Ein Komparatives Query by Example System

2008-2009

Lukas Maczejka (0426085)
Christian Seidl (0426409)

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument beschreibt die grundlegende Systemarchitektur sowie die einzelnen implementierten Features und Funktionalitäten. Weiters werden Details der implementierten Algorithmen behandelt sowie die Verwendung des Systems erklärt. Die Vorgehensweise zur Erweiterung des Systems um zusätzliche Extraktions- bzw. Vergleichsalgorithmen wird behandelt.

1.1 Systemarchitektur

Das System besteht aus drei miteinander in Verbindung stehenden Hauptmodulen. Verschiedene Metadaten werden aus MIDI- Quellen gelesen und in einer eigenen Bibliothek abgelegt. Abhängig vom Format der Metadaten sind mehrere Vergleichsalgorithmen implementiert, die den Unterschied zweier extrahierter Features numerisch ausdrücken. Sowohl das Bibliothekssystem als auch das Vergleichssystem können jederzeit einfach um weitere Algorithmen erweitert werden.

1.2 Metadaten

Metadaten werden mit einem eindeutigen Identifier in der Datenbank (in einer Verzeichnisstrukur) gespeichert. Das System bietet Interfaces zur Ablage und zum Einlesen der Daten aus dem Dateisystem an. Weiters wird eine Methode angeboten, die einen Vergleichsalgorithmus auswählt und auf zwei gleichartige Metadaten anwendet. Außerdem bietet das System Preprocessing der MIDI- Daten an, um einfaches und möglichst fehlerfreies extrahieren der Features zu ermöglichen.

Pro Quell-MIDI wird eine Metadatei angelegt. Die ersten 3 Zeilen dieser Metadatei definierend deren Header, und beinhalten Versionsinformationen und den relativen und absoluten Pfad zum Quell-MIDI. Anschliessend wird jedes extrahierte Feature im folgenden Format abgelegt:

1: Name des Features
2: Anzahl Zeilen
3-N: Extrahierte Daten

Alle Daten werden in menschlich lesbarem Text gespeichert.

Implementierte Feature-Extraction Algorithmen:

- Parson's Code (Up/Down Sequenzen)
- Note Duration (Dauer der Einzelnoten)
- Note Histogram (Notenhistogramm nach Notenname)

- Lempel Ziv - Längste Sequenzen (Adaption des Lempel Ziv 78 Algorithmus zu Extraktion der häufigsten Sequenzen)
- Lempel Ziv - Häufigste Sequenzen
- Note Sequence
- Statistics

1.3 Bibliothek

Das Bibliothekssystem verwaltet eine Liste von Metadaten und stellt diese für Vergleiche zur Verfügung. Die Bibliothek kann einfach um neue MIDI- Quellen erweitert werden. Nach Extraktion der Features werden diese in einem verwalteten Verzeichnis abgelegt.

1.4 Vergleichsalgorithmen

Ein Vergleichsalgorithmus vergleicht jeweils zwei Instanzen von Metadaten. Ergebnis ist eine Ähnlichkeit (bzw. Differenz), die numerisch zwischen 0 und 1 ausgedrückt wird, wobei 1 laut Algorithmus idente Features beschreibt, währen 0 für zwei komplett verschiedene Inputs steht.

Implementierte Vergleichsalgorithmen:

- Dice's Coefficient (Bi-Gram Vergleich von zwei Strings)
- Euklidische Distanz (Zum Vergleich numerischer Vektoren)
- Hamming Distance (Vergleich zweier Strings gleicher Länge)
- Levenshtein Distance (Vergleich zweier Sequenzen unterschiedlicher Länge)

1.5 Interface

Für dieses System wurde ein Benutzerinterface zum einfachen Zugriff auf die einzelnen Funktionen entwicklet. Das Interface erlaubt die Verwaltung der Bibliothek und den Vergleich der in der Bibliothek gespeicherten Daten. In weiterer Folge soll das Interface den Benutzer bzw. Entwickler dabei unterstützen, die Anwendbarkeit und Qualität unterschiedlicher Audio- Feature- Exctraction Algorithmen zu vergleichen.

2 Feature- Extraktion

Dieses Kapitel wird die einzelnen implementierten Feature-Extraktionsalogrithmen im Detail behandeln.

2.1 Preprocessing

Vor der Extraktion der jeweiligen Metadaten können folgende Preprocessing- Tools angewandt werden um die Qualität der Ergebnisse zu erhöhen:

Percussion Track. MIDI Spur 10 ist für beat/percussion reserviert. Laut Standard darf diese Spur also keine melodierelevanten Informationen enthalten, und kann somit für die meisten extrahierten Features ignoriert werden.

Beat Filter. Auch andere Spuren können nicht-melodische Informationen enthalten. Der Beat Filter berechnet die durchschnittliche Anzahl an Veränderungen im Parson's Code und filtert die Spur aus, sofern die berechnete Zahl unter einem bestimmten Schwellenwert liegt.

Phrase Merger. MIDI Spuren sind in mehreren *Phrasen* abgespeichert. Das sequenzielle Abarbeiten dieser Phrasen innerhalb einer Spur führt zu verfälschten Ergebnissen, da oft mehrere Noten aus unterschiedlichen Phrasen innerhalb einer Spur gleichzeitig abgespielt werden (Akkorde). Hier wird ein Skyline- Algorithmus angewandt um einen Akkord auf jeweils eine Note zu reduzieren und überlappende Phrasen in ein einheitliches, zeitlich kohärentes Format zu bringen.

2.2 Parson's Code

Der Parson's Code (auch Up-Down Sequenz) repräsentiert die Folge der Tonlagen eines Stücks als Liste von drei möglichen Veränderungen. Die aktuelle Note kann entweder die *gleiche* Tonlage der vorhergehenden Note haben, oder *höher* bzw. *tiefer* sein. Dadurch entsteht ein String von Veränderungswerten, der zum Vergleich eines Stücks herangezogen werden kann.

Folgendes Beispiel zeigt den Auszug aus einem Parson's Code:

0021111110022200001020010011200211111100222000010200100 11

Abbildung 1: Parson's Code

Im implementierten System steht 0 für eine Veränderung der Tonhöhe nach unten, 1 für eine Veränderung nach oben, und 2 für keine Veränderung.

2.3 Note Duration

Noten werden im MIDI-Format mit einer bestimmten Länge abgespeichert. Dieser Extraktionsalgorithmus speichert die Längen der einzelnen aufeinanderfolgenden Noten ab.

Folgendes Beispiel zeigt einen Auszug der extrahierten Notenlängen:

0.5;0.34375;1.0520833333333333;2.947916666666667;0.5520833333333334;0.53125;0.53125;

Abbildung 2: Note Duration

Dieses Feature erlaubt Distanzvergleiche der Längenvektoren verschiedener Stücke.

2.4 Note Histogram

Das Notenhistogramm repräsentiert die Anzahl des Vorkommens einzelner Noten in einem Stück. Im Unterschied zum Parson's Code bzw. der Note Duration, wird dieser Algorithmus über alle MIDI-Spuren (sofern nicht im Preprocessing herausgefiltert) angewandt, sodass ein Gesamtergebnis entsteht und nicht ein Ergebnis pro MIDI-Spur.

D:226
C:225
B:39
C#:33
A:134
Bb:114
G:78
F:290
E:144

Abbildung 3: Note Histogram

2.5 Lempel-Ziv

Der Lempel-Ziv Komprimierungsalgorithmus kann angewandt werden um effizient wiederkehrende Sequenzen innerhalb eines Strings zu identifizieren. Der Algorithmus verwendet ein Dictionary zur temporären Speicherung immer längerer, wiederkehrender Sequenzen. Dieses Dictionary kann verwendet werden um häufig wiederkehrende String- Sequenzen innerhalb eines Stückes zu finden.

Da dieser Algorithmus einen Eingabestring benötigt, basiert er auf dem vorher bereits extrahierten Parson's Code.

16:1021010102
15:010102101010
15:1010210101
16:10210101021
20:1010102101
7:0101021011
23:0101021010
4:10101021011
15:01010210101
9:1010101101

Abbildung 4: Lempel-Ziv

Am obigen Beispiel erkennt man die Häufigkeit der Sequenzen sowie den zugehörigen Parson's Code. Zwei Methoden wurden ausgewählt um die Anwendbarkeit des Algorithmus zu testen.

2.5.1 Längste Sequenzen

Hier werden die jeweils längsten Sequenzen als repräsentativ angesehen, die mindestens zweimal vorkommen.

2.5.2 Häufigste Sequenzen

Hier werden die Sequenzen gespeichert, die am häufigsten vorkommen. Um nicht repräsentative Sequenzen auszuklammern wurde eine Mindestlänge von 4 Zeichen definiert.

2.6 Note Sequence

Ähnlich dem Parson's Code wird die Notenfolge innerhalb eines Stücks gespeichert. Allerdings wird bei diesem Algorithmus nicht nur die Veränderung, sondern die textuelle Repräsentation der einzelnen Noten abgelegt.

Folgendes Beispiel zeigt eine extrahierte Notensequenz:

G F F A B C# D E F E D D D D C Bb

Abbildung 5: Note Sequence

2.7 Statistische Daten

Es werden statistische Einzelwerte über das Stück in einem Featurevektor gespeichert. Diese Werte sind:

- **Time Signature Denominator**. Nenner der Taktangabe.
- **Time Signature Numerator**. Zähler der Taktangabe.
- **Highest Pitch**. Höchste gemessene Tonhöhe.
- **Lowest Pitch**. Niedrigste gemessene Tonhöhe.
- **Key Quality (minor/major)**. Tongeschlecht (Dur/Moll).
- **Key Signature**. Vorzeichnung des Musikstücks.
- **Longest Rhythm Value**. Längste Notenlänge.
- **Shortest Rhythm Value**. Kürzeste Notenlänge.
- **Tempo**.
- **Volume**.
- **Changes/same**. Anzahl der gleichbleibenden Elemente des Parson's Code.
- **Changes/up**. Anzahl der Veränderungen nach oben im Parson's Code.
- **Changes/down**. Anzahl der Veränderungen nach unten im Parson's Code.
- **avg pitch**. Durchschnittliche Tonhöhe.
- **avg rhythm**. Durchschnittliche Notenlänge.
- **highest frequency**. Höchsten Tonfrequenz.
- **lowest frequency**. Niedrigste Tonfrequenz.
- **avg frequency**. Durchschnittliche Tonfrequenz.
- **score size**. Größe des MIDI- Notensatzes.
- **rests**. Anzahl der Pausen.
- **flats**. Anzahl der Halbtonschritt- Erniedrigungen.
- **sharps**. Anzahl der Halbtonschritt- Erhöhungen.
- **pitch variance**. Varianz der Tonhöhen.
- **pitch skewness**. Schiefe der Tonhöhen.
- **pitch kurtosis**. Wölbung der Tonhöhen.

Die Extraktion der Werte für höchste, niedrigste und durchschnittliche Frequenz wurde zwar implementiert aber in der finaler Version ausgepart. Das liegt an der direkten Korrelation zur Tonhöhe. Die Frequenz lässt sich eindeutig aus der Tonhöhe berechnen, wenn auch die Definition der Frequenz für einzelne Töne historisch variabel ist. Innerhalb der MIDI-Daten gibt es allerdings zu jeder MIDI-Tonhöhe eine entsprechende unveränderbare Frequenz.

Das System bietet außerdem die Möglichkeit zur Ausgabe dieses Featurevektors im SomLib Format.

3 Vergleichsalgorithmen

Dieses Kapitel behandelt die implementierten Vergleichsalgorithmen im Detail.

3.1 Dice's Coefficient

Der Dice Koeffizient vergleicht zwei Strings indem diese in zwei Mengen von Bi-grams X und Y unterteilt werden. Wird auf diese Menge die Formel

$$s = \frac{2|X \cap Y|}{|X| + |Y|} \tag{1}$$

angewandt, erhält man eine Ähnlichkeitsmetrik für die beiden Strings. Diese Methode ist besonders beim Parson's Code problematisch, da es hier aufgrund der eingeschränkten Zeichenmenge nur zu sehr kleinen Variationen innerhalb des Strings kommen kann.

3.2 Euklidische Distanz

Die euklidische Distanz berechnet den Abstand zwischen zwei Vektoren und ist wie folgt definiert:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2}$$

Normalisiert man diesen Wert mit Hilfe einer Maximaldistanz (gegen 0, da alle extrahierten Werte positiv sind), erhält man eine Ähnlichkeitsmetrik für numerische Vektoren.

3.3 Hamming Distance

Die Hamming-Distanz berechnet den Unterschied zweier Strings gleicher Länge, indem die Anzahl der Positionen gezählt wird an denen die Strings unterschiedlich sind. Auch hier kann eine Maximaldistanz berechnet werden und das Ergebnis damit normalisiert werden. Es stellt sich das Problem des Vergleichs von Strings unterschiedlicher Länge, welches mit einer mehrfachen Berechnung mittels eines Suchfensters und Auswahl der geringsten gefundenen Distanz gelöst wird.

3.4 Levenshtein Distance

Die Levenshtein-Distanz ist eine Generalisierung der Hamming-Distanz und kann auf Strings unterschiedlicher Länge angewandt werden. Es wird die minimale Anzahl an Operationen gezählt, die nötig ist um einen String in einen anderen zu transformieren. Nach Normalisierung kann diese Metrik ebenfalls zum Vergleich von stringbasierenden Features herangezogen werden.

4 Erweiterung des Systems

Dieses Kapitel beschreibt die Objektstrukturen des Systems und die Erweiterung um neue Extraktions- bzw. Vergleichsalgorithmen.

4.1 ACFeature

Jeder Algorithmus zur Extraktion von Features ist von der Klasse *ACFeature* abgeleitet. Identifiziert wird dieses Feature über die Eigenschaft *featureID*. Der anzuwendende Vergleichsalgorithmus wird in der Eigenschaft *compareAlgorithm* des jeweiligen Interfaces abgelegt.

Folgende Methoden müssen implementiert werden:

void extract (Score s). Extraktion der Metadaten aus einer jMusic Score und interne Ablage der Daten.

double compare (ACFeature f). Vergleich gegen andere Metadaten mit Hilfe des ausgewählten Vergleichsalgorithmus.

void print(). Ausgabe der Metadaten am Bildschirm.

String serialise(). Umwandlung der Metadaten in menschlich lesbares Format zur Ablage in einer der Bibliotheksdatei.

void deserialise(BufferedReader in). Umwandlung des Speicherformats in die interne Repräsentation der Metadaten.

String allowedCompareAlgorithms(). Liefert eine Liste der Identifier von Vergleichsalgorithmen zurück, die auf diese Metadaten angewandt werden können.

Weiters muss die Existenz des neuen Features der Hauptklasse bekannt gegeben werden (zur Inkludierung in das Interface). Die Funktion **getFeatureIds()** muss um den Identifier des neuen Features erweitert werden und die Funktion **featureFactory()** muss das zugehörige neue Objekt zurückliefern.

Sind all diese Bedingungen erfüllt, ist die neue Feature- Klasse automatisch in das Interface eingebunden.

4.2 ACComparison

Ähnlich der Klasse ACFeature müssen alle Vergleichsalgorithmen von ACComparison abgeleitet sein. Auch hier gibt es einen Identifier (**CName**), und die Art des Vergleichs wird über die Eigenschaft **CType** definiert. Je nach Typ muss die Methode **compareString()** bzw. **compareDoublevect()** implementiert werden, und die Funktion **comparisonFactory()** der Elternklasse um den neuen Algorithmus erweitert werden.

5 Performance

Aufgrund der Abhängigkeit von jMusic und den großen Datenmengen bei der Analyse einer komplexeren MIDI-Datei kann es bei der initalen Extraktion der Metadaten zu längeren Berechnungszeiten kommen. Die Vergleiche der Metadaten untereinander beschränken sich auf String- bzw. Vektorvergleiche, und nehmen Aufgrund der reduzierten Natur der Daten weit weniger Zeit in Anspruch. Bei Anwendung auf große Datenbanken kann es allerdings auch hier zu längeren Berechnungszeiten kommen.

Bei der Analyse einer Testdatenbank von 856 MIDI-Dateien wurde auf einem durchschnittlichen modernen Gerät (AMD Turion X2 2.2 GHz, 4 GB RAM, Windows Vista) eine Gesamtberechnungszeit von 16 Stunden und 27 Minuten benötigt, was einer durchschnittlichen Berechnungszeit von 1 Minuten und 9 Sekunden pro MIDI entspricht. Je komplexer und größer das Eingabe-MIDI ist, umso länger wird auch die Berechnungsdauer.

6 jMusic

Das System basiert auf der offenen Bibliothek **jMusic** von Andrew Sorensen und Andrew Brown der Queensland University of Technology. Neben der Bereitstellung von Tools

zur Analyse von MIDI- Daten und der Extraktion von Einzelwerten aus einem MIDI-File bietet diese Bibliothek auch Tools zur Darstellung, Wiedergabe und Erstellung von MIDI- Dateien. Eine ausführliche, wenn auch nicht erschöpfende Dokumentation sowie die Bibliothek selbst können unter

`http://jmusic.ci.qut.edu.au/`

gefunden werden.

7 Erkenntnisse

Erste Analysen und Tests die im Laufe der Entwicklung gemacht wurden deuten darauf hin dass unspezialisierte Vergleichsalgorithmen meist ungenaue Ergebnisse liefern. So sind die Werte eines Hamming-Distanzvergleichs aufgrund der Applikation auf unterschiedlich lange Strings weitaus schlechter als die Werte des Levenshtein-Vergleichs, der eine Spezialisierung auf Strings variabler Länge aufweist.

Ebenso lässt sich erkennen dass die mit den beiden Lempel-Ziv Algorithmen extrahierten Sequenzen beim Vergleich kurzer Lieder schlechtere Ergebnisse liefern als beim Vergleich längerer Stücke.

Insgesamt scheinen die Werte der Parson's Code Vergleiche, insbesondere in Kombination mit dem Levenshtein- Algorithmus, sowie die Werte der Statistikvergleiche die besten Ergebnisse zu liefern.

Als Testdaten wurde eine Datenbank von 856 MIDI-Dateien aus den Bereichen *Klassik*, *Jazz* und *Pop* herangezogen. Aufgrund der bestehenden Einteilung dieser Testdaten lässt sich ausserdem eine klare Tendenz zur Erkennung des jeweiligen Genres herauslesen, da die höchsten Ähnlichkeitswerte im jeweiligen gleichen Genrebereich zu finden waren.

Literatur

[1] William Birmingham, Roger Dannenberg, and Bryan Pardo. Query by humming with the vocalsearch system. *Commun. ACM*, 49(8):49–52, 2006.

[2] Roger B. Dannenberg, William P. Birmingham, Bryan Pardo, Ning Hu, Colin Meek, and George Tzanetakis. A comparative evaluation of search techniques for query-by-humming using the musart testbed. *JASIST*, 58(5):687–701, 2007.

[3] Roger B. Dannenberg and Ning Hu. Understanding search performance in query-by-humming systems. In *ISMIR 2004 5th International Conference on Music Information Retrieval*, 2004.

[4] Alexander Duda, Andreas Nürnberger, and Sebastian Stober. Towards query by singing/humming on audio databases. In *ISMIR 2007 Proceedings of the 8th International Conference on Music Information Retrieval*, 2007.

[5] Asif Ghias, Jonathan Logan, David Chamberlin, and Brian C. Smith. Query by humming – musical information retrieval in an audio database. In *ACM Multimedia 95 - Electronic Proceedings*, 1995.

[6] Akinori Ito, Sung-Phil Heo, Motoyuki Suzuki, and Shozo Makino. Comparison of features for dp- matching based query-by-humming system. In *ISMIR 2004 5th International Conference on Music Information Retrieval*, 2004.

[7] Jyh-Sing Roger Jang, Chao-Ling Hsu, and Hong-Ru Lee. Continuous hmm and its enhancement for singing/humming query retrieval. In *ISMIR 2005 Proceedings of the 6th International Conference on Music Information Retrieval*, 2005.

[8] Ming Li and Ronan Sleep. Improving melody classification by discriminant feature extraction and fusion. 2004.

[9] David Little, David Reffensperger, and Bryan Pardo. A query by humming system that learns from experience. In *ISMIR 2007 Proceedings of the 8th International Conference on Music Information Retrieval*, 2007.

[10] Matija Marolt. Audio meldoy extraction based on timbral similarity of melodic fragments. *EUROCON 2005*, 2005.

[11] José Martinez, Rob Koenen, and Fernando Pereira. Mpeg-7 the generic multimedia content description standard.

[12] José M. Martinez. Overview of mpeg-7 description tools.

[13] José M. Martinez. Mpeg-7 overview. http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm, 2004.

[14] Rodger J. McNab, Lloyd A. Smith, Ian H. Witten, and Clare L. Henderson. Tune retrieval in the multimedia library. 2000.

[15] Rodger J. McNab, Lloyd A. Smith, Ian H. Witten, Clare L. Henderson, and Sally Jo Cunningham. Towards the digital music library: tune retrieval from acoustic input. pages 11–18, 1996.

[16] Colin Meek and William P. Birmingham. The dangers of parsimony in query-by-humming applications. In *ISMIR 2003 Proceedings of the 4th International Conference on Music Information Retrieval*, 2003.

[17] Giyasettin OZCAN, Cihan ISIKHAN, and Adil ALPKOCAK. Melody extraction on midi music files. 2005.

[18] Bryan Pardo and William P. Birmingham. Query by humming: How good can it get? *Workshop on the Evaluation of Music Information Retrieval (MIR) Systems*, 2003.

[19] Lutz Prechelt and Rainer Typke. An interface for melody input. *ACM Trans. Comput.-Hum. Interact.*, 8(2):133–149, 2001.

[20] M. Anand Raju, Bharat Sundaram, and Preeti Rao. Tansen: A query-by-humming based music retrieval system. 2003.

[21] Hsuan-Huei Shih, Shrikanth S. Narayanan, and C.-C. Jay Kuo. Automatic main melody extraction from midi files with a modified lempel-ziv algorithm. 2001.

[22] Wolfgang Theimer and Andree Ross. Melody retrieval by fault-tolerant string meatching using a novel ranking order algorithm. 2007.

[23] Alexandra. L. Uitdenbogerd and Justin Zobel. Manipulation of music for melody matching.

[24] Eugene Weinstein. Query by humming: A survey. 2006.