbox.org/>. [Stand: 11.2.2014].
- [Link-7] <http://www.discogs.com>. [Stand: 11.4.2014].
- [Link-8] [http://de.wikipedia.org/wiki/Hierarchische\\_Clusteranalyse](http://de.wikipedia.org/wiki/Hierarchische_Clusteranalyse). [Stand 12.4.2014].
- [Link-9] <https://pypi.python.org/pypi/bidict>. [Stand: 14.4.2014].
- [Link-10] <http://www.last.fm>. [Stand: 11.2.2014].
- [Link-11] <https://github.com/gfairchild/pyxDamerauLevenshtein>. [Stand: 14.4.2014].
- [Link-12] [https://musicbrainz.org/doc/General\\_FAQ](https://musicbrainz.org/doc/General_FAQ). [Stand: 13.4.2014].
- [Link-13] <https://gist.github.com/sampsyo/1241307>. [Stand: 12.4.2014].
- [Link-14] [http://en.wikipedia.org/wiki/List\\_of\\_popular\\_music\\_genres](http://en.wikipedia.org/wiki/List_of_popular_music_genres). [Stand: 15.4.2014].
- [Link-15] [http://en.wikipedia.org/wiki/List\\_of\\_music\\_styles](http://en.wikipedia.org/wiki/List_of_music_styles). [Stand: 15.4.2014].
- [Link-16] [https://bitbucket.org/spirit/guess\\_language](https://bitbucket.org/spirit/guess_language). [Stand: 13.4.2014].
- [Link-17] <http://pythonhosted.org/pyenchant/>. [Stand: 14.4.2014].
- [Link-18] <http://www.abisource.com/projects/enchant/>. [Stand: 14.4.2014].
- [Link-19] <https://github.com/bartdag/pymining>. [Stand: 13.4.2014].
- [Link-20] <https://github.com/piem/aubio>. [Stand: 10.4.2014].
- [Link-21] <http://gstreamer.freedesktop.org/>. [Stand: 18.4.2014].
- [Link-22] <https://github.com/sloria/TextBlob>. [Stand: 10.4.2014].
- [Link-23] <http://igraph.sourceforge.net/>. [Stand: 13.4.2014].
- [Link-24] <http://cairographics.org/>. [Stand: 13.4.2014].
- [Link-25] <http://www.graphviz.org/>. [Stand: 13.4.2014].
- [Link-26] <http://www.inkscape.org>. [Stand: 15.4.2014].
- [Link-27] <http://libmunin.readthedocs.org/en/latest/media.html>. [Stand: 14.4.2014].
- [Link-28] <http://libmunin.readthedocs.org/en/latest/docs.html>. [Stand: 14.4.2014].

## **Eidesstattliche Erklärung**

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer, als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken, sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form, keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

*Hof, den 11. Mai 2014*

---

*Christopher Pahl*