Universität Osnabrück,
Fachbereich 03: Erziehungs- und Kulturwissenschaften

Hausarbeit im Rahmen der Ersten Staatsprüfung
für das Lehramt an Gymnasien

Konzeption und Implementation eines computergestützten Systems zur Melodiesuche

Vorgelegt von
Christian Datzko
Buersche Str. 70
49084 Osnabrück
Telefon: 0541-6853370
Fax: 0541-6853380
Email: christian@datzko.ch
Matrikel-Nr.: 852245

Tag der Abgabe: 02. Juli 2004.

Erstgutachter: Prof. Dr. phil. habil. Bernd Enders

Dieses Dokument wurde mit LaTeX 2$_\varepsilon$ und LilyPond in Zusammenarbeit mit teTeX formatiert.

Stand der Datei: 27. Juni 2004.

Inhaltsverzeichnis

1 Einleitung

1.1 Abgrenzung des Themas

Das Thema dieser Examensarbeit ist die *Konzeption und Implementation eines computergestützten Systems zur Melodiesuche*. Dies beinhaltet zum einen die theoretische Auseinandersetzung mit Melodiesuche und zum anderen die praktische Entwicklung eines Computerprogramms, mit dem man in Datenbanken nach Melodien suchen kann.

Die theoretische Auseinandersetzung mit dem Thema Melodiesuche geschieht in Kapitel 2. Im Kapitel 3 werden paradigmatische Verfahren von Melodiesuchen genauer beschrieben und analysiert. Die Umsetzung dieser Verfahren in Algorithmen wird im Kapitel 4 erläutert. Dort wird auch der Rahmen erläutert, der die verschiedenen Melodiesuchen in eine Benutzerschnittstelle integriert.

Diese Examensarbeit beschränkt sich auf einen Teilbereich der Melodiesuche. Zum ersten beschränken sich die hier vorgestellten Konzepte auf monophone Musikstücke und monophone Suchmelodien. Eine Erweiterung in die Polyphonie ist jedoch einfach, wenn man die Musik in einzelnen (wiederum monophonen) Stimmen vorliegen hat. Aus einer polyphonen Wolke von Tönen einzelne Stimmen herauszufiltern ist jedoch eine Aufgabe, die den Rahmen dieser Examensarbeit sprengen würde. Allerdings können aufgrund dieser Einschränkung harmonische Informationen, die für die Melodiesuche relevant sein könnten, nicht beachtet werden.

Diese Examensarbeit gibt keinen umfassendern Überblick über alle Konzepte und Ansätze, sondern gibt nur einen Einblick in den derzeitigen Stand der

Forschung. Trotzdem wird der Versuch unternommen, relevante Entwicklungen darzustellen und exemplarisch zu implementieren.

Der große Bereich der Tonhöhen- und Event-Erkennung, bei dem aus Audio-Dateien Noten generiert werden, wird nicht beachtet, da auch dieser den Rahmen der Examensarbeit sprengen würde.

Somit besteht der Kernbereich dieser Examensarbeit aus verschiedenen Algorithmen, die dazu geeignet sind, auf einer Datenbank mit monophonen Melodien nach Melodien bzw. Melodiefragmenten zu suchen.

Diese Examensarbeit ist integriert in das Projekt *MUSITECH – Music- and Sound-Objects in Information Technology (Multimedia Publishing, Archiving, and Teleteaching)* der Forschungsstelle für Musik- und Medientechnologie der Universität Osnabrück[1].

Dies bedeutet zum einen, dass Klassen genutzt werden, die von Musitech zur Verfügung gestellt werden (so zum Beispiel die Klasse `de.uos.fmt.musitech.data.structure.Piece`, die ein Musikstück darstellt), und zum anderen, dass die im Rahmen dieser Examensarbeit erstellten Klassen wiederum von Musitech genutzt werden können. Die Abgrenzung zwischen den Klassen der Examensarbeit und den übrigen Bereichen von Musitech ist jedoch ohne weiteres gegeben, da alle für diese Examensarbeit relevanten Klassen, und zwar nur diese in dem Paket `ch.datzko.melodyRetrieval` gesammelt sind.

Die Integration in das Projekt erforderte eine Zusammenarbeit zwischen Dr. Tillman Weyde, der als wissenschaftlicher Mitarbeiter der Universität Osnabrück das Projekt betreut, und mir. An dieser Stelle möchte ich ihm für die freundliche und fruchtbare Zusammenarbeit danken.

[1] Näheres zum Projekt *MUSITECH* siehe http://musitech.fmt.uni-osnabrueck.de/.

Das Projekt gibt einige Rahmenbedingungen vor, die hier zum besseren Verständnis erwähnt werden sollten. Erstens sind alle Programmteile im Musitech-Projekt in der Programmiersprache Java[2] geschrieben. Zweitens sind alle Kommentare innerhalb der Quelltexte auf Englisch geschrieben. Von einer Übersetzung dieser Kommentare ins Deutsche für diese Examensarbeit wurde abgesehen, da der Quelltext in dieser Examensarbeit ausführlich erläutert wird. Es wird nicht jede einzelne Zeile des Quelltextes besprochen, sondern nur die relevanten Ausschnitte. Ein komplettes Listing der Quelltexte ist in Anhang A zu finden.

1.2 Bezeichnungen

In dieser Arbeit werden verschiedene Bezeichnungen verwendet, die hier definiert werden sollen.

Definition 1.1. Eine Sequenz S von N Tonhöhen im MIDI-Standard: $S = (s_1, s_2, \ldots, s_N)$, wobei die s_i im MIDI-Standard vorliegen. Im MIDI-Standard ist die Tonhöhe wie folgt festgelegt: Eine Tonhöhe $h \in 0, 1, \ldots, 127 =: \Sigma_{\text{MIDI}}$ wird abgebildet auf die Klaviatur, wobei $h = 69$ dem Kammerton a' entspricht, und eine Veränderung von h um 1 einem Halbtonschritt entspricht.

Die folgenden Definitionen für die Alphabete Σ_7, Σ_{12}, Σ_{21} und Σ_{40} sind von Selfridge-Field[3] definiert.

Definition 1.2. Das Alphabet Σ_7 der diatonischen Tonhöhen: Viele der in abendländischer Musik gebräuchlichen Tonleitern basieren auf einem Modell diatonischer Tonhöhen, das in regelmäßigen Abständen Ganztonschritte und Halbtonschritte ordnet. Für das Alphabet Σ_7 sei hier eine Notation gewählt,

[2] Java 2 Software Development Kit, Standard Edition, Version 1.4.2_04, siehe http://java.sun.com/j2se/1.4.2/.
[3] Selfridge-Field (1998).

die der Dur-Tonleiter entspricht, sie ist jedoch verschiebbar und in jedes Tongeschlecht und in jede Tonart transponierbar. Die Zuordnung der einzelnen Töne auf ihre Zahlwerte ist in Abbildung 1 für C-Dur notiert.

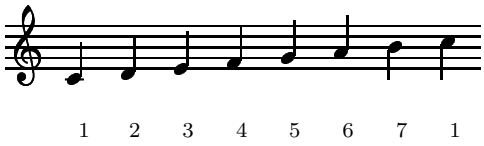


Abbildung 1: *Das Alphabet Σ_7 der diatonischen Tonhöhen*

Definition 1.3. Das Alphabet Σ_{12} der chromatischen Tonhöhen: Etwas genauer ist das Alphabet der chromatischen Tonhöhen. Hier sind nur noch Halbtonschritte enthalten (siehe Abbildung 2). Die Auflösung dieses Alphabets entspricht der des MIDI-Standards.

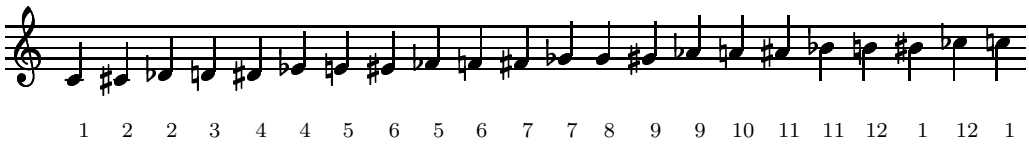


Abbildung 2: *Das Alphabet Σ_{12} der chromatischen Tonhöhen*

Definition 1.4. Das Alphabet Σ_{21}: Ein wenig genauer ist das Alphabet Σ_{21}, das im Gegensatz zu Σ_{12} eine Unterscheidung zwischen C♯ und D♭ erlaubt (siehe Abbildung 3).

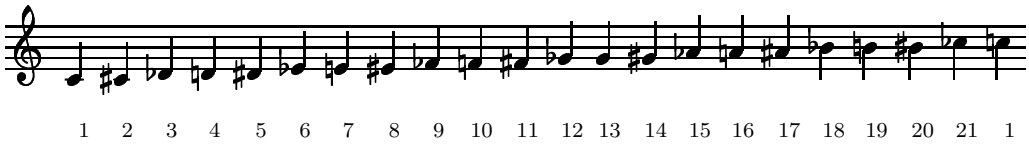


Abbildung 3: *Das Alphabet Σ_{21}*

Definition 1.5. Das Alphabet Σ_{40}: Als letztes sei hier das Alphabet Σ_{40} erwähnt, welches Differenzierungen bis hin zum Doppelkreuz und zum Doppelbe erlaubt und zusätzlich geeignet ist, durch Differenzen eindeutige Intervalle zu bilden (siehe Abbildung 4). Dieses sowie das Alphabet Σ_{21} sind jedoch nicht aus MIDI-Dateien konstruierbar und können daher für diese Examensarbeit nicht verwendet werden.

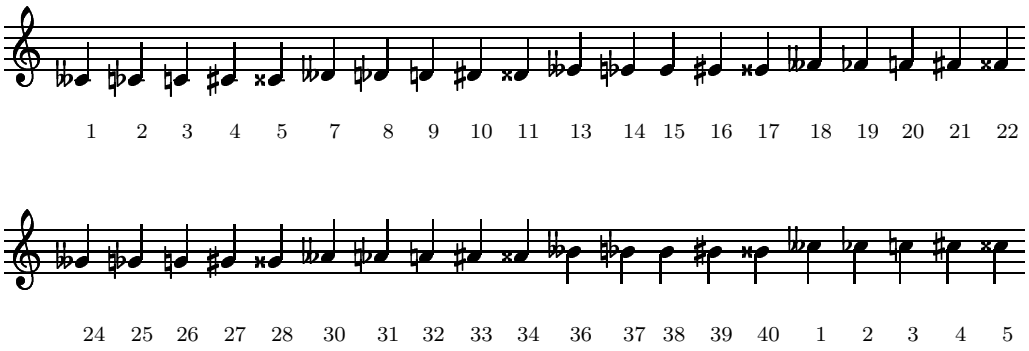


Abbildung 4: *Das Alphabet* Σ_{40}

Definition 1.6. Bezeichnungen: Der Buchstabe $S = (s_1, s_2, \ldots, s_N)$ wird in der Regel für eine zu durchsuchende Melodie verwendet, der Buchstabe $Q = (q_1, q_2, \ldots, q_M)$ in der Regel für die Such-Melodie. In S sind $|S| = N$ Töne, in Q sind $|Q| = M$ Elemente. Die Anzahl aller Melodien in der Datenbank, die durchsucht wird, soll L sein.

Definition 1.7. Teilsequenzen: $S_{i\ldots j}$ bezeichnet die Teilsequenz $(s_i, s_{i+1}, \ldots, s_{j-1}, s_j)$ von S, wobei bei $i > j$ die Teilsequenz leer ist. Spezielle Teilsequenzen sind $S_{1\ldots i}$, der *Anfang* einer Sequenz, und $S_{i\ldots N}$, das *Ende* einer Sequenz.

Definition 1.8. Intervalle: Ein Intervall $i = s_j - s_k$ ist die Anzahl der Halbtonschritte, um von der j-ten Note zur k-ten Note zu kommen, wenn die s_l aus Σ_{MIDI} sind. Es ist negativ, wenn s_k tiefer als s_j ist, positiv, wenn s_k höher als s_j ist, und 0, wenn $s_j = s_k$. Da alle $s \in \Sigma_{\text{MIDI}} = 0, \ldots, 127$, sind alle

$i \in (-127, -126, \ldots, 126, 127) =: {\Sigma_{\text{MIDI}}}^*$. Einige Intervalle sind in Tabelle 1 aufgelistet.

Tabelle 1: *Intervalle und ihre Entsprechungen in* ${\Sigma_{\text{MIDI}}}^*$

Intervallname	**Wert in** ${\Sigma_{\text{MIDI}}}^*$
Einklang	0
kleine Sekunde aufwärts	1
kleine Sekunde abwärts	-1
große Terz aufwärts	4
reine Quinte abwärts	-7
große Oktave aufwärts	12

1.3 Verwendete Software

Für diese Examensarbeit habe ich folgende Software verwendet:

- Für die Programmierung:
  - das Java 2 Runtime Environment und das Java 2 Software Development Kit[4]
  - die Entwicklungsumgebung Eclipse[5]
- Für die Ausarbeitung:
  - das Graphikprogramm dia[6]
  - das UML-Paket EclipseUML von Omondo[7]
  - die LaTeX-Distribution teTeX[8], die zusammen mit dem Paket LilyPond[9] und der Cygwin-Shell[10].

[4] Java 2 Software Development Kit, Standard Edition, Version 1.4.2_04, siehe http://java.sun.com/j2se/1.4.2/.
[5] Eclipse Platform, Version 3.0.0, siehe http://www.eclipse.org/.
[6] dia, Version 0.92, siehe http://www.lysator.liu.se/~alla/dia/.
[7] EclipseUML 1.0.0.20040524 Studio, siehe http://www.omondo.com/.
[8] teTeX, Version 2.0.2-13, http://www.tug.org/teTeX/.
[9] LilyPond, Version 2.0.1, siehe http://www.lilypond.org/.
[10] Cygwin, Version 2.05b, siehe http://www.cygwin.com/.

Die gesamte verwendete Software steht kostenlos zur Verfügung, im Falle von EclipseUML zumindest akademischen Instituten.

1.4 CD-ROM

Das Programm wurde auf verschiedensten MIDI-Dateien getestet. Zum einen wurden selbst erstellte MIDI-Dateien mit bestimmten Test-Fällen verwendet, um zu überprüfen, ob die Algorithmen korrekt arbeiten. Zudem wurden die Datensammlungen „Digital Tradition“[11] und die des Evangelischen Gesangbuches[12] verwendet. Diese beiden Datensammlungen sind auf der CD-ROM in der Tasche im rückwärtigen Deckel zu finden.

Dort befinden sich zudem eine ausführbare Version des für diese Examensarbeit entwickelten Programmes, der komplette Quellcode und die aus dem Quellcode mit dem Programm `javadoc` erstellte Dokumentation.

Zur Referenz sind alle erwähnten und digital verfügbaren Literaturquellen und die verwendete Software vorhanden. Dort ist auch eine PDF-Datei mit dieser Examensarbeit.

[11] Greenhaus (2004), im MIDI-Format unter Rickheit (2004).
[12] Deutsche Bibelgesellschaft (2004).

2 Grundlagen zur Melodiesuche

In diesem Kapitel werden sowohl allgemeine Überlegungen zur Melodiesuche dargelegt, als auch eine Kategorisierung für Melodiesuchen vorgeschlagen. Dieses Kapitel soll als theoretisches Modell zum Kapitel 3 und Kapitel 4 hinführen und die Auswahl der Verfahren begründen.

2.1 Arbeitsweise von Melodiesuchen

Eine Melodiesuche bezeichnet einen Algorithmus, der eine Melodie, die *Suchmelodie*, in einer Sammlung von Melodien, den *zu durchsuchenden Melodien*, sucht. Das Ergebnis einer Melodiesuche ist eine (eventuell auch leere) Teilmenge der Sammlung von Melodien, die auf bestimmte, dem Algorithmus eigene Art und Weise mit der Suchmelodie übereinstimmen. Eventuell ist mit jeder Melodie ein Wert für die Ähnlichkeit verknüpft, die dann als Paare bezeichnet werden.

Die Sammlung der Melodien wird in einer *Datenbank* gespeichert. Da diese eventuell sehr groß sein kann, gibt es für die Datenbank einen von der Arbeitsweise der Melodiesuche abhängigen *Index*, in dem die für die Melodiesuche relevanten Informationen jeder einzelnen Melodie gespeichert werden.

Um eine Melodiesuche zu realisieren, sind im Wesentlichen zwei Schritte nötig. Jede zu durchsuchende Melodie, die in die Datenbank aufgenommen wird, muss zuerst *indiziert* werden, das heißt, dass sie auf ihre für die konkrete Melodiesuche relevanten Eigenschaften untersucht werden muss. Dieser Index verweist dann auf die Einträge der Melodien in der Datenbank. Der zweite notwendige Schritt ist die *Suche*. Hier wird die Suchmelodie analysiert und mit den Einträgen im Index verglichen. Alle zu durchsuchenden Melodien, die die Melodiesuche aufgrund ihres Indexeintrages als Treffer wertet, werden als Ergebnis

zurückgegeben, eventuell, wenn die Melodiesuche dies ermöglicht, sortiert nach der Güte der Übereinstimmung.

Die Suchmelodie kann auf zweierlei Art und Weise eingegeben werden: Entweder wird sie als musikalische Information wie zum Beispiel als *MIDI-Datei* eingegeben, oder sie wird als *Audio-Datei* eingegeben, die jedoch (mit gewissen Unschärfen) nach MIDI konvertiert werden kann. Im Rahmen dieser Arbeit soll der Aspekt der Konvertierung von Audio-Dateien zu MIDI jedoch ausgeblendet werden, wie schon in der Einleitung (Kapitel 1.1) dargelegt. In Abbildung 5 ist der prinzipielle Aufbau jeder Melodiesuche graphisch dargestellt.

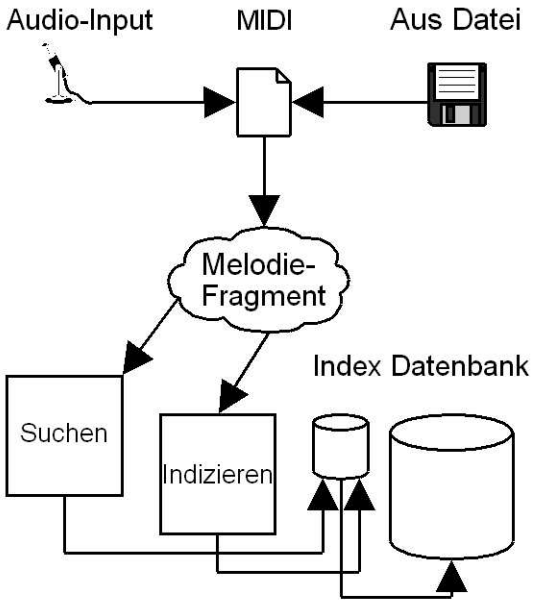


Abbildung 5: *Arbeitsweise von Melodiesuchen*

Interessant sind die individuellen Arbeitsweisen von verschiedenen Melodiesuchen. In Kapitel 2.3 wird deshalb eine Kategorisierung von derzeit möglichen Arbeitsweisen vorgeschlagen werden. Zuvor jedoch einige prinzipielle Überlegungen, welche Parameter für eine Melodiesuche relevant sein könnte.

Es sei gewarnt: Verschiedene Melodiesuchen kommen zu unterschiedlichen Ergebnissen. Zwei Melodien können nach einer bestimmten Suche ähnlich sein

und nach der anderen wiederum sehr verschieden. Eine Vergleichbarkeit von Suchergebnissen über verschiedene Suchenmethoden hinweg ist nicht möglich. Nur innerhalb einer Suchemethode ist es möglich, Ergebnisse, die unterschiedliche Suchmelodien ergeben, zu vergleichen.

Für die Frage, was für eine Melodiesuche relevant ist, gibt die Musikpsychologie wichtige Antworten. In Experimenten hat man festgestellt, dass folgende musikalische Phänomene für die Wiedererkennung von Musik relevant sind[13]:

1. Tondauer
2. Tonhöhe
3. Harmonie
4. Klangfarbe.

Dabei sind die einzelnen Punkte unabhängig voneinander variierbar. Die Reihenfolge, in der die Punkte genannt sind, zeigt ihre jeweilige Relevanz: Die Tondauer bzw. Abfolge von Tondauern trägt am meisten zur Wiedererkennung bei. Die Tonhöhe bzw. die Abfolge von Tonhöhen ist ebenfalls noch wichtig, während Harmonien und Harmoniefolgen unwichtiger sind. Die Klangfarbe ist noch relevant, aber längst nicht so stark wie die drei erstgenannten Phänomene.

Melodiesuchen verwenden einen oder mehrere dieser vier Punkte, um Musik wiederzuerkennen, meist jedoch nur den ersten und den zweiten. Da es in dieser Arbeit um monophone Melodien geht (siehe Kapitel 1.1), werden Melodiesuchen, die Harmonien analysieren, nicht aufgenommen. Die Klangfarbe wird zur Wiedererkennung in den hier behandelten Melodiesuchen nicht verwendet, zum einen, da in MIDI-Dateien nur eine begrenzte Anzahl von verschiedenen

[13] Bleeck (1996) zum Beispiel hat Experimente zu diesen musikalischen Phänomenen durchgeführt.

Klangfarben zur Verfügung steht, und zum anderen, da es von den vier Punkten der am wenigsten relevante ist. Außerdem sind die Daten, die in größeren Datensammlungen vorliegen, oftmals nicht mit Klanginformationen versehen.

2.2 Informatische Aspekte

Jenseits von musikalischen Aspekten von Melodiesuchen ist es wichtig, informatische Aspekte zu berücksichtigen. Vor allem zwei Punkte sind wichtig, an denen jeder Algorithmus zur Melodiesuche gemessen werden sollte:

1. Laufzeit der Algorithmen
2. Platzbedarf im Speicher

Der erste Punkt, *Laufzeit der Algorithmen*, wird üblicherweise in der O-Notation angegeben, die die Laufzeit relativ zur Länge der Eingabe angibt[14].

Definition 2.1. O-Notation: Seien $f, g : \mathbb{N} \to \mathbb{N}$. Dann gilt „$f$ ist höchstens von der Größenordnung g“, wenn ein n_0 und ein $c \in \mathbb{N}$ existieren, so dass gilt:

$$\forall n \geq n_0 : f(n) \leq c \cdot g(n) \tag{1}$$

Für „f ist höchstens von der Größenordnung g“ sagt man auch $f \in O(g)$ oder abkürzend $f = O(g)$.

Übliche Klassen der O-Notation sind:

$$\log n < n < n \cdot \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n! \tag{2}$$

[14] Siehe zum Beispiel Vornberger et al. (2001).

Wenn die Laufzeit von mehreren Eingaben abhängt, so kann man eine mehrdimensionale O-Notation verwenden.

Definition 2.2. Mehrdimensionale O-Notation: Sei $f, g : \mathbb{N}_1 \times \ldots \times \mathbb{N}_m \rightarrow \mathbb{N}$. Dann gilt „$f$ ist höchstens von der Größenordnung g“, wenn für alle $1 \leq i \leq n$ bei konstanten Eingaben $n_1, \ldots n_{i-1}, n_{i+1}, n_m$ es ein n_{i_0} und ein $c \in \mathbb{N}$ gibt, so dass gilt:

$$\forall n_i \geq n_{i_0} : f(n_1, \ldots, n_i, \ldots, n_m) \leq c \cdot g(n_1, \ldots, n_i, \ldots, n_m) \tag{3}$$

Abkürzend schreibt man zum Beispiel $O(L \cdot M \cdot N)$ anstelle von $O(L, M, N)$.

Hätte man also zum Beispiel zwei verschiedene Melodiesuchen, die auf einer Datenbank mit L Melodien, welche jeweils kleiner oder gleich N Töne lang sind, eine Suchmelodie von M Tönen zu suchen, so könnte man nun die Laufzeiten vergleichen. Würde die Melodiesuche A eine Laufzeit von $O(M \cdot N \cdot L)$ haben und eine Melodiesuche B eine Laufzeit von $O(M^2 \cdot N \cdot L)$, so wäre die Melodiesuche B prinzipiell langsamer als die Melodiesuche A. Dies muss im konkreten Fall, wenn man die Laufzeit in Sekunden mit einer Stoppuhr messen würde, nicht stimmen, da der konstante Faktor c nicht näher spezifiziert ist und sehr groß sein könnte. Im Allgemeinen gilt die Kategorisierung in langsamere und schnellere Algorithmen jedoch.

Als zweiten Punkt sollte man den *Platzbedarf* betrachten. Da alle Melodien in der Datenbank gespeichert sind, ist dieser Platzbedarf für alle Melodiesuchen konstant, jedoch benötigen der Index und eventuell die Suche zusätzlichen Platz. Dieser Platzbedarf kann genauso wie die Laufzeit in O-Notation geschrieben und dann verglichen werden.

2.3 Kategorisierung verschiedener Algorithmen zur Melodiesuche

Crawford, Iliopoulos und Raman haben in ihrem Aufsatz „String-Matching Techniques for Musical Similarity and Melodic Recognition“[15] eine Kategorisierung für verschiedene Algorithmen zur Melodiesuche vorgeschlagen. Diese Kategorisierung ist auf polyphone Musik zugeschnitten und bezieht sich zudem auf Kategorisierungen aus dem Bereich des String-Matchings. Da hier jedoch nur monophone Musik betrachtet werden soll, und einige mögliche Kategorien dort nicht betrachtet wurden, ist hier eine Bearbeitung wichtig.

Ich schlage vor, die verschiedenen Algorithmen in folgende Kategorien zu unterteilen:

1. Exakte Algorithmen (Kapitel 2.3.1)
2. Melodiekontur-Vergleiche (Kapitel 2.3.2)
3. Fehlertolerante Algorithmen (Kapitel 2.3.3)
4. Musikalische Analyse und Motiv-Vergleich (Kapitel 2.3.4).

Diese Kategorien stellen meines Erachtens grob die heutige Situation der Melodiesuchen dar, mit der Einschränkung, dass es sich hier um monophone MIDI-Informationen handelt.

2.3.1 Exakte Algorithmen

Am einfachsten, sowohl im Konzept als auch in der Implementation, sind exakte Algorithmen, die Ton für Ton vergleichen, ob die Suchmelodie mit der zu

[15] Crawford et al. (1998).

durchsuchenden Melodie übereinstimmt. Niclas Fogwalle hat in seinem Artikel *„The search for a notation index“*[16] die beiden am weitesten verbreiteten Varianten vorgestellt.

Als erstes kann eine direkte Übereinstimmung überprüft werden. Da dies jedoch für viele Anwendungen nicht reicht, kann es sinnvoll sein, die Übereinstimmung unabhängig von ihrer Transposition zu machen. Eine Möglichkeit ist es, ein transpositionsunabhängiges Alphabet zu verwenden, wie zum Beispiel das auf Guido Aretinus zurückzuführende Solfeggio (Do, Re, Mi, ...) oder eine Melodie nach C-Dur bzw. C-Moll zu transponieren.

Dieses Konzept haben Barlow und Morgenstern im *„A Dictionary of Musical Themes“*[17] schon 1948 verwendet. In ihrem Index sind 10.000 Themen nach C-Dur bzw. C-Moll transponiert und dann in absoluten Tonhöhen angegeben. Eine Unabhängigkeit vom Beginn der zu durchsuchenden Melodie haben sie in Ansätzen realisiert, indem sie zweite Themen zusätzlich in ihren Index aufgenommen haben.

Das Beispiel *„Der Mai ist gekommen“* (Abbildung 6) würde folgendermaßen übersetzt werden: „D E F♯ F♯ G H A F♯ A G G A F♯“. Dies würde dann nach C-Dur transponiert werden, also „C D E E F A G E G F F G E“ lauten.

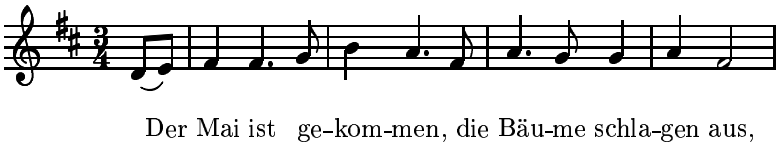


Abbildung 6: *Der Anfang von „Der Mai ist gekommen“*

Dies bringt jedoch Schwierigkeiten bei Stücken mit sich, die entweder auf Kirchentonleitern komponiert sind, oder die nicht mehr mit klassischer Harmonie-

[16] Fogwall (2004).
[17] Barlow und Morgenstern (1948), später veröffentlichten sie noch das „A Dictionary of Vocal Themes“, Barlow und Morgenstern (1950), das auch als „A Dictionary of Opera and Song Themes“ veröffentlicht wurde.

lehre analysiert werden können. Deshalb schlägt Barlow vor, im Zweifelsfall alle möglichen Transpositionen einer Melodie zu überprüfen.

Weiterhin kann es sinnvoll sein, die exakte Übereinstimmung nicht nur von der ersten Note der zu durchsuchenden Melodie an zuzulassen, sondern auch zu erlauben, dass die Suchmelodie an allen anderen Noten der Melodie anfangen kann. So können zum Beispiel auch Refrains oder Seitenthemen aus Stücken gefunden werden, obwohl immer noch eine exakte Übereinstimmung der Suchmelodie mit einem Ausschnitt der zu durchsuchenden Melodie gewährleistet ist.

2.3.2 Melodiekontur-Vergleiche

Eine Verallgemeinerung der exakten Algorithmen sind Melodiekontur-Vergleiche. Hier werden die Intervallstrukturen verglichen und nicht die Notennamen. Dabei kann unterschiedliche Genauigkeit verwendet werden.

Für den Vergleich einer Melodiekontur sind mindestens 3 Symbole nötig: Ob von einem Ton zum anderen nach oben (U) oder nach unten (D) gegangen wird, oder ob die Tonhöhe beibehalten wird (S). Ein Alphabet könnte demnach so aussehen: ${\Sigma_3}^* = (\mathrm{D}, \mathrm{S}, \mathrm{U})$, wobei D für „runter“, S für „gleich“ und U für „hoch“ steht.

Ein wenig genauer ist das Modell, in dem Tonschritte und Tonsprünge unterschieden werden. Ein Alphabet hierfür wäre ${\Sigma_5}^* = (\mathrm{D}, \mathrm{d}, \mathrm{s}, \mathrm{u}, \mathrm{U})$, bei dem die großen Buchstaben für Sprünge und die kleinen für Schritte stehen.

Eine weitere, noch genauere Möglichkeit ist ein diatonisches Modell, das die Tonschritte zwischen den beiden Noten auszählt, und notiert. Hierbei ist die Differenz der beiden Töne zur Basis 7 (siehe Kapitel 1.2) relevant. Das Alphabet

$\Sigma_7{}^* = (-\infty, \ldots, -1, 0, 1, \ldots, \infty)$ ist zwar unendlich groß, hat in der Praxis höchstens 50 Einträge, die verwendet werden.

Inhaltlich zwischen $\Sigma_5{}^*$ und $\Sigma_7{}^*$ steht das Alphabet $\Sigma_9{}^*$ von Pauws[18]. Es beschränkt sich auf diatonische Intervalle und ordnet alle Intervalle größer oder gleich einer reinen Quinte in eine Kategorie. In Tabelle 2 sind die Kategorien im Überblick gezeigt.

Tabelle 2: *Kategorien für Intervalle* $\Sigma_9{}^*$

Intervall	**Intervallgröße [Halbtonschritte]**	**Kodierung**
absteigende reine Quinte oder größer	< -6	-4
absteigende reine oder übermäßige Quarte	-5 oder -6	-3
absteigende kleine oder große Terz	-3 oder -4	-2
absteigende kleine oder große Sekunde	-1 oder -2	-1
Einklang	0	0
aufsteigende kleine oder große Sekunde	1 oder 2	1
aufsteigende kleine oder große Terz	3 oder 4	2
aufsteigende reine oder übermäßige Quarte	5 oder 6	3
aufsteigende reine Quinte oder größer	> 6	4
Startelement „·“	-	„·“

Als zusätzliches Symbol ist immer ein Startsymbol notwendig, es wird meistens mit „*“ oder „·“ notiert, wie schon in Tabelle 2 erwähnt.

Genauere Alphabete wie Σ_{12}, Σ_{21} und Σ_{40} (siehe Kapitel 1.2) bieten genauere Such-Möglichkeiten, erheben jedoch sehr hohe Qualitätsansprüche an die Melodien. Sie sind – bis auf Transpositionen – mit den exakten Algorithmen übereinstimmend.

Als Beispiel sei hier die Publikation von Parsons[19] aus dem Jahr 1975 genannt. Er benutzt in seinem Buch $\Sigma_3{}^*$ und indiziert ungefähr 10.000 Stücke. Der Al-

[18] Pauws (2002).
[19] Parsons (1975).

gorithmus CubyHum von Pauws[20] verwendet zwar auch das Alphabet $\Sigma_9{}^*$, ist aber zusätzlich fehlertolerant. Daher gehört er hier nicht zu den Beispielen.

Im Buch *„The Directory of Tunes and musical Themes“* von Denys Parsons wird folgende Notation verwendet: Der erste Ton wird mit einem „*“ markiert, und alle nachfolgenden Töne werden entweder als U („up“, hoch), S („same“, der gleiche Ton) oder D („down“, runter) indiziert, jeweils in Relation zu dem direkt vor ihm stehenden Ton. Sein Ziel war es, Suchindizes auch für Nicht-Musiker leichter zugänglich zu machen, die oftmals weder exakte Tonfolgen aufschreiben noch diese transponieren können.

Das oben verwendete Beispiel (Abbildung 6 auf Seite 21) würde also wie folgt übersetzt: „*UUSUUDDUDSUD“.

Das System Parsons' ist in Kapitel 3.1 genauer beschrieben, die Implementation davon ist in Kapitel 4.3 beschrieben.

2.3.3 Fehlertolerante Algorithmen

Fehlertolerante Algorithmen bauen auf Melodiekontur-Vergleichen auf, bieten jedoch zusätzlich zur Abstraktion von den absoluten Notenwerten auf die Melodiekontur die Möglichkeit, dass gewisse Fehler im Vergleich erlaubt sind. Dadurch erhält man ein Maß der Ähnlichkeit der Melodien, die dann zu mehr oder weniger wahrscheinlichen Übereinstimmungen führen.

Übliche Fehler, die erlaubt werden, sind die Folgenden:

1. Notenlöschung
2. Noteneinfügung

[20] Pauws (2002).

3. Notenersetzung

4. Intervalländerung.

Solche Modelle sind erst seit der Verbreitung des Computers effizient durchführbar.

Das Modell, das hinter dieser Methode steht, ist das der *„Edit Distance“* bzw. *„Levenshtein Distance“*[21]. In diesem Algorithmus sind als mögliche Änderungen Buchstabenlöschungen, -einfügungen und -ersätzungen erlaubt. Dies entspricht der Notenlöschung, Noteneinfügung und Notenersetzung. In der Musik sind zusätzlich Rückungen möglich, weshalb als viertes die Intervalländerung hinzugefügt wurde.

Man kann fehlertolerante Algorithmen nach dem *Levenshtein Distance*-Modell auf allen bei den Melodiekontur-Vergleichen erwähnten Alphabeten implementieren. Als Beispiel sei hier der Algorithmus CubyHum von Pauws[22] genannt, der auf dem Alphabet $\Sigma_9{}^*$ alle oben genannten Fehler zulässt. CubyHum rechnet zusätzlich zu den Fehlern, die durch das *Levenshtein Distance*-Modell auftreten, Fehler hinzu, die durch rhythmische Unterschiede entstehen. Das Kapitel 3.2.2 beschreibt das Verfahren von CubyHum genau.

2.3.4 Musikalische Analyse und Motiv-Vergleich

Als letzter Ansatz soll in dieser Examensarbeit eine Methode entwickelt werden, die musikalische Eigenarten stärker berücksichtigt. Bei dieser Methode wird die Melodie in einzelne Motive segmentiert, die dann als kleinste, thematisch relevante Bausteine der Melodie untereinander verglichen werden. Je häufiger ein

[21] Diese Methode wurde nach Vladimir Levenshtein benannt, einem russischen Wissenschaftler, der diesen Algorithmus für String Matching 1965 entwickelt hat. Siehe Gilleland (2004) und Stephen (1994).
[22] Pauws (2002).

solcher Baustein auftritt, desto wichtiger ist er für die Melodie selber. Handelt es sich zusätzlich um einen in der gesamten Datenbank selten vorkommenden Baustein, ist es unwahrscheinlich, dass er nur durch Zufall zum Beispiel sowohl in der Suchmelodie als auch in der zu durchsuchenden Melodie vorkommt.

Um ein Stück zu indizieren, sind folgende Schritte nötig:

1. Segmentierung der Melodie
2. Analyse der einzelnen Segmente
3. Indizierung der Segmente in einem Suchindex.

Die Segmentierung kann nach Rhythmus, Intervallen, Lautstärke und Klangfarbe erfolgen, wobei der Rhythmus Segmente am deutlichsten voneinander trennt[23]. In Abbildung 7 ist eine mögliche Segmentierung des Anfangs von „Der Mai ist gekommen“ aufgeschrieben.

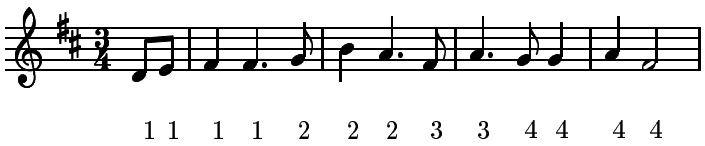


Abbildung 7: *Mögliche Segmentierung von „Der Mai ist gekommen“*

Für die Analyse der Segmente ist kein besonders elaborierter Algorithmus notwendig, da dies nur eine Vorsortierung für die spätere Grobsuche bieten soll. Hier bietet es sich an, die Melodiekontur der einzelnen Segmente oder die rhythmische Kontur zu speichern.

Die Indizierung geschieht anhand dieser Analyse in einem einfachen Index, der effizient nach dem Ergebnis der Analyse durchsucht werden kann.

[23] Weyde (2002).

Wenn man nun in der Datenbank nach einem Stück suchen möchte, ist folgender Ablauf notwendig:

1. Segmentierung und Analyse der Melodie, die genauso wie bei der Indizierung einer Melodie abläuft
2. Grobsuche auf dem Index nach gleichen und ähnlichen Segmenten
3. Gewichtung der gefundenen Ergebnisse nach der Häufigkeit der Segmente
4. Feinsuche auf den Ergebnissen nach tatsächlicher Übereinstimmung der Segmente.

Wenn man eine Melodie mit derselben Methode segmentiert und analysiert hat, wie die Melodien, die in der Datenbank gespeichert sind, kann man die einzelnen Motive, die in Form der Segmente vorliegen, mit denen der Melodien in der Datenbank vergleichen.

Dies geschieht zuerst in der Grobsuche. In ihr werden die Konturen der einzelnen Segmente der Suchmelodie mit den Konturen der einzelnen Segmente im Index verglichen. Je näher die Konturen beieinaner liegen, desto wahrscheinlicher ist es, dass die damit zusammenhängenden Motive einander ähnlich sind.

Zusätzlich kann man eine Gewichtung einführen, durch welche oft auftretende Motive (wie zum Beispiel Schlussfloskeln oder Figuren) geringer und seltener auftretende Motive (die eventuell für die Melodie typisch sind) stärker gewichtet werden.

Als letzten Schritt kann man eine Feinsuche einführen, welche die einzelnen Motive selbst und nicht nur ihre Konturen vergleicht. Diese Feinsuche kann anhand einer der schon beschriebenen Melodiesuchen vorgenommen werden. So schließt man Fehler aus, die bei der Abstraktion der Segmente auf ihre Konturen passieren können und erhält zusätzlich eine Verfeinerung der Ergebnisse.

Alternativ zur hier vorgestellten Methode könnten auch nur die Themen der zu durchsuchenden Melodie herausgefunden werden. Einen solchen Ansatz hat Smith[24] für Fugen verwendet. Der Nachteil dieser Methode ist, dass hier keine musikalisch relevanten Parameter für die Abgrenzung möglicher Themen herangezogen werden außer der Tatsache, dass sich Themen durch Wiederholung auszeichnen. Zudem wird hier ein exaktes Pattern-Matching verwendet, was mögliche Variationen eines Themas ausschließt.

2.4 Zusammenfassung

Melodiesuchen sind Algorithmen, die in einer Menge von *zu durchsuchenden Melodien* nach Melodien suchen, die einer *Suchmelodie* möglichst ähneln.

Mögliche Ansätze zur Melodiesuche sind in vier Kategorien zu unterteilen, die *exakten Algorithmen*, die *Melodiekontur-Vergleiche*, *fehlertolerante Algorithmen* und *musikalische Analysen und Motiv-Vergleiche*.

Algorithmen dieser Kategorien können trotz ihrer unterschiedlichen Ansätze nach verschiedenen Kriterien verglichen und bewertet werden.

[24] Smith und Medina (2001).

3 Verfahren zur Melodiesuche

In diesem Abschnitt sollen einige paradigmatische Verfahren zur Melodiesuche genauer betrachtet werden. Dabei soll aus den in Kapitel 2.3 aufgestellten Kategorien je ein typisches Verfahren angeführt und seine Funktionsweise erklärt werden. Ausgenommen werden die exakten Algorithmen, die trivial sind.

Die Analyse der Verfahren erfolgt in drei Schritten. Als erstes wird die Indizierung der Melodien beschrieben, als zweites wird die Suche nach Melodien im Index beschrieben, und als drittes werden Qualitäts- und Effizienzbetrachtungen durchgeführt.

Zur *Indizierung* wird erklärt, welche Schritte durchgeführt werden, um eine Melodie in die Datenbank und den Index zur Datenbank aufzunehmen.

Zur *Suche* wird erklärt, was getan werden muss, um eine Melodie in der Datenbank zu suchen. Außerdem wird dargelegt, wie zwei Melodien miteinander verglichen werden.

Bei den *Qualitäts- und Effizienzbetrachtungen* sind verschiedene Fragen zu stellen: Wie gut sind die Ergebnisse der Suche? Wie schnell werden diese Suchergebnisse gefunden? Wie viel Platz wird benötigt, um die Suche durchzuführen? Die Grundlagen hierzu, die auch eine Vergleichbarkeit erlauben, sind in Kapitel 2.2 gelegt worden.

3.1 Melodiekontur-Vergleich: Denys Parsons

Wenn in historischen Betrachtungen zur Melodiesuche Beispiele erster systematischer Kataloge von Melodien genannt werden, so wird der von Denys Parsons verwendete Melodiekonturvergleich meist mit angeführt[25].

[25] So zum Beispiel in Fogwall (2004).

3.1.1 Indizierung

Das Buch *„The Directory of Tunes and musical Themes“*[26] nutzt hierbei lediglich die Information aus, ob der jeweils nächste Ton nach oben (U = „up“) geht, der gleiche (S = „same“) bleibt oder nach unten geht (D = „down“). Dies entspricht dem Alphabet $\Sigma_3{}^*$. Der erste Ton wird mit einem „*“ markiert, da er nicht in Relation zu einem vorhergehenden steht. Dieser Algorithmus wird für diese Examensarbeit adaptiert.

Das schon in Kapitel 2.3.2 verwendete Beispiel in Abbildung 6 auf Seite 21 (hier noch einmal zum Vergleich in Abbildung 8 angeführt) würde wie folgt übersetzt: „*UUSUUDDUDSUD“.

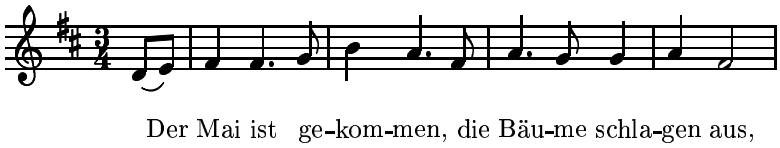


Abbildung 8: *Der Anfang von „Der Mai ist gekommen“*

Diese Struktur lässt sich gut in Form eines ternären Indexbaumes darstellen, wie er exemplarisch für die Rekursionstiefe 3 in Abbildung 9 dargestellt ist. Demnach würde das Beispiel „Der Mai ist gekommen“ dem Blatt „*UU“ ganz rechts zugeordnet werden.

Je größer die Rekursionstiefe ist, desto genauer wird auch die Suche. Jedoch wird exponentiell mehr Platz für die Datenstruktur verbraucht: Die im Kapitel 4.3 vorgestellte Implementation dieses Algorithmus benutzt eine Rekursionstiefe von 10, was also Platz für $\sum_{i=0}^{\text{Rekursionstiefe}-1} 3^i = 29524$ Knoten erfordert. Wäre die Rekursionstiefe 11, so würden schon 88573 Knoten benötigt werden, bei 12 gar 265720. Aus diesem Grund wird die Rekursionstiefe vorher fest eingestellt.

[26] Parsons (1975).

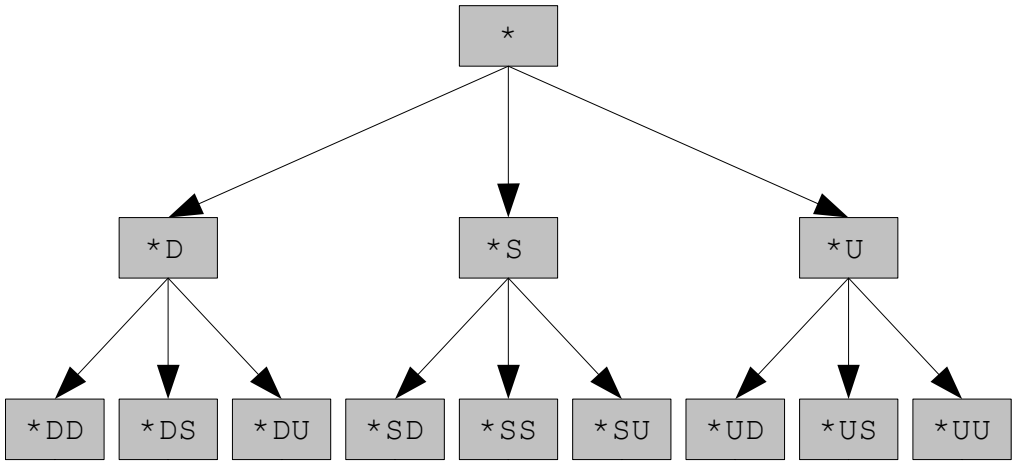


Abbildung 9: *Beispiel für einen ternären Indexbaum*

Das Beispiel aus Abbildung 8 mit der Kontur „*UUSUUDDUDSUD“ würde also an dem Knoten „*UUSUUDDUD“ eingeordnet werden. Alle Melodien, deren Melodiekontur der ersten 10 Töne gleich ist, würden auch zu diesem Knoten eingeordnet werden. Dies hat allerdings den Nachteil, dass vom 11. Ton an alle Informationen, die von der Melodie zur Verfügung stehen, ignoriert werden.

Ist eine Melodie nicht lang genug (das heißt, sie hat weniger Noten als die Rekursionstiefe groß ist), so wird der Suchstring beim Indizieren mit „S“ verlängert, bis sie lang genug ist. Dies ist jedoch bei dieser Algorithmisierung des Verfahrens so.

3.1.2 Suche

Die Suche generiert den Index für die Suchmelodie mit derselben Methode, die für alle zu durchsuchenden Melodien angewendet wurde. Auch hier brauchen nur die ersten Töne betrachtet werden, bis die Rekursionstiefe erreicht ist. Allerdings werden Melodien, die nicht lang genug sind, nicht künstlich verlängert.

Anhand dieses Indexes wird nun durch den Baum gegangen, an jedem Knoten wird entsprechend dem Suchstring entschieden, welcher Unterbaum genom-

men wird. Sind entweder die Rekursionstiefe oder das Ende der Melodie erreicht, werden alle Melodien, die in diesem Knoten bzw. in allen Knoten dieses Subbaumes gespeichert sind, als Ergebnisse zurückgegeben. Das sind alle Melodien in der Datenbank außer denjenigen, deren Melodiekonturen der ersten Töne nicht der der Suchmelodie gleichen.

3.1.3 Qualitäts- und Effizienzbetrachtungen

Auf den ersten Blick ist die oben beschriebene Melodiesuche sehr effizient. Die Laufzeit der Indizierung ist konstant ($O(1)$) bei einer Rekursionstiefe von r, da höchstens r Schritte benötigt werden, um ein Stück in den Index einzuordnen. Diese Anzahl ist nicht abhängig von der Länge der Melodie $|S| = N$.

Auch die Laufzeit der Suche ist konstant ($O(1)$), da höchstens soviele Schritte benötigt werden, wie die Rekursionstiefe ist. Dies ist eine konstante Anzahl von Schritten, um die Ergebnisse im Suchbaum zu finden. Die Anzahl der Schritte ist nicht abhängig von der Länge der Eingabe $|Q| = M$.

Der Platzbedarf der Melodiesuche hält sich ebenfalls in Grenzen, neben dem konstanten Platzbedarf für den Suchbaum (der jedoch nicht unbedingt gering sein muss, wie in Kapitel 3.1.1 gezeigt wurde) ist für jedes Stück nur ein Link nötig, der Platzbedarf des Indexes ist also $O(L)$.

Die Suche benötigt noch weniger Platz, nämlich nur einen konstanten Speicherplatz für den Suchstring, also $O(1)$.

Jedoch gibt es bei dieser Melodiesuche einige Probleme:

- Suchmelodien, für deren Länge gilt: $|Q| = M < r$, müssen künstlich verlängert werden, um in den Suchbaum eingefügt zu werden. So könnte eine

zu durchsuchende Melodie aus drei Tönen, deren Index „*SSD“ ist, von einer Suchmelodie, deren Index „*SSDSSS“ ist, gefunden werden.

- Sind Fehler in Suchmelodien oder den zu durchsuchenden Melodien, die die Melodiekontur verändern, so zum Beispiel eingefügte Tonrepetitionen oder Durchgangsnoten, dann werden Melodien nicht gefunden, die eigentlich gefunden werden sollten.

- Bei großen Datenbanken werden zu viele Suchergebnisse zurückgegeben, die verschieden sein können. Dies ist der Fall, wenn die Anzahl der verschiedenen Suchergebnisse nahe bei $3^{\text{Rekursionstiefe}}$ liegt oder gar darüber, im konkreten Fall werden höchstens $3^{10} = 19683$ verschiedene Melodiekonturen unterschieden.

3.2 Fehlertoleranter Algorithmus: CubyHum

Steffen Pauws beschreibt in seinem Aufsatz *„CubyHum: A Fully Operational Query by Humming System“*[27] ein komplettes System zur Melodiesuche. Die Schritte, die er beschreibt, sind Folgende:

1. Tonhöhen-Erkennung
2. Ereignis-Erkennung
3. Melodie-Repräsentation
4. Melodie-Vergleich.

Der komplette Aufbau von CubyHum ist in Abbildung 10 dargestellt[28].

[27] Pauws (2002).
[28] Die Graphik stammt aus der Publikation von Steffen Pauws Pauws (2002).

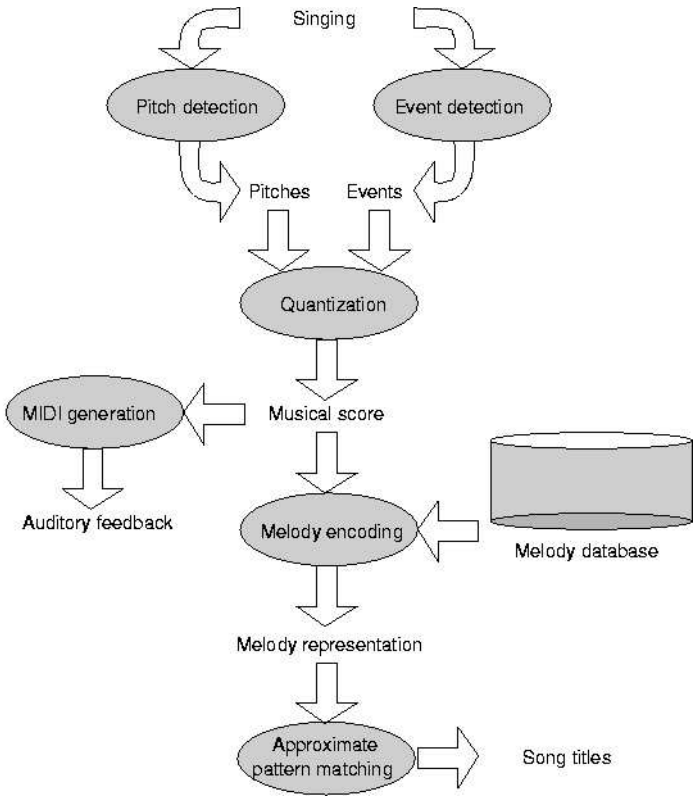


Abbildung 10: *Aufbau von CubyHum*

Die beiden ersten Schritte, Tonhöhen-Erkennung und Ereignis-Erkennung, beziehen sich auf die Analyse von Audio-Dateien und werden hier nicht weiter berücksichtigt, da hier von MIDI-Dateien ausgegangen wird, die sowohl die Tonhöhe als auch den Zeitpunkt des Auftretens der Noten beinhalten. Für die Indizierung ist die Melodie-Repräsentation relevant, für die Suche der Melodie-Vergleich.

3.2.1 Indizierung

Die Melodie-Repräsentation beschreibt eine Methode, eine vorliegende Melodie auf die für sie relevanten Intervall-Gruppen zu abstrahieren. Eine Abstraktion des Rhythmus ist an dieser Stelle nicht vorgesehen, sie wird erst später während des Melodie-Vergleichs vorgenommen.

Eine Sequenz S von N Tonhöhen ($S = (s_1, s_2, \ldots, s_N)$) wird in die Sequenz der Tonhöhenänderungen $S' = (\cdot, (s_2 - s_1), (s_3 - s_2), \ldots, (s_N - s_{N-1}))$ transformiert, wobei das erste Element „$\cdot$“ lediglich als Startelement dient.

Zusätzlich wird die Sequenz der (chromatischen) Tonhöhenänderungen auf Intervallgruppen reduziert. Hierbei gehören der Einklang in eine Gruppe, die kleine und große Sekunde in eine Gruppe, die kleine und große Terz in eine Gruppe, die reine und übermäßige Quarte in eine Gruppe und alle größeren Intervalle in eine Gruppe. Zudem wird unterschieden, ob die Intervalle nach oben oder nach unten gerichtet sind. Damit ergeben sich Gruppen wie in Tabelle 3, die mit ${\Sigma_9}^*$ bezeichnet werden. Die Funktion, die diese Abbildung erfüllt, wird mit $MR : {\Sigma_{\text{MIDI}}}^* \rightarrow {\Sigma_9}^*$ bezeichnet.

Tabelle 3: *Kategorien für Intervalle* ${\Sigma_9}^*$

Intervall	**Intervallgröße [Halbtonschritte]**	**Kodierung**
absteigende reine Quinte oder größer	< -6	-4
absteigende reine oder übermäßige Quarte	-5 oder -6	-3
absteigende kleine oder große Terz	-3 oder -4	-2
absteigende kleine oder große Sekunde	-1 oder -2	-1
Einklang	0	0
aufsteigende kleine oder große Sekunde	1 oder 2	1
aufsteigende kleine oder große Terz	3 oder 4	2
aufsteigende reine oder übermäßige Quarte	5 oder 6	3
aufsteigende reine Quinte oder größer	> 6	4
Startelement „$\cdot$“	-	„$\cdot$“

Der musikalische Hintergrund für die Konstruktion dieser Kategorien ist, dass der empfundene Unterschied zwischen einer großen und kleinen Terz wesentlich kleiner ist als der Unterschied zwischen einer großen Terz und einer Quarte, ähnlich zwischen einer großen und kleinen Sekunde im Vergleich zum Unterschied zwischen einer großen Sekunde und kleinen Terz. Intervalle, die größer als ein Tritonus sind, sind viel schwerer zu hören und kommen viel seltener in

Melodien vor, so dass man sie gut in einer gemeinsame Gruppe zusammenfassen kann.

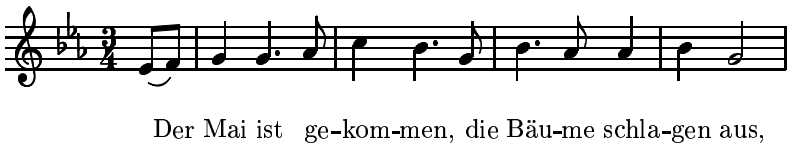


Abbildung 11: *Der Anfang von „Der Mai ist gekommen“*

Diese Kategorisierung ist am Beispiel von *„Der Mai ist gekommen“* (siehe Abbildung 11) in Gleichung 4 einmal durchgeführt.

$$\begin{aligned} S &= (63, 65, 67, 67, 68, 72, 70, 67, 70, 68, 68, 70, 67) \in \Sigma_{\text{MIDI}} \\ \Rightarrow S' &= (\cdot, 2, 2, 0, 1, 4, -2, -3, 3, -2, 0, 2, -3) \in {\Sigma_{\text{MIDI}}}^* \\ \Rightarrow MR(S') &= (\cdot, 1, 1, 0, 1, 2, -1, -2, 2, -1, 0, 1, -2) \in {\Sigma_9}^* \end{aligned} \tag{4}$$

Hierbei werden die absoluten Tonhöhen S (hier dargestellt in Σ_{MIDI}) zuerst in Intervalle (dargestellt in ${\Sigma_{\text{MIDI}}}^*$) umgerechnet und dann in die gröberen Kategorien von ${\Sigma_9}^*$ übersetzt.

3.2.2 Suche

Das Kernstück der Suche ist eine 2-dimensionale $M \times N$-Matrix (mit $M = |Q|$ und $N = |S|$). In ihr werden die Fehler aufsummiert, die auftreten, wenn beim Melodie-Vergleich von der i-ten zur $i + 1$-ten Note übergegangen wird.

Ein Schritt nach unten in der Matrix bedeutet einen Schritt vorwärts in der zu durchsuchenden Melodie und ein Schritt nach rechts bedeutet einen Schritt vorwärts in der Suchmelodie. Die erste Zeile und die erste Spalte werden mit Anfangsbedingungen initialisiert.

Dabei können die folgenden Fehler kompensiert werden:

1. Die Suchmelodie ist eine *Teilsequenz* der zu durchsuchenden Melodie, jedoch nicht ihr Anfang
2. *Minimale Tonhöhendifferenzen*, wie sie zum Beispiel beim Singen auftreten
3. spontane *Rückungen* von einer Tonart in eine andere
4. *Tonersetzungen*
5. *Noten-Einfügungen*, insbesondere Durchgänge und Verzierungsnoten
6. *Noten-Löschungen*
7. *Intervall-Einfügungen*
8. *Intervall-Löschungen*
9. *Tonlängen-Fehler*.

Die Kompensierung der einzelnen Fehler geschieht wie folgt:

Wenn die Suchmelodie Q eine *Teilsequenz* der zu durchsuchenden Melodie S ist, die nicht gleich am Anfang von S zu finden ist, sondern erst später, so wird dem dadurch Rechnung getragen, dass eine Melodie von jedem Punkt in der ersten Spalte starten kann.

Minimale Tonhöhendifferenzen werden zum einen dadurch kompensiert, dass das Alphabet $\Sigma_9{}^*$ nur Ganztonschritte unterscheidet und Intervalle größer oder gleich einer Quinte zusammengefasst werden. Zum anderen kann zum Beispiel eine Überschreitung der Grenze zwischen einer großen Sekunde und einer kleinen Terz mit zwei Rückungen hintereinander verglichen werden, ein Fehler, der durch die Fehlertoleranz des Algorithmus kompensiert wird. Im Beispiel aus Abbildung 12 sind bei „*“ zwei beispielhafte Tonhöhendifferenzen, zuerst eine,

die vom Alphabet $\Sigma_9{}^*$ aufgefangen wird, und dann eine, die als zwei Rückungen hintereinander interpretiert werden kann.

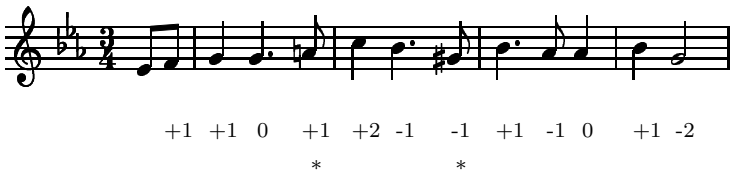


Abbildung 12: *„Der Mai ist gekommen“ – Tonhöhendifferenzen*

Eine *Rückung* ist nichts anderes, als dass ein Intervall durch ein anderes vertauscht wird, ohne dass dies sofort wieder rückgängig gemacht wird, das heißt, dass das Intervall $q_i \neq s_j$ ist. Die Stärke des Fehlers wird durch die absolute Differenz der beiden Intervalle $|q_i - s_j|$ festgestellt, normiert durch die Größe des Alphabets $\Sigma_9{}^*$, $|\Sigma_9{}^*| = 9 =: \sigma$, und gewichtet durch eine Konstante C. Der Fehler, der durch eine Rückung eingeführt wird, ist in Gleichung 5 beschrieben.

$$D_{i,j} = D_{i-1,j-1} + \frac{C}{\sigma} \cdot |q_i - s_j| \tag{5}$$

Die Konstante C ist in diesem Fall ein Maß, das angibt, wie stark eine Intervallvertauschung gewichtet wird. Steffen Pauws nimmt für diese Konstante $C = 1$ an, so dass ein Fehler durch Intervallvertauschung maximal so viel wiegt wie jeder andere Fehler. Ich halte diese Konstante für zu gering, und ersetze sie deshalb durch 2.0. Beim Wert 1 kann es bei kürzeren Suchmelodien oft vorkommen, dass Intervalleinfügungen oder Noteneinfügungen als solche nicht erkannt werden. Wird der Wert jedoch höher als 2.0 angesetzt, kann eine tatsächliche Modulation durch eine Intervalllöschung und eine Intervalleinfügung ersetzt werden.

Eine *Tonersetzung* entspricht zwei Rückungen in gegensätzlicher Richtung hintereinander und wird von daher von der Fehlermöglichkeit der Rückungen schon erfaßt.

Schwieriger ist eine *Noten-Einfügung* zu erkennen. Es gibt zwei verschiedene Möglichkeiten, die im Beispiel in Abbildung 13 jeweils bei „*“ aufgezeigt sind: Durchgänge und Verzierungsnoten. In diesen Fällen gilt, dass die Tonhöhendifferenz zweier Noten in der Suchmelodie der Tonhöhendifferenz einer Note in der zu durchsuchenden Melodie gleicht. In diesem Beispiel wäre die Summe der 5. und 6. Tonhöhendifferenz bzw. die Summe der 10. und 11. Tonhöhendifferenz gleich der 4. bzw. der 9. Tonhöhendifferenz. Dies wird kompensiert, indem (mit einem ganzen Fehler) einfach eine Note in der Suchmelodie übersprungen wird. In Gleichung 6 ist dies beschrieben.

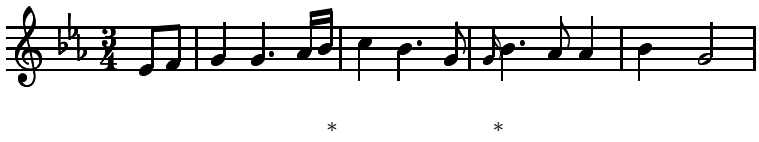


Abbildung 13: *„Der Mai ist gekommen“ – Noteneinfügungen*

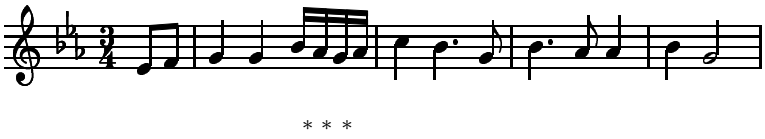
$$D_{i,j} = D_{i-2,j-1} + 1 \tag{6}$$

Bei *Noten-Löschungen* verhält es sich ähnlich wie bei Noten-Einfügungen. Eine Noten-Löschung muss man sich so vorstellen, als wenn in der zu durchsuchenden Melodie eine Note eingefügt worden ist. Dementsprechend verhält sich die Gleichung wie bei der Noten-Einfügung, nur dass rekursiv auf $D_{i-1,j-2}$ zurückgegriffen wird und nicht auf $D_{i-2,j-1}$. Gleichung 7 beschreibt dies.

$$D_{i,j} = D_{i-1,j-2} + 1 \tag{7}$$

Zusätzlich zur Möglichkeit, einzelne Noten hinzuzufügen bzw. zu löschen, muss es auch möglich sein, Teilsequenzen einer Melodie hinzuzufügen bzw. von ihr zu löschen, die länger als eine Note sind. Dies ist über die Möglichkeit von

Intervall-Einfügungen und *Intervall-Löschungen* gegeben. Ein Beispiel, in dem eine solche kleine Sequenz eingefügt ist, ist in Abbildung 14 gegeben. Um dies zu kompensieren gibt es für jeden neuen Ton einen ganzen Fehler, die dazugehörige Formel ist in Gleichung 8 dargestellt. Entsprechend gilt Gleichung 9 für Intervall-Löschungen.



* * *

Abbildung 14: *„Der Mai ist gekommen“ – Sequenz-Einfügung*

$$D_{i,j} = D_{i,j-1} + 1 \tag{8}$$

$$D_{i,j} = D_{i-1,j} + 1 \tag{9}$$

Im Vergleich zu Noteneinfügungen bzw. -löschungen sind Intervalleinfügungen bzw. -löschungen allgemeiner. Sie erlauben mehrere Möglichkeiten. Jedoch werden auch mehr Fehlerpunkte angerechnet. Während eine Noteneinfügung zum Beispiel nur einen ganzen Fehler gibt, würde die Entsprechung mit einer Intervalleinfügung und einer Rückung mehr als einen ganzen Fehler geben. Hinzu käme der unten angesprochen rhythmische Fehler.

Als letzte Fehlergruppe, die von den anderen relativ unabhängig ist, ist die Gruppe der *Tonlängen-Fehler* zu nennen. Sie werden im Vergleich zu den anderen genannten Fehlern nicht so stark gewichtet.

Im Folgenden muss eine Abstraktion von absoluten Werten zu relativen Werten geschehen. Dies wird erreicht, indem die Länge jeder Note in Relation zu der Länge ihrer vorausgehenden Note gesetzt wird, und zwar mit dem Quotienten

$\frac{L(q_i)}{L(q_{i-1})}$ bzw. $\frac{L(s_i)}{L(s_{i-1})}$. Die absolute Differenz dieser Quotienten ist ein Maß für den relativen Unterschied zweier Töne bezüglich ihrer Länge. Deshalb wird zu jeder der oben genannten Fehlerformeln noch die Fehlerformel Gleichung 10 hinzugefügt. Dieser Fehler ist zusätzlich mit einem konstanten Faktor K gewichtet, dessen ideale Größe nur empirisch festgestellt werden kann, da er die subjektive Gewichtung von rhythmischen Fehlern gegenüber den anderen genannten Fehlern darstellt. In CubyHum wird eine Größe von $K = 0.2$ vorgeschlagen, was ich für realistisch halte.

$$D_{i,j} = * + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} - \frac{L(s_j)}{L(s_{j-1})} \right| \tag{10}$$

Da immer vom geringsten Fehler ausgegangen werden muss, wird von den Ergebnissen der Gleichungen immer das Minimum genommen. Die komplette Formel, wie die Einträge der Matrix berechnet werden, ist in Gleichung 11 beschrieben.

$$D_{i,j} = \min \begin{cases} D_{i-1,j} + 1 + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} \right| & \text{(Intervall-Löschung)} \\ D_{i-2,j-1} + 1 + K \cdot \left| \frac{L(q_{i-1}) + L(q_i)}{L(q_{i-2})} - \frac{L(s_j)}{L(s_{j-1})} \right|, \\ \qquad \text{wenn } q_{i-1} + q_i = s_j, i > 2 & \text{(Noten-Löschung)} \\ D_{i-1,j-1} + \frac{C}{\sigma} \cdot |q_i - s_j| + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} - \frac{L(s_j)}{L(s_{j-1})} \right| & \text{(Rückung)} \\ D_{i-1,j-2} + 1 + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} - \frac{L(s_{j-1}) + L(s_j)}{L(s_{j-2})} \right|, \\ \qquad \text{wenn } q_i = s_{j-1} + s_j, j > 2 & \text{(Noten-Einfügung)} \\ D_{i,j-1} + 1 + K \cdot \left| \frac{L(s_j)}{L(s_{j-1})} \right| & \text{(Intervall-Einfügung)} \end{cases} \tag{11}$$

Die Anfangsbedingungen der Matrix sind in Gleichung 12 beschrieben. Sie sind in der 1. Spalte jeweils 0, um zu ermöglichen, dass eine Suchmelodie an jedem Ton der zu durchsuchenden Melodie anfangen kann, und in der 1. Zeile jeweils so berechnet, als würde ein Ton von der zu durchsuchenden Melodie am Anfang fehlen.

$$D = \begin{pmatrix} 0 & D_{1,1}+1 & \cdots & D_{i-1,1}+1 & \cdots & D_{N-1,1}+1 \\ & +K \cdot \frac{L(q_2)}{L(q_1)} & & +K \cdot \frac{L(q_i)}{L(q_{i-1})} & & +K \cdot \frac{L(q_N)}{L(q_{N-1})} \\ \vdots & ? & & \cdots & & ? \\ \vdots & \vdots & & \ddots & & \vdots \\ 0 & ? & & \cdots & & ? \end{pmatrix} \tag{12}$$

In der letzten Spalte finden sich dann die aufsummierten Fehler, die eine Suchmelodie hat, um an die zu durchsuchende Melodie angepaßt zu werden. Das Minimum der Einträge in der letzten Spalte ist die minimale Fehlersumme, die nötig ist, um die Suchmelodie so zu transformieren, dass sie auf die zu durchsuchende Melodie paßt.

Als Beispiel sei hier der Beginn der Melodie von *„Der Mai ist gekommen“* in Abbildung 15, mit einer Teilsequenz der Melodie in Abbildung 16 verglichen. Die Matrix mit den Ergebnissen ist in Gleichung 13 dargestellt.

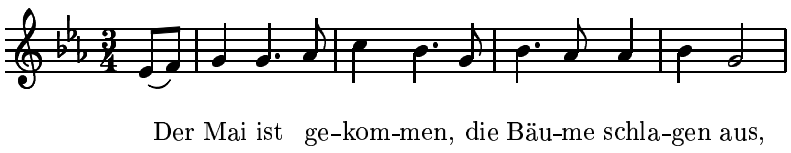


Abbildung 15: *„Der Mai ist gekommen“ – zu durchsuchende Melodie*

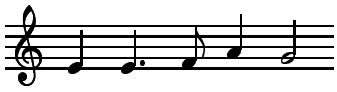


Abbildung 16: *„Der Mai ist gekommen“ – Suchmelodie*

$$D = \begin{pmatrix} 0.00 & 1.20 & 2.20 & 3.60 & 5.00 \\ 0.00 & 0.21 & 1.28 & 2.51 & 3.91 \\ \mathbf{0.00} & 0.21 & 0.54 & 1.39 & 2.73 \\ 0.00 & \mathbf{0.00} & 0.56 & 0.87 & 1.60 \\ 0.00 & 0.34 & \mathbf{0.00} & 1.00 & 1.42 \\ 0.00 & 0.32 & 0.79 & \mathbf{0.00} & 1.33 \\ 0.00 & 0.11 & 0.78 & 1.22 & \mathbf{0.10} \\ 0.00 & 0.46 & 0.44 & 1.56 & 1.17 \\ 0.00 & 0.22 & 0.80 & 0.54 & 1.94 \\ 0.00 & 0.34 & 0.44 & 1.47 & 0.88 \\ 0.00 & 0.50 & 1.19 & 1.07 & 1.68 \\ 0.00 & 0.21 & 0.63 & 1.50 & 1.49 \\ 0.00 & 0.32 & 0.88 & 1.08 & 1.61 \end{pmatrix} \tag{13}$$

Das Minimum der Matrix in der letzten Spalte, also der minimale Fehler, ist in $D_{5,7}$ zu finden. Man kann gut nachvollziehen, dass der einzige Fehler der ist, dass die letzte Note der Suchmelodie im Vergleich zur zu durchsuchenden Melodie um ein Drittel länger ist: $D_{5,7} = D_{4,6} + \frac{1}{9} \cdot |-1 - -1| + 0.2 \cdot \left|\frac{0.5}{0.25} - \frac{0.375}{0.25}\right| = 0 + \frac{1}{9} \cdot 0 + 0.2 \cdot 0.5 = 0.1$. Der Weg durch die Matrix zum minimalen Fehler ist **fett** hervorgehoben.

Die Entscheidungen der Matrix in Gleichung 13 sind in Gleichung 14 aufgezeichnet.

$$W = \begin{pmatrix}
. & \text{Int.-Lösch.} & \text{Int.-Lösch.} & \text{Int.-Lösch.} & \text{Int.-Lösch.} \\
. & \text{Rückung} & \text{Int.-Lösch.} & \text{Rückung} & \text{Int.-Lösch.} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Rückung} \\
. & \textbf{Rückung} & \text{Rückung} & \text{Rückung} & \text{Rückung} \\
. & \text{Rückung} & \textbf{Rückung} & \text{Rückung} & \text{Rückung} \\
. & \text{Rückung} & \text{Rückung} & \textbf{Rückung} & \text{Rückung} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \textbf{Rückung} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Int.-Einfg.} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Int.-Lösch.} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Rückung} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Noten-Einfg.} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Rückung} \\
. & \text{Rückung} & \text{Rückung} & \text{Rückung} & \text{Rückung}
\end{pmatrix} \tag{14}$$

Die so errechneten Werte werden normalisiert, indem sie durch die Länge der Suchmelodie geteilt werden. Zusätzlich wird dieser Wert von 1 abgezogen. So sind die Werte vergleichbar, und ein höherer Wert entspricht einer größeren Übereinstimmung.

Als Ergebnis der Suche über die Melodien in der Datenbank werden Verweise auf alle diejenigen Melodien zurückgegeben, deren Vergleich einen bestimmten Wert, den `threshhold`, übersteigen. Dieser Wert ist normalerweise auf 0.85 eingestellt, kann jedoch bei der Initialisierung frei angegeben werden.

3.2.3 Qualitäts- und Effizienzbetrachtungen

CubyHum sucht als fehlertoleranter Algorithmus sehr gut, wenn die Suchmelodie in der zu durchsuchenden Melodie mehr oder weniger vollständig enthalten ist. Mögliche Resultate können nach Ähnlichkeit sortiert werden. Man kann die Änderungen, die notwendig sind, um eine Suchmelodie an die gefundene Teilsequenz der zu durchsuchenden Melodie anzupassen, graphisch darstellen. Viel schwerer ist es jedoch, Melodien zu finden, deren thematisches Material nicht in derselben Reihenfolge wie in der Suchmelodie vorliegt.

Die Indizierung verläuft relativ schnell, es ist nur eine Laufzeit von $O(N)$ notwendig, da jeder Ton der zu indizierenden Melodie nur mit seinem Vorgänger verglichen werden und das Resultat im Alphabet $\Sigma_9{}^*$ gespeichert werden muss. Der Platzbedarf des Indexes ist jedoch vergleichsweise groß, hier ist $O(N)$ für jede Melodie notwendig, also insgesamt ein Platzbedarf von $O(N \cdot L)$.

Die Suche ist ineffizienter. Da hier eine $M \times N$-Matrix rekursiv aufgebaut werden muss, braucht es $O(M \cdot N)$ Schritte für den Vergleich mit jeder Melodie, also insgesamt $O(M \cdot N \cdot L)$ Schritte.

Der Platzbedarf ist groß. Es werden für jeden Vergleich $O(M \cdot N)$ Speicherplätze für die Matrix gebraucht. Diesen Platzbedarf kann man jedoch (mit einem geringfügig höheren Aufwand an Datenmanagement) auf $O(\min\{M, N\})$ reduzieren. Man benötigt jeweils die gerade berechnete Zeile und die beiden darüber sowie das bisherige Minimum, da auf die anderen Daten nicht mehr zugegriffen werden muss. Alternativ kann man anstelle der Zeilen auch die Spalten reduzieren.

3.3 Musikalische Analyse und Motiv-Vergleich

Als dritter Suchtypus soll hier eine selbst entwickelte Suche beschrieben werden, die nach den Prinzipien aus Kapitel 2.3.4 arbeiten soll. Sie soll die Melodien in Motive segmentieren und diese vergleichen.

Arnold Schönberg schreibt in seinem Buch „Die Grundlagen der musikalischen Komposition“ über das Motiv:

> „Das Motiv erscheint in der Regel in charakteristischer und eindrucksvoller Weise am Anfang eines Stücks. Seine Merkmale sind *Intervall* und *Rhythmus*, deren Vereinigung eine Gestalt und Kontur ergibt, die sich dem Gedächtnis einprägt und im allgemeinen eine darin enthaltene Harmonie ausdrückt. Insoweit beinahe jede Figur in einem Stück eine Verwandtschaft zu dem Grundmotiv aufweist, wird es oft als *Keim* der musikalischen Idee angesehen. Da es mindestens einige Elemente jeder folgenden musikalischen Figur enthält, kann es als „kleinstes gemeinsames Vielfaches“ angesehen werden und da es in jeder folgenden Figur enthalten ist, auch als „größter gemeinsamer Nenner“.
>
> [...] Im Laufe eines Stücks erscheint das Motiv fortwährend: das bedeutet *Wiederholung*. Bloße Wiederholung aber könnte bald zur *Eintönigkeit* führen; diese kann nur durch *Variation* überwunden werden.“[29]

Demnach ist die Suche nach Motiven in einem Stück erfolgversprechend. Selbstverständlich stellt Schönberg nur eine eingeschränkte und subjektive Sicht der Dinge vor, nicht zufällig spricht er in diesem Zusammenhang von einer *entwickelnden Variation*, die er als Fortschritt in der Musik zu seiner Zeit ansah. Da

[29] Schönberg (1979, Seite 15).

jedoch die Musikgeschichte kein einheitlicher Entwicklungsstrang ist, sondern sich immer in verschiedenste Richtungen entwickelt hat, und daher ein Fortschreiten noch lange kein Fortschritt bedingt[30], müssen die hier geforderten Eigenschaften für Motive in der Praxis empirisch nachgewiesen werden.

Die Melodien werden in Segmente unterteilt, die man als Motive ansehen kann. Diese werden dann in Kategorien einsortiert und später mit denen der Suchmelodie verglichen.

3.3.1 Indizierung

Im Folgenden werden die in Kapitel 2.3.4 genannten Schritte zur Indizierung einer Melodie im Einzelnen beschrieben:

1. Segmentierung der Melodie
2. Analyse der einzelnen Segmente
3. Indizierung der Segmente in einem Suchindex.

Segmentierung der Melodie

Um eine Reihe von Tönen in Teile zu unterteilen, muss es Vorschriften geben, nach denen diese Teile erstellt werden. Dies kann nach verschiedensten Aspekten geschehen.

Die von Schönberg genannten möglichen Kriterien zur Segmentierung von Melodien sind Intervalle und Rhythmus. In empirischen Studien hat sich herausgestellt, dass der Rhythmus stärker als Intervalle zur Segmentierung beiträgt[31]. Deshalb wird hier aufgrund des Rhytmus segmentiert.

[30] Eggebrecht (2002, Seite 650-657).
[31] Weyde (2002).

Der hier verwendete Segmentierer ist Teil des ISSM[32] und schon fertig im Projekt *MUSITECH* integriert. Im Kommentar der Implementation heißt es:

> „This class is a basic segmenter for NoteLists, setting boundaries at the relative maxima of notes' inter-onset-intervals."[33]

Das heißt, dass immer dann, wenn die Zeitdauer vom Beginn einer Note bis zum Beginn der nachfolgenden Note relativ groß ist, ein Einschnitt erfolgt. Ob eine Zeitdauer relativ groß ist, muss natürlich im Verhältnis zu den anderen Zeitdauern um sie herum gesehen werden.

So wird zum Beispiel der Anfang der Melodie von „Der Mai ist gekommen" wie in Abbildung 17 dargestellt in vier Teile geteilt.

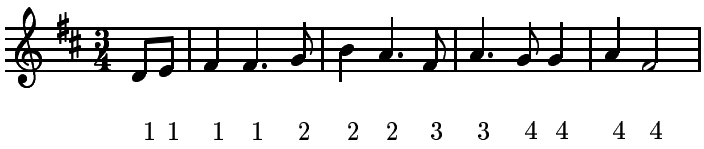


Abbildung 17: *Segmentierung von „Der Mai ist gekommen" durch den Segmentier des ISSM*

Dies ist eine zulässige Methode, um Segmente einer Melodie zu erhalten, die musikalisch als mögliche Motive gelten können. Weyde schreibt dazu in seiner Dissertation[34]:

> Größere Einsatzabstände bewirken bei ansonsten gleichen Parametern eine Gruppengrenze.

Was er dort als Gruppengrenze bezeichnet, ist hier die Grenze zwischen zwei Segmenten. Die Tonhöhe bewirkt auch eine Segmentierung, jedoch nicht so stark

[32] Weyde (2002) und Weyde (2003).
[33] Siehe `segmenter.java` im Anhang A.10.
[34] Weyde (2002, Seite 45).

wie der Rhythmus[35]. Klangfarbe und Dynamik können in diesem Zusammenhang nicht zur Segmentierung herangezogen werden. Dynamik zum Beispiel ist zwar sehr relevant für Segmentierungn, sie ist auch prinzipiell in MIDI-Informationen enthalten, aber bei manuell über Sequencer oder Notationssysteme eingegebenen Noten, gerade von Melodien, ist die Dynamik nicht notwendigerweise mit der Dynamik einer live eingespielten Melodie vergleichbar. Über die Relevanz von Klangfarbe, insbesondere im Rahmen der 128 verschiedenen MIDI-Klänge, ist noch wenig geforscht worden. Auch hier gilt die Einschränkung, dass die Daten eher unzuverlässig sind.

Eine genauere Segmentierung könnte zum Beispiel durch die Melo-Rubette, die für das Projekt Rubato[36] entwickelt wurde, bieten. Dies erfordert jedoch eine Portierung der entsprechenden Module und wäre mit einem grossen Aufwand verbunden.

Für diese Such-Methode ist eine sehr genaue Segmentierung aufgrund der Fehlertoleranz und dem statistischen Vergleich nicht notwendig.

Analyse der einzelnen Segmente

Im Zitat auf Seite 46 nennt Schönberg die Wiederholung als wesentliches Merkmal eines Motives. Gleichzeitig jedoch fordert er auch Variationen von Motiven. Deshalb wird jedes Segment abstrahiert und zusätzlich seine Melodiekontur und Rhythmuskontur gespeichert. Diese Konturen sind jedoch andere als die, die Schönberg mit „Gestalt und Kontur“ bezeichnet hat. Es ist lediglich die Information, ob die jeweils nachfolgende Note höher, gleich hoch oder tiefer und kürzer, gleich lang oder länger ist.

So kann man eine Abbildung definieren, die ähnlich wie die Melodiekontur von Denis Parsons (siehe Kapitel 3.1) funktioniert, jedoch zusätzlich den Rhyth-

[35] Weyde (2003) und Weyde (2004).
[36] Mazzola (2004).

mus berücksichtigt. In den Gleichungen 15 und 16 ist diese Funktion definiert. Hierbei ist M eine Melodie und $L(n)$ die Länge einer Note und $H(n)$ die Höhe einer Note.

$$\begin{aligned} f(M) &= f(M_{1...N-1}) + k(m_{N-1}, m_N) \quad \text{für} \quad N >= 2 \\ f(M) &= \emptyset \qquad\qquad\qquad\qquad\qquad\quad \text{für} \quad N < 2 \end{aligned} \tag{15}$$

$$k(n_1, n_2) = \left\{ \begin{array}{lll} \text{„U“} & \text{für} & H(n_1) < H(n_2) \\ \text{„E“} & \text{für} & H(n_1) = H(n_2) \\ \text{„D“} & \text{für} & H(n_1) > H(n_2) \end{array} \right\} + \left\{ \begin{array}{lll} \text{„L“} & \text{für} & L(n_1) < L(n_2) \\ \text{„E“} & \text{für} & L(n_1) = L(n_2) \\ \text{„S“} & \text{für} & L(n_1) > L(n_2) \end{array} \right\} \tag{16}$$

Die Segmente aus dem Beispiel in Abbildung 17 auf Seite 48 haben also die Konturen UEULEL, ULDL, UL und ELUEDL.

Indizierung der Segmente in einem Suchindex

Um die Konturen effizient speichern zu können und später schnell nach ihnen suchen zu können, werden sie zusätzlich binär kodiert, so dass eine Kontur in eine Integer-Variable mit 32 Bit gespeichert werden kann. Hierdurch kann diese Kontur nämlich als Hash-Code auf einer Hashtable verwendet werden. Die (umkehrbare) Abbildungsfunktion ist in den Gleichungen 17 und 18 definiert.

$$\begin{aligned} e(K) &= e(K_{1...N-2}) \cdot 2^4 + b(k_{N-1}, k_N) \\ e(\emptyset) &= 0 \end{aligned} \tag{17}$$

$$b(k_1, k_2) = \left\{ \begin{array}{lll} 0100b & \text{für} & k_1 = \text{„U“} \\ 1000b & \text{für} & k_1 = \text{„E“} \\ 1100b & \text{für} & k_1 = \text{„D“} \end{array} \right\} + \left\{ \begin{array}{lll} 01b & \text{für} & k_2 = \text{„L“} \\ 10b & \text{für} & k_2 = \text{„E“} \\ 11b & \text{für} & k_2 = \text{„S“} \end{array} \right\} \quad (18)$$

Sollte eine Kontur aus mehr als 9 Tönen, also aus mehr als 8 Einträge zu je 4 Bit bestehen, so dass sie nicht mehr in 32 Bits gespeichert werden kann, werden die letzten Einträge abgeschnitten. Dies ist aus zweierlei Gründen problemlos möglich: Zum einen wird der Fall äußerst selten vorkommen, dass die Kontur der ersten 9 Töne gleich ist und danach erst Unterschiede auftreten, und zum zweiten sind Gruppen von 8 oder mehr Tönen selten. Miller zum Beispiel stellt fest[37], dass die Fähigkeit, mehr als 7 Ereignisse kognitiv als eine Gruppe wahrzunehmen, außerhalb der Fähigkeiten des Kurzzeitgedächtnisses liegen. Somit sind Gruppen von mehr als 7 Tönen unwahrscheinlich.

Die Konturen der Segmente aus dem Beispiel in Abbildung 17 werden also wie folgt kodiert: UEULEL = $011001011001b$ = 1625, ULDL = $01011101b$ = 93, UL = 0101b = 5 und ELUEDL = $100101101101b$ = 2413.

In einem Index wird dann gespeichert, wie häufig die verschiedenen Konturen in der Melodie vorkommen. Mit einer Hashtable und der kodierten Kontur als Hash-Code kann man dies ohne großen Aufwand erreichen. In diesem Index existiert dann für jedes Stück eine Hashtable, aus der man zum einen alle verschiedenen Melodie-Konturen speichern kann, und zu denen man dann auch weiß, wie oft sie in dem Stück vorkommen. Zusätzlich wird die Segmentierung und die Anzahl der Segmente gespeichert, so dass man bei der Suche schnell auf sie zugreifen kann und dies nicht extra berechnen muss.

[37] Miller (1956).

Zusätzlich wird in einer globalen Hashtable gespeichert, wie häufig eine bestimmte Kontur in der ganzen Datenbank vorkommt. So kann man seltene von häufig vorkommenden Motiven unterscheiden und eine zusätzliche Gewichtung vornehmen. Wenn ein selten vorkommendes Motiv sowohl in der zu durchsuchenden als auch in der Suchmelodie vorkommt, ist es wahrscheinlicher, dass das zu durchsuchende Stück ein Treffer ist, als wenn ein häufig vorkommendes Motiv in einer Melodie gefunden wird.

3.3.2 Suche

Die Suche errechnet für jede Melodie einen Wert, der die Ähnlichkeit zwischen der Suchmelodie und der zu durchsuchenden Melodie darstellt. Hierfür wird für jede Segmentklasse in der zu durchsuchenden Melodie die am besten passende Segmentklasse aus der zu suchenden Melodie gesucht. Der Vergleichswert für die am besten passende Segmentklasse wird dann nach der Häufigkeit der Segmentklasse in der zu durchsuchenden Melodie gewichtet. Alle diese Werte werden aufsummiert.

Jede Segmentklasse der zu durchsuchenden Melodie wird also mit allen Segmentklassen der zu suchenden Melodie verglichen. Dabei wird jeweils ein lokaler Wert errechnet, dessen Maximum dann als höchstmögliche Übereinstimmung gewertet wird.

Grobsuche

Für die Grobsuche errechnet sich der lokale Wert l wie in Gleichung 19 dargestellt.

$$0 \leq l = \frac{1}{d(S,Q)+1} \cdot \frac{n_1}{n_2} \leq 1 \tag{19}$$

In dieser Gleichung ist $d(S, Q)$ die Levenshtein Distance zwischen den Konturen von S und Q. Sie errechnet sich ähnlich wie die Levenshtein Distance-Methode über Strings[38], gibt jedoch nur einen Fehlerpunkt für einen rhythmischen oder intervallischen Fehler. Sollte beides gleichzeitig falsch sein, gibt es ebenfalls nur einen Fehlerpunkt. Auf diese Weise wird die Ähnlichkeit der Konturen gemessen.

Ausserdem bezeichnet n_1 die Anzahl der Segmente in der aktuelle Segmentklasse der Suchmelodie und n_2 die Anzahl der Segmente in der Suchmelodie insgesamt. $\frac{n_1}{n_2}$ ist also die Gewichtung der Häufigkeit der Kontur in der Suchmelodie.

Gewichtung der Segmentklassen

Optional kann an dieser Stelle nach der Häufigkeit der Segmentklassen in der Datenbank gewichtet werden. Dies geschieht relativ einfach, indem die Segmentklassen in drei annähernd gleich große Gruppen unterteilt werden: Segmentklassen, die selten vorkommen, Segmentklassen, die durchschnittlich vorkommen und Segmentklassen, die häufig vorkommen. Die Funktion hierfür wird in Gleichung 20 beschrieben.

$$l_{\text{neu}} = l \cdot \begin{cases} 1 & \text{für: die Segmentklasse kommt selten vor} \\ 0.75 & \text{für: die Segmentklasse kommt durchschnittlich vor} \\ 0.5 & \text{für: die Segmentklasse kommt häufig vor} \end{cases} \tag{20}$$

[38] Gilleland (2004).

Feinsuche

Zuletzt wird genauer untersucht, wie sehr sich die Motive ähneln. Hierzu werden alle Segmente der zu durchsuchenden Melodie, die in der Segmentklasse der zu durchsuchenden Melodie sind, mit allen Segmenten der Suchmelodie, die in der Segmentklasse der Suchmelodie sind, verglichen.

Die Feinsuche braucht relativ viel Zeit, deshalb kann man hier unterschiedlich intensive Vergleichsmethoden auswählen, die sogenannten Motiv-Komparatoren. Deren Ergebnis liegt zwischen 0 und 1.

Am einfachsten ist die Methode, die eine 1 zurück gibt, und die Melodien gar nicht erst vergleicht. Diese Methode ist dazu da, die Grobsuche alleine zu verwenden.

Als Gegenstück dazu gibt es eine Methode, die nur eine 1 zurück gibt, wenn die beiden verglichenen Segmente identisch sind.

Als dritte Methode wird die Vergleichsmethode von CubyHum (siehe Kapitel 3.2) aufgerufen und somit ein Levenshtein Distance-ähnlicher Vergleich zwischen den beiden Segmenten errechnet. Diese Methode differenziert sehr stark und ist somit für ein möglichst exaktes Ergebnis zu bevorzugen.

Von den vielen Vergleichen der Feinsuche wird das Maximum der Übereinstimmungen ermittelt und mit dem lokalen Wert multipliziert.

Gesamtsumme

Das Maximum dieser lokalen Werte, $l_{\max}$, ist der höchste erreichbare Wert in Bezug auf die Übereinstimmung der betrachteten Segmentklasse der zu durchsuchenden Melodie mit den Segmentklassen der Suchmelodie.

Da Gleichung 21 gilt, gilt auch die Gleichung 22.

$$\sum_{\text{alle Segmentklassen}} n_{1_i} = n_2 \tag{21}$$

$$0 \leq s = \sum_{\text{alle Segmentklassen}} l_{\text{max}_i} \leq 1 \tag{22}$$

Somit kann für jede zu durchsuchende Melodie ein Wert s_i zwischen 0 und 1 errechnet werden, der die Ähnlichkeit zwischen der Suchmelodie und der jeweiligen zu durchsuchenden Melodie darstellt. Die Melodien werden nach ihren Ähnlichkeiten sortiert, und alle, deren Wert größer als ein Grenzwert, dem `threshhold`, von 0.25 ist, werden zurückgegeben. Dieser Grenzwert ist natürlich einstellbar.

3.3.3 Qualitäts- und Effizienzbetrachtungen

Die Ergebnisse der Suche sind unabhängig von der Reihenfolge der Motive. Selbst wenn nur einige Motive zufällig verteilt in der zu durchsuchenden Melodie vorkommen, kann eine Ähnlichkeit erkannt werden. Sowohl die Wichtigkeit eines Motives als auch die Unterscheidung zwischen typischen Motiven für eine Melodie und üblichen „Floskeln“ wird vorgenommen.

Ein gewichtiger Nachteil ist, dass die gewünschten Treffer, wie zum Beispiel identische Melodien, nicht notwendigerweise mit einer 1 gewertet werden. Besteht eine Melodie zum Beispiel aus üblichen „Floskeln“, wirkt sich dies auf die Gewichtung aus. Trotz identischer Motive und gleicher Anzahl der Motivklassen wird die Ähnlichkeit der Melodien abgewertet.

Die Laufzeit der Indizierung ist nur abhängig von der Länge der Melodie – jeder Ton muss einzeln betrachtet werden, und zwar bei der Segmentierung. Die Klassifizierung der Segmente und der Eintrag in der Datenbank sind auch im Rahmen von $O(N)$, da hier die Anzahl der Segmente ausschlaggebend ist, die wiederum linear von der Länge der Melodie abhängt. Die Laufzeit der Indizierung ist also $O(N)$.

Im Index wird die Segmentierung gespeichert. Da diese linear abhängig von der Länge der Melodie ist, ist der Platzbedarf für den Index $O(N \cdot L)$.

Die Laufzeit der Suche ist jedoch sehr hoch. Da letzlich alle Segmente aus der Suchmelodie mit allen Segmenten aus allen zu durchsuchenden Melodien miteinander verglichen werden, ist die Laufzeit mindestens in der Klasse $O(N \cdot M \cdot L)$. Hier kommt noch der variable Zeitaufwand der Motiv-Komparatoren bei der Feinsuche hinzu, im Falle des `CubyHumMotifComparator` ist die Laufzeit also in der Klasse $O(N^2 \cdot M^2 \cdot L)$. Die Gewichtung nach Anzahl der Motive in der Datenbank hat vom Prinzip her keinen Einfluss – obwohl natürlich jeder zusätzliche Rechenschritt den Algorithmus länger dauern lässt.

Dafür hält der Platzbedarf sich in Grenzen – hier wird nur $O(M + N)$ Platz gebraucht, um im schlimmsten Fall alle Segmente aus der Suchmelodie und der zu suchenden Melodie gleichzeitig im temporären Speicher zu haben.

Wenn nur eine begrenzte Anzahl von guten Ergebnissen gefunden werden soll, kann man einige Optimierungen durchführen. So kann man zum Beispiel nur die Grobsuche durchführen und mit den 10 besten Ergebnissen die Feinsuche durchführen lassen. Danach wird bei der jeweils nächsten Melodie die Feinsuche durchgeführt, bis das Ergebnis der Feinsuche des 10. Treffers größer ist als das Ergebnis der besten noch nicht feiner gesuchten Grobsuche. So kann man die Laufzeit wieder auf $O(N \cdot M \cdot L)$ begrenzen, da die Feinsuche dann in der Klasse $O(N \cdot M)$, also in einer Unterklasse, liegt.

4 Implementation

Die in Kapitel 3 beschriebenen Melodiesuch-Algorithmen sind in dem Paket `ch.datzko.melodyRetrieval` in der Programmiersprache Java[39] implementiert. In diesem Kapitel soll die Implementation dieser Algorithmen beschrieben und analysiert werden.

4.1 Aufbau

Die verschiedenen Klassen des Paketes sind in Abbildung 18 und in Abbildung 19 als Überblick gezeichnet. Dabei gibt es verschiedene Gruppen von Klassen. Die Bedeutung der verschiedenen Icons im Diagramm ist auf der Dokumentationsseite zum Programm Eclipse[40] erklärt. Alle diese Klassen sind als Teil dieser Examensarbeit im Anhang A aufgeführt.

4.1.1 Melodiesuche-Klassen

Die wichtigste Gruppe von Klassen enthält die abstrakte Klasse `MelodyRetrieval` und alle, die sie beerben, also:

- `ParsonsMelodyRetrieval`
- `CubyHumMelodyRetrieval`
- `MotifMelodyRetrieval`.

Der entsprechende Ausschnitt aus dem UML-Klassendiagramm für diese Klassen ist in Abbildung 18 dargestellt.

[39] Java 2 Software Development Kit, Standard Edition, Version 1.4.2_04, siehe http://java.sun.com/j2se/1.4.2/.

[40] Siehe http://help.eclipse.org/help21/topic/org.eclipse.jdt.doc.user/reference/ref-156.htm.

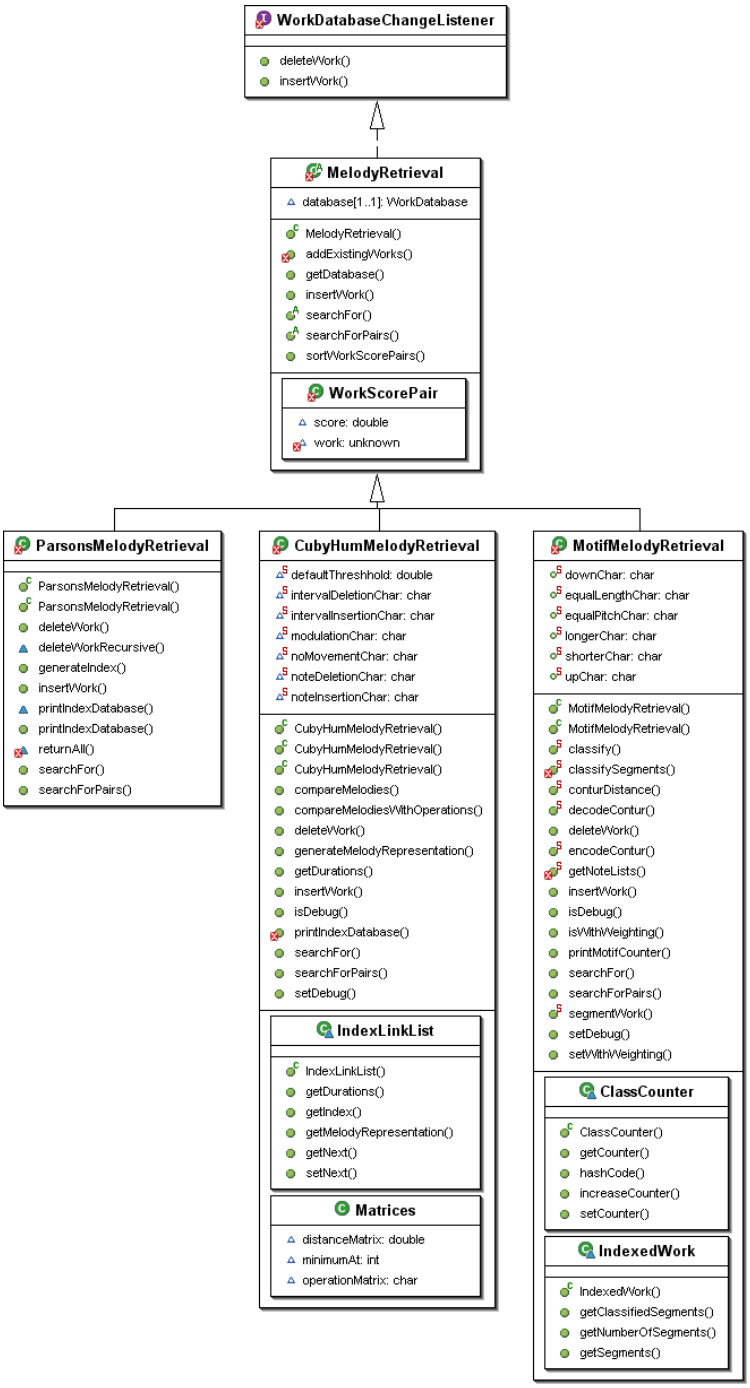


Abbildung 18: *UML Klassendiagramm (Vererbung) der* `MelodyRetrieval`-*Klassen*

Diese Klassen stellen jeweils alle notwendigen Methoden und Strukturen zur Verfügung, um eine Melodiesuche durchzuführen, jede nach einem anderen Konzept. `ParsonsMelodyRetrieval` ist nach dem Konzept von Denys Par-

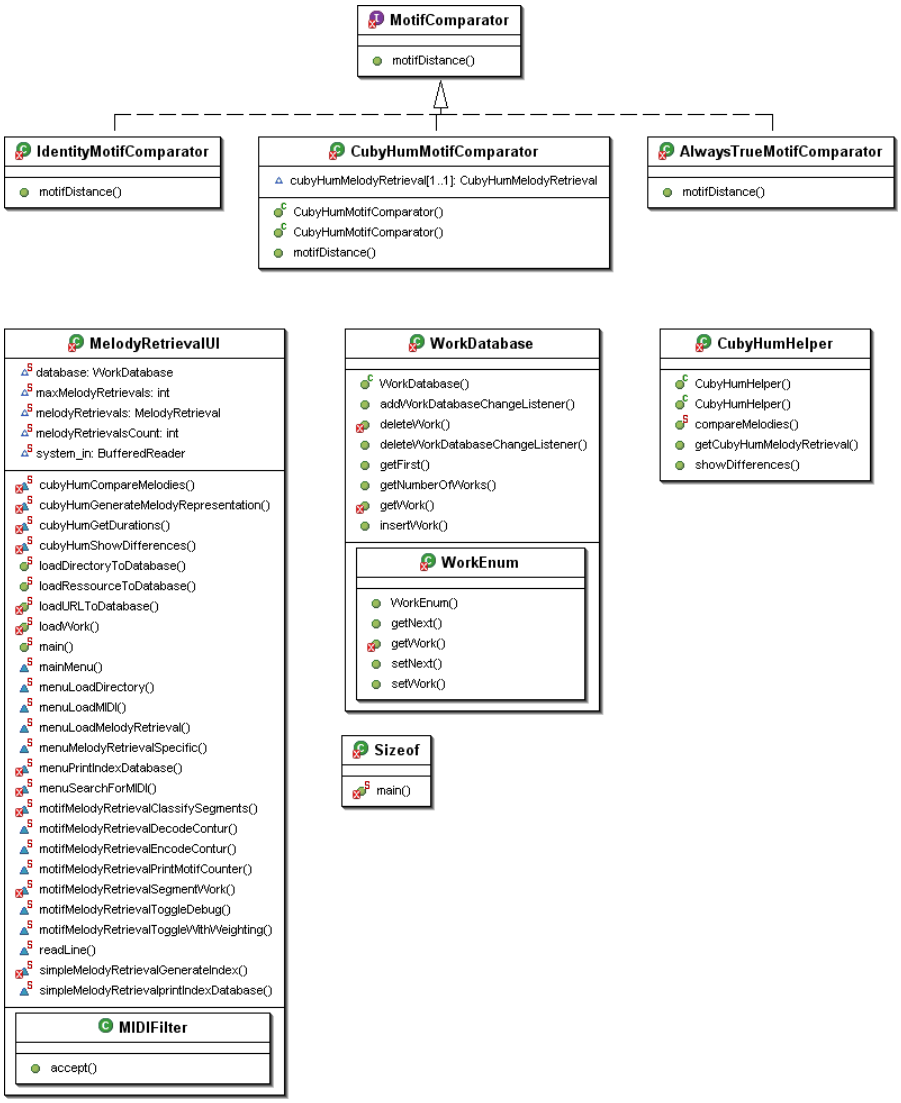


Abbildung 19: *UML Klassendiagramm (Vererbung) aller anderen Klassen*

sons (siehe Kapitel 3.1) aufgebaut, `CubyHumMelodyRetrieval` nach dem Konzept von Steffen Pauws (siehe Kapitel 3.2) und `MotifMelodyRetrieval` nach dem in dieser Examensarbeit entwickelten Konzept (siehe Kapitel 3.3).

Zu diesen Klassen gibt es kleine Klassen, die bestimmte Struktur-Aufgaben innerhalb der verschiedenen Melodiesuchen ausführen. Dazu gehören alle `MotifComparator`-Klassen wie `AlwaysTrueMotifComparator`, `CubyHumMotifComparator`, `IdentityMotifComparator` und natür-

lich das Interface `MotifComparator` selber. Die Klasse `CubyHumHelper` übernimmt einige zusätzliche optionale Funktionen der Klasse `CubyHumMelodyRetrieval`.

Das allgemeine System der Klasse `MelodyRetrieval` ist in Kapitel 4.2 beschrieben. Die drei `Melody-Retrieval`-Klassen werden in den Kapiteln 4.3, 4.4 und 4.5 genauer besprochen.

4.1.2 Datenbank

Die Klasse `PieceDatabase` stellt eine einfache Datenbank dar, in der die Stücke gespeichert werden, die von den `MelodyRetrieval`-Klassen durchsucht werden sollen. Die abstrakte Klasse `MelodyRetrieval` implementiert dazu das Interface `PieceDatabaseChangeListener`, so dass alle `MelodyRetrieval`-Klassen, die an der Datenbank angemeldet sind, bei jeder Änderung der Datenbank benachrichtig werden und ihren Index aktualisieren können. Auch hierzu gibt es zwei kleine Hilfsklassen, die für die Datenstruktur innerhalb der Datenbank zuständig sind, die inneren Klassen `PieceEnum` und `ChangeListenerEnum`.

Hervorzuheben ist bei der Datenbank, dass sie mehrere Melodiesuche-Klassen gleichzeitig erlaubt. So kann auf einer großen Datenbank mit verschiedenen Methoden gesucht werden. Auch sind die Melodiesuche-Klassen so aufgebaut, dass man sie jederzeit an eine existierende Datenbank anbinden kann, so dass alle schon in der Datenbank aufgenommenen Stücke automatisch indiziert werden.

Die Datenbank selbst basiert auf einer einfachen Link-Liste, die alle Stücke der Reihe nach speichert. Nach der Löschung eines Stücks wird einfach nur der Link auf das Stück von der Link-Liste aus gelöscht, so dass es keine Verschiebungen im Index gibt. Diese Implementation ist zwar recht einfach, und im Umgang mit

vielen Stücken nicht sehr effizient, erlaubt jedoch einen dynamischen Platzverbrauch nach Bedarf und ist vor allem leicht zu überblicken.

Innerhalb der Datenbank sind alle `MelodyRetrieval`-Klassen, die auf dieser Datenbank arbeiten, registriert. Immer wenn ein neues Stück in die Datenbank eingefügt wird, werden alle `MelodyRetrieval`-Klassen benachrichtigt, dass sie ihren Index aktualisieren können. Dies geschieht über das Interface `PieceDatabaseChangeListener`. Ebenso werden beim Löschen eines Stücks alle `MelodyRetrieval`-Klassen benachrichtigt.

Die Klasse `Sizeof` von Vladimir Roubtsov[41] dient dazu, die Größe von bestimmten Datenstrukturen herauszufinden, so zum Beispiel von einem Stück im Arbeitsspeicher.

4.1.3 Benutzeroberfläche

Es gibt eine statische Klasse, `MelodyRetrievalUI`, die eine rudimentäre Benutzeroberfläche darstellt, mit der die Basisfunktionalität der Melodiesuchen angewendet werden kann. Aufgabe dieser Klasse ist es, die Funktionalität der verschiedenen Melodiesuchen zur Verfügung zu stellen, so dass Tests und Vergleiche durchgeführt werden können.

Die genaue Struktur der Benutzeroberfläche und alle ihre Menü-Einträge sind in Kapitel 4.6 beschrieben.

4.2 MelodyRetrieval

Die abstrakte Klasse MelodyRetrieval stellt den Prototyp einer Implementation eines Systems zur Melodiesuche dar. Sie ist auf der einen Seite Schnittstelle zu ei-

[41] Roubtsov (2004).

ner `PieceDatabase`, indem sie einen `PieceDatabaseChangeListener` implementiert, und stellt auf der anderen Seite abstrakte Methoden bereit, die für die Melodiesuche verwendet werden können.

Dazu enthält sie die Variable `database` vom Typ `PieceDatabase`, die ein Link zu einer Datenbank ist.

Die Methode `public MelodyRetrieval(PieceDatabase db)` erzeugt ein neues Objekt der Klasse `MelodyRetrieval`. Für das Objekt ist eine Datenbank für die Stücke nötig, die als Parameter übergeben werden muss.

Für den Fall, dass die Datenbank, die dem Konstruktor einer MelodyRetrieval-Implementation übergeben wird, nicht leer ist, muss dafür gesorgt werden, dass die schon in der Datenbank vorhandenen Stücke bei Bedarf indiziert werden. Dazu ist die Methode `public void addExistingPieces()` vorhanden, die vom Konstruktor einer MelodyRetrieval-Implementation aufgerufen wird, nachdem die für die Indizierung notwendigen Objekt-Strukturen initialisiert sind, und die dieses erledigt.

Die abstrakte Methode `public abstract Piece[] searchFor(Piece melodyFragment)` soll in einer Implementation die tatsächliche Suche im Index der Datenbank nach möglichen Treffern durchführen. Hierzu wird der Methode ein Melodiefragment im Format eines `Piece` übergeben, nach dem gesucht werden soll. Als Ergebnis werden alle gefundenen Stücke der Datenbank in einer Liste zurückgegeben.

Die Methode `public void insertPiece(Piece piece)` führt den notwendigen Schritt durch, um ein Stück, das als Parameter `piece` übergeben wird, in die Datenbank ein- und dem Index hinzuzufügen. Da die Klasse `MelodyRetrieval` das Interface `PieceDatabaseChangeListener` implementiert, genügt es an dieser Stelle, `database.insertPiece(piece)` aufzu-

rufen. Der Sinn dieser Methode ist es, einem Index-Objekt ein Stück geben zu können, um es in die Datenbank einzufügen.

4.3 Implementation eines Melodiesuch-Systems nach Denys Parsons

Dieser Abschnitt beschreibt die Implementation des Systems, das Denys Parsons 1975 für das Buch „Directory of Tunes and Musical Themes“ erfunden hat. Die notwendigen Änderungen für die Umsetzung dieses Systems für den Computer sind schon im Kapitel 3.1 beschrieben.

4.3.1 Indizierungs-Funktion

Wie in Kapitel 3.1.1 beschrieben, verwendet der Index des Melodiesuch-Systems nach Denys Parsons einen ternären Suchbaum. Die Entscheidung, zu welchem Unterbaum bzw. Blatt gegangen wird, ist von der Melodiekontur abhängig. So ist also zunächst die Melodiekontur zu errechnen. Dies wird von der Methode `generateIndex()` übernommen, die die Noten einer Melodie der Reihe nach durchgeht und die Melodiekontur zurückgibt:

```
Note note = null;
int previousPitch = 0;
Iterator iter = sortedNotes.iterator();
if (iter.hasNext()) {
    note = (Note) iter.next();
    previousPitch = note.getPerformanceNote().getPitch();
}
while (iter.hasNext() && returnString.length() < recursionDepth + 1)
{
    note = (Note) iter.next();
    int currentPitch = note.getPerformanceNote().getPitch();
    // ignore all rests
    if (currentPitch != 0) {
        if (currentPitch > previousPitch)
            returnString = returnString + "U";
        else if (currentPitch == previousPitch)
            returnString = returnString + "S";
        else returnString = returnString + "D";
```

```
        previousPitch = currentPitch;
    }
}
```

Die Entscheidung, ob ein „U“, ein „S“ oder ein „D“ angehängt wird, ist davon abhängig, ob die derzeitige Tonhöhe höher, gleich hoch oder niedriger als die vorherige ist. Hierzu wird von der aktuellen Note die Tonhöhe ermittelt (`currentPitch`) und mit der vorherigen Tonhöhe (`previousPitch`) verglichen.

Die Eintragung im Suchbaum selber ist dann einfach. Man muss nur den Suchbaum durchlaufen, indem man immer die richtigen Unterbäume wählt, bis man bei der Rekursionstiefe angekommen ist. Dort wird das Stück eintragen:

```
String index = generateIndex(piece);
// It is assumed that the generated index is valid.
// If it is not long enough it is padded with Ss.
while (index.length() <= recursionDepth)
    index = index + "S";
DSUIndexTree t = indexTree;
for (int i = 1; i <= recursionDepth; i++) {
    if (index.charAt(i) == 'D')
        t = t.getDown();
    else if (index.charAt(i) == 'S')
        t = t.getSame();
    else if (index.charAt(i) == 'U')
        t = t.getUp();
}
DSUIndexLeaf u = (DSUIndexLeaf)t;
u.insertIndex(newIndex);
```

4.3.2 Such-Funktion

Eine ähnliche Vorgehensweise funktioniert dann ebenfalls bei der Suche. Von der Suchmelodie wird ein Index erstellt. Im Suchbaum wird für jede Tonänderung ein Knoten nach unten gegangen und danach alle an diesem Suchbaum noch hängenden Melodien als Treffer zurückgegeben:

```
for (int i = 1; (i < index.length()) && (i < recursionDepth); i++) {
```

```
    if (index.charAt(i) == 'D')
        t = t.getDown();
    else if (index.charAt(i) == 'S')
        t = t.getSame();
    else if (index.charAt(i) == 'U')
        t = t.getUp();
}
return returnAll(t);
```

Es ist hierbei zu beachten, dass beim Eintragen einer Melodie in den Suchbaum die Kontur so lange verlängert wird, bis sie lang genug ist, um die Melodie bei einem Blatt einzutragen. Suchmelodien jedoch werden nicht künstlich verlängert, damit auch Melodien gefunden werden, die genauso wie die Suchmelodie in Bezug auf ihre Kontur anfangen.

4.4 Implementation eines Melodiesuch-Systems nach Steffen Pauws

Die Implementation eines Melodiesuch-Systems nach Steffen Pauws entspricht ziemlich genau dem in seiner Publikation[42] beschriebenen Algorithmus.

4.4.1 Melodie-Repräsentierung

Die Melodie, die in Form von Tonhöhen im Alphabet Σ_{MIDI} vorliegt, soll zunächst in das Alphabet ${\Sigma_9}^*$ umgewandelt werden, das zum einen unabhängig von der absoluten Tonhöhe ist, indem es nur die jeweiligen Intervallunterschiede von einem Ton zum nächsten aufschreibt, und zum anderen nur diatonische Intervalle speichert und alle Intervalle, die größer oder gleich einer reinen Quinte sind, einer eigenen Kategorie zuordnet. Die entsprechende Abbildungsvorschrift $MR : \Sigma_{\text{MIDI}}* \rightarrow {\Sigma_9}^*$ von den absoluten Tonhöhenunterschieden nach ${\Sigma_9}^*$ ist in Tabelle 4 noch einmal notiert.

[42] Pauws (2002).

Tabelle 4: *Kategorien für Intervalle* $\Sigma_9{}^*$

Intervall	**Intervallgröße [Halbtonschritte]**	**Kodierung**
absteigende reine Quinte oder größer	< -6	-4
absteigende reine oder übermäßige Quarte	-5 oder -6	-3
absteigende kleine oder große Terz	-3 oder -4	-2
absteigende kleine oder große Sekunde	-1 oder -2	-1
Einklang	0	0
aufsteigende kleine oder große Sekunde	1 oder 2	1
aufsteigende kleine oder große Terz	3 oder 4	2
aufsteigende reine oder übermäßige Quarte	5 oder 6	3
aufsteigende reine Quinte oder größer	> 6	4
Startelement ·	-	·

Die Melodie-Repräsentation wird erstellt, indem von der ersten Note startend die Differenz `currentPitch` − `previousPitch` (jetzige Tonhöhe − vorherige Tonhöhe) errechnet und der entsprechende Wert Tonhöhen in die Liste der Intervalle `intervals[]` eingetragen wird. Eventuell auftretende Pausen werden ignoriert, indem die Tonhöhendifferenz zwischen den tatsächlich gespielten Noten davor und danach gemessen wird.

Im Folgenden der relevante Ausschnitt aus der Methode, die diese Funktion erfüllt, `public int[] generateMelodyRepresentation(Piece piece)`:

```
Iterator iter = sortedNotes.iterator();
Note note = (Note) iter.next();
int previousPitch = note.getPerformanceNote().getPitch();
for (; iter.hasNext();) {
    note = (Note) iter.next();
    int currentPitch = note.getPerformanceNote().getPitch();
    // ignore alle rests
    if (currentPitch != 0) {
        if (currentPitch - previousPitch < -6)
            intervals[intervalCounter] = -4;
        else if (currentPitch - previousPitch < -4)
            intervals[intervalCounter] = -3;
        else if (currentPitch - previousPitch < -2)
            intervals[intervalCounter] = -2;
        else if (currentPitch - previousPitch < 0)
            intervals[intervalCounter] = -1;
        else if (currentPitch - previousPitch < 1)
            intervals[intervalCounter] = 0;
        else if (currentPitch - previousPitch < 3)
```

```
            intervals[intervalCounter] = 1;
        else if (currentPitch - previousPitch < 5)
            intervals[intervalCounter] = 2;
        else if (currentPitch - previousPitch < 7)
            intervals[intervalCounter] = 3;
        else
            intervals[intervalCounter] = 4;
        previousPitch = currentPitch;
        intervalCounter++;
    }
}
```

4.4.2 Suche

Die Suche ist umfangreicher. Alle Melodien müssen mit der Suchmelodie abgeglichen werden. Die Such-Methode arbeitet die Melodien in der Datenbank Stück für Stück durch und vergleicht sie mit der Suchmelodie. Die Fehleranzahl `comparismScore` wird für jeden Vergleich normalisiert und auf ein Prozentähnliches Maß gebracht. Die Ergebnisse sind nicht genaue Prozentwerte, jedoch wird durch die Formel $1 - \frac{\text{Fehlerpunkte}}{\text{Anzahl der Noten}}$ ein annähernd guter Wert ermittelt. Da theoretische Fälle auch Werte < 0 ergeben können, wird der Minimalwert auf 0 gesetzt. Ist der so errechnete prozentähnliche Wert größer als eine bestimmte Grenze `threshhold`, so wird das Stück zusammen mit seinem errechneten Wert gespeichert und am Ende mit allen anderen gefundenen Stücken nach der Wahrscheinlichkeit sortiert[43] zurückgegeben.

Im Folgenden ist der relevante Ausschnitt aus der Methode `public PieceScorePair[] searchForPairs(Piece melodyFragment)` abgebildet:

```
IndexLinkList ll = indexLinkList;
while (ll != null) {
    double comparismScore = compareMelodies(queryMR, queryD,
            ll.getMelodyRepresentation(), ll.getDurations());
    // The next calculation somewhat generates a normalisation to
    // 0.0 .. 1.0. Since in rare cases values below 0.0 can exist
    // it is minimised with 0.0. The higher the comparismScore, the
```

[43] Zur Sortierung wird der weit verbreitete und effiziente Quicksort-Algorithmus verwendet, siehe Vornberger et al. (2001).

```
    // greater its similarity between the piece and the melodyFragment.
    comparismScore =  1 - (comparismScore / queryD.length);
    if (comparismScore < 0.0)
        comparismScore = 0.0;
    if (comparismScore > threshhold) {
        pieces[foundPieces].piece = database.getPiece(ll.getIndex());
        pieces[foundPieces].score = comparismScore;
        foundPieces++;
    }
    ll = ll.getNext();
}
```

Die in `public PieceScorePair[] searchForPairs(Piece melodyFragment)` aufgerufene Methode `public double compareMelodies(int[] q_MR, long[] q_D, int[] s_MR, long[] s_D)` ist das Herzstück der Suche. Dort werden zwei Melodien verglichen, indem die Matrix D erstellt wird. Dazu sind drei Schritte nötig.

Zuerst muss die Matrix initialisiert werden, und zwar wie in Gleichung 23 notiert.

$$D = \begin{pmatrix} 0 & D_{1,1}+1 & \cdots & D_{i-1,1}+1 & \cdots & D_{N-1,1}+1 \\ & +K \cdot \frac{L(q_2)}{L(q_1)} & & +K \cdot \frac{L(q_i)}{L(q_{i-1})} & & +K \cdot \frac{L(q_N)}{L(q_{N-1})} \\ \vdots & ? & & \cdots & & ? \\ \vdots & \vdots & & \ddots & & \vdots \\ 0 & ? & & \cdots & & ? \end{pmatrix} \tag{23}$$

Im Folgenden der relevante Ausschnitt aus `public double compareMelodies(int[] q_MR, long[] q_D, int[] s_MR, long[] s_D)` (dabei sind die Variablennamen `q_MR` und `q_D` die Melodierepräsentation und Längen der Suchmelodie und `s_MR` und `s_D` die Melodierepräsentationen der zu durchsuchenden Melodie):

```
matrices.distanceMatrix[0][0] = 0.0;
for (int i = 1; i < q_D.length; i++) {
```

```
    matrices.distanceMatrix[i][0] =
        matrices.distanceMatrix[i - 1][0] + 1 + k *
        Math.abs(q_D[i] / q_D[i-1]);
}
for (int j = 1; j < s_D.length; j++) {
    matrices.distanceMatrix[0][j] = 0.0;
}
```

Als zweiter Schritt muss die Matrix errechnet werden. Dazu muss die Funktion in Gleichung 24 für alle Einträge außer denen in der ersten Zeile und ersten Spalte nacheinander errechnet werden.

$$D_{i,j} = \min \begin{cases} D_{i-1,j} + 1 + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} \right| & \text{(Intervall-Löschung)} \\ D_{i-2,j-1} + 1 + K \cdot \left| \frac{L(q_{i-1})+L(q_i)}{L(q_{i-2})} - \frac{L(s_j)}{L(s_{j-1})} \right|, \\ \qquad \text{wenn } q_{i-1} + q_i = s_j, i > 2 & \text{(Noten-Löschung)} \\ D_{i-1,j-1} + \frac{C}{\sigma} \cdot |q_i - s_j| + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} - \frac{L(s_j)}{L(s_{j-1})} \right| & \text{(Rückung)} \\ D_{i-1,j-2} + 1 + K \cdot \left| \frac{L(q_i)}{L(q_{i-1})} - \frac{L(s_{j-1})+L(s_j)}{L(s_{j-2})} \right|, \\ \qquad \text{wenn } q_i = s_{j-1} + s_j, j > 2 & \text{(Noten-Einfügung)} \\ D_{i,j-1} + 1 + K \cdot \left| \frac{L(s_j)}{L(s_{j-1})} \right| & \text{(Intervall-Einfügung)} \end{cases} \tag{24}$$

Im Folgenden der relevante Ausschnitt aus `public double compareMelodies(int[] q_MR, long[] q_D, int[] s_MR, long[] s_D)`:

```
for (int i = 1; i < q_D.length; i++)
    for (int j = 1; j < s_D.length; j++) {
        // interval deletion:
        double interval_deletion =
            matrices.distanceMatrix[i - 1][j] + 1 + k *
            Math.abs((double)q_D[i] / (double)q_D[i-1]);
```

```
        // note deletion:
        double note_deletion = Double.POSITIVE_INFINITY;
        if (i > 2 && (q_MR[i - 1] + q_MR[i] == s_MR[j]))
            note_deletion = matrices.distanceMatrix[i - 2][j - 1] +
            1 + k * Math.abs(((double)q_D[i - 1] +
                    (double)q_D[i]) / (double)q_D[i - 2] -
                    (double)s_D[j] / (double)s_D[j - 1]);
        // modulation or no error:
        double modulation = matrices.distanceMatrix[i - 1][j - 1] +
        (c / sigma) * Math.abs(q_MR[i] - s_MR[j]) + k *
        Math.abs((double)q_D[i] / (double)q_D[i - 1] -
                (double)s_D[j] / (double)s_D[j - 1]);
        // note insertion:
        double note_insertion = Double.POSITIVE_INFINITY;
        if (j > 2 && (q_MR[i] == s_MR[j - 1] + s_MR[j]))
            note_insertion = matrices.distanceMatrix[i - 1][j - 2] +
            1 + k * Math.abs((double)q_D[0] / (double)q_D[i - 1] -
                    ((double)s_D[j - 1] + (double)s_D[j]) /
                    (double)s_D[j - 2]);
        // interval insertion:
        double interval_insertion =
            matrices.distanceMatrix[i][j - 1] + 1 + k *
            Math.abs((double)s_D[j] / (double)s_D[j - 1]);
        matrices.distanceMatrix[i][j] = Math.min(interval_deletion,
                Math.min(note_deletion,
                        Math.min(modulation,
                                Math.min(note_insertion,
                                        interval_insertion))));
    }
```

Als letztes muss das Minimum der Einträge in der letzten Spalte gefunden werden. Hier der relevante Ausschnitt aus `public double compareMelodies(int[] q_MR, long[] q_D, int[] s_MR, long[] s_D)`:

```
double minimum = Double.POSITIVE_INFINITY;
matrices.minimumAt = 0;
for (int j = 0; j < s_D.length; j++) {
    minimum = Math.min(minimum,
            matrices.distanceMatrix[q_D.length - 1][j]);
    if (minimum == matrices.distanceMatrix[q_D.length - 1][j])
        matrices.minimumAt = j;
}
```

Die Ergebnisse werden sortiert, wie oben beschrieben normalisiert und zurückgegeben.

4.5 Implementation eines Motiv-Melodiesuch-Systems

Die in dieser Examensarbeit entwickelte Motiv-Melodiesuche erfordert zuallererst eine Segmentierung der vorgegebenen Melodien. Diese wird in Kapitel 4.5.1 beschrieben. Zur Speicherung der Motivklassen im Index wird eine Kodierung der Melodie- und Rhythmus-Kontur gebraucht, die diese in einem einzigen Integer speichert. In Kapitel 4.5.2 wird die Klassifizierung der Melodiekontur besprochen und in Kapitel 4.5.3 die Kodierung davon. In Kapitel 4.5.4 wird die Suche selbst, die in Grobsuche, Gewichtung nach Häufigkeit der Motivklassen und Feinsuche unterteilt ist, beschrieben. Zuletzt werden in Kapitel 4.5.5 verschiedene Motiv-Komparatoren beschrieben, die bei der Feinsuche verwendet werden.

4.5.1 Segmentierung der Melodie

Die Segmentierung ist Teil des ISSM[44] und von Tillman Weyde implementiert worden. Sie berücksichtigt den relativen Rhythmus, um eine Melodielinie in Segmente zu unterteilen, im Prinzip wird immer nach einer langen Note ein Einschnitt gemacht.

Im Quellcode sieht das so aus:

```
public Segmentation segment( NoteList list) {
    ArrayList boundaries = new ArrayList();
    // special cases
    // cases size() 0 or 1
    if(list.size()<2)
        return segment(list,boundaries);
    // count note positions in relations to current position (starting with 4).
    Note note1 = (Note) list.get(0); // note at position -3
    Note note2 = (Note) list.get(1); // note at position -2
    if(list.size()==2) {
        if(note2.getTime() - note1.getTime() > maxDuration) {
            boundaries.add(new Integer(1));
        }
        return segment(list,boundaries);
```

[44] Weyde (2002) und Weyde (2003).

```
    }
    Note note3 = (Note) list.get(2);
    long dist1 = note2.getTime() - note1.getTime();
    long dist2 = note3.getTime() - note2.getTime();
    if(list.size()==3) {
        if(dist1 < dist2) {
            boundaries.add(new Integer(1));
        }
        else {
            boundaries.add(new Integer(2));
        }
    }
    Note note4 = (Note) list.get(3);
    long dist3 = note4.getTime() - note3.getTime();
    if(dist2>dist3 && dist2>dist1) {
        boundaries.add(new Integer(2));
    }
    for (int i = 4; i < list.size(); i++) {
        note1 = note2;
        note2 = note3;
        note3 = note4;
        note4 = (Note) list.get(i);
        dist1 = dist2;
        dist2 = dist3;
        dist3 = note4.getTime() - note3.getTime();
        if(dist2>dist3 && dist2>dist1) {
            boundaries.add(new Integer(i-1));
        }
    }
    return segment(list,boundaries);
}
```

Eine Trennung zwischen zwei Segmenten wird immer genau dann gemacht, wenn eine zeitliche Differenz des Beginns zweier Noten größer ist als die Differenz der vorhergehenden mit der ersten Note und die Differenz mit der zweiten und der nachfolgenden Note. In Abbildung 20 ist eine solche Segmentierung gegeben. Die Grenzen der Segmente werden zwischen der 4. und 5. Note, zwischen der 7. und 8. sowie zwischen der 9. und 10. Note gezogen.

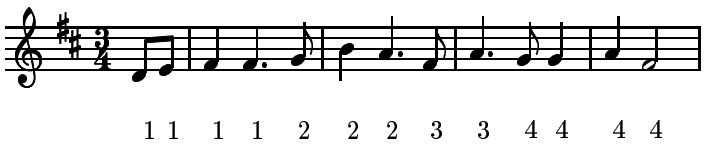


Abbildung 20: *Segmentierung von „Der Mai ist gekommen“*

Diesen Segmentierer könnte man noch verbessern, indem man gezielt nach Wiederholungen innerhalb der Melodie sucht. Schönberg selbst hat dazu eine

Liste von Merkmalen erstellt, die bei Wiederholungen von Motiven verändert werden können. Er schreibt:

> „Der *Rhythmus* wird verändert:
>
> 1. Durch Veränderung der Tondauer
> 2. Durch Tonwiederholung
> 3. Durch Wiederholung einiger rhythmischer Gestalten
> 4. Durch Verlegung von rhythmischen Gestalten auf andere Taktteile
> 5. Durch Hinzufügung von Auftakten
> 6. Durch Veränderung der Taktart – nur selten innerhalb eines Stücks zu verwenden
>
> Die *Intervalle* werden verändert:
>
> - Durch Veränderung der ursprünglichen Reihenfolge oder Richtung der Töne
> - Durch Hinzufügung oder Weglassung von Intervallen
> - Durch Ausfüllen des Intervallraums mit Nebennoten
> - Durch Reduktion mit Hilfe von Weglassung oder Kondensierung
> - Durch Wiederholung von Motiv-Merkmalen
> - Durch Verlegung von Motiv-Merkmalen auf andere Taktteile“[45]

Er gibt zu jedem seiner Punkte Beispiele und geht zusätzlich auf die Harmonie ein, die jedoch für die Melodie selbst irrelevant ist.

[45] Schönberg (1979, Seite 17).

Es bleibt festzuhalten, dass diese Liste nicht vollständig ist. Sie ist unter dem Gesichtspunkt einer Kompositionslehre von einem Komponisten einer bestimmten Schule geschrieben worden. Ob solche Variationen für die Melodiesuche berücksichtigt werden müssen, bleibt offen.

Smith zum Beispiel hat in seinem Poster „Discovering Themes by Exact Pattern Matching“[46] allein die exakte Wiederholungen von Motiven dazu verwendet, um Themen aus Musikstücken zu extrahieren. Mit diesem Konzept hatte er in begrenztem Umfang Erfolg. Da die analysierten Stücke Fugen waren, half die musikalischer Struktur sehr, Themen durch exakte Wiederholungen zu finden.

Dieses Beispiel lässt aber die Annahme gelten, dass die Bedingungen dafür, dass ein Motiv relevant genannt werden kann, sehr eng gefasst sein können, um trotzdem brauchbare Ergebnisse zu erhalten. Deshalb soll dieser Segmentierer für diese Implementation ausreichen.

4.5.2 Klassifizierung der Melodiekontur

Die Motive einer auf diese Weise segmentierten Melodie sollten dann in Gruppen geordnet werden. So können häufige Motive herausgefunden werden. Dies geschieht anhand ihrer melodischen und rhythmischen Kontur. Dazu werden die Intervalle zwischen allen Noten in das Alphabet Σ_3 übersetzt. Ähnlich wird mit dem Rhythmus verfahren: Es wird gespeichert, ob die Länge einer Note länger, gleich lang oder kürzer ist als die der jeweils vorhergehenden Note.

$$\begin{aligned} f(M) &= f(M_{1\ldots N-1}) + k(m_{N-1}, m_N) \quad \text{für} \quad N >= 2 \\ f(M) &= \emptyset \qquad\qquad\qquad\qquad\qquad\quad \text{für} \quad N < 2 \end{aligned} \tag{25}$$

[46] Smith und Medina (2001).

$$k(n_1, n_2) = \left\{ \begin{array}{lll} \text{„U“} & \text{für} & H(n_1) < H(n_2) \\ \text{„E“} & \text{für} & H(n_1) = H(n_2) \\ \text{„D“} & \text{für} & H(n_1) > H(n_2) \end{array} \right\} + \left\{ \begin{array}{lll} \text{„L“} & \text{für} & L(n_1) < L(n_2) \\ \text{„E“} & \text{für} & L(n_1) = L(n_2) \\ \text{„S“} & \text{für} & L(n_1) > L(n_2) \end{array} \right\} \tag{26}$$

Die Gleichungen 25 und 26 beschreiben dieses Abstraktion von Melodie und Rhythmus. Dabei sind $L(n)$ die Länge und $H(n)$ die Höhe einer Note sowie M das zu untersuchende Segment. Der folgende Quellcode setzt diese Gleichungen um:

```
public static int classify(NoteList seq) {
    String s = "";
    Iterator iter = seq.iterator();
    if (!iter.hasNext()) {
        return 0;
    }
    else {
        Note note = (Note) iter.next();
        int i = 0;
        while (iter.hasNext()) {
            int previousPitch = note.getPerformanceNote().getPitch();
            long previousLength = note.getPerformanceNote().getDuration();
            note = (Note) iter.next();
            if (note.getPerformanceNote().getPitch() > previousPitch)
                s = s + upChar;
            else
                if (note.getPerformanceNote().getPitch() == previousPitch)
                    s = s + equalPitchChar;
                else
                    s = s + downChar;

            if (note.getPerformanceNote().getDuration() > previousLength)
                s = s + longerChar;
            else
                if (note.getPerformanceNote().getDuration() == previousLength)
                    s = s + equalLengthChar;
                else
                    s = s + shorterChar;
        }
        return encodeContur(s);
    }
}
```

Die Tabelle 5 zeigt an, welche Buchstaben für welches Ereignis verwendet werden.

Tabelle 5: *Buchstaben für die Melodie- und Rhythmus-Kontur*

Variablen-Name	**Bedeutung**	**Buchstabe**
downChar	Intervalle nach unten	D
equalPitchChar	selbe Tonhöhe	S
upChar	Intervalle nach oben	U
shorterChar	Note kürzer als die vorige	S
equalLengthChar	Note genauso lang wie die vorige	E
longerChar	Note länger als die vorige	L

Als Ergebnis wird ein String zurückgegeben, in dem von der zweiten Note an für jede Note mit jeweils zwei Zeichen angegeben wird, was für ein Intervall zwischen den Tönen ist, und was für eine rhythmische Änderung vorhanden ist. Das vorhin verwendete Beispiel in Abbildung 20 auf Seite 72 hat folgende Konturen: UEULEL, ULDL, UL und ELUEDL.

4.5.3 Kodierung der Melodie-Rhythmus-Konturen

Um die so errechneten Konturen effizient speichern zu können und sie vor allem als Hash-Funktion verwenden zu können, müssen sie in einer Integer-Variablen gespeichert werden. Die Funktion, die die Kontur in eine Integer-Variable umrechnet, sollte umkehrbar sein. Daher wird die in Gleichung 27 und 28 beschriebene Funktion verwendet.

$$\begin{aligned} e(K) &= e(K_{1\ldots N-2}) \cdot 2^4 + b(k_{N-1}, k_N) \\ e(\emptyset) &= 0 \end{aligned} \tag{27}$$

$$b(k_1, k_2) = \left\{ \begin{array}{lll} 0100b & \text{für} & k_1 = \text{„U“} \\ 1000b & \text{für} & k_1 = \text{„E“} \\ 1100b & \text{für} & k_1 = \text{„D“} \end{array} \right\} + \left\{ \begin{array}{lll} 01b & \text{für} & k_2 = \text{„L“} \\ 10b & \text{für} & k_2 = \text{„E“} \\ 11b & \text{für} & k_2 = \text{„S“} \end{array} \right\} \tag{28}$$

Sie erfüllt die Bedingung der Umkehrbarkeit mit der einzigen Einschränkung, dass aufgrund des geringen Platzes von 32 Bit in einer Integer-Variablen nur die ersten 9 Noten (mit 8 Intervallen und Rhythmusänderungen zu je 4 Bits) gespeichert werden. Motive, die länger als 9 Noten sind und bei denen zusätzlich die Kontur der ersten 9 Noten gleich ist, sind jedoch selten und von daher macht es keinen großen Unterschied, ob diese alle in eine oder in mehrere Kategorien sortiert werden. Zusätzlich hat Miller[47] den Einwand erhoben, dass Gruppierungen von mehr als 7 Ereignissen vom Kurzzeitgedächtnis gar nicht als Einheit wahrgenommen werden und von daher gar keine Segmente im Sinne dieser Arbeit sind.

Die Kodierung der Kontur selbst stellt keinen großen Aufwand dar, es werden einfach alle Buchstaben der Reihe nach kodiert. Dies geschieht, indem beim ersten Buchstaben begonnen wird, und die Variable immer um 2 Bits nach links verschoben wird, so dass dann die Kontur vom nächsten Buchstaben hinzu addiert werden kann. Sollten unzulässige Buchstaben vorkommen, wird die Kodierung abgebrochen. Der Quellcode ist dieser:

```
public static int encodeContur(String s) {
    int i = 0;
    int j = 0;
    while ((j < s.length()) && (j < 16)) {
        i <<= 2;
        switch (s.charAt(j)) {
            case upChar :
                i += 1;
                break;
            case equalPitchChar :
                i += 2;
                break;
            case downChar :
                i += 3;
                break;
            default :
                throw new IllegalArgumentException("This string is no " +
                        "valid contur string: " + s + " at position " + j);
        }
        i <<= 2;
        j++;
        switch (s.charAt(j)) {
            case longerChar :
                i += 1;
```

[47] Miller (1956).

```
                break;
            case equalLengthChar :
                i += 2;
                break;
            case shorterChar :
                i += 3;
                break;
            default :
                throw new IllegalArgumentException("This string is no " +
                        "valid contur string: " + s + " at position " + j);
        }
        j++;
    }
    return i;
}
```

Die Dekodierung ist ein sehr ähnlicher Vorgang, nur dass hier der letzte Buchstabe zuerst herausgerechnet wird, da er als letztes hinzugefügt wurde. Da jedes Mal von der Variablen die letzten beiden Bits abgetrennt werden, und die Kodierungen nur 1, 2 und 3 enthalten, ist die Variable gleich 0, sobald keine weiteren Konturen mehr in ihr gespeichert sind. Der Quellcode sieht wie folgt aus:

```
public static String decodeContur(int i) {
    String s = "";
    while (i != 0) {
        switch (i & 3) {
            case 1 :
                s = longerChar + s;
                break;
            case 2 :
                s = equalLengthChar + s;
                break;
            case 3 :
                s = shorterChar + s;
                break;
            default :
                throw new IllegalArgumentException("This int is no " +
                        "valid contur integer: " + Integer.
                        toBinaryString(i));
        }
        i >>>= 2;
        switch (i & 3) {
            case 1 :
                s = upChar + s;
                break;
            case 2 :
                s = equalPitchChar + s;
                break;
            case 3 :
                s = downChar + s;
                break;
            default :
                throw new IllegalArgumentException("This int is no " +
                        "valid contur integer: " + Integer.
                        toBinaryString(i));
        }
```

```
        i >>>= 2;
    }
    return s;
}
```

4.5.4 Suche

Die Suchfunktion hat drei Abschnitte. Zuerst wird die Grobsuche durchgeführt, dann die Gewichtung der Motive und zuletzt die Feinsuche.

Diese drei Abschnitte werden für jede Segmentklasse der zu suchenden Melodie und für jede Segmentklasse der zu durchsuchenden Melodie und zwar für jede zu durchsuchende Melodie gemacht, also eine dreifache Verschachtelung. Hierbei ist die Variable `classCounter1` der Zähler für die aktuelle Segmentklasse des aktuell zu durchsuchenden Stücks, `classCounter2` der Zähler für die aktuelle Segmentklasse des zu suchenden Stücks und `classCounter3` der Zähler für die Anzahl der Motive der Klasse von `classCounter1` in der gesamten Datenbank. Die Funktion `hashCode()` eines `ClassCounter`-Objekts gibt an, welche Kontur die Klasse hat, und die Funktion `getCounter()` gibt an, wie häufig Motive dieser Klasse in der Melodie bzw. in der Datenbank sind. Zuletzt gibt `searchedPiece.getNumberOfSegments()` an, wie viele Segmente in der durchsuchten Melodie insgesamt vorhanden sind.

```
for (int i = 0; i < indexArrayList.size(); i++) {
    c1++;
    scores[i] = 0.0;
    if (indexArrayList.get(i) != null) {
        IndexedPiece searchedPiece = (IndexedPiece) indexArrayList.get(i);
        Iterator searchedPieceSegIter =
            searchedPiece.getClassifiedSegments().values().iterator();
        // counter for the fragments
        int c2 = 0;
        // for all segments in the piece that is searched in
        while (searchedPieceSegIter.hasNext()) {
            c2++;
            ClassCounter classCounter1 =
                (ClassCounter) searchedPieceSegIter.next();
            NoteList[] noteSeq1 =
                getNoteLists(searchedPiece.getSegments(),
                        classCounter1.hashCode());
            ClassCounter classCounter3 =
```

```
                (ClassCounter) motifCounter.get(new Integer(
                        classCounter1.hashCode()));
            if (classCounter3 == null) {
                        System.out.println("error");
            }
            Iterator melFragSegIter =
                melFragClasSeg.values().iterator();
            double tempMax = 0.0;
            // for all segments in the piece that is searched for
            while (melFragSegIter.hasNext()) {
                ClassCounter classCounter2 =
                    (ClassCounter) melFragSegIter.next();

                [...]

            }
            scores[i] += tempMax;
        }
    }
    if (debug)
        System.out.println(" score for piece #" + c1 + " = " +
                scores[i]);
}
```

Grobsuche

Bei der Grobsuche soll die Gleichung 29 umgesetzt werden, wobei S die durchsuchte Melodie ist, Q die Suchmelodie, n_1 die Anzahl der Motive in den Klassen des aktuell durchsuchten Segmentes und n_2 die Anzahl der Motive in der durchsuchten Melodie insgesamt.

$$l = \frac{1}{d(S,Q)+1} \cdot \frac{n_1}{n_2} \tag{29}$$

Die Funktion $d(S,Q)$ bezeichnet eine Form der Levenshtein Distance für Melodie-Rhythmus-Konturen. Sie ähnelt der Levenshtein Distance-Funktion für Strings, mit dem Unterschied, dass hier die beiden Buchstaben für Intervall und Rhythmusänderung als ein Zeichen betrachtet werden.

Konkret bedeutet dies, dass eine $m \times n$-Matrix M gefüllt wird, mit m der Länge der ersten Melodie - 1 und n der Länge der zweiten Melodie - 1. Das Element $(0,0)$ wird mit 0 initialisiert und die erste Zeile mit den Werten i für $(i,0)$ und j für $(0,j)$. Alle anderen Elemente werden rekursiv errechnet: Für den Punkt (i,j)

werden die Werte $M(i-1,j)+1$ und $M(i,j-1)+1$ errechnet, und sollte die Kontur des $i-1$-ten Eintrags der ersten Melodie gleich der Kontur des $j-1$-ten Eintrags der zweiten Melodie sein, wird der Wert $M(i-1,j-1)$ hinzugenommen, ansonsten der Wert $M(i-1,j-1)+1$. Davon wird das Minimum an der Stelle (i,j) eingetragen. Somit wird für keinen Fehler in der Kontur auch kein Fehler gegeben und für das Einfügen, Löschen oder Ändern der Kontur an einer Stelle 1 Fehlerpunkt gegeben. Die Distanz der beiden Konturen voneinander lässt sich dann am letzten Eintrag in der Matrix, $M(m-1,n-1)$ ablesen. Die Gleichung sieht dann so aus:

$$M(i,j) = \min \begin{cases} M(i-1,j)+1 \\ M(i,j-1)+1 \\ M(i-1,j-1)+1, \text{wenn die Konturen an } (i,j) \text{ nicht gleich sind} \\ M(i-1,j-1), \text{wenn die Konturen an } (i,j) \text{ gleich sind} \end{cases} \tag{30}$$

Der Quellcode hierzu ist dieser:

```
public static int conturDistance(int i1, int i2) {
    String s1 = decodeContur(i1);
    String s2 = decodeContur(i2);

    int[][] distances = new int[s1.length() / 2 + 1][s2.length() / 2 + 1];
    for (int i = 0; i < s1.length() / 2 + 1; i++) {
        distances[i][0] = i;
    }
    for (int i = 0; i < s2.length() / 2 + 1; i++) {
        distances[0][i] = i;
    }

    for (int i = 0; i < s1.length() / 2; i++) {
        for (int j = 0; j < s2.length() / 2; j++) {
            distances[i + 1][j + 1] = Math.min(distances[i][j + 1] + 1,
                    Math.min(distances[i + 1][j] + 1, distances[i][j] +
                            (((s1.charAt(i * 2) == s2.charAt(j * 2)) &&
                                    (s1.charAt(i * 2 + 1) ==
                                        s2.charAt(j * 2 + 1)))? 0 : 1)));
        }
    }

    return distances[s1.length() / 2][s2.length() / 2];
}
```

Somit ist der Quelltext der Suchfunktion für die Grobsuche der Folgende:

```
double localScore = ((1.0 / (conturDistance(classCounter1.hashCode(),
    classCounter2.hashCode()) + 1)) * classCounter1.getCounter() /
    searchedPiece.getNumberOfSegments());
```

Gewichtung

Für den zweiten Schritt, die Gewichtung der Motive nach ihrer relativen Häufigkeit innerhalb der Datenbank, muss zunächst festgestellt werden, wie häufig die einzelnen Motive überhaupt vorkommen, und wie viele es insgesamt sind. Danach werden die Variablen mit den Namen `lowerBorder` und `higherBorder` mit Werten belegt, die die Grenzen zwischen den seltenen, durchschnittlich häufig vorkommenden und häufigen Motiven festlegen. Als Modell gilt, dass jeweils ungefähr ein Drittel der vorkommenden Motive in jeder Gruppe sein soll.

Zur Auszählung der Motive wird aufsummiert, wie oft Motive mit einer bestimmten Häufigkeit auftreten. Zusätzlich wird gezählt, wieviele Motive insgesamt auftreten. Dann wird von der kleinsten Häufigkeit der Motive, nämlich nur 1 mal in der ganzen Datenbank, angefangen und aufsummiert wie oft diese Häufigkeit vorkommt. Sobald diese Summe $\frac{1}{3}$ der Gesamtanzahl der Motive erreicht hat, wird die untere Grenze gesetzt, bei $\frac{2}{3}$ die obere Grenze.

Der entsprechende Quellcode ist Folgender:

```
int[] numOfOccurances = new int[maxCounters];
int totalCounters = 0;
Iterator motifIter = motifCounter.values().iterator();
while (motifIter.hasNext()) {
    ClassCounter cc = (ClassCounter) motifIter.next();
    if (cc.getCounter() >= maxCounters)
        numOfOccurances[maxCounters - 1]++;
    else
        numOfOccurances[cc.getCounter()]++;
    totalCounters++;
}
```

```
int currentTotal = 0;
for (int i = 0; i < numOfOccurances.length; i++) {
    currentTotal += numOfOccurances[i];
    if (lowerBorder == 0 && currentTotal > (1.0 / 3.0 * totalCounters))
        lowerBorder = i;
    if (higherBorder == 0 && currentTotal > (2.0 / 3.0 * totalCounters))
        higherBorder = i;
}
```

Somit sind die Grenzen `lowerBorder` und `higherBorder` mit entsprechenden Werten belegt und können angewendet werden. Hierzu ist lediglich zu überprüfen, in welchem Bereich sich `classCounter3` befindet, welcher der Zähler der Segmente in der Datenbank ist, die dieselbe Segmentklasse wie dem aktuellen Segment aus der zu durchsuchenden Melodie haben. Entsprechend wird der lokale Wert mit 1, $\frac{3}{4}$ oder $\frac{1}{2}$ multipliziert. Der Quellcode ist recht kurz:

```
if (withWeighting) {
    if (classCounter3.getCounter() > higherBorder)
        localScore *= 0.5;
    else
        if (!(classCounter3.getCounter() <= lowerBorder))
            localScore *= 0.75;
}
```

Feinsuche

Die Feinsuche ruft eine externe `MotifComparator`-Klasse auf. Mit dieser Methode wird jedes Segment der zu durchsuchenden Melodie, das zur Segmentklasse `classCounter1` gehört, mit jedem Segment der Suchmelodie, das zur Segmentklasse `classCounter2` gehört, verglichen. Die einzelnen Motiv-Komparatoren werden in Kapitel 4.5.5 vorgestellt. Das Maximum der Übereinstimmungen wird dann mit dem lokalen Wert multipliziert.

```
NoteList[] noteSeq1 =
    getNoteLists(searchedPiece.getSegments(), classCounter1.hashCode());
[...]
NoteList[] noteSeq2 = getNoteLists(melFragSeg, classCounter2.hashCode());
double localFineMax = 0.0;
for (int j = 0; j < noteSeq1.length; j++)
    for (int k = 0; k < noteSeq2.length; k++) {
```

```
        localFineMax = Math.max(localFineMax,
            motifComparator.motifDistance(noteSeq1[j], noteSeq2[k]));
    }
localScore *= localFineMax;
```

Ermittlung der Summen

Da dieser Vorgang für jedes Motiv durchgeführt wird, wird das Maximum dieser Werte ermittelt und jedes dieser Maxima für das jeweils durchsuchte Stück aufsummiert. Die beiden Zeilen aus dem Quellcode dazu sind diese:

```
tempMax = Math.max(tempMax, localScore);
```

und

```
scores[i] += tempMax;
```

In dem Array `scores[]` ist nun für jede durchsuchte Melodie ein Wert zwischen 0 und 1 gespeichert, der die Ähnlichkeit der jeweiligen Melodie mit der Suchmelodie darstellt. Paare von Werten zusammen mit dem Index der Melodie in der Datenbank werden nach ihren Werten sortiert und in einem Array als Suchergebnis zurückgegeben.

4.5.5 Motiv-Komparatoren

Für die Feinsuche (siehe Kapitel 4.5.4) werden Motiv-Komparatoren verwendet. Diese sollen auf unterschiedliche Art und Weise die Ähnlichkeit zweier Motive errechnen.

Die Schnittstelle zu Motiv-Komparatoren ist das Interface `MotifComparator`:

```
public interface MotifComparator {
    public double motifDistance(NoteList seq1, NoteList seq2);
}
```

Die beiden zu vergleichenden Motive sind in Form einer `NoteList` zu übergeben. Als Resultat soll ein Wert zwischen 0 und 1 zurückgegeben werden, wobei eine 1 eine Übereinstimmung nach dem jeweiligen Vergleichsalgorithmus darstellen soll.

Die drei verfügbaren Motiv-Komparatoren bieten alle Möglichkeiten der Melodiesuche. Der `IdentityMotifComparator` ist ein exakter Algorithmus und der `CubyHumMotifComparator` ein fehlertoleranter Algorithmus. Der `AlwaysTrueComparator` ermöglicht, den Melodiekontur-Vergleich der Grobsuche für sich stehen zu lassen. Musikalische Analyse und Motiv-Vergleiche auf der Ebene von Motiven ist natürlich nicht mehr möglich.

`AlwaysTrueMotifComparator`

Dieser Motiv-Komparator dient alleine dem Zweck, die Feinsuche zu ignorieren. Sein Ergebnis ist immer 1, egal mit welchen Motiven er aufgerufen wird. Eine `MotifMelodyRetrieval`-Instanz mit einem `AlwaysTrueMotifComparator` führt im Endeffekt nur die Grobsuche und bei Bedarf eine Gewichtung durch.

`CubyHumMotifComparator`

Der `CubyHumMotifComparator` benutzt die Vergleichsfunktion der Klasse `CubyHumMelodyRetrieval`. Der `CubyHumMotifComparator` ist also ein fehlertoleranter Algorithmus (siehe Kapitel 2.3.3). Dazu wird eine Instanz dieser Klasse geladen. Die beiden `NoteLists` werden in den Typ `Piece` konvertiert. Mit diesen beiden Stücken wird die Methode `compareMelodies()` aufgerufen. Diese Methode ist nicht symmetrisch, das

heißt, dass `compareMelodies(piece1, piece2)` nicht unbedingt dasselbe Ergebnis wie `compareMelodies(piece2, piece1)` ergibt. Deshalb werden beide Möglichkeiten aufgerufen und der Wert der größten Übereinstimmung zurückgegeben.

Der Quellcode dieses Motiv-Komparators ist der Folgende:

```
Piece piece1 = new Piece();
Piece piece2 = new Piece();

piece1.setNotePool(seq1);
piece2.setNotePool(seq2);

// 1) how many minimum errors to fit seq2 in seq1
double score1 = cubyHumMelodyRetrieval.compareMelodies(
        cubyHumMelodyRetrieval.generateMelodyRepresentation(piece1),
        cubyHumMelodyRetrieval.getDurations(piece1),
        cubyHumMelodyRetrieval.generateMelodyRepresentation(piece2),
        cubyHumMelodyRetrieval.getDurations(piece2));
score1 /= cubyHumMelodyRetrieval.getDurations(piece1).length;
if (score1 < 0.0)
    score1 = 0.0;
else
    if (score1 > 1.0)
        score1 = 1.0;

// 2) how many minimum errors to fit seq1 in seq2
double score2 = cubyHumMelodyRetrieval.compareMelodies(
cubyHumMelodyRetrieval.generateMelodyRepresentation(piece2),
        cubyHumMelodyRetrieval.getDurations(piece2),
        cubyHumMelodyRetrieval.generateMelodyRepresentation(piece1),
        cubyHumMelodyRetrieval.getDurations(piece1));
score2 /= cubyHumMelodyRetrieval.getDurations(piece2).length;
if (score2 < 0.0)
    score2 = 0.0;
else
    if (score2 > 1.0)
        score2 = 1.0;

// return minimum error to fit one seq in the other using the
// CubyHumMelodyRetrieval algorithm
return 1.0 - ((score1 < score2) ? score1 : score2);
```

`IdentityMotifComparator`

Das Gegenstück zum `AlwaysTrueMotifComparator` ist der `IdentityMotifComparator`. Er gibt immer 0 zurück, es sei denn, dass die beiden Sequenzen bis auf Transposition und Augmentation bzw. Dimi-

nution identisch sind, in diesem Fall wird 1 zurückgegeben. Somit ist der `IdentityMotifComparator` ein exakter Algorithmus (siehe Kapitel 2.3.1).

Sein Quellcode ist:

```
double score = 1.0;

Iterator iter1 = seq1.iterator();
Iterator iter2 = seq2.iterator();

// one sequence is empty
if (!iter1.hasNext() || !iter2.hasNext())
    score = 0.0;
else {
    Note note11 = (Note) iter1.next();
    Note note12 = (Note) iter2.next();
    while (iter1.hasNext()) {
        Note note21 = (Note) iter1.next();
        // different lengths of the sequences
        if (!iter2.hasNext())
            score = 0.0;
        else {
            Note note22 = (Note) iter2.next();
            // different intervals
            if ((note21.getPerformanceNote().pitch -
                    note11.getPerformanceNote().pitch)
            != (note22.getPerformanceNote().pitch -
                    note12.getPerformanceNote().pitch))
                score = 0.0;
            // different relative lengths
            if (((double)note21.getPerformanceNote().length /
                    (double)note11.getPerformanceNote().length)
            !=  ((double)note22.getPerformanceNote().length /
                    (double)note12.getPerformanceNote().length))
                score = 0.0;
        }
    }
    // different lengths of the sequences
    if (iter2.hasNext())
        score = 0.0;
}
return score;
```

4.6 Benutzerschnittstelle

Die Klasse `MelodyRetrievalUI` stellt eine simple Benutzerschnittstelle für die verschiedenen Melodiesuch-Klassen dar. In ihr kann man die wesentlichen Aufgabe der Melodiesuch-Klassen ausprobieren. Hier soll nur beschrieben werden, wie man sie anwendet, da der Aufbau und die Programmierung der Benutzerschnittstelle für diese Arbeit nicht relevant ist.

Mit dem Befehl `java -jar MelodyRetrievalUI.jar` startet man das Programm. Es empfiehlt sich, der virtuellen Maschine von Java mehr Arbeitsspeicher zur Verfügung zu stellen, deshalb sollte man den Befehl `java -Xmx768M -jar MelodyRetrievalUI.jar` ausführen. Das folgende Hauptmenü stellt die verschiedenen Funktionen zur Verfügung:

```
-------------------- main menu --------------------
 1) load MIDI file into database
 2) load alle MIDI files of a directory into database
 3) load MelodyRetrieval class
 4) print Database
 5) search for MIDI file
 6) do MelodyRetrieval-specific tasks
 7) exit
```

`MelodyRetrievalUI` hat beim Start schon eine Datenbank initialisiert, auf der nun gearbeitet werden kann. Der Punkt 1 ist dazu da, einzelne MIDI-Dateien in die Datenbank einzufügen, und der Punkt 2 ermöglicht dies für alle MIDI-Dateien in einem Verzeichnis. Der Punkt 3 lädt eine weitere Melodiesuch-Klasse. Der Punkt 4 zeigt den momentanen Inhalt der Datenbank an und der Punkt 5 gibt die Möglichkeit, die Datenbank mit den schon geladenen Melodiesuch-Klassen zu durchsuchen. Der Punkt 6 erlaubt, spezifische Funktionen einzelner Melodiesuchen auszuführen und der Punkt 7 beendet die Benutzerschnittstelle.

4.6.1 MIDI-Dateien laden

Der Punkt 1 fordert auf, einen Dateinamen einzugeben, zum Beispiel `C:\Temp\mid\tester.mid`. Der Dateiname muss im Dateisystem, auf dem das Programm läuft, eindeutig identifizierbar sein, zum Beispiel wäre unter Linux der Dateiname `/home/cdatzko/mid/tester.mid` eine korrekte Bezeichnung.

Der Punkt 2 fordert auf, ein Verzeichnis anzugeben. Alle MIDI-Dateien dieses Verzeichnisses werden dann in die Datenbank eingefügt. Der Pfad

`C:\Temp\mid` würde zum Beispiel unter anderem die Datei `tester.mid` laden.

Diese beiden Punkte verwenden die Klasse `MidiReader` aus dem Paket `Musitech`.

4.6.2 Melodiesuch-Klasse laden

Der Punkt 3 bietet die verschiedenen zur Verfügung stehenden Melodiesuch-Klassen an, im Falle der Klasse `MotifMelodyRetrieval` mit allen verschiedenen möglichen Motiv-Komparatoren. In Klammern ist der Autor des dahintersteckenden Prinzips zur leichteren Identifikation hinzugefügt:

```
------------------- select menu -------------------
 1) ParsonsMelodyRetrieval (Denys Parsons)
 2) CubyHumMelodyRetrieval (Steffen Pauws)
 3) MotifMelodyRetrieval with AlwaysTrueMotifComparator (Christian Datzko)
 4) MotifMelodyRetrieval with IdentityMotifComparator (Christian Datzko)
 5) MotifMelodyRetrieval with CubyHumMotifComparator (Christian Datzko)
```

Es können bis zu 10 verschiedene oder gleiche Melodiesuch-Klassen geladen werden. Jede geladene Klasse wird automatisch an die Datenbank angebunden und alle Stücke in der Datenbank werden in der Melodiesuch-Klasse wie in Kapitel 4.1.2 beschrieben indiziert.

4.6.3 Datenbank ausdrucken

Durch diesen Befehl werden der Index und der interne Name (durch die `.toString()`-Methode ermittelt) für jeden Eintrag in der Datenbank aufgegeben. Wenn zum Beispiel `4123 Imported from URL file:/c:/temp/eg/W786C.MID` ausgegeben wird, so bedeutet dies, dass unter dem Index 4123 das Stück, das aus der Datei `C:\temp\eg\W786C.MID` in die Datenbank importiert wurde, gespeichert ist.

4.6.4 Nach einer MIDI-Datei suchen

Alle anderen Funktionen der Benutzerschnittstelle sind nur Hilfsfunktionen für diese Funktion: Man wird aufgefordert, anzugeben welche Datei geladen werden soll, damit nach dieser dann in der Datenbank gesucht werden kann. Diese Datei selbst wird nicht in die Datenbank aufgenommen. Es werden alle Melodiesuch-Klassen, die momentan geladen sind, aufgerufen und diese durchsuchen dann die Datenbank nach der Datei. Die Ergebnisse werden zusammen mit den errechneten Werten auf den Bildschirm ausgegeben.

Eine beispielhafte Ausgabe einer solchen Suche nach der Datei `C:\temp\mid\tester.mid` ist diese:

```
ch.datzko.melodyRetrieval.ParsonsMelodyRetrieval:
1.0 Imported from URL file:/c:/temp/mid/tester.mid
ch.datzko.melodyRetrieval.CubyHumMelodyRetrieval:
1.0 Imported from URL file:/c:/temp/mid/tester.mid
ch.datzko.melodyRetrieval.MotifMelodyRetrieval:
1.0 Imported from URL file:/c:/temp/mid/tester.mid
0.34660493827160493 Imported from URL file:/c:/temp/mid/mai.mid
```

Dieses Ergebnis ist so zu lesen, dass die drei geladenen Melodiesuch-Methoden der Reihe nach aufgerufen werden und alle zurückgegebenen Ergebnisse mit den jeweiligen Werten ausgedruckt werden.

4.6.5 Spezifische Funktionen der Melodiesuchen

Für die einzelnen Melodiesuchen sind spezifische Funktionen möglich:

1. `ParsonsMelodyRetrieval`
    a) den Index anzeigen: dies stellt einen Suchbaum einer `ParsonsMelodyRetrieval`-Klasse dar, unabhängig davon, ob

schon eine `ParsonsMelodyRetrieval`-Klasse geladen ist, oder nicht.

b) für ein einzelnes Stück die Melodiekontur errechnen: dies lädt eine MIDI-Datei und gibt die Melodiekontur, die zur Indizierung verwendet wird, zurück.

2. `CubyHumMelodyRetrieval`

a) eine Melodie-Repräsentation anhand des CubyHum Algorithmus' errechnen: dies führt die Funktion aus, die die Melodie in das Alphabet $\Sigma_9{}^*$ überführt.

b) die absoluten Längen der Noten ausgeben: Dies gibt die absoluten Längen der einzelnen Noten in Millisekunden aus.

c) zwei Melodien mit zusätzlichen Informationen vergleichen: Dies gibt alle interessanten Informationen aus, die beim Vergleich zweier Melodien mit dem CubyHum-Algorithmus entstehen: Die Noten der beiden zu vergleichenden Melodien, die jeweiligen Melodie-Repräsentationen und Tonlängen, die Matrix der Levenshtein Distance-Werte und die Matrix, welche Entscheidungen für die lokalen Minima in der Matrix getroffen wurden. Dies ist ein Beispiel eines solchen Vergleichs:

```
Imported from URL file:/c:/temp/mid/tester.mid
SCORE c1, onset=0/960, dur=1/2 MIDI 60 (pitch) from 0 for 1000 on channel 0
SCORE c1, onset=1/2, dur=1/2 MIDI 60 (pitch) from 1000 for 1000 on channel 0
SCORE b0, onset=1, dur=1/4 MIDI 59 (pitch) from 2000 for 500 on channel 0
SCORE a0, onset=5/4, dur=1/2 MIDI 57 (pitch) from 2500 for 1000 on channel 0
SCORE c1, onset=7/4, dur=1/4 MIDI 60 (pitch) from 3500 for 500 on channel 0
SCORE d1, onset=2, dur=1/2 MIDI 62 (pitch) from 4000 for 1000 on channel 0
SCORE c1, onset=5/2, dur=1/2 MIDI 60 (pitch) from 5000 for 1000 on channel 0

Imported from URL file:/c:/temp/mid/mai.mid
SCORE d#1, onset=1/2, dur=1/8 MIDI 63 (pitch) from 1000 for 250 on channel 0
SCORE f1, onset=5/8, dur=1/8 MIDI 65 (pitch) from 1250 for 250 on channel 0
SCORE g1, onset=3/4, dur=1/4 MIDI 67 (pitch) from 1500 for 500 on channel 0
SCORE g1, onset=1, dur=3/8 MIDI 67 (pitch) from 2000 for 750 on channel 0
SCORE g#1, onset=11/8, dur=1/8 MIDI 68 (pitch) from 2750 for 250 on channel 0
SCORE c2, onset=3/2, dur=1/4 MIDI 72 (pitch) from 3000 for 500 on channel 0
SCORE a#1, onset=7/4, dur=3/8 MIDI 70 (pitch) from 3500 for 750 on channel 0
SCORE g1, onset=17/8, dur=1/8 MIDI 67 (pitch) from 4250 for 250 on channel 0
```

```
SCORE a#1, onset=9/4, dur=3/16 MIDI 70 (pitch) from 4500 for 375 on channel 0
SCORE g#1, onset=39/16, dur=1/16 MIDI 68 (pitch) from 4875 for 125 on channel 0
SCORE g#1, onset=5/2, dur=1/4 MIDI 68 (pitch) from 5000 for 500 on channel 0
SCORE a#1, onset=11/4, dur=1/4 MIDI 70 (pitch) from 5500 for 500 on channel 0
SCORE g1, onset=3, dur=1/2 MIDI 67 (pitch) from 6000 for 1000 on channel 0

Query Melody Representation : 0 0 -1 -1 2 1 -1
Query Durations             : 1000 1000 500 1000 500 1000 1000
Search Melody Representation: 0 1 1 0 1 2 -1 -2 2 -1 0 1 -2
Search Durations            : 250 250 500 750 250 500 750 250 375 125 500 500 1000
distanceMatrix:
 0.00  1.20  2.20  3.60  4.60  6.00  7.20
 0.00  0.22  1.32  2.72  3.60  4.80  6.00
 0.00  0.42  0.97  1.77  2.52  3.60  4.80
 0.00  0.10  0.84  1.29  2.39  2.84  3.92
 0.00  0.36  0.58  1.62  1.54  2.72  3.42
 0.00  0.64  1.32  1.24  1.92  1.77  2.97
 0.00  0.32  0.84  1.42  2.11  2.47  1.87
 0.00  0.58  0.58  1.40  2.34  3.11  2.82
 0.00  0.54  1.44  1.34  1.60  2.67  3.87
 0.00  0.36  0.58  1.78  2.04  2.38  2.80
 0.00  0.60  1.28  1.20  2.30  2.67  3.04
 0.00  0.22  1.14  1.92  1.52  2.50  3.11
 0.00  0.64  0.74  1.32  2.42  2.19  2.92
Minimum: 1.8666666666666667

operationMatrix:
. d d d d d d
. m d d D m d
. m m m D m d
. m m m d m m
. m m m m m m
. m m m m m d
. m m m m m m
. m m m m m m
. m m m m m d
. m m m m m m
. m m m d m D
. m m m m m m
. m m D d m m
```

d) Unterschiede anzeigen: Dieser Punkt öffnet ein Fenster, in dem die Unterschiede zwischen den beiden verglichenen Stücken in Noten farbig dargestellt sind. In Abbildung 21 ist ein Beispiel eines solchen Vergleiches dargestellt. Die Farben bedeuten:

- **Grau:** die erste Note - diese wird nicht mit der entsprechenden Note der anderen Melodie verglichen.
- **Schwarz:** Keine Fehler oder einfache Modulationen.
- **Rot:** Gelöschte Note.
- **Orange:** Gelöschtes Intervall.
- **Gelb:** Eingefügtes Intervall.

- **Grün:** Eingefügte Note.

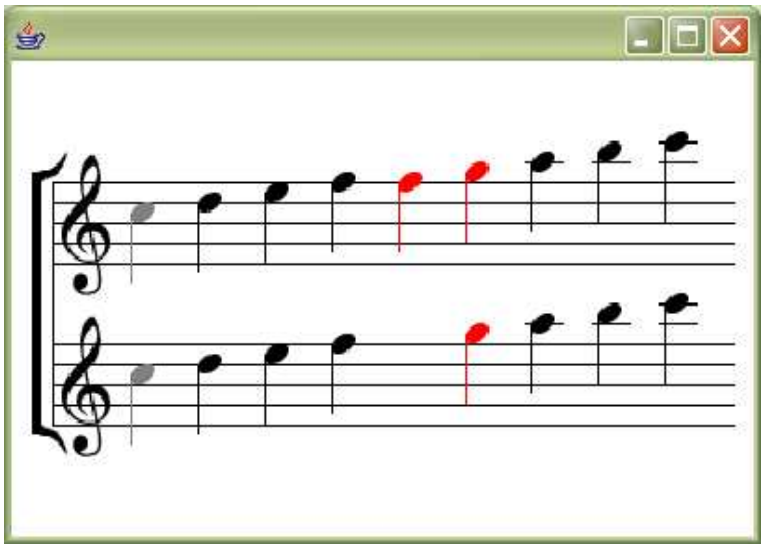


Abbildung 21: *Vergleich einer Tonleiter mit einer Tonleiter mit einem eingefügten Ton*

3. `MotifMelodyRetrieval`

   a) Eine Melodie segmentieren: Dies ruft den Segmentierer auf und gibt die einzelnen Segmente aus.

   b) Eine Melodie segmentieren und klassifizieren: Hier werden zusätzlich noch die Konturen generiert und somit die Klassifikationen errechnet.

   c) Eine Kontur kodieren: Hiermit kodiert man eine Melodie- und Rhythmus-Kontur.

   d) Eine Kontur dekodieren: Dieses dekodiert eine Melodie- und Rhythmus-Kontur.

   e) `debug` umschalten: Dies schaltet bei allen `MotifMelodyRetrieval`-Klassen das `debug`-Bit um, so dass zusätzliche Informationen während der Suche ausgegeben werden.

   f) `withWeightung`: Mit diesem Punkt kann man die Gewichtung nach Motiv-Häufigkeit in der Datenbank bei allen `MotifMelodyRetrieval`-Klassen an- bzw. ausschalten.

g) momentanen `MotifCounter` ausgeben: Dies gibt aus, welche Motivklassen mit welcher Häufigkeit momentan in allen `MotifMelodyRetrieval`-Klassen eingetragen sind.

4.7 Tests

Dieses Kapitel soll einige Testergebnisse darstellen. Als Suchmelodie wird ein Ausschnitt des Lieds „Der Lindenbaum“ aus der Winterreise von Franz Schubert verwendet. Der Ausschnitt ist in Abbildung 22 dargestellt. Die zu durchsuchenden Melodien sind aus den Datenbanken „Digital Tradition“[48] und „Evangelisches Gesangbuch“[49] übernommen, insgesamt über 8.000 Melodien. Zuletzt ist die Suchmelodie selbst noch in der Datenbank.

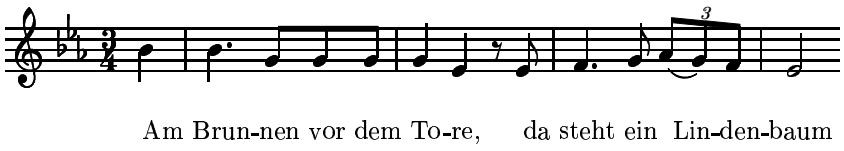


Abbildung 22: *Der Anfang von „Der Lindenbaum“*

Die Ergebnisse der Suche nach der Melodie sind in Tabelle 6 auf Seite 97 aufgelistet. Wie zu erwarten, ist die Melodie „Der Lindenbaum“, hier mit „brunnen“ bezeichnet, immer an erster Stelle oder zumindest punktgleich mit dem ersten Platz. Das Ziel, dass identische Melodien auch gefunden werden, ist also erreicht.

Des weiteren fällt auf, dass innerhalb der verschiedenen Motiv-Vergleiche auf den ersten Plätzen dieselben Stücke sind, wenn auch mit kleinen Unterschieden. Im direkten Vergleich zwischen den Suchmethoden mit Gewichtung und denen

[48] Greenhaus (2004), im MIDI-Format unter Rickheit (2004).
[49] Deutsche Bibelgesellschaft (2004).

ohne Gewichtung sind für die drei ersten Plätze gar keine Unterschiede zu erkennen. Diese treten erst später, ab dem 7. Platz auf.

Anders hingegen sieht es bei einer Bearbeitung der Suchmelodie von Friedrich Silcher aus. Abbildung 23 zeigt diese Bearbeitung der Melodie. Die Suche in der Datenbank nach ihr zeigt auf, wie stark kleine Unterschiede sich auswirken können.

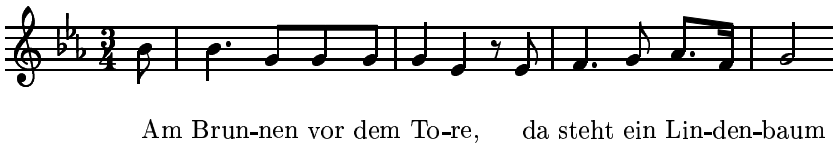


Abbildung 23: *Der Anfang von „Der Lindenbaum“ (Fassung von Friedrich Silcher)*

Mit den beiden ersten Such-Methoden ist das Original noch zu finden, aber in den nachfolgenden Such-Methoden ist es, wenn überhaupt, erst weit hinten zu finden, auf Plätzen über 200. Das bedeutet, dass das Original unter den ersten 5% der Ergebnisse zu finden ist. Dies ist jedoch kein sehr gutes Ergebnis. Das hängt sehr damit zusammen, dass die beiden Melodien unterschiedlich Segmentiert werden, wie in Abbildung 24 gezeigt, die jeweiligen Konturen sind: „EL“, „EEEEELDE“, „UL“ und „USDSDLDL“ sowie „EL“, „EEEEELDS“, „UL“, „UL“ und „UL“.

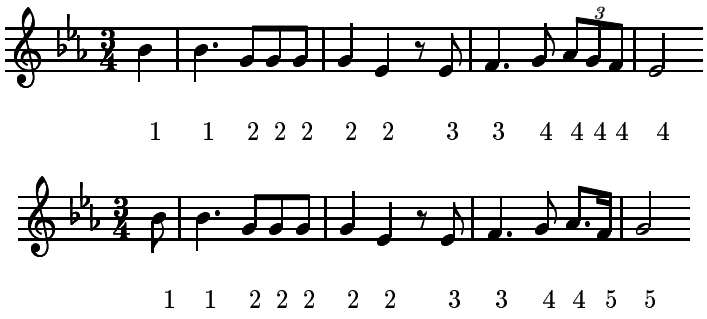


Abbildung 24: *Segmentierungen der beiden Fassungen von „Der Lindenbaum“*

Alles in allem sind diese Ergebnisse zwar befriedigend, jedoch noch längst nicht perfekt. Diese beschriebenen Tests sind nur Beispiele, eine genauere Untersuchung mit ausführlichen Testfällen würde ein genaueres Bild von der Qualität der verschiedenen Such-Methoden geben. Dies ist aber im Rahmen dieser Examensarbeit nicht zu leisten.

Tabelle 6: *Ergebnisse der Tests*

Suche	Komparator	Gewichtung	Platz 1	Platz 2	Platz 3
Suche nach dem Original					
Melodiekontur-Vergleich	-	-	r581 (EG) 1.00	rwl581 (EG) 1.00	brunnen 1.00
Fehlertoleranter Algorithmus	-	-	brunnen 1.00	-	-
Motiv-Vergleich	AlwaysTrue	nein	brunnen 1.00	oe000_6e (EG) 1.00	simsflot (DT) 0.91
Motiv-Vergleich	Identity	nein	brunnen 1.00	remalamo (DT) 0.78	sambass (DT) 0.74
Motiv-Vergleich	CubyHum	nein	brunnen 1.00	oe000_6e (EG) 0.93	remalamo (DT) 0.85
Motiv-Vergleich	AlwaysTrue	ja	oe000_6e (EG) 0.50	brunnen 0.50	simsflot (DT) 0.46
Motiv-Vergleich	Identity	ja	brunnen 0.50	remalamo (DT) 0.39	sambass (DT) 0.37
Motiv-Vergleich	CubyHum	ja	brunnen 0.50	oe000_6e (EG) 0.46	remalamo (DT) 0.43
Suche nach der Bearbeitung von Friedrich Silcher					
Melodiekontur-Vergleich	-	-	r581 (EG) 1.00	rwl581 (EG) 1.00	brunnen 1.00
Fehlertoleranter Algorithmus	-	-	brunnen 0.86	-	-
Motiv-Vergleich	AlwaysTrue	nein	oe000_6e (EG) 1.00	simsflot (DT) 0.91	cucknes2 (DT) 0.88
Motiv-Vergleich	Identity	nein	simsflot (DT) 0.85	remalamo (DT) 0.78	cucknes2 (DT) 0.77
Motiv-Vergleich	CubyHum	nein	oe000_6e (EG) 0.93	simsflot (DT) 0.90	remalamo (DT) 0.86
Motiv-Vergleich	AlwaysTrue	ja	oe000_6e (EG) 0.50	simsflot 0.46	remalamo (DT) 0.44
Motiv-Vergleich	Identity	ja	simsflot 0.43	remalamo (DT) 0.39	cucknes2 (DT) 0.38
Motiv-Vergleich	CubyHum	ja	oe000_6e (EG) 0.46	simsflot (DT) 0.45	remalamo (DT) 0.43

5 Fazit und Ausblick

Melodiesuche mit Computern lässt verschiedene Ansätze zu. Bei den Suchen sind von simplen Algorithmen bis hin zu komplizierten Algorithmen diverse Schattierungen möglich. Die Ansätze haben alle ihre Vor- und Nachteile – Treffergenauigkeit, Fehlertoleranz, Laufzeit und Platzbedarf sind unterschiedliche Parameter, die je nach Algorithmus verschieden gut umgesetzt werden.

Ziel dieser Examensarbeit war es, verschiedene Ansätze zur Melodiesuche zu beleuchten und konkrete Algorithmen zu implementieren. Durch die drei Algorithmen, den *Melodiekontur-Vergleich von Denys Parsons*, den *fehlertoleranten Algorithmus CubyHum von Steffen Pauws* und den selbst entwickelten Algorithmus der *musikalischen Analyse und des Motiv-Vergleichs* ist ein breites Spektrum an Möglichkeiten aufgezeigt worden. Das modulare Konzept ermöglicht es, bei Bedarf die vorhandenen Algorithmen anzupassen oder neue Algorithmen hinzuzufügen.

Je nach Bedarf und Anspruch an die Ergebnisse wird man sich in der Praxis für verschiedene Ansätze entscheiden. Für große Datenbanken, die online im Internet abgefragt werden können, ist die Zeit, die für eine einzelne Suche benötigt wird, wichtig. Für wissenschaftliche Zwecke ist dagegen die Genauigkeit der Ergebnisse von größerer Bedeutung, auch wenn es länger dauert, diese zu bekommen.

So haben die drei vorgestellten Algorithmen von Parsons, Pauws und die Motiv-Melodiesuche jeweils ihre eigenen Anwendungsgebiete. Anhand der Beschreibungen sowie der Qualitäts- und Effizienzbetrachtungen zu jedem von ihnen ist eine genaue Einschätzung möglich. Aufgrund dieser können sich mögliche Anwender dann entscheiden, welchen Algorithmus sie für ihre Zwecke verwenden wollen.

Mit dieser Examensarbeit wird nicht alles, was mit Melodiesuche zu tun hat, ausgeschöpft. An vielen Stellen bieten sich Möglichkeiten zur Optimierung und Erweiterung der Algorithmen. Ein paar Möglichkeiten, die ich für relevant halte, sind die Folgenden:

- *empirische Qualitätsprüfung:* Im Rahmen dieser Examensarbeit hat keine empirische Forschung stattgefunden, um die Qualität der Melodiesuchen zu überprüfen. Es sind zwar Testfälle durchgeführt worden, aber eine Überprüfung von Melodien, die von Laien als auch Fachleuten als ähnliche Melodien bezeichnen werden, wurde nicht gemacht. Hier sollte man in längeren Studien mit verschiedensten Testpersonen Testfälle durcharbeiten.

- *GUI:* Die textorientierte Benutzeroberfläche könnte graphisch aufgearbeitet werden. Dies könnte durch dynamisch erzeugtes HTML auf einem Server geschehen oder durch eine graphische Benutzeroberfläche für die Rechner, auf denen das Programm verwendet werden soll. Die Aufarbeitung würde die Bedienung und das Verständnis des Programmes erleichtern.

- *Methoden für Datenbanken, die größer als der Arbeitsspeicher sind:* Mit den beschriebenen Methoden lassen sich aufgrund des zu den jeweiligen Algorithmen beschriebenen Platzbedarfs nur eine begrenzte Anzahl von Melodien gleichzeitig im Arbeitsspeicher halten – im Falle der Datenbanken „Digital Tradition“[50] und „Evangelisches Gesangbuch“[51] nur ein paar Tausend gleichzeitig. Um die Anzahl zu erhöhen, müssen Wege gefunden werden, den maximal verfügbaren Arbeitsspeicher für eine virtuelle Java-Maschine effizienter auszunutzen. Dies könnte zum Beispiel dadurch geschehen, dass die MIDI-Dateien der Datenbank nur analysiert, aber nicht

[50] Greenhaus (2004).
[51] Deutsche Bibelgesellschaft (2004).

im Arbeitsspeicher gespeichert werden, wenn es für die Such-Methode nicht notwendig ist, auf sie zuzugreifen.

- *Audio-Dateien:* Das Verfahren wird auch als Query-by-Humming bezeichnet. Die wenigsten Anwender haben ein MIDI-Keyboard an ihrem Arbeitsplatz, viele wissen auch nicht, wie sie eine Melodie, die sie in ihrem Kopf haben, dort eingeben können. Deshalb ist es zusätzlich praktisch, eine gesummte oder gesungene Melodie auf ihre Tonhöhen zu analysieren und dies als mögliche Eingabe zu akzeptieren.

- *mehrstimmige MIDI-Dateien:* Eine zusätzliche nützliche Erweiterung wäre die Möglichkeit, mit Mehrstimmigkeit umgehen zu können. So könnten weitere Parameter zum Vergleich herangezogen werden, und so zumindest in mehrstimmigen Stücken nach einstimmigen Melodien gesucht werden kann.

Alle diese Optimierungen und Erweiterungen wären im Rahmen des hier vorgestellten Systems möglich, da die einzelnen Teile modular konstruiert sind. Dieses System bietet somit eine gute Grundlage für weitere Forschung, als auch für praktische Anwendungen.

Literatur

BARLOW, HAROLD und MORGENSTERN, SAM (1948): *A Dictionary of Musical Themes*. Crown Publishers, New York.

BARLOW, HAROLD und MORGENSTERN, SAM (1950): *A Dictionary of Vocal Themes*. Crown Publishers, New York.

BLEECK, STEFAN (1996): *Psychophysikalische Untersuchung von spektralen und zeitlichen Mechanismen des auditorischen Systems anhand harmonischer und unharmonischer Amplitudenmodulationen: relatives und absolutes Gehör*. Diplomarbeit, Fachbereich Physik der Technischen Hochschule Darmstadt.
URL: http://www.tonhoehe.de/diplom.html.

CRAWFORD, TIM, ILIOPOULOS, COSTAS S. und RAMAN, RAJEEV (1998): *String-Matching Techniques for Musical Similarity and Melodic Recognition*. In: WALTER B. HEWLETT und ELEANOR SELFRIDGE-FIELD, Hg., *Melodic Similarity – Concepts, Procedures, and Applications*. MIT Press, Cambridge.

DEUTSCHE BIBELGESELLSCHAFT (2004): *Evangelisches Gesangbuch elektronisch. CD- ROM für Windows 3.1, 95, 98 oder NT.*
URL: http://c-software.web-am-main.de/index.html?nr=081911&k=3&f=0.

EGGEBRECHT, HANS HEINRICH (2002): *Musik im Abendland : Prozesse und Stationen vom Mittelalter bis zur Gegenwart*. Piper, München.

FOGWALL, NICLAS (2004): *The search for a notation index*.
URL: http://www.af.lu.se/~fogwall/notation.html.

GILLELAND, MICHAEL (2004): *Levenshtein Distance, in Three Flavors*.
URL: http://www.merriampark.com/ld.htm.

GREENHAUS, DICK (2004): *The Digital Tradition*.
URL: http://www.mudcat.org/DigiTrad-blurb.cfm.

MAZZOLA, GUERINO (2004): *Rubato*.
URL: http://www.rubato.org/.

MILLER, GEORGE A. (1956): *The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information*. *The Psychological Review*, 63, S. 81–97.
URL: http://www.well.com/user/smalin/miller.html.

PARSONS, DENYS (1975): *Directory of Tunes and Musical Themes*. Spencer Brown, Cambridge.

PAUWS, STEFFEN (2002): *CubyHum: A Fully Operational Query by Humming System*. In: MICHAEL FINGERHUT, Hg., *ISMIR 2002 Conference Proceedings – Third International Conference on Music Information Retrieval*. IRCAM – Centre Pompidou, Paris.
URL: http://ismir2002.ismir.net/proceedings/02-FP06-2.pdf.

RICKHEIT, ERICH (2004): *Yet Another Digital Tradition Page*.
URL: http://sniff.numachi.com/~rickheit/dtrad/.

ROUBTSOV, VLADIMIR (2004): *Java Tip 130: Do you know your data size? – Don't pay the price for hidden class fields*.
URL: http://www.javaworld.com/javaworld/javatips/jw-javatip130.html.

SCHÖNBERG, ARNOLD (1979): *Die Grundlagen der musikalischen Komposition*. Universal Edition, Wien.

SELFRIDGE-FIELD, ELEANOR (1998): *Conceptual and Representational Issues in Melodic Comparison*. In: WALTER B. HEWLETT und ELEANOR SELFRIDGE-FIELD, Hg., *Melodic Similarity – Concepts, Procedures, and Applications*. MIT Press, Cambridge.

SMITH, LLOYD und MEDINA, RICHARD (2001): *Discovering Themes by Exact Pattern Matching*. In: J. STEPHEN DOWNIE und DAVID BAINBRIDGE, Hg., *ISMIR*

2001 – 2nd Annual International Symposium on Music Information Retrieval. Indiana University, Bloomington.
URL: http://ismir2001.ismir.net/posters/smith.pdf.

STEPHEN, GRAHAM A. (1994): *Lecture notes series on computing 3: String searching algorithms*. World Scientific, Singapore.

VORNBERGER, OLIVER, MÜLLER, OLAF und KUNZE, RALF (2001): *Algorithmen*, Band 127 von *Osnabrücker Schriften zur Mathematik: Reihe V*. Selbstverlag der Universität Osnabrück Fachbereich Mathematik/Informatik, Osnabrück.

WEYDE, TILLMAN (2002): *Lern- und wissensbasierte Analyse von Rhythmen*. Dissertation, Universität Osnabrück, Fachbereich Erziehungs- und Kulturwissenschaften.

WEYDE, TILLMAN (2003): *Optimising Parameter Weights in Models for Melodic Segmentation*. In: *Proceedings of the ESCOM 2003 Conference*. Hochschule für Musik und Theater Hannover.

WEYDE, TILLMAN (2004): *The Influence of Pitch on Melodic Segmentation*. Für die Veröffentlichung in *Proceedings of the International Conference on Music Information Retrieval* eingereicht.

Abbildungsverzeichnis

Tabellenverzeichnis

A Quelldateien

A.1 MelodyRetrieval.java

```
/*
 * Created on 24.11.2003
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.Piece;

/**
 * <pre>
 * Abstract class for Melody Retrieval.
 *
 * A structure for the index of the database must be user supplied - maybe
 * binary or other search trees, maybe hash tables, maybe other means of
 * structuring data.
 * </pre>
 *
 * @author Christian Datzko
 */
public abstract class MelodyRetrieval implements PieceDatabaseChangeListener {

    /**
     * <pre>
     * Creates a new MelodyRetrieval object. Needs a database of music to piece
     * on (which can be empty, but not null).
     * </pre>
     *
     * @param db the database to piece on
     */
    public MelodyRetrieval(PieceDatabase db) {
        if (db != null)
            database = db;
        else
            throw new IllegalArgumentException("Error during initialization "
                    + "of MelodyRetrieval: Need a database to piece on!");
        db.addPieceDatabaseChangeListener(this);
    }

    /**
     * <pre>
     * This method needs to be called from the constructor as soon as all index
     * structures are set up to add all already existing pieces in the database
     * to the index.
     *
     * </pre>
     */
    public void addExistingPieces() {
        // if there are already pieces in the database, update the index
        PieceDatabase.PieceEnum enum = database.getFirst();
        int i = 1;
        while (enum != null) {
            this.insertPiece(enum.getPiece(), i);
            enum = enum.getNext();
            i++;
        }
    }

    /**
     * <pre>
     * the database to which this MelodyRetrieval class is attached to
```

```
     * </pre>
     */
    PieceDatabase database;

    /**
     * <pre>
     * Returns the associated database.
     * </pre>
     *
     * @return the database
     */
    public PieceDatabase getDatabase() {
        return database;
    }


    /**
     * <pre>
     * A pair of a piece and a score. The score is a percentage between 0.0 and
     * 1.0.
     * </pre>
     *
     * @author Christian Datzko
     */
    public class PieceScorePair {

        /**
         * <pre>
         * the piece
         * </pre>
         */
        Piece piece;

        /**
         * <pre>
         * the score, between 0.0 and 1.0
         * </pre>
         */
        double score;
    }

    /**
     * <pre>
     * Quicksort algorithm to sort PieceScorePairs after their score. The
     * PieceScorePair with the highest score is first.
     * This algorithm is an adaption of a standard Quicksort algorithm, that can
     * be found for example in: Oliver Vornberger, Olaf Müller, Ralf Kunze:
     * Osnabrücker Schriften zur Mathematik: Reihe V, Heft 127: Algorithmen.
     * </pre>
     *
     * @param array the array to sort
     * @param bottom the first index to sort
     * @param top the last index to sort
     */
    public void sortPieceScorePairs(PieceScorePair[] array,
            int bottom, int top) {
        if (array.length != 0) {
            PieceScorePair temp;
            int i = bottom;
            int j = top;
            PieceScorePair x = array[(bottom + top) / 2];
            do {
                while (array[i].score > x.score)
                    i++;
                while (array[j].score < x.score)
                    j--;
```

```
                if (i <= j) {
                    temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                    i++;
                    j--;
                }
            } while (i <= j);

            if (bottom < j)
                sortPieceScorePairs (array, bottom, j);
            if (i < top)
                sortPieceScorePairs (array, i, top);
        }
    }

    /**
     * <pre>
     * This method returns pairs of pieces with their scores which are
     * calculated when trying to compare the melodyFragment with each
     * piece of the PieceDatabase.
     *
     * In certain implementations only pieces with a score larger than the
     * threshhold are returned. If you want more pieces, lower the threshhold.
     * </pre>
     *
     * @param melodyFragment the melody fragment to search for
     * @return an array of piece and score pairs, sorted from the one with the
     * highest score down
     */
    public abstract PieceScorePair[] searchForPairs(Piece melodyFragment);

    /**
     * <pre>
     * Searches within the database for a melody fragment.
     * </pre>
     *
     * @param melodyFragment the melody fragment to search for in the database
     * @return the pieces are returned in an array
     */
    public abstract Piece[] searchFor(Piece melodyFragment);

    /**
     * <pre>
     * Adds a piece to the database.
     * </pre>
     *
     * @param piece the piece to add to the database
     */
    public void insertPiece(Piece piece) {
        database.insertPiece(piece);
    }
}
```

A.2 ParsonsMelodyRetrieval.java

```
/*
 * Created on 24.11.2003
 */
package ch.datzko.melodyRetrieval;

import java.util.Iterator;

import de.uos.fmt.musitech.data.structure.Context;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.container.Containable;
import de.uos.fmt.musitech.data.structure.container.Container;
import de.uos.fmt.musitech.data.structure.form.NoteList;


/**
 * <pre>
 * This class is a (very simple) Melody Retrieval class - to test all the
 * methods and structures in MelodyRetrieval.java, PieceDatabase.java und
 * PieceDatabaseChangeListener.java.
 *
 * The index method pieces via the mechanism suggested by Denys Parsons
 * in his book "The Directory of Tunes and musical Themes". His idea was to
 * abstract from the melody only wether the next note is higher, lower or
 * the same as its previous note. The first note is marked with a *. The
 * famous song "Silent Night" would be
 *
 *     *UDDUUDDUSDUSDUSUDDDUDD...
 *
 * The index is arranged as a tree: up to a depth of 10 each piece is entered
 * at its specific characteristic:
 *
 *                                      *
 *                  D                   S                   U
 *            D     S     U       D     S     U       D     S     U
 *          D S U D S U D S U   D S U D S U D S U   D S U D S U D S U
 *                                     ...
 *        ... lots of leafs ... ... lots of leafs ... lots of leafs ...
 *
 * Some practical annotations: A melody must be at least 10 notes long (if
 * not, Ss are added), for now it only pieces on monophonic music.
 *
 * See also Denys Parsons: The Directory of Tunes and Musical Themes, London:
 * Spencer Brown, 1975.
 * </pre>
 *
 * @author Christian Datzko
 */
public class ParsonsMelodyRetrieval extends MelodyRetrieval {

    /**
     * <pre>
     * Initiates the database and the index tree.
     * </pre>
     */
    public ParsonsMelodyRetrieval () {
        super(new PieceDatabase());
        indexTree = new DSUIndexTree();
        addNewLevelToDSUIndexTree(indexTree, 1);
    }

    /**
     * <pre>
```

```
 * Initiates with the supplied database and initiates the index tree.
 * </pre>
 */
public ParsonsMelodyRetrieval (PieceDatabase db) {
    super(db);
    indexTree = new DSUIndexTree();
    addNewLevelToDSUIndexTree(indexTree, 1);
    addExistingPieces();
}

/**
 * <pre>
 * recursion depth of the index tree
 * </pre>
 */
private int recursionDepth = 9;

/**
 * <pre>
 * Down-Same-Up-Index-Tree. This data structure is used to save the index
 * of the database.
 * </pre>
 *
 * @author Christian Datzko
 */
private class DSUIndexTree {

    /**
     * <pre>
     * A new, pure and empty DSUIndexTree.
     * </pre>
     */
    public DSUIndexTree() {
        down = null;
        same = null;
        up = null;
    }

    private DSUIndexTree down;

    /**
     * <pre>
     * Returns the index tree resp. leaf when the next note is D (down)
     * </pre>
     * @return the index tree resp. leaf when the next note is D (down)
     */
    public DSUIndexTree getDown() {
        return down;
    }

    /**
     * <pre>
     * Sets a new down - don't use this when data is in the tree, the
     * structure might corrupt
     * </pre>
     * @param newDown sets a new down - don't use this when data is in the
     * tree, the structure might corrupt
     */
    public void setDown(DSUIndexTree newDown) {
        down = newDown;
    }

    private DSUIndexTree same;

    /**
     * <pre>
```

```
     * Returns the index tree resp. leaf when the next note is S (same)
     * </pre>
     * @return the index tree resp. leaf when the next note is S (same)
     */
    public DSUIndexTree getSame() {
        return same;
    }

    /**
     * <pre>
     * Sets sets a new same - don't use this when data is in the tree, the
     * structure might corrupt
     * </pre>
     * @param newSame sets a new same - don't use this when data is in the
     * tree, the structure might corrupt
     */
    public void setSame(DSUIndexTree newSame) {
        same = newSame;
    }

    private DSUIndexTree up;

    /**
     * <pre>
     * Returns the index tree resp. leaf when the next note is U (up)
     * </pre>
     * @return the index tree resp. leaf when the next note is U (up)
     */
    public DSUIndexTree getUp() {
        return up;
    }

    /**
     * <pre>
     * Sets a new up - don't use this when data is in the tree, the
     * structure might corrupt
     * </pre>
     * @param newUp sets a new up - don't use this when data is in the
     * tree, the structure might corrupt
     */
    public void setUp(DSUIndexTree newUp) {
        up = newUp;
    }
}

/**
 * <pre>
 * Down-Same-Up-Index-Leaf. This is the bottom-most element of a
 * DSUIndexTree. It doesn't contain and more trees or leafs, but
 * contains a leafList.
 * </pre>
 *
 * @author Christian Datzko
 */
private class DSUIndexLeaf extends DSUIndexTree {

    private class IndexLeafLinkList {

        public IndexLeafLinkList(int i) {
            index = i;
        }

        private IndexLeafLinkList next = null;

        public IndexLeafLinkList getNext() {
            return next;
```

```
    }

    public void setNext(IndexLeafLinkList newNext) {
        next = newNext;
    }

    private int index;

    public int getIndex() {
        return index;
    }

    public void setIndex(int i) {
        index = i;
    }
}

/**
 * <pre>
 * Creates a new DSUIndexLeaf with an empty list.
 * </pre>
 */
public DSUIndexLeaf() {
}

IndexLeafLinkList leafList = null;

/**
 * <pre>
 * </pre>
 * Inserts a piece @ index i into the leaf.
 * @param i the index of the piece in the database
 */
public void insertIndex(int i) {
    if (leafList == null)
        leafList = new IndexLeafLinkList(i);
    else {
        IndexLeafLinkList ll = leafList;
        while (ll.getNext() != null)
            ll = ll.getNext();
        ll.setNext(new IndexLeafLinkList(i));
    }
}

/**
 * <pre>
 * Deletes an index to piece i from the leaf.
 * </pre>
 *
 * @param i the index of the piece to delete
 */
public void deleteIndex(int i) {
    while (leafList != null && leafList.getIndex() == i) {
        leafList = leafList.getNext();
    }
    if (leafList != null) {
        IndexLeafLinkList ll = leafList;
        while (ll.getNext() != null)
            if (ll.getNext().getIndex() == i)
                ll.setNext(ll.getNext().getNext());
            else
                ll = ll.getNext();
    }
}

/**
```

```
         * <pre>
         * Returns a new index list with all the indices of the leaf.
         * </pre>
         *
         * @return the int[] of the indices
         */
        public int[] returnIndex() {
            int[] indexList = new int[leafListLength()];

            // retrieve entries of the link list
            IndexLeafLinkList ll = leafList;
            int i = 0;
            while (ll != null) {
                indexList[i] = ll.getIndex();
                ll = ll.getNext();
                i++;
            }

            // return indices
            return indexList;
        }

        public int leafListLength() {
            IndexLeafLinkList ll = leafList;
            int i = 0;
            while (ll != null) {
                ll = ll.getNext();
                i++;
            }
            return i;
        }

        /**
         * <pre>
         * Since there should be no more knots, don't add one.
         * </pre>
         */
        public void setDown(DSUIndexTree newDown) {
        }

        /**
         * <pre>
         * Since there should be no more knots, don't add one.
         * </pre>
         */
        public void setSame(DSUIndexTree newSame) {
        }

        /**
         * <pre>
         * Since there should be no more knots, don't add one.
         * </pre>
         */
        public void setUp(DSUIndexTree newUp) {
        }
    }

    /**
     * <pre>
     * The one DSUIndexTree for the ParsonsMelodyRetrieval index of the
     * database.
     * </pre>
     */
    private DSUIndexTree indexTree;
```

```
/**
 * <pre>
 * Builds up recursivly a new DSUIndexTree.
 * </pre>
 *
 * @param tree the current knot to continue building
 * @param previousLevel the current level on which to add
 */
private void addNewLevelToDSUIndexTree(DSUIndexTree tree,
        int previousLevel) {
    if ((tree != null) && previousLevel < recursionDepth) {
        tree.setDown(new DSUIndexTree());
        tree.setSame(new DSUIndexTree());
        tree.setUp(new DSUIndexTree());
        previousLevel++;
        addNewLevelToDSUIndexTree(tree.getDown(), previousLevel);
        addNewLevelToDSUIndexTree(tree.getSame(), previousLevel);
        addNewLevelToDSUIndexTree(tree.getUp(), previousLevel);
    }
    else if (tree != null) {
        tree.setDown(new DSUIndexLeaf());
        tree.setSame(new DSUIndexLeaf());
        tree.setUp(new DSUIndexLeaf());
    }
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 *      searchForPairs(de.uos.fmt.musitech.music.Piece)
 */
public PieceScorePair[] searchForPairs(Piece melodyFragment) {
    Piece[] pieces = searchFor(melodyFragment);
    PieceScorePair[] pieceScorePairs = new PieceScorePair[pieces.length];
    for (int i = 0; i < pieces.length; i++) {
        pieceScorePairs[i] = new PieceScorePair();
        pieceScorePairs[i].score = 1.0;
        pieceScorePairs[i].piece = pieces[i];
    }
    return pieceScorePairs;
}

/*
 * This method is the implementation of the method defined in the abstract
 * class MelodyRetrieval.
 *
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 *      searchFor(de.uos.fmt.musitech.music.Piece)
 */
public Piece[] searchFor(Piece melodyFragment) {
    String index = generateIndex(melodyFragment);
    DSUIndexTree t = indexTree;
    for (int i = 1; (i < index.length()) && (i < recursionDepth); i++) {
        if (index.charAt(i) == 'D')
            t = t.getDown();
        else if (index.charAt(i) == 'S')
            t = t.getSame();
        else if (index.charAt(i) == 'U')
            t = t.getUp();
    }
    return returnAll(t);
}

/*
 * This method is the implementation of the method defined in the interface
 * PieceDatabaseChangeListener. Should only called by a PieceDatabase.
 *
```

```
     * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#
     *      insertPiece(de.uos.fmt.musitech.music.Piece, int)
     */
    public void insertPiece(Piece piece, int newIndex) {
        String index = generateIndex(piece);
        // It is assumed that the generated index is valid.
        // If it is not long enough it is padded with Ss.
        while (index.length() <= recursionDepth)
            index = index + "S";
        DSUIndexTree t = indexTree;
        for (int i = 1; i <= recursionDepth; i++) {
            if (index.charAt(i) == 'D')
                t = t.getDown();
            else if (index.charAt(i) == 'S')
                t = t.getSame();
            else if (index.charAt(i) == 'U')
                t = t.getUp();
        }
        DSUIndexLeaf u = (DSUIndexLeaf)t;
        u.insertIndex(newIndex);
    }

    /*
     * This method is the implementation of the method defined in the interface
     * PieceDatabaseChangeListener. Should only called by a PieceDatabase.
     *
     * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#deletePiece(int)
     */
    public void deletePiece(int index) {
        deletePieceRecursive(indexTree, index);
    }

    /**
     * <pre>
     * Generates an index from a piece
     * </pre>
     *
     * @param piece the piece to analyse and to generate the index from
     * @return The index' first letter is a *, the others may only be "D", "S"
     * or "U" for down, same and up.
     */
    public String generateIndex(Piece piece) {
        String returnString = "*";
        if (piece != null) {
            // get the notes of the piece
            // from here on it is assumed that the piece is only monophonic
            Container noteCollection = piece.getNotePool();
            Containable conts[] = noteCollection.getContentsRecursive();
            NoteList sortedNotes = new NoteList(new Context(piece));
            for (int i = 0; i < conts.length; i++) {
                if(conts[i] instanceof Note){
                    sortedNotes.add(conts[i]);
                    // for debugging: printout of the notes in the piece
                    // System.out.println(conts[i]);
                }
            }

            Note note = null;
            int previousPitch = 0;
            Iterator iter = sortedNotes.iterator();
            if (iter.hasNext()) {
                note = (Note) iter.next();
                previousPitch = note.getPerformanceNote().getPitch();
            }
            while (iter.hasNext() && returnString.length() < recursionDepth + 1)
            {
```

```
            note = (Note) iter.next();
            // for debugging: printout of the notes in the NoteList
            // System.out.println(note);
            int currentPitch = note.getPerformanceNote().getPitch();
            // ignore all rests
            if (currentPitch != 0) {
                if (currentPitch > previousPitch)
                    returnString = returnString + "U";
                else if (currentPitch == previousPitch)
                    returnString = returnString + "S";
                else returnString = returnString + "D";
                previousPitch = currentPitch;
            }
            // System.out.println(previousPitch + " " + note.
            //      getPerformanceNote().getPitch() + " " + returnString);
        }
    }
    return returnString;
}

/**
 * <pre>
 * Returns all pieces starting from t in an array.
 * </pre>
 *
 * @param t the DSUIndexTree to start from
 * @return Array containing all pieces from below t
 */
Piece[] returnAll(DSUIndexTree t) {
    if (t instanceof DSUIndexLeaf) {
        int[] indices = ((DSUIndexLeaf)t).returnIndex();
        Piece[] pieces = new Piece[indices.length];
        for (int i = 0; i < indices.length; i++) {
            pieces[i] = database.getPiece(indices[i]);
        }
        return pieces;
    }
    else {
        Piece[] pieces1 = returnAll(t.getDown());
        Piece[] pieces2 = returnAll(t.getSame());
        Piece[] pieces3 = returnAll(t.getUp());
        Piece[] pieces = new Piece[pieces1.length + pieces2.length +
                                   pieces3.length];
        for (int i = 0; i < pieces1.length; i++)
            pieces[i] = pieces1[i];
        for (int i = 0; i < pieces2.length; i++)
            pieces[i + pieces1.length] = pieces2[i];
        for (int i = 0; i < pieces3.length; i++)
            pieces[i + pieces1.length + pieces2.length] = pieces3[i];
        return pieces;
    }
}

/**
 * <pre>
 * This method is only used by deletePiece to find and delete a link from
 * the index tree.
 * </pre>
 *
 * @param t current branch
 * @param index index to delete
 */
void deletePieceRecursive(DSUIndexTree t, int index) {
    if (t instanceof DSUIndexLeaf) {
        // this is a leaf, no more branches below
        DSUIndexLeaf l = (DSUIndexLeaf) t;
```

```
            l.deleteIndex(index);
        } else {
            // this is a branch, recurse to all the subbranches
            deletePieceRecursive(t.getDown(), index);
            deletePieceRecursive(t.getSame(), index);
            deletePieceRecursive(t.getUp(), index);
        }
    }

    /**
     * <pre>
     * This method is used internally for the recursion in the database.
     * </pre>
     *
     * @param t current branch
     * @param path path up to this branch
     */
    void printIndexDatabase(DSUIndexTree t, String path) {
        if (t instanceof DSUIndexLeaf) {
            if (((DSUIndexLeaf) t).leafListLength() != 0) {
                int[] indices = ((DSUIndexLeaf) t).returnIndex();
                System.out.print(path + ":");
                for (int i = 0; i < indices.length; i++) {
                    System.out.print(" " + indices[i]);
                }
                System.out.println();
            }
        }
        else {
            printIndexDatabase(t.getDown(), path + "D");
            printIndexDatabase(t.getSame(), path + "S");
            printIndexDatabase(t.getUp(), path + "U");
        }
    }

    /**
     * <pre>
     * Prints out all entries in the database to System.out for debugging
     * purposes.
     * </pre>
     */
    public void printIndexDatabase() {
        printIndexDatabase(indexTree, "*");
    }
}
```

A.3 CubyHumMelodyRetrieval.java

```
/*
 * Created on 17.02.2004
 */
package ch.datzko.melodyRetrieval;

import java.util.Iterator;

import de.uos.fmt.musitech.data.structure.Context;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.container.Containable;
import de.uos.fmt.musitech.data.structure.container.Container;
import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 * <pre>
 * This class is a Melody Retrieval implementation based on the CubyHum System
 * described by Steffen Pauws in Steffen Pauws: CubyHum: "A Fully Operational
 * Query by Humming System" in: Michael Fingerhut (Editor): "ISMIR 2002
 * Conference Proceedings -- Third International Conference on Music
 * Information Retrieval", Paris, 2002. This is also available at
 * http://ismir2002.ismir.net/proceedings/02-FP06-2.pdf.
 *
 * The basic idea of the melody retrieval part of CubyHum is to abstract the
 * interval structure of a melody to a melody contour with 9 different steps,
 * one being unison and the others being a second, a third, a fourth or greater
 * up or down.
 *
 * The search algorithm allows for differences in the compared melodies, they
 * are:
 * - Melodic sequences of variable length
 * - Amount of mistuning
 * - Modulation by interval replacement
 * - Note replacement
 * - Note insertion
 * - Note deletion
 * - Duration errors
 *
 * In addition a change in rhythm is taken in account.
 *
 * A total score subtracting (errors / number of notes) from 100% matches is
 * calculated for each piece in the database.
 *
 * The method searchFor(Piece piece) implemented from MelodyRetrieval simply
 * returns the matches with a score larger than a specified threshhold (0.85
 * default) sorted with the best match first, while the method searchForPairs
 * returns also the score for each piece.
 * </pre>
 *
 * @author Christian Datzko
 */
public class CubyHumMelodyRetrieval extends MelodyRetrieval {

    /**
     * <pre>
     * Creates a new CubyHumMelodyRetrieval object with a supplied database
     * and a threshhold from which on a query melody is assumed as found
     * </pre>
     *
     * @param db the database to connect to
     * @param found_threshhold the threshhold from which on a query melody is
     * assumed as found (in %)
     */
```

```
public CubyHumMelodyRetrieval(PieceDatabase db, double found_threshhold) {
    super(db);
    threshhold = found_threshhold;
    addExistingPieces();
}

/**
 * <pre>
 * Creates a new CubyHumMelodyRetrieval object with a supplied database
 * </pre>
 *
 * @param db the database to connect to
 */
public CubyHumMelodyRetrieval(PieceDatabase db) {
    this(db, defaultThreshhold);
}

/**
 * <pre>
 * Creates a new CubyHumMelodyRetrieval object and initiates a new database
 *
 * </pre>
 */
public CubyHumMelodyRetrieval() {
    this(new PieceDatabase());
}

/**
 * <pre>
 * the default threshhold. See "int threshhold".
 * </pre>
 */
final static double defaultThreshhold = 0.85;

/**
 * <pre>
 * char representation of interval deletion.
 * </pre>
 */
final static char intervalDeletionChar = 'd';

/**
 * <pre>
 * char representation of interval insertion.
 * </pre>
 */
final static char intervalInsertionChar = 'i';

/**
 * <pre>
 * char representation of note deletion.
 * </pre>
 */
final static char noteDeletionChar = 'D';

/**
 * <pre>
 * char representation of note insertion.
 * </pre>
 */
final static char noteInsertionChar = 'I';

/**
 * <pre>
 * char representation of modulation (or no error).
 * </pre>
```

```
 */
final static char modulationChar = 'm';

/**
 * <pre>
 * char representation of no movement (for (i,0) and (0,j)).
 * </pre>
 */
final static char noMovementChar = '.';

/**
 * <pre>
 * the maximum length of a melody
 * </pre>
 */
private int maxMelodyLength = 1000;

/**
 * <pre>
 * set true to have extra debug information on the console
 * </pre>
 */
private boolean debug = false;

/**
 * @return Whether debuggin information is outputted or not.
 */
public boolean isDebug() {
    return debug;
}

/**
 * @param debug true: Debugging information is printet do System.out,
 * false: Debugging information is not printed.
 */
public void setDebug(boolean debug) {
    this.debug = debug;
}


/**
 * <pre>
 * This IndexLinkList is the way to store the indices for the database.
 * For each piece the link to the PieceDatabase "index", a melody
 * representation "melodyRepresentation" and the durations "durations" are
 * stored.
 * </pre>
 *
 * @author Christian Datzko
 */
class IndexLinkList {

    /**
     * <pre>
     * Creates a new IndexLinkList node. You have to supply it with
     * an index of the piece in the PieceDatabase, with a melody
     * representation of this piece and with the durations of the piece.
     * </pre>
     *
     * @param newIndex the index of the piece in the PieceDatabase
     * @param newMelodyRepresentation a melody representation of the piece
     * in the PieceDatabase
     * @param newDurations the durations of the piece in the PieceDatabase
     */
    public IndexLinkList(int newIndex, int[] newMelodyRepresentation,
            long[] newDurations) {
```

```
        next = null;
        index = newIndex;
        melodyRepresentation = newMelodyRepresentation;
        durations = newDurations;
    }

    /**
     * <pre>
     * the next node in the IndexLinkList
     * </pre>
     */
    private IndexLinkList next;

    /**
     * <pre>
     * Returns the next node in the IndexLinkList.
     * </pre>
     *
     * @return the next node
     */
    public IndexLinkList getNext() {
        return next;
    }

    /**
     * <pre>
     * Sets a new next node in the IndexLinkList.
     * </pre>
     *
     * @param newNext the new next node
     */
    public void setNext(IndexLinkList newNext) {
        next = newNext;
    }

    /**
     * <pre>
     * the index of the current piece in the PieceDatabase
     * </pre>
     */
    private int index;

    /**
     * <pre>
     * Returns the index of the current piece in the PieceDatabase.
     * </pre>
     *
     * @return the index of the current piece
     */
    public int getIndex() {
        return index;
    }

    /**
     * <pre>
     * a melody representation of the piece in the PieceDatabase
     * </pre>
     */
    private int[] melodyRepresentation;

    /**
     * <pre>
     * Returns a melody representation of the current piece in the
     * PieceDatabase.
     * </pre>
     *
```

```
     * @return a melody representation of the current piece
     */
    public int[] getMelodyRepresentation() {
        return melodyRepresentation;
    }

    /**
     * <pre>
     * the durations of the notes in the PieceDatabase
     * </pre>
     */
    private long[] durations;

    /**
     * <pre>
     * Returns the durations of the current piece in the PieceDatabase.
     * </pre>
     *
     * @return the durations of the current piece
     */
    public long[] getDurations() {
        return durations;
    }
}

/**
 * <pre>
 * the link list where all the indices and informations about the pieces are
 * stored
 * </pre>
 */
private IndexLinkList indexLinkList = null;


/**
 * <pre>
 * The threshhold from when on a piece is considered as a match. Should be
 * below or equal to 1.0 and should be greater than or equal to 0.0 to piece.
 * Default is set to defaultThreshhold.
 * </pre>
 */
private double threshhold;

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 *      searchForPairs(de.uos.fmt.musitech.music.Piece)
 */
public PieceScorePair[] searchForPairs(Piece melodyFragment) {
    final int maxFoundPieces = 1000;
    int[] queryMR = generateMelodyRepresentation(melodyFragment);
    long[] queryD = getDurations(melodyFragment);
    PieceScorePair[] pieces = new PieceScorePair[maxFoundPieces];
    for (int i = 0; i < maxFoundPieces; i++) {
        pieces[i] = new PieceScorePair();
    }
    int foundPieces = 0;
    IndexLinkList ll = indexLinkList;
    while (ll != null) {
        double comparismScore = compareMelodies(queryMR, queryD,
                ll.getMelodyRepresentation(), ll.getDurations());
        // The next calculation somewhat generates a normalisation to
        // 0.0 .. 1.0. Since in rare cases values below 0.0 can exist
        // it is minimised with 0.0. The higher the comparismScore, the
        // greater its similarity between the piece and the melodyFragment.
        comparismScore =  1 - (comparismScore / queryD.length);
        if (comparismScore < 0.0)
```

```
            comparismScore = 0.0;
        if (comparismScore > threshhold) {
            pieces[foundPieces].piece = database.getPiece(ll.getIndex());
            pieces[foundPieces].score = comparismScore;
            foundPieces++;
        }
        ll = ll.getNext();
    }
    PieceScorePair[] returnPieces = new PieceScorePair[foundPieces];
    for (int i = 0; i < foundPieces; i++)
        returnPieces[i] = pieces[i];
    sortPieceScorePairs(returnPieces, 0, foundPieces - 1);
    return returnPieces;
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 *      searchFor(de.uos.fmt.musitech.music.Piece)
 */
public Piece[] searchFor(Piece melodyFragment) {
    PieceScorePair[] pieceScorePairs = searchForPairs(melodyFragment);
    Piece[] returnPieces = new Piece[pieceScorePairs.length];
    for (int i = 0; i < pieceScorePairs.length; i++) {
        returnPieces[i] = pieceScorePairs[i].piece;
    }
    return returnPieces;
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#
 *      insertPiece(de.uos.fmt.musitech.music.Piece, int)
 */
public void insertPiece(Piece piece, int newIndex) {
    if (indexLinkList == null) {
        indexLinkList = new IndexLinkList(newIndex,
                generateMelodyRepresentation(piece), getDurations(piece));
    }
    else {
        IndexLinkList ll = indexLinkList;
        while (ll.getNext() != null) {
            ll = ll.getNext();
        }
        ll.setNext(new IndexLinkList(newIndex,
                generateMelodyRepresentation(piece), getDurations(piece)));
    }
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#deletePiece(int)
 */
public void deletePiece(int index) {
    while (indexLinkList != null && indexLinkList.getIndex() == index) {
        indexLinkList = indexLinkList.getNext();
    }
    IndexLinkList previous = indexLinkList;
    IndexLinkList ll = indexLinkList.getNext();
    while (ll != null) {
        if (ll.getIndex() == index)
            previous.setNext(ll.getNext());
        else
            previous = ll;
        ll = ll.getNext();
    }
}

/**
```

```
 * <pre>
 * This class holds all the information that is generated when comparing
 * two melodies: The distances, the operations and where the minimum is.
 * The minimum itself is not contained since it is returned from the method.
 * The dimensions of operationMatrix and distanceMatrix will be the same
 * after running compareMelodiesWithOperations(...).
 * </pre>
 *
 * @author Christian Datzko
 */
public static class Matrices {
    /**
     * <pre>
     * at each (i,j) is the minimum sum of the errors necessary to get to
     * that point.
     * </pre>
     */
    double[][] distanceMatrix = null;

    /**
     * <pre>
     * at each (i,j) the chosen operations are stored. they can be any of
     * intervalDeletionChar, intervalInsertionChar, noteDeletionChar,
     * noteInsertionChar, modulationChar or noMovementChar. With this
     * information you can track back the way through the matrix taken.
     * </pre>
     */
    char[][] operationMatrix = null;

    /**
     * <pre>
     * the row in which the minimum value in the last column of the
     * distanceMatrix is located
     * </pre>
     */
    int minimumAt = 0;
}

/**
 * <pre>
 * This is an implementation of the algorithm describted in CubyHum, chapter
 * 6. The melody is represented by a list of integers which are generated
 * using the following function:
 * each intervall between two notes is mapped in the following way:
 *
 * interval name                       | interval size      | integer code
 * ------------------------------------+--------------------+--------------
 * descending fifth and greater        | < -6 semitones     | -4
 * descending perfect/augmented fourth | -5 or -6 semitones | -3
 * descending minor/major third        | -3 or -4 semitones | -2
 * descending minor/major second       | -1 or -2 semitones | -1
 * unison                              | 0 semitones        |  0
 * ascending minor/major second        | 1 or 2 semitones   |  1
 * ascending minor/major third         | 3 or 4 semitones   |  2
 * ascending perfect/augmented fourth  | 5 or 6 semitones   |  3
 * ascending fith and greater          | > 6 semitones      |  4
 * </pre>
 *
 * @param piece the piece to analyse
 * @return array of integers of -4, -3, ..., 3, 4.
 */
public int[] generateMelodyRepresentation(Piece piece) {
    int[] intervals = new int[maxMelodyLength];
    int intervalCounter = 1;
    intervals[0] = 0; // first interval is no interval
    if (piece != null) {
```

```
            // get the notes of the piece
            // from here on it is assumed that the piece is only monophonic
            Container noteCollection = piece.getNotePool();
            Containable conts[] = noteCollection.getContentsRecursive();
            NoteList sortedNotes = new NoteList(new Context(piece));
            if (debug)
                System.out.println(piece.getName());
            for (int i = 0; i < conts.length; i++) {
                if(conts[i] instanceof Note){
                    sortedNotes.add(conts[i]);
                    // for debugging: printout of the notes in the piece
                    if (debug)
                        System.out.println(conts[i]);
                }
            }
            if (debug)
                System.out.println();

            Iterator iter = sortedNotes.iterator();
            Note note = null;
            int previousPitch = 0;
            if (iter.hasNext()) {
                note = (Note) iter.next();
                previousPitch = note.getPerformanceNote().getPitch();
            }
            for (; iter.hasNext();) {
                note = (Note) iter.next();
                // for debugging: printout of the notes in the NoteList
                // if (debug)
                //  System.out.println(note);
                int currentPitch = note.getPerformanceNote().getPitch();
                // ignore all rests
                if (currentPitch != 0) {
                    if (currentPitch - previousPitch < -6)
                        intervals[intervalCounter] = -4;
                    else if (currentPitch - previousPitch < -4)
                        intervals[intervalCounter] = -3;
                    else if (currentPitch - previousPitch < -2)
                        intervals[intervalCounter] = -2;
                    else if (currentPitch - previousPitch < 0)
                        intervals[intervalCounter] = -1;
                    else if (currentPitch - previousPitch < 1)
                        intervals[intervalCounter] = 0;
                    else if (currentPitch - previousPitch < 3)
                        intervals[intervalCounter] = 1;
                    else if (currentPitch - previousPitch < 5)
                        intervals[intervalCounter] = 2;
                    else if (currentPitch - previousPitch < 7)
                        intervals[intervalCounter] = 3;
                    else
                        intervals[intervalCounter] = 4;
                    previousPitch = currentPitch;
                    if (intervalCounter < maxMelodyLength)
                        intervalCounter++;
                }
            }
        }
        int[] returnIntervals = new int[intervalCounter];
        for (int i = 0; i < intervalCounter; i++) {
            returnIntervals[i] = intervals[i];
        }
        if (intervalCounter == 1)
            return new int[0];
        else
            return returnIntervals;
    }
```

```
/**
 * <pre>
 * This returns an array of the durations of all the notes (excluding
 * rests). The durations are in absolute time (ms).
 * </pre>
 *
 * @param piece The piece to analyse, must be monophonic
 * @return an array with the durations of all the notes (excluding rests)
 */
public long[] getDurations(Piece piece) {
    long[] durations = new long[maxMelodyLength];
    int durationsCounter = 0;
    if (piece != null) {
        // get the notes of the piece
        // from here on it is assumed that the piece is only monophonic
        Container noteCollection = piece.getNotePool();
        Containable conts[] = noteCollection.getContentsRecursive();
        NoteList sortedNotes = new NoteList(new Context(piece));
        for (int i = 0; i < conts.length; i++) {
            if(conts[i] instanceof Note){
                sortedNotes.add(conts[i]);
                // for debugging: printout of the notes in the piece
                // if (debug)
                //  System.out.println(conts[i]);
            }
        }

        Iterator iter = sortedNotes.iterator();
        Note note = null;
        if (iter.hasNext()) {
            note = (Note) iter.next();
            for (; iter.hasNext();) {
                // for debugging: printout of the notes in the NoteList
                // if (debug)
                //  System.out.println(note);
                long currentDuration =
                    note.getPerformanceNote().getDuration();
                // add up all rests to the previous duration, but ignore
                //  rests at the beginning
                if (note.getPerformanceNote().getPitch() != 0) {
                    durations[durationsCounter] = currentDuration;
                    durationsCounter++;
                } else if (durationsCounter != 0)
                    durations[durationsCounter - 1] =
                        durations[durationsCounter - 1] + currentDuration;
                note = (Note) iter.next();
            }
            // for debugging: printout of the notes in the NoteList
            // if (debug)
            //  System.out.println(note);
            long currentDuration = note.getPerformanceNote().getDuration();
            if (note.getPerformanceNote().getPitch() != 0) {
                durations[durationsCounter] = currentDuration;
                durationsCounter++;
            }
            else if (durationsCounter != 0)
                durations[durationsCounter - 1] =
                    durations[durationsCounter - 1] + currentDuration;
        }
    }
    long[] returnDurations = new long[durationsCounter];
    for (int i = 0; i < durationsCounter; i++) {
        returnDurations[i] = durations[i];
    }
    return returnDurations;
```

```
}

/**
 * <pre>
 * This function compares two melodies using the algorithm described for
 * CubyHum. The function is this:
 *
 * d_Matrix[0][0] ... d_Matrix[0][M - 1] = 0.0
 * d_Matrix[i][0] = d_Matrix[i - 1][0] + 1 + k * |q_D[i] / q_D[i-1]|
 *    for i = 1 ... N - 1
 * d_Matrix[i][j] =
 *     /                             | q_D[i] |
 *     | d_Matrix[i-1][j] + 1 + k * |--------|
 *     |                             |q_D[i-1]|
 *     |
 *     |                               |q_D[i-1] + q_D[i]    s_D[i] |
 *     | d_Matrix[i-2][j-1] + 1 + k * |----------------- - --------|, *)
 *     |                               |    q_D[i-2]        s_D[i-1]|
 *     |
 *     |                              c
 * min | d_Matrix[i-1][j-1] + 1 + ----- * |q_MR[i] - s_MR[i]| +
 *     |                            sigma
 *     |                                   | q_D[i]     s_D[j]  |
 *     |                                   |-------- - -------- |
 *     |                                   |q_D[i-1]   s_D[j-1] |
 *     |
 *     |                               | q_D[i]    s_D[j-1] + s_D[j]|
 *     | d_Matrix[i-1][j-2] + 1 + k * |-------- - -----------------|, **)
 *     |                               |q_D[i-1]       s_D[j-2]     |
 *     |
 *     |                             | s_D[j] |
 *     | d_Matrix[i][j-1] + 1 + k * |--------|
 *     \                             |s_D[j-1]|
 *
 * *) if q_MR[i-1] + q_MR[i] = s_MR[i], i > 2
 * **) if q_MR[i] = s_MR[j-1] + s_MR[j], j > 2
 *
 * sigma = 9 = size of the alphabet of the melody representation used
 * k = 0.2 = the weight of a rhythm error agains any other error
 * c = the maximum weight of a modulation error agains all other errors
 *
 * The first choice accounts for interval deletions, the second choice for
 * note deletions, the third choice for modulation errors or no error, the
 * fourth choice for note insertions and the fifth choice for interval
 * insertions.
 * </pre>
 *
 * @param queryMelodyRepresentation a representation of the query melody
 * @param queryDurations the durations of the query melody
 * @param searchMelodyRepresentation a representation of the searched melody
 * @param searchDurations the duration of the searched melody
 * @param matrices matrices to store the ways taken
 * @return the sum of the minimum error points of the melody comparism. Is
 * most probably (but not guaranteed) smaller then the length of the query
 * melody
 */
public double compareMelodiesWithOperations(int[] q_MR, long[] q_D,
        int[] s_MR, long[] s_D,
        Matrices matrices) {
    if (q_MR.length == 0 || s_MR.length == 0)
        return Double.POSITIVE_INFINITY;
    else {
        if (q_MR.length != q_D.length) {
            throw new IllegalArgumentException("q_MR and q_D do not belo" +
            "ng together!");
        }
```

```
if (s_MR.length != s_D.length)
    throw new IllegalArgumentException("s_MR and s_D do not belo" +
    "ng together!");

// a MxN-Matrix
matrices.distanceMatrix = new double[q_D.length][s_D.length];

// a MxN-Matrix for storing the decisions which way was taken
matrices.operationMatrix = new char[q_D.length][s_D.length];
for (int i = 0; i < q_D.length; i++)
    for (int j = 0; j < s_D.length; j++)
        matrices.operationMatrix[i][j] = noMovementChar;
for (int i = 1; i < q_D.length; i++)
    matrices.operationMatrix[i][0] = intervalDeletionChar;

// size of the melody representation alphabet
double sigma = 9.0;
// the weight of a rhythm error against any other error
double k = 0.2;
// the maximum weight of a modulation error agains all other errors
double c = 2.0;

// initialise matrix
matrices.distanceMatrix[0][0] = 0.0;
for (int i = 1; i < q_D.length; i++) {
    matrices.distanceMatrix[i][0] =
        matrices.distanceMatrix[i - 1][0] + 1 + k *
        Math.abs(q_D[i] / q_D[i-1]);
}
for (int j = 1; j < s_D.length; j++) {
    matrices.distanceMatrix[0][j] = 0.0;
}

// fill matrix
for (int i = 1; i < q_D.length; i++)
    for (int j = 1; j < s_D.length; j++) {
        // interval deletion:
        double interval_deletion =
            matrices.distanceMatrix[i - 1][j] + 1 + k *
            Math.abs((double)q_D[i] / (double)q_D[i-1]);
        // note deletion:
        double note_deletion = Double.POSITIVE_INFINITY;
        if (i > 2 && (q_MR[i - 1] + q_MR[i] == s_MR[j]))
            note_deletion = matrices.distanceMatrix[i - 2][j - 1] +
            1 + k * Math.abs(((double)q_D[i - 1] +
                    (double)q_D[i]) / (double)q_D[i - 2] -
                    (double)s_D[j] / (double)s_D[j - 1]);
        // modulation or no error:
        double modulation = matrices.distanceMatrix[i - 1][j - 1] +
        (c / sigma) * Math.abs(q_MR[i] - s_MR[j]) + k *
        Math.abs((double)q_D[i] / (double)q_D[i - 1] -
                (double)s_D[j] / (double)s_D[j - 1]);
        // note insertion:
        double note_insertion = Double.POSITIVE_INFINITY;
        if (j > 2 && (q_MR[i] == s_MR[j - 1] + s_MR[j]))
            note_insertion = matrices.distanceMatrix[i - 1][j - 2] +
            1 + k * Math.abs((double)q_D[0] / (double)q_D[i - 1] -
                    ((double)s_D[j - 1] + (double)s_D[j]) /
                    (double)s_D[j - 2]);
        // interval insertion:
        double interval_insertion =
            matrices.distanceMatrix[i][j - 1] + 1 + k *
            Math.abs((double)s_D[j] / (double)s_D[j - 1]);
        matrices.distanceMatrix[i][j] = Math.min(interval_deletion,
                Math.min(note_deletion,
                        Math.min(modulation,
```

```
                                    Math.min(note_insertion,
                                            interval_insertion))));

            // fill the operationMatrix
            if (matrices.distanceMatrix[i][j] == interval_deletion)
                matrices.operationMatrix[i][j] = intervalDeletionChar;
            else if (matrices.distanceMatrix[i][j] == note_deletion)
                matrices.operationMatrix[i][j] = noteDeletionChar;
            else if (matrices.distanceMatrix[i][j] == modulation)
                matrices.operationMatrix[i][j] = modulationChar;
            else if (matrices.distanceMatrix[i][j] == note_insertion)
                matrices.operationMatrix[i][j] = noteInsertionChar;
            else if (matrices.distanceMatrix[i][j] ==
                interval_insertion)
                matrices.operationMatrix[i][j] = intervalInsertionChar;
        }

    // find minimum
    double minimum = Double.POSITIVE_INFINITY;
    matrices.minimumAt = 0;
    for (int j = 0; j < s_D.length; j++) {
        minimum = Math.min(minimum,
                matrices.distanceMatrix[q_D.length - 1][j]);
        if (minimum == matrices.distanceMatrix[q_D.length - 1][j])
            matrices.minimumAt = j;
    }
    // output matrix and results
    if (debug) { // set true for debugging
        System.out.print("Query Melody Representation : ");
        for (int i = 0; i < q_MR.length; i++) {
            System.out.print(q_MR[i] + " ");
        }
        System.out.println();
        System.out.print("Query Durations             : ");
        for (int i = 0; i < q_D.length; i++) {
            System.out.print(q_D[i] + " ");
        }
        System.out.println();
        System.out.print("Search Melody Representation: ");
        for (int i = 0; i < s_MR.length; i++) {
            System.out.print(s_MR[i] + " ");
        }
        System.out.println();
        System.out.print("Search Durations            : ");
        for (int i = 0; i < s_D.length; i++) {
            System.out.print(s_D[i] + " ");
        }
        System.out.println();
        System.out.println();
        System.out.println("distanceMatrix:");
        for (int j = 0; j < s_D.length; j++) {
            for (int i = 0; i < q_D.length; i++) {
                String s =
                    (Math.round(matrices.distanceMatrix[i][j] * 100)
                            / 100.0) + "";
                while (s.indexOf(".") < 2)
                    s = " " + s;
                while (s.length() < 5)
                    s = s + "0";
                s = s + " ";
                System.out.print(s);
            }
            System.out.println();
        }
        System.out.println("Minimum: " + minimum);
        System.out.println();
```

```
                System.out.println("operationMatrix:");
                for (int j = 0; j < s_D.length; j++) {
                    for (int i = 0; i < q_D.length; i++) {
                        System.out.print(matrices.operationMatrix[i][j] + " ");
                    }
                    System.out.println();
                }
            }
            return minimum;
        }
    }

    /**
     * <pre>
     * This function compares two melodies using the algorithm described for
     * CubyHum. For more information see compareMelodiesWithWay.
     * </pre>
     *
     * @param queryMelodyRepresentation a representation of the query melody
     * @param queryDurations the durations of the query melody
     * @param searchMelodyRepresentation a representation of the searched melody
     * @param searchDurations the duration of the searched melody
     * @return the sum of the minimum error points of the melody comparism. Is
     * most probably (but not guaranteed) smaller then the length of the query
     * melody
     */
    public double compareMelodies(int[] q_MR, long[] q_D, int[] s_MR,
            long[] s_D) {
        return compareMelodiesWithOperations(q_MR, q_D, s_MR, s_D,
                new Matrices());
    }

    /**
     * <pre>
     * Prints out all entries in the database to System.out for debugging
     * purposes.
     * </pre>
     */
    public void printIndexDatabase() {
        IndexLinkList ll = indexLinkList;
        while (ll != null) {
            System.out.print(database.getPiece(ll.getIndex()).getName() + ":");
            int[] melodyRepresentation = ll.getMelodyRepresentation();
            for (int i = 0; i < melodyRepresentation.length; i++) {
                System.out.print(" " + melodyRepresentation[i]);
            }
            System.out.println();
            long[] durations = ll.getDurations();
            for (int i = 0; i < durations.length; i++) {
                System.out.print(" " + durations[i]);
            }
            System.out.println();
            ll = ll.getNext();
        }
    }
}
```

A.4 CubyHumHelper.java

```
/*
 * Created on 17.03.2004
 */
package ch.datzko.melodyRetrieval;

import java.awt.Color;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Iterator;

import javax.swing.JFrame;

import ch.datzko.melodyRetrieval.CubyHumMelodyRetrieval.Matrices;
import de.uos.fmt.musitech.data.score.NotationStaff;
import de.uos.fmt.musitech.data.score.NotationSystem;
import de.uos.fmt.musitech.data.score.NotationVoice;
import de.uos.fmt.musitech.data.score.ScoreNote;
import de.uos.fmt.musitech.data.structure.Context;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.RenderingNote;
import de.uos.fmt.musitech.data.structure.container.Containable;
import de.uos.fmt.musitech.data.structure.container.Container;
import de.uos.fmt.musitech.data.structure.form.NoteList;
import de.uos.fmt.musitech.score.gui.ScoreMapper;
import de.uos.fmt.musitech.score.gui.ScorePanel;
import de.uos.fmt.musitech.utility.math.Rational;

/**
 * <pre>
 * This helper class provides some additional methods for the class
 * CubyHumMelodyRetrieval which will rarely be used and thus do not need to be
 * loaded each and every time.
 *
 * Most interesting is the method showDifferences which provides a graphical
 * explanation of the differences of two melodies. The color coding is as
 * follows:
 *
 * Color.GRAY: The first note is colored gray to show that it is not compared
 * to allow transposition invariance.
 * Color.BLACK: No error or simple Modulations are colored black.
 * Color.RED: Deleted notes are colored red.
 * Color.ORANGE: Deleted Intervals are colored orange.
 * Color.YELLOW: Inserted Intervals are colored yellow.
 * Color.GREEN: Inserted Notes are colored green.
 *
 * For more informations @see ch.datzko.melodyRetrieval.CubyHumMelodyRetrieval
 * </pre>
 *
 * @author Christian Datzko
 */

public class CubyHumHelper {

    /**
     * <pre>
     * This constructs a new CubyHumHelper class and attaches it to the passed
     * CubyHumMelodyRetrieval class.
     * </pre>
     *
     * @param mr a CubyHumMelodyRetrieval class
     */
    public CubyHumHelper(CubyHumMelodyRetrieval mr) {
```

```
    chmr = mr;
}

/**
 * <pre>
 * This constructs a new CubyHumHelper class and also constructs a new
 * CubyHumMelodyRetrieval class.
 * </pre>
 */
public CubyHumHelper() {
    chmr = new CubyHumMelodyRetrieval();
}

/**
 * <pre>
 * This WindowAdapter is used internally to wait until the window is closed.
 * </pre>
 *
 * @author Christian Datzko
 */
private class ExitWindowAdapter extends WindowAdapter {
    /**
     * <pre>
     * Externally visible variable, check this to wait until it is true.
     * </pre>
     */
    boolean exit = false;

    public void windowClosing(WindowEvent ev) {
        exit = true;
    }
}

/**
 * <pre>
 * The CubyHumMelodyRetrieval class to which this class is attached.
 * </pre>
 */
private CubyHumMelodyRetrieval chmr;


/**
 * <pre>
 * Returns the CubyHumMelodyRetrieval class to which this class is attached.
 * </pre>
 *
 * @return the CubyHumMelodyRetrieval class to which this class is attached.
 */
public CubyHumMelodyRetrieval getCubyHumMelodyRetrieval() {
    return chmr;
}

/**
 * <pre>
 * Compares two monophonic melodies, shows a window with the differences and
 * returns after the window is closed.
 *
 * The opened window provides a graphical explanation of the differences of
 * two melodies. The color coding is as follows:
 *
 * Color.GRAY: The first note is colored gray to show that it is not
 * compared to allow transposition invariance.
 * Color.BLACK: No error or simple Modulations are colored black.
 * Color.RED: Deleted notes are colored red.
 * Color.ORANGE: Deleted Intervals are colored orange.
 * Color.YELLOW: Inserted Intervals are colored yellow.
```

```
     * Color.GREEN: Inserted Notes are colored green.
     *
     * For more informations @see ch.datzko.melodyRetrieval.
     * CubyHumMelodyRetrieval
     * </pre>
     *
     * @param queryPiece must be monophonic!
     * @param searchedPiece must be monophonic!
     */
    public void showDifferences(Piece queryPiece, Piece searchedPiece) {
        // get the way matrix
        Matrices way = new Matrices();
        chmr.compareMelodiesWithOperations(chmr.generateMelodyRepresentation(
                queryPiece),
                chmr.getDurations(queryPiece),
                chmr.generateMelodyRepresentation(searchedPiece),
                chmr.getDurations(searchedPiece), way);

        // find the changes
        String changes = "";
        int begin = way.minimumAt;
        for (int i = way.operationMatrix.length - 1; i > -1; ) {
            switch (way.operationMatrix[i][begin]) {
                case CubyHumMelodyRetrieval.intervalDeletionChar :
                    i--;
                changes = CubyHumMelodyRetrieval.intervalDeletionChar +
                changes;
                break;
                case CubyHumMelodyRetrieval.intervalInsertionChar :
                    begin--;
                changes = CubyHumMelodyRetrieval.intervalInsertionChar +
                changes;
                break;
                case CubyHumMelodyRetrieval.noteDeletionChar :
                    i--;
                i--;
                begin--;
                changes = CubyHumMelodyRetrieval.noteDeletionChar + changes;
                break;
                case CubyHumMelodyRetrieval.noteInsertionChar :
                    i--;
                begin--;
                begin--;
                changes = CubyHumMelodyRetrieval.noteInsertionChar +
                changes;
                break;
                case CubyHumMelodyRetrieval.modulationChar :
                    i--;
                begin--;
                changes = CubyHumMelodyRetrieval.modulationChar + changes;
                break;
                case CubyHumMelodyRetrieval.noMovementChar :
                    i--;
                changes = CubyHumMelodyRetrieval.noMovementChar + changes;
                break;
                default :
                    break;
            }
        }

        if (chmr.isDebug()) {
            System.out.println(changes);
            System.out.println(begin);
        }

        // build the panel
```

```
// sort the notes of the queryPiece
Container queryNoteCollection = queryPiece.getNotePool();
Containable queryConts[] = queryNoteCollection.getContentsRecursive();
NoteList queryPieceSortedNotes = new NoteList(
        new Context(queryPiece));
for (int i = 0; i < queryConts.length; i++) {
    if(queryConts[i] instanceof Note){
        queryPieceSortedNotes.add(queryConts[i]);
        // for debugging: printout of the notes in the piece
        // System.out.println(conts[i]);
    }
}

// sort the notes of the searchedPiece
Container searchedNoteCollection = searchedPiece.getNotePool();
Containable searchedConts[] = searchedNoteCollection.
getContentsRecursive();
NoteList searchedPieceSortedNotes = new NoteList(
        new Context(searchedPiece));
for (int i = 0; i < searchedConts.length; i++) {
    if(searchedConts[i] instanceof Note){
        searchedPieceSortedNotes.add(searchedConts[i]);
        // for debugging: printout of the notes in the piece
        // System.out.println(conts[i]);
    }
}

Context context = new Context(new Piece());
NotationSystem system = new NotationSystem(context);
NotationStaff queryStaff = new NotationStaff(context, system);
system.add(queryStaff);
NotationVoice queryVoice = new NotationVoice(context, queryStaff);

NotationStaff searchedStaff = new NotationStaff(context, system);
system.add(searchedStaff);
NotationVoice searchedVoice = new NotationVoice(context, searchedStaff);
searchedStaff.add(searchedVoice);
queryStaff.add(queryVoice);

Iterator queryIter = queryPieceSortedNotes.iterator();
Note queryNote = (Note) queryIter.next();
Iterator searchedIter = searchedPieceSortedNotes.iterator();
Note searchedNote = null;

// no time signature
system.getContext().getWork().setMetricalTimeLine(null);

// skip ignored notes in searched piece
for (int i = 0; i < begin; i++) {
    searchedNote = (Note) searchedIter.next();
    if (i > begin - 5)
        searchedVoice.add(new RenderingNote(new ScoreNote(searchedNote.
                getScoreNote().getMetricTime(),
                searchedNote.getScoreNote().getMetricDuration(),
                searchedNote.getScoreNote().getDiatonic(),
                searchedNote.getScoreNote().getOctave(),
                searchedNote.getScoreNote().getAccidental()),
                null,
                Color.GRAY));
}

Rational metricTime;

if (searchedNote != null)
    metricTime = searchedNote.getScoreNote().getMetricTime().add(
```

```
            searchedNote.getScoreNote().getMetricDuration());
else
    metricTime = new Rational(0,4);

queryIter = queryPieceSortedNotes.iterator();
int k = 0;
while (k < changes.length()) {
    switch (changes.charAt(k)) {
        case CubyHumMelodyRetrieval.intervalDeletionChar :
            queryNote = (Note) queryIter.next();
        queryVoice.add(new RenderingNote(new ScoreNote(metricTime,
                queryNote.getScoreNote().getMetricDuration(),
                queryNote.getScoreNote().getDiatonic(),
                queryNote.getScoreNote().getOctave(),
                queryNote.getScoreNote().getAccidental()),
                null,
                Color.ORANGE));
        metricTime = metricTime.add(queryNote.getScoreNote().
                getMetricDuration());
        break;
        case CubyHumMelodyRetrieval.intervalInsertionChar :
            searchedNote = (Note) searchedIter.next();
        searchedVoice.add(new RenderingNote(new ScoreNote(
                metricTime,
                searchedNote.getScoreNote().getMetricDuration(),
                searchedNote.getScoreNote().getDiatonic(),
                searchedNote.getScoreNote().getOctave(),
                searchedNote.getScoreNote().getAccidental()),
                null,
                Color.YELLOW));
        metricTime = metricTime.add(searchedNote.getScoreNote().
                getMetricDuration());
        break;
        case CubyHumMelodyRetrieval.noteDeletionChar :
            queryNote = (Note) queryIter.next();
        queryVoice.add(new RenderingNote(new ScoreNote(metricTime,
                searchedNote.getScoreNote().getMetricDuration(),
                queryNote.getScoreNote().getDiatonic(),
                queryNote.getScoreNote().getOctave(),
                queryNote.getScoreNote().getAccidental()),
                null,
                Color.RED));
        metricTime = metricTime.add(searchedNote.getScoreNote().
                getMetricDuration());
        queryNote = (Note) queryIter.next();
        queryVoice.add(new RenderingNote(new ScoreNote(metricTime,
                searchedNote.getScoreNote().getMetricDuration(),
                queryNote.getScoreNote().getDiatonic(),
                queryNote.getScoreNote().getOctave(),
                queryNote.getScoreNote().getAccidental()),
                null,
                Color.RED));
        searchedNote = (Note) searchedIter.next();
        searchedVoice.add(new RenderingNote(new ScoreNote(
                metricTime,
                searchedNote.getScoreNote().getMetricDuration(),
                searchedNote.getScoreNote().getDiatonic(),
                searchedNote.getScoreNote().getOctave(),
                searchedNote.getScoreNote().getAccidental()),
                null,
                Color.RED));
        metricTime = metricTime.add(searchedNote.getScoreNote().
                getMetricDuration());
        break;
        case CubyHumMelodyRetrieval.noteInsertionChar :
            searchedNote = (Note) searchedIter.next();
```

```
searchedVoice.add(new RenderingNote(new ScoreNote(
        metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        searchedNote.getScoreNote().getDiatonic(),
        searchedNote.getScoreNote().getOctave(),
        searchedNote.getScoreNote().getAccidental()),
        null,
        Color.GREEN));
metricTime = metricTime.add(searchedNote.getScoreNote().
        getMetricDuration());
queryNote = (Note) queryIter.next();
searchedNote = (Note) searchedIter.next();
queryVoice.add(new RenderingNote(new ScoreNote(metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        queryNote.getScoreNote().getDiatonic(),
        queryNote.getScoreNote().getOctave(),
        queryNote.getScoreNote().getAccidental()),
        null,
        Color.GREEN));
searchedVoice.add(new RenderingNote(new ScoreNote(
        metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        searchedNote.getScoreNote().getDiatonic(),
        searchedNote.getScoreNote().getOctave(),
        searchedNote.getScoreNote().getAccidental()),
        null,
        Color.GREEN));
metricTime = metricTime.add(searchedNote.getScoreNote().
        getMetricDuration());
break;
case CubyHumMelodyRetrieval.modulationChar :
    queryNote = (Note) queryIter.next();
searchedNote = (Note) searchedIter.next();
queryVoice.add(new Note(new ScoreNote(metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        queryNote.getScoreNote().getDiatonic(),
        queryNote.getScoreNote().getOctave(),
        queryNote.getScoreNote().getAccidental()),
        null));
searchedVoice.add(new Note(new ScoreNote(metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        searchedNote.getScoreNote().getDiatonic(),
        searchedNote.getScoreNote().getOctave(),
        searchedNote.getScoreNote().getAccidental()),
        null));
metricTime = metricTime.add(searchedNote.getScoreNote().
        getMetricDuration());
break;
case CubyHumMelodyRetrieval.noMovementChar :
    queryNote = (Note) queryIter.next();
searchedNote = (Note) searchedIter.next();
queryVoice.add(new RenderingNote(new ScoreNote(metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        queryNote.getScoreNote().getDiatonic(),
        queryNote.getScoreNote().getOctave(),
        queryNote.getScoreNote().getAccidental()),
        null,
        Color.GRAY));
searchedVoice.add(new RenderingNote(new ScoreNote(
        metricTime,
        searchedNote.getScoreNote().getMetricDuration(),
        searchedNote.getScoreNote().getDiatonic(),
        searchedNote.getScoreNote().getOctave(),
        searchedNote.getScoreNote().getAccidental()),
        null,
        Color.GRAY));
```

```
                metricTime = metricTime.add(searchedNote.getScoreNote().
                        getMetricDuration());
                break;
                default :
                    break;
            }
            k++;
        }

        for (int i = 0; i < 4; i++) {
            if (searchedIter.hasNext()) {
                searchedNote = (Note) searchedIter.next();
                searchedVoice.add(new RenderingNote(new ScoreNote(metricTime,
                        searchedNote.getScoreNote().getMetricDuration(),
                        searchedNote.getScoreNote().getDiatonic(),
                        searchedNote.getScoreNote().getOctave(),
                        searchedNote.getScoreNote().getAccidental()),
                        null,
                        Color.GRAY));
                metricTime = metricTime.add(searchedNote.getScoreNote().
                        getMetricDuration());
            }
        }

        ScorePanel panel = new ScorePanel(system);
        ScoreMapper mapper = new ScoreMapper(panel, system);
        JFrame frame = new JFrame();
        ExitWindowAdapter ewa = new ExitWindowAdapter();
        frame.addWindowListener(ewa);
        frame.getContentPane().add(panel);
        frame.setSize(500, 400);
        frame.setVisible(true);

        while (!ewa.exit) {
            try {
                java.lang.Thread.sleep(100);
            }
            catch (InterruptedException e) {
            }
        }
        frame.dispose();
    }

    /**
     * <pre>
     * This static method initialises its own CubyHumMelodyRetrieval and
     * compares the two melodies in piece1 and piece2 and returns the minimum
     * necessary error to transform piece1 into piece2. Beware that
     * compareMelodies(pieceA, pieceA) doesn't necessarily return the same as
     * comapreMelodies(pieceB, pieceA). piece1 is the "query melody" and piece2
     * the "searched melody". For further details @see ch.datzko.
     * melodyRetrieval.CubyHumMelodyRetrieval#compareMelodiesWithOperations(...)
     * </pre>
     *
     * @param piece1 must be monophonic!
     * @param piece2 must be monophonic!
     * @return
     */
    public static double compareMelodies(Piece piece1, Piece piece2) {
        CubyHumMelodyRetrieval mr = new CubyHumMelodyRetrieval();
        return mr.compareMelodies(mr.generateMelodyRepresentation(piece1),
                mr.getDurations(piece1),
                mr.generateMelodyRepresentation(piece2),
                mr.getDurations(piece2));
    }
}
```

A.5 MotifMelodyRetrieval.java

```
/*
 * Created on 09.03.2004
 */
package ch.datzko.melodyRetrieval;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import structure.Segmentation;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.container.Containable;
import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 * <pre>
 * This is a MelodyRetrieval implementation developed by Christian Datzko, it
 * analyses the melodies for segmentation and then searches for segments rather
 * than for complete melodies.
 *
 * This class uses the Segmenter of ISSM from the MusicalStructure package which
 * seperates the segments according to their rhythmical structure. One could use
 * and other segmenter though as long as the segmentation algorithm is the same
 * for all the pieces.
 *
 * This class requires a MotifComparator class that does a fine comparism of the
 * motifs. This MotifComparator can be simple, like the
 * AlwaysTrueMotifComparator, or quite elaborate, like the
 * CubyHumMotifComparator.
 *
 * The search function piece this way:
 *
 * First for all segment classes in each searched piece a best fitting segment
 * class in the searched for piece is selected. This is done using a edit
 * distance method. This is multiplied with the number of occurances in the
 * searched piece and divided through the total number of segments in the
 * searched piece. This gives a weighting of the importance of the segment class.
 * The formula is:
 *
 *   l = 1 / (d(S,Q)+ 1) * n1 / n2
 *
 * Second optionally a weighting of the uniqueness of the segment class is
 * performed where the occurances of the segment class in the complete database
 * is taken into account. The formula is:
 *
 *               / 1    for rare segment classes
 *   l_new = l * | 0.75 for average segment classes
 *               \ 0.5  for common segment classes
 *
 * where each group of segment classess is about 1/3 of the segments total.
 *
 * Third a fine seach is conducted with all segments in the segment class of
 * the two pieces and the best fit is selected. The selection is based upon a
 * similarity measure using the supplied MotifComparator class. This value
 * is multiplied with l or l_new, depending on whether weighting is performed
 * or not.
 * </pre>
 *
 * @author Christian Datzko
 */
public class MotifMelodyRetrieval extends MelodyRetrieval {
```

```
/**
 * <pre>
 * Creates a new MotifMelodyRetrieval class. db is the database to which
 * this class should connect to, mc the comparism function to compare two
 * motifs during the search.
 * </pre>
 *
 * @param db the database to connect to
 * @param mc the MotifComparator to use during search
 */
public MotifMelodyRetrieval(PieceDatabase db, MotifComparator mc) {
    super(db);
    motifComparator = mc;
    indexArrayList = new ArrayList();
    motifCounter = new HashMap();
    threshhold = defaultThreshhold;
    addExistingPieces();
}

/**
 * <pre>
 * Creates a new MotifMelodyRetrieval class. Since no database is supplied
 * a new, empty one will be created. Otherwise it has the same
 * functionality as public MotifMelodyRetrieval(PieceDatabase db,
 * MotifComparator mc).
 * </pre>
 *
 * @param mc the MotifComparator to use during search
 */
public MotifMelodyRetrieval(MotifComparator mc) {
    this(new PieceDatabase(), mc);
}

/**
 * <pre>
 * the default threshhold. See "int threshhold".
 * </pre>
 */
final static double defaultThreshhold = 0.25;

/**
 * <pre>
 * This char is used to encode that from one note to another it goes up.
 * </pre>
 */
public static final char upChar = 'U';

/**
 * <pre>
 * This char is used to encode that from one note to another it stays at
 * the same pitch.
 * </pre>
 */
public static final char equalPitchChar = 'E';

/**
 * <pre>
 * This char is used to encode that from one note to another it goes down.
 * </pre>
 */
public static final char downChar = 'D';

/**
 * <pre>
 * This char is used to encode that from the note is longer than the
```

```
 * previous one.
 * </pre>
 */
public static final char longerChar = 'L';
/**
 * <pre>
 * This char is used to encode that from the note has the same length as
 * the previous one.
 * </pre>
 */
public static final char equalLengthChar = 'E';

/**
 * <pre>
 * This char is used to encode that from the note is shorter than the
 * previous one.
 * </pre>
 */
public static final char shorterChar = 'S';

/**
 * <pre>
 * Set this variable via the public void setDebug(boolean debug) to true
 * and debugging information will be written to System.out during the
 * execution of the routines.
 * </pre>
 */
private boolean debug = false;

/**
 * <pre>
 * Returns whether debug is on or off.
 * </pre>
 *
 * @return Returns the debug.
 */
public boolean isDebug() {
    return debug;
}

/**
 * <pre>
 * Sets debug to on or off. See the description of debug for more detail.
 * </pre>
 *
 * @param debug The debug to set.
 */
public void setDebug(boolean debug) {
    this.debug = debug;
}


/**
 * <pre>
 * This turns on or off whether the motifs are weighted according to their
 * occurance in the database. This affects the method public PieceScorePair[]
 * searchForPairs(Piece melodyFragment) and all methods that call this
 * method. If weighting is turnd on, more common motifs are counted less.
 * </pre>
 */
private boolean withWeighting = true;

/**
 * <pre>
 * This returns whether withWeighting is set to on or off.
 * </pre>
```

```
     *
     * @return Returns the withWeighting.
     */
    public boolean isWithWeighting() {
        return withWeighting;
    }

    /**
     * <pre>
     * This sets the withWeighting. Set it to true to turn on weighting of
     * motifs according to their number of occurance in the database.
     * </pre>
     *
     * @param withWeighting The withWeighting to set.
     */
    public void setWithWeighting(boolean withWeighting) {
        this.withWeighting = withWeighting;
    }


    /**
     * <pre>
     * This small class stores how many times motifs with the certain contur
     * are in a piece or database. Two informations are stored: The counter and
     * the contur itself as hashCode.
     * </pre>
     *
     * @author Christian Datzko
     */
    static class ClassCounter {

        /**
         * <pre>
         * Creates a new ClassCounter with the contur hash supplied.
         * </pre>
         *
         * @param hash the contur to count
         */
        public ClassCounter(int hash) {
            hashCode = hash;
        }

        /**
         * The counter of a contur class starts with one.
         */
        private int counter = 1;

        /**
         * <pre>
         * Increase the counter by one.
         * </pre>
         */
        public void increaseCounter() {
            counter++;
        }

        /**
         * <pre>
         * How often does this contur occur in the motifs?
         * </pre>
         *
         * @return Returns the counter.
         */
        public int getCounter() {
            return counter;
        }
```

```
    /**
     * <pre>
     * Sets the counter to a new value.
     * </pre>
     *
     * @param counter The counter to set.
     */
    public void setCounter(int counter) {
        this.counter = counter;
    }

    /**
     * The hash code of this class is the contur of the contur class.
     */
    private int hashCode;

    /* (non-Javadoc)
     * @see java.lang.Object#hashCode()
     */
    public int hashCode() {
        return hashCode;
    }
}

/**
 * <pre>
 * This class stores all relevant information about an indexed piece that
 * might be needed again.
 *
 * The Map classifiedSegments contains ClassCounters
 * that count the occurances of all motifs that belong to a certain contur.
 * The contur is used as hash function in the map.
 *
 * The segmentation contains a pre-calculated segmentation of the piece.
 *
 * The numberOfSegments included the total number of segments (which can
 * and most probably will be smaller than the number of unique (according
 * to its contur) segments.
 * </pre>
 *
 * @author Christian Datzko
 */
class IndexedPiece {

    /**
     * <pre>
     * Creates a new IndexedPiece. You need to supply a map with the
     * classified segments and the segments itself. The number of
     * segments is calculated.
     * </pre>
     *
     * @param clsSeg
     * @param segmts
     */
    public IndexedPiece(Map clsSeg, Segmentation segmts) {
        classifiedSegments = clsSeg;
        segments = segmts;
        Iterator iter = clsSeg.values().iterator();
        numberOfSegments = 0;
        while (iter.hasNext()) {
            ClassCounter classCounter = (ClassCounter) iter.next();
            numberOfSegments += classCounter.getCounter();
        }
    }
```

```
    /**
     * <pre>
     * Here the Map with the classified segments is stored.
     * </pre>
     */
    private Map classifiedSegments;

    /**
     * <pre>
     * Returns the classified segments.
     * </pre>
     *
     * @return Returns the classifiedSegments.
     */
    public Map getClassifiedSegments() {
        return classifiedSegments;
    }

    /**
     * <pre>
     * Here the segmentation is stored.
     * </pre>
     */
    private Segmentation segments;

    /**
     * <pre>
     * Returns the segmentation.
     * </pre>
     *
     * @return Returns the segments.
     */
    public Segmentation getSegments() {
        return segments;
    }

    /**
     * <pre>
     * Here the number of segments in segments is stored. This value is
     * calculated in the constructor.
     * </pre>
     */
    private int numberOfSegments;

    /**
     * <pre>
     * Returns the numbers of segments in segments. This is NOT the number
     * of unique segments!
     * </pre>
     *
     * @return Returns the numberOfSegments
     */
    public int getNumberOfSegments() {
        return numberOfSegments;
    }
}

/**
 * <pre>
 * In this ArrayList for each piece in the database an IndexedPiece is
 * stored. The position in the ArrayList corresponds to the index in the
 * database.
 * </pre>
 */
private ArrayList indexArrayList;
```

```
/**
 * <pre>
 * Here is kept track of the number of how often a motif (according to its
 * contur) occurs in the database. This is useful when weighting a match
 * whether a rather rare or a rather common motif is found.
 * </pre>
 */
private Map motifCounter;


/**
 * <pre>
 * This method is for debugging and returns the classes and the numbers of
 * motifs in the motifCounter, the global motif counter of all the motifs
 * in the database.
 * </pre>
 */
public void printMotifCounter() {
    Iterator iter = motifCounter.values().iterator();
    while (iter.hasNext()) {
        ClassCounter cc = (ClassCounter) iter.next();
        System.out.println(cc.getCounter() + "x " + cc.hashCode());
    }
}

/**
 * This is the class that is called during search to compare two motifs.
 */
private MotifComparator motifComparator;


/**
 * <pre>
 * The threshhold from when on a piece is considered as a match. Should be
 * below or equal to 1.0 and should be greater than or equal to 0.0 to piece.
 * Default is set to defaultThreshhold.
 * </pre>
 */
private double threshhold;

/**
 * The threshhold from when on a piece is considered as a match. Should be
 * below or equal to 1.0 and should be greater than or equal to 0.0 to piece.
 * Default is set to defaultThreshhold.
 *
 * @return Returns the threshhold.
 */
public double getThreshhold() {
    return threshhold;
}

/**
 * The threshhold from when on a piece is considered as a match. Should be
 * below or equal to 1.0 and should be greater than or equal to 0.0 to piece.
 * Default is set to defaultThreshhold.
 *
 * @param threshhold The threshhold to set.
 */
public void setThreshhold(double threshhold) {
    this.threshhold = threshhold;
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 * searchFor(de.uos.fmt.musitech.music.Piece)
 */
```

```
public Piece[] searchFor(Piece melodyFragment) {
    PieceScorePair[] pieceScorePairs = searchForPairs(melodyFragment);
    Piece[] returnPieces = new Piece[pieceScorePairs.length];
    for (int i = 0; i < pieceScorePairs.length; i++) {
        returnPieces[i] = pieceScorePairs[i].piece;
    }
    return returnPieces;
}

private final static int maxCounters = 1000;

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.MelodyRetrieval#
 * searchForPairs(de.uos.fmt.musitech.music.Piece)
 */
public PieceScorePair[] searchForPairs(Piece melodyFragment) {
    // segment the melodyFragment
    Segmentation melFragSeg = segmentPiece(melodyFragment);
    Map melFragClasSeg = classifySegments(melFragSeg);

    int lowerBorder = 0;
    int higherBorder = 0;
    if (withWeighting) {
        // calculate the borders of of uniqueness - 1/3, 1/3, 1/3
        int[] numOfOccurances = new int[maxCounters];
        int totalCounters = 0;
        Iterator motifIter = motifCounter.values().iterator();
        while (motifIter.hasNext()) {
            ClassCounter cc = (ClassCounter) motifIter.next();
            if (cc.getCounter() >= maxCounters)
                numOfOccurances[maxCounters - 1]++;
            else
                numOfOccurances[cc.getCounter()]++;
            totalCounters++;
        }
        int currentTotal = 0;
        for (int i = 0; i < numOfOccurances.length; i++) {
            currentTotal += numOfOccurances[i];
            if (lowerBorder == 0 && currentTotal > (1.0 / 3.0 * totalCounters))
                lowerBorder = i;
            if (higherBorder == 0 && currentTotal > (2.0 / 3.0 * totalCounters))
                higherBorder = i;
        }
    }

    // this array of ArrayLists will contain the similar segments for each
    // searched piece - first the segment within the searched Piece and then
    // the segment within the melodyFragment

    // sum up positive scores - the higher the better, always a value
    // between 0 and 1
    double[] scores = new double[indexArrayList.size()];

    // counter for the pieces
    int c1 = 0;
    // search for similar segments

    // for all pieces
    for (int i = 0; i < indexArrayList.size(); i++) {
        c1++;
        if (debug)
            System.out.println("piece #" + c1 + ":");
        scores[i] = 0.0;
        if (indexArrayList.get(i) != null) {
            IndexedPiece searchedPiece = (IndexedPiece) indexArrayList.get(i);
            Iterator searchedPieceSegIter =
```

```
    searchedPiece.getClassifiedSegments().values().iterator();
// counter for the fragments
int c2 = 0;
// for all segments in the piece that is searched in
while (searchedPieceSegIter.hasNext()) {
    c2++;
    if (debug)
        System.out.println(" fragment #" + c2 + ":");
    ClassCounter classCounter1 =
        (ClassCounter) searchedPieceSegIter.next();
    NoteList[] noteSeq1 =
        getNoteLists(searchedPiece.getSegments(),
                classCounter1.hashCode());
    ClassCounter classCounter3 =
        (ClassCounter) motifCounter.get(new Integer(
                classCounter1.hashCode()));
    if (classCounter3 == null) {
        System.out.println("error");
    }
    Iterator melFragSegIter =
        melFragClasSeg.values().iterator();
    double tempMax = 0.0;
    // for all segments in the piece that is searched for
    while (melFragSegIter.hasNext()) {
        ClassCounter classCounter2 =
            (ClassCounter) melFragSegIter.next();
        if (debug) {
            System.out.println("  searchd:" +
                    classCounter1.getCounter() + "x " +
                    classCounter1.hashCode);
            System.out.println("  melFrag: " +
                    classCounter2.getCounter() + "x " +
                    classCounter2.hashCode);
            System.out.print("  local score = 1/" +
                    (conturDistance(classCounter1.hashCode(),
                            classCounter2.hashCode()) + 1) +
                            " * " + classCounter1.getCounter() +
                            "/" + searchedPiece.
                            getNumberOfSegments() + " = ");
            System.out.println(((1.0 /
                    (conturDistance(classCounter1.hashCode(),
                            classCounter2.hashCode()) + 1)) *
                            classCounter1.getCounter() /
                            searchedPiece.
                            getNumberOfSegments()));
        }
        // rough search:
        // (1 / Edit Distance) *
        // (#segmente mit Kontur x / #segmente insgesamt)
        double localScore = ((1.0 / (conturDistance(
                classCounter1.hashCode(),
                classCounter2.hashCode()) + 1)) *
                classCounter1.getCounter() /
                searchedPiece.getNumberOfSegments());

        if (withWeighting) {
            // weighting for uniqueness of the motif
            if (classCounter3.getCounter() > higherBorder)
                localScore *= 0.5;
            else
                if (!(classCounter3.getCounter() <=
                    lowerBorder))
                    localScore *= 0.75;
        }

        // fine search:
```

```
                    NoteList[] noteSeq2 =
                        getNoteLists(melFragSeg, classCounter2.hashCode());

                    double localFineMax = 0.0;
                    for (int j = 0; j < noteSeq1.length; j++)
                        for (int k = 0; k < noteSeq2.length; k++) {
                            localFineMax =
                                Math.max(localFineMax,
                                        motifComparator.motifDistance(
                                                noteSeq1[j], noteSeq2[k]));
                        }

                    if (debug)
                        System.out.println("  local fine max = " +
                                localFineMax);
                    localScore *= localFineMax;

                    tempMax = Math.max(tempMax, localScore);
                }
                if (debug)
                    System.out.println("  tempMax = " + tempMax);
                scores[i] += tempMax;
            }
        }
        if (debug)
            System.out.println(" score for piece #" + c1 + " = " +
                    scores[i]);
    }

    PieceScorePair[] wsp = new PieceScorePair[indexArrayList.size()];
    for (int i = 0; i < wsp.length; i++) {
        wsp[i] = new PieceScorePair();
        wsp[i].piece = database.getPiece(i + 1);
        wsp[i].score = scores[i];
    }
    sortPieceScorePairs(wsp, 0, indexArrayList.size() - 1);

    int i = 0;
    while ((i < indexArrayList.size()) && (wsp[i].score > threshhold))
        i++;

    PieceScorePair[] wsp2 = new PieceScorePair[i];

    for (int j = 0; j < i; j++) {
        wsp2[j] = wsp[j];
    }

    return wsp2;
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#
 * insertPiece(de.uos.fmt.musitech.music.Piece, int)
 */
public void insertPiece(Piece piece, int newIndex) {
    // generate abstracts and add piece to array
    Segmentation segmentation = segmentPiece(piece);
    Map clsSeg = classifySegments(segmentation);
    IndexedPiece indexedPiece = new IndexedPiece(clsSeg, segmentation);
    indexArrayList.add(newIndex - 1, indexedPiece);

    // add motifs to motif counter
    Iterator iter = clsSeg.values().iterator();
    while (iter.hasNext()) {
        ClassCounter classCounter = (ClassCounter) iter.next();
        ClassCounter mc = (ClassCounter) motifCounter.get(
```

```
                new Integer(classCounter.hashCode()));
        if (mc == null) {
            mc = new ClassCounter(classCounter.hashCode());
            mc.setCounter(0);
        }
        mc.setCounter(classCounter.getCounter() + mc.getCounter());
        motifCounter.put(new Integer(classCounter.hashCode()), mc);
    }
}

/* (non-Javadoc)
 * @see ch.datzko.melodyRetrieval.PieceDatabaseChangeListener#deletePiece(int)
 */
public void deletePiece(int index) {
    // delete motifs from motifCounter
    IndexedPiece indexedPiece = (IndexedPiece) indexArrayList.get(index - 1);
    Iterator iter = indexedPiece.getClassifiedSegments().values().iterator();
    while (iter.hasNext()) {
        ClassCounter classCounter = (ClassCounter) iter.next();
        ClassCounter mc = (ClassCounter) motifCounter.get(
                new ClassCounter(classCounter.hashCode()));
        if (mc != null)
            mc.setCounter(mc.getCounter() - classCounter.getCounter());
        motifCounter.put(mc, mc);
    }

    // delete entry and add an empty entry
    indexArrayList.remove(index - 1);
    indexArrayList.add(index - 1, null);
}

/**
 * <pre>
 * This static method encodes a melody contur. The entered string must be
 * of the format (ab)* where a is either "D", "E" or "U" and b is either
 * "L", "E" or "S". The first character represents the change in pitch and
 * the second character represents the change in length. The returned int
 * is a bitwise encoding of the contur string with the following mappings:
 *
 * D -> 01
 * E -> 10
 * U -> 11
 * L -> 01
 * E -> 10
 * S -> 11
 *
 * The earlier characters are encoded in higher bits, thus the String
 * "DEDS" would encode to 01100111.
 * </pre>
 *
 * @param s a syntactical correct contur string
 * @return an integer representation of the contur string
 */
public static int encodeContur(String s) {
    int i = 0;
    int j = 0;
    while ((j < s.length()) && (j < 16)) {
        i <<= 2;
        switch (s.charAt(j)) {
            case upChar :
                i += 1;
                break;
            case equalPitchChar :
                i += 2;
                break;
            case downChar :
```

```
                i += 3;
                break;
            default :
                throw new IllegalArgumentException("This string is no " +
                        "valid contur string: " + s + " at position " + j);
        }
        i <<= 2;
        j++;
        switch (s.charAt(j)) {
            case longerChar :
                i += 1;
                break;
            case equalLengthChar :
                i += 2;
                break;
            case shorterChar :
                i += 3;
                break;
            default :
                throw new IllegalArgumentException("This string is no " +
                        "valid contur string: " + s + " at position " + j);
        }
        j++;
    }
    return i;
}

/**
 * <pre>
 * This is the counter function to encodeContur, it decodes a melody contur
 * int. The format is described in the description of the function
 * encodeContur.
 * </pre>
 *
 * @param i a correct encoded contur string
 * @return the contur string
 */
public static String decodeContur(int i) {
    String s = "";
    while (i != 0) {
        switch (i & 3) {
            case 1 :
                s = longerChar + s;
                break;
            case 2 :
                s = equalLengthChar + s;
                break;
            case 3 :
                s = shorterChar + s;
                break;
            default :
                throw new IllegalArgumentException("This int is no " +
                        "valid contur integer: " + Integer.
                        toBinaryString(i));
        }
        i >>>= 2;
        switch (i & 3) {
            case 1 :
                s = upChar + s;
                break;
            case 2 :
                s = equalPitchChar + s;
                break;
            case 3 :
                s = downChar + s;
                break;
```

```
            default :
                throw new IllegalArgumentException("This int is no " +
                        "valid contur integer: " + Integer.
                        toBinaryString(i));
        }
        i >>>= 2;
    }
    return s;
}

/**
 * <pre>
 * Calculates the edit distance of two melody contur strings. This is not
 * the edit distance of two strings since the fact is considered that two
 * characters belong to each other.
 * </pre>
 *
 * @param i1 the first contur
 * @param i2 the second contur
 * @return the edit distance between the two conturs
 */
public static int conturDistance(int i1, int i2) {
    String s1 = decodeContur(i1);
    String s2 = decodeContur(i2);

    int[][] distances = new int[s1.length() / 2 + 1][s2.length() / 2 + 1];
    for (int i = 0; i < s1.length() / 2 + 1; i++) {
        distances[i][0] = i;
    }
    for (int i = 0; i < s2.length() / 2 + 1; i++) {
        distances[0][i] = i;
    }


    for (int i = 0; i < s1.length() / 2; i++) {
        for (int j = 0; j < s2.length() / 2; j++) {
            distances[i + 1][j + 1] = Math.min(distances[i][j + 1] + 1,
                    Math.min(distances[i + 1][j] + 1, distances[i][j] +
                            (((s1.charAt(i * 2) == s2.charAt(j * 2)) &&
                                    (s1.charAt(i * 2 + 1) ==
                                        s2.charAt(j * 2 + 1)))? 0 : 1)));
        }
    }

    return distances[s1.length() / 2][s2.length() / 2];
}

/**
 * <pre>
 * This helper method peels out all the NoteLists out of a segmentation
 * that have the contur contur.
 * </pre>
 *
 * @param seg a segmentation
 * @param contur the contur to match the segments
 * @return an array of NoteLists that match the contur
 */
public static NoteList[] getNoteLists(Segmentation seg, int contur) {
    Iterator iter = seg.iterator();
    ArrayList foundSeg = new ArrayList();
    while (iter.hasNext()) {
        NoteList sequence = (NoteList) iter.next();
        if (classify(sequence) == contur)
            foundSeg.add(sequence);
    }
    return (NoteList[]) foundSeg.toArray(new NoteList[]{});
```

```
}

/**
 * <pre>
 * Calculates a contur from a NoteList. The contur saves two
 * informations for each note change: 1. whether the melody goes up,
 * stays on the same pitch or goes down and 2. whether the note is longer,
 * equal length of shorter than the previous one. The contur is encoded as
 * an integer using the encodeContur method described above.
 * </pre>
 *
 * @param seq the sequence to analyse
 * @return the contur of seq
 */
public static int classify(NoteList seq) {
    String s = "";
    Iterator iter = seq.iterator();
    if (!iter.hasNext()) {
        return 0;
    }
    else {
        Note note = (Note) iter.next();
        int i = 0;
        while (iter.hasNext()) {
            int previousPitch = note.getPerformanceNote().getPitch();
            long previousLength = note.getPerformanceNote().getDuration();
            note = (Note) iter.next();
            if (note.getPerformanceNote().getPitch() > previousPitch)
                s = s + upChar;
            else
                if (note.getPerformanceNote().getPitch() == previousPitch)
                    s = s + equalPitchChar;
                else
                    s = s + downChar;

            if (note.getPerformanceNote().getDuration() > previousLength)
                s = s + longerChar;
            else
                if (note.getPerformanceNote().getDuration() == previousLength)
                    s = s + equalLengthChar;
                else
                    s = s + shorterChar;
        }
        return encodeContur(s);
    }
}

/**
 * <pre>
 * This method segments the piece piece and returns a segmentation of it.
 * This method simply calls the method Segmenter in the issm package of
 * MusicalStructure. You could use any other segmenter here, as long as
 * the same segmenter is used throughout the complete database.
 * </pre>
 *
 * @param piece
 * @return
 */
public static Segmentation segmentPiece(Piece piece) {
    // peel out all the notes of the piece
    Containable[] contents = piece.getContainerPool().getContentsRecursive();
    NoteList noteList = new NoteList(piece.getContext());
    for (int i = 0; i < contents.length; i++) {
        if (contents[i] instanceof Note) {
            noteList.add((Note) contents[i]);
        }
```

```
        }
        // segment
        issm.Segmenter segmenter = new issm.Segmenter();
        return segmenter.segment(noteList);
    }

    /**
     * <pre>
     * This method classifies all segments of a given segmentation. The method
     * classify is called to do the classification via the contur of each
     * segment. Returned is a map of ClassCounters that count the occurances
     * of a contur in the segments (motifs) of the segmentation.
     * </pre>
     *
     * @param segmentation the segmentation to analyse
     * @return a map containing ClassCounters
     */
    public static Map classifySegments(Segmentation segmentation) {
        Map counters = new HashMap();
        Iterator iter = segmentation.iterator();
        while (iter.hasNext()) {
            NoteList noteList = (NoteList) iter.next();
            int sequenceClass = classify(noteList);
            ClassCounter classCounter =
                (ClassCounter) counters.get(new Integer(sequenceClass));
            if (classCounter == null)
                classCounter = new ClassCounter(sequenceClass);
            else
                classCounter.increaseCounter();
            counters.put(new Integer(sequenceClass), classCounter);
        }
        return counters;
    }
}
```

A.6 MotifComparator.java

```
/*
 * Created on 31.03.2004
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 * <pre>
 * This class is used in MotifMelodyRetrieval to compare two segments
 * represented in notelists to each other.
 * </pre>
 *
 * @author Christian Datzko
 */
public interface MotifComparator {

    /**
     * <pre>
     * This returns the distance between seq1 and seq2.
     * </pre>
     *
     * @param seq1 the first sequence
     * @param seq2 the second sequence to compare seq1 to
     * @return value between 0.0 and 1.0
     */
    public double motifDistance(NoteList seq1, NoteList seq2);

}
```

A.7 AlwaysTrueMotifComparator.java

```
/*
 * Created on 16.04.2004
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 * <pre>
 * This MotifComparator always returns 1.0 regardless of what NoteLists are
 * being put into the method public double motifDistance(...). This is useful
 * to test the MotifMelodyRetrieval when only the rough search is supposed to
 * be tested.
 * </pre>
 *
 * @author Christian Datzko
 */
public class AlwaysTrueMotifComparator implements MotifComparator {

    /* (non-Javadoc)
     * @see ch.datzko.melodyRetrieval.MotifComparator#
     * motifDistance(de.uos.fmt.musitech.music.NoteList,
     * de.uos.fmt.musitech.music.NoteList)
     */
    public double motifDistance(NoteList seq1, NoteList seq2) {
        return 1.0;
    }
}
```

A.8 CubyHumMotifComparator.java

```
/*
 * Created on 16.04.2004
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 * <pre>
 * This MotifComparator calls the method compareMelodies in
 * CubyHumMelodyRetrieval and returns this result.
 * </pre>
 *
 * @author Christian Datzko
 */
public class CubyHumMotifComparator implements MotifComparator {

    /**
     * <pre>
     * The CubyHumMelodyRetrieval class to call when comparing
     * </pre>
     */
    CubyHumMelodyRetrieval cubyHumMelodyRetrieval;


    /**
     * <pre>
     * Creates a new CubyHumMotifComparator using the supplied
     * CubyHumMelodyRetrieval class. Most of the time you might want to use the
     * other constructor which creates its own CubyHumMelodyRetrieval class.
     * </pre>
     *
     * @param chmr the CubyHumMelodyRetrieval class to use
     */
    public CubyHumMotifComparator(CubyHumMelodyRetrieval chmr) {
        super();
        cubyHumMelodyRetrieval = chmr;
    }

    /**
     * <pre>
     * Creates a new CubyHumMotifComparator and creates its own
     * CubyHumMelodyRetrieval class.
     * </pre>
     */
    public CubyHumMotifComparator() {
        this(new CubyHumMelodyRetrieval());
    }

    /* (non-Javadoc)
     * @see ch.datzko.melodyRetrieval.MotifComparator#
     * motifDistance(de.uos.fmt.musitech.music.NoteList,
     * de.uos.fmt.musitech.music.NoteList)
     */
    public double motifDistance(NoteList seq1, NoteList seq2) {
        Piece piece1 = new Piece();
        Piece piece2 = new Piece();

        piece1.setNotePool(seq1);
        piece2.setNotePool(seq2);

        // 1) how many minimum errors to fit seq2 in seq1
```

```
        double score1 = 1.0;
        try {
            score1 = cubyHumMelodyRetrieval.compareMelodies(
                    cubyHumMelodyRetrieval.generateMelodyRepresentation(piece1),
                    cubyHumMelodyRetrieval.getDurations(piece1),
                    cubyHumMelodyRetrieval.generateMelodyRepresentation(piece2),
                    cubyHumMelodyRetrieval.getDurations(piece2));

        } catch (Exception e) {
        }
        score1 /= cubyHumMelodyRetrieval.getDurations(piece1).length;
        if (score1 < 0.0)
            score1 = 0.0;
        else
            if (score1 > 1.0)
                score1 = 1.0;

            // 2) how many minimum errors to fit seq1 in seq2
        double score2 = 1.0;
        try {

        } catch (Exception e) {
            score2 = cubyHumMelodyRetrieval.compareMelodies(
                    cubyHumMelodyRetrieval.generateMelodyRepresentation(piece2),
                    cubyHumMelodyRetrieval.getDurations(piece2),
                    cubyHumMelodyRetrieval.generateMelodyRepresentation(piece1),
                    cubyHumMelodyRetrieval.getDurations(piece1));
        }
        score2 /= cubyHumMelodyRetrieval.getDurations(piece2).length;
        if (score2 < 0.0)
            score2 = 0.0;
        else
            if (score2 > 1.0)
                score2 = 1.0;

            // return minimum error to fit one seq in the other using the
            // CubyHumMelodyRetrieval algorithm
        return 1.0 - ((score1 < score2) ? score1 : score2);
    }
}
```

A.9 IdentityMotifComparator.java

```
/*
 * Created on 31.03.2004
 */
package ch.datzko.melodyRetrieval;

import java.util.Iterator;

import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.form.NoteList;

/**
 *
 * <pre>
 * This class is a motif comparator to be used with MotifMelodyRetrieval.java
 * in the same package, it returns 1 if the two NoteLists contain the same notes
 * and 0 if the NoteLists contain different notes.
 *
 * The identity is checked only for the intervals between the notes and the
 * relative rhythms. This class returns 1 if seq2 is a transposition of seq1
 * or if seq2 is a rhythmically streched or condensed version of seq1!
 * </pre>
 *
 * @author Christian Datzko
 */
public class IdentityMotifComparator implements MotifComparator {

    /* (non-Javadoc)
     * @see ch.datzko.melodyRetrieval.MotifComparator#motifDistance(null, null)
     */
    public double motifDistance(NoteList seq1, NoteList seq2) {
        double score = 1.0;

        Iterator iter1 = seq1.iterator();
        Iterator iter2 = seq2.iterator();

        // one sequence is empty
        if (!iter1.hasNext() || !iter2.hasNext())
            score = 0.0;
        else {
            Note note11 = (Note) iter1.next();
            Note note12 = (Note) iter2.next();
            while (iter1.hasNext()) {
                Note note21 = (Note) iter1.next();
                // different lengths of the sequences
                if (!iter2.hasNext())
                    score = 0.0;
                else {
                    Note note22 = (Note) iter2.next();
                    // different intervals
                    if ((note21.getPerformanceNote().getPitch() -
                            note11.getPerformanceNote().getPitch())
                            != (note22.getPerformanceNote().getPitch() -
                                    note12.getPerformanceNote().getPitch()))
                        score = 0.0;
                    // different relative lengths
                    if (((double)note21.getPerformanceNote().getDuration() /
                            (double)note11.getPerformanceNote().getDuration())
                            !=  ((double)note22.getPerformanceNote().getDuration() /
                                    (double)note12.getPerformanceNote().getDuration()))
                        score = 0.0;
                }
            }
            // different lengths of the sequences
```

```
            if (iter2.hasNext())
                score = 0.0;
        }
        return score;
    }
}
```

A.10 Segmenter.java

```
/*
 * File Segmenter.java
 */

package issm;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import structure.Segmentation;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.form.NoteList;


/**
 * This class is a basic segmenter for NoteLists, seting boundaries at
 * the relative maxima of notes' inter-onset-intervals.
 * @author Tillman Weyde
 */
public class Segmenter {

    // the minimal and maximal number of notes in a segment.
    int minLength = 1;
    int maxLength = 5;

    // the maximal duration of a segment in milliseconds.
    int maxDuration = 1600;

    /**
     * This method segments the given NoteList by
     * distributing its notes into new segments (Containers).
     * @param list The container with the notes to segment.
     * @return the generated segmentation.
     */
    public Segmentation segment( NoteList list) {
        ArrayList boundaries = new ArrayList();
        // special cases
        // cases size() 0 or 1
        if(list.size()<2)
            return segment(list,boundaries);
        // count note positions in relations to current position (starting with 4).
        Note note1 = (Note) list.get(0); // note at position -3
        Note note2 = (Note) list.get(1); // note at position -2
        if(list.size()==2) {
            if(note2.getTime() - note1.getTime() > maxDuration) {
                boundaries.add(new Integer(1));
            }
            return segment(list,boundaries);
        }
        Note note3 = (Note) list.get(2);
        long dist1 = note2.getTime() - note1.getTime();
        long dist2 = note3.getTime() - note2.getTime();
        if(list.size()==3) {
            if(dist1 < dist2) {
                boundaries.add(new Integer(1));
            }
            else {
                boundaries.add(new Integer(2));
            }
        }
        Note note4 = (Note) list.get(3);
        long dist3 = note4.getTime() - note3.getTime();
```

```
    if(dist2>dist3 && dist2>dist1) {
        boundaries.add(new Integer(2));
    }
    for (int i = 4; i < list.size(); i++) {
        note1 = note2;
        note2 = note3;
        note3 = note4;
        note4 = (Note) list.get(i);
        dist1 = dist2;
        dist2 = dist3;
        dist3 = note4.getTime() - note3.getTime();
        if(dist2>dist3 && dist2>dist1) {
            boundaries.add(new Integer(i-1));
        }
    }
    return segment(list,boundaries);
}

/**
 * Generates a segmentation from noteList using a list of boundaries.
 * @param list the notes to segement
 * @param boundaries the boundaries at which to segment
 * @return the resulting segmentation
 */
private Segmentation segment(NoteList list, List boundaries){
    Segmentation sgmtn = new Segmentation(list.getContext());
    // first segment starts at 0
    int noteIndex=0;
    // iterate through the boundaries
    for (Iterator iter = boundaries.iterator(); iter.hasNext(); ) {
        int nextBoundary = ((Integer) iter.next()).intValue();
        NoteList nl = makeSegment(list,noteIndex,nextBoundary);
        noteIndex += nl.size();
        sgmtn.add(nl);
    }
    // the last segment end at the last note
    sgmtn.add(makeSegment(list,noteIndex,list.size()));
    return sgmtn;
}

/**
 * Makes a segment (as NoteList) form list starting a noteIndex (inclusive) and
 * ending at nextBoundary (exclusive).
 * @param list The NoteList from which to take the segment.
 * @param noteIndex the start index of the segment (inclusive).
 * @param nextBoundary the end index of the segment (exclusive).
 * @return A <code>NoteList</code> containing the note of this segment.
 */
private NoteList makeSegment(NoteList list, int noteIndex, int nextBoundary){
    NoteList segment = new NoteList();
    if(nextBoundary>list.size()){
        System.out.println(this.getClass()+
        ".makeSegement(NoteList,int,int) called with nextBoundary ("+
        nextBoundary+") out of range.");
        nextBoundary = list.size();
    }
    for (;noteIndex<nextBoundary;noteIndex++) {
        segment.add(list.get(noteIndex));
    }
    return segment;
}

public static void main(String[] args) {
    NoteList test1 =  new NoteList("4c2d4e2f4g");
    Segmenter segmenter = new Segmenter();
    Segmentation sgmtn = segmenter.segment(test1);
```

```
        System.out.println(sgmtn);
    }

}
```

A.11 Sizeof.java

```
/*
 * Created on 26.03.2004
 */
package ch.datzko.melodyRetrieval;

/**
 * A simple class to experiment with your JVM's garbage collector
 * and memory sizes for various data types.
 *
 * Source: http://www.javaworld.com/javaworld/javatips/jw-javatip130.html
 *
 * @author <a href="mailto:vlad@trilogy.com">Vladimir Roubtsov</a>
 */
public class Sizeof {

    public static void main(String[] args) throws Exception {
        // Warm up all classes/methods we will use
        runGC();
        usedMemory();

        MelodyRetrievalUI.loadPiece("C:\\TEMP\\MID\\TESTER.MID");

        // Array to keep strong references to allocated objects
        final int count = 100;
        Object[] objects = new Object[count];

        long heap1 = 0;

        // Allocate count+1 objects, discard the first one
        for(int i = -1; i < count; ++i)
        {
            Object object = null;

            // Instantiate your data here and assign it to object

            object = MelodyRetrievalUI.loadPiece("C:\\TEMP\\DT\\humblhrt.MID");
            // Object();
            //object = new Integer(i);
            //object = new Long(i);
            //object = new String();
            //object = new byte[128][1]

            if(i >= 0)
                objects[i] = object;
            else
            {
                object = null; // Discard the warm up object
                runGC();
                heap1 = usedMemory(); // Take a before heap snapshot
            }
        }

        runGC();
        long heap2 = usedMemory(); // Take an after heap snapshot:

        final int size = Math.round(((float)(heap2 - heap1))/count);
        System.out.println("'before' heap: " + heap1 +
                ", 'after' heap: " + heap2);
        System.out.println("heap delta: " +(heap2 - heap1) +
                ", {" + objects[0].getClass() + "} size = " + size + " bytes");

        for(int i = 0; i < count; ++i) objects[i] = null;
        objects = null;
```

```
    }

    private static void runGC() throws Exception
    {
        // It helps to call Runtime.gc()
        // using several method calls:
        for(int r = 0; r < 4; ++ r) _runGC();
    }

    private static void _runGC() throws Exception
    {
        long usedMem1 = usedMemory(), usedMem2 = Long.MAX_VALUE;
        for(int i = 0;(usedMem1 < usedMem2) &&(i < 500); ++ i)
        {
            s_runtime.runFinalization();
            s_runtime.gc();
            Thread.yield();

            usedMem2 = usedMem1;
            usedMem1 = usedMemory();
        }
    }

    private static long usedMemory()
    {
        return s_runtime.totalMemory() - s_runtime.freeMemory();
    }

    private static final Runtime s_runtime = Runtime.getRuntime();

} // End of class}
```

A.12 MelodyRetrievalUI.java

```
/*
 * Created on 02.03.2004
 */
package ch.datzko.melodyRetrieval;

import java.io.File;
import java.io.FilenameFilter;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.Iterator;
import java.util.Map;

import structure.Segmentation;
import de.uos.fmt.musitech.data.structure.Note;
import de.uos.fmt.musitech.data.structure.Piece;
import de.uos.fmt.musitech.data.structure.form.NoteList;
import de.uos.fmt.musitech.performance.midi.MidiReader;

/**
 * <pre>
 * This class is a simple text-based user interface so that the MelodyRetrieval
 * classes can be tested. You can load files, directories and MelodyRetrieval
 * classes and search for a melody in the database using the methods of the
 * different MelodyRetrieval classes. You can also print out the current
 * contents of the database.
 *
 * It also offers three static methods for simple loading of a file, a ressource
 * or all MIDI files (*.mid) in a directory into a database.
 * </pre>
 *
 * @author Christian Datzko
 */
public class MelodyRetrievalUI {

    /**
     * <pre>
     * Loads a ressource into the database using the getRessource method.
     * </pre>
     *
     * @param ressourceName the ressource to load, for example "mid/tester.mid"
     * @param db the database to add the piece to
     */
    public static void loadRessourceToDatabase(String ressourceName,
            PieceDatabase db) {
        URL url = MelodyRetrievalUI.class.getResource(ressourceName);
        loadURLToDatabase(url, db);
    }

    /**
     * <pre>
     * Loads a URL into the database. Catches the RuntimeException that can be
     * thrown by the MidiReader, if the file is not found or could not be loaded
     * for any other reason.
     * </pre>
     *
     * @param url the url to the piece to load
     * @param db the database to add the piece to
     */
    public static void loadURLToDatabase(URL url, PieceDatabase db) {
        MidiReader midiReader = new MidiReader();
        Piece piece = null;
        try {
            piece = midiReader.getWork(url);
```

```
        } catch (RuntimeException e) {
            System.err.println("Could not load " + url);
            e.printStackTrace();
        }
        if (piece != null)
            db.insertPiece(piece);
    }

    /**
     * <pre>
     * This simple class is a FilenameFilter used to sort out the MIDI files in
     * a directory.
     * </pre>
     *
     * @author Christian Datzko
     */
    public static class MIDIFilter implements FilenameFilter {

        /**
         * <pre>
         * Returns true if name ends with ".mid", case ignored.
         * </pre>
         *
         * @param dir the directory in which the file was found - ignored
         * @param name the name of the file
         * @return true, if the file ends with ".mid", case ignored, false
         * otherwise
         */
        public boolean accept(File dir, String name) {
            return name.toLowerCase().endsWith(".mid");
        }
    }

    /**
     * <pre>
     * Loads all MIDI files in a directory (without subdirectories) to the
     * database.
     * </pre>
     *
     * @param directory the directory in which to search for MIDI files
     * @param db the database to add the pieces to
     */
    public static void loadDirectoryToDatabase(String directory,
            PieceDatabase db) {
        if (!directory.endsWith("\\") && !directory.endsWith("/"))
            directory = directory + "/";
        File dir = new File(directory);
        String[] files = dir.list(new MIDIFilter());
        for (int i = 0; (files != null) && i < files.length; i++) {
            try {
                loadURLToDatabase(new URL("file:/" + directory + files[i]),
                        db);
            }
            catch (MalformedURLException e) {
                System.err.println("Could not load " + "file:/" + directory +
                        files[i]);
                e.printStackTrace();
            }
        }
    }

    /**
     * <pre>
     * Loads a piece from a given filename and returns it.
     * </pre>
     *
```

```
 * @param filename the file name to load
 * @return the loaded piece
 */
public static Piece loadPiece(String filename) {
    String fn = filename;
    if (fn == null) {
        System.err.println("Coud not load not specified file.");
        return null;
    }
    else {
        MidiReader midiReader = new MidiReader();
        Piece piece = null;
        if (!fn.startsWith("file:/"))
            fn = "file:/" + fn;
        try {
            piece = midiReader.getWork(new URL(fn));
        } catch (MalformedURLException e) {
            System.err.println("Could not load " + fn + ".");
            e.printStackTrace();
        }
        return piece;
    }
}

/**
 * <pre>
 * a buffered reader for reading from standard IO
 * </pre>
 */
static final java.io.BufferedReader system_in = new java.io.BufferedReader(
        new java.io.InputStreamReader(System.in));

/**
 * <pre>
 * Reads a line of text from System.in.
 * </pre>
 *
 * @return the next input line; null at end of file (or when an error
 * occurs).
 */
static String readLine() {
    try
    {
        return system_in.readLine();
    }
    catch (java.io.IOException e)
    {
        System.err.println(e);
        return null;
    }
}

/**
 * <pre>
 * the database to piece with
 * </pre>
 */
static PieceDatabase database = new PieceDatabase();

/**
 * <pre>
 * maximum MelodyRetrievals to be loaded
 * </pre>
 */
static final int maxMelodyRetrievals = 10;
```

```
/**
 * <pre>
 * the MelodyRetrieval classes
 * </pre>
 */
static MelodyRetrieval[] melodyRetrievals =
    new MelodyRetrieval[maxMelodyRetrievals];

/**
 * <pre>
 * how many classes are already loaded
 * </pre>
 */
static int melodyRetrievalsCount = 0;

/**
 * <pre>
 * This is the main menu. It loops until exit is chosen and calls all other
 * menu methods.
 * </pre>
 */
static void mainMenu() {
    final int exitAction = 7;
    int action;
    do {
        System.out.println();
        System.out.println("-------------------- main menu --------------" +
        "------");
        System.out.println(" 1) load MIDI file into database");
        System.out.println(" 2) load all MIDI files of a directory into " +
        "database");
        System.out.println(" 3) load MelodyRetrieval class");
        System.out.println(" 4) print Database");
        System.out.println(" 5) search for MIDI file");
        System.out.println(" 6) do MelodyRetrieval-specific tasks");
        System.out.println(" 7) exit");
        String input = readLine();
        if (input == null)
            action = 0;
        else {
            try {
                action = Integer.parseInt(input);
            }
            catch (NumberFormatException e) {
                action = 0;
            }
            switch (action) {
                case 1 :
                    System.out.println("1) load MIDI file into database");
                    menuLoadMIDI();
                    break;
                case 2:
                    System.out.println("2) load all MIDI files of a " +
                    "directory into database");
                    menuLoadDirectory();
                    break;
                case 3:
                    System.out.println("3) load MelodyRetrieval class");
                    menuLoadMelodyRetrieval();
                    break;
                case 4:
                    System.out.println("4) print database");
                    menuPrintIndexDatabase();
                    break;
                case 5:
                    System.out.println("5) search for MIDI file");
```

```
                    menuSearchForMIDI();
                    break;
                case 6:
                    System.out.println("6) do MelodyRetrieval-specific " +
                    "tasks");
                    menuMelodyRetrievalSpecific();
                    break;
                case 7:
                    System.out.println("7) exit");
                    break;
                default :
                    break;
            }

        }
    } while (action != exitAction);
}

/**
 * <pre>
 * Sub menu to load a MIDI file into the database.
 * </pre>
 */
static void menuLoadMIDI() {
    System.out.println();
    System.out.print("Please enter a file to load>");
    String input = readLine();
    if (input == null) {
        System.err.println("Error reading file name.");
    }
    else {
        if (!input.startsWith("file:/"))
            input = "file:/" + input;
        try {
            loadURLToDatabase(new URL(input), database);
        }
        catch (MalformedURLException e) {
            System.err.println("Could not load " + input);
            e.printStackTrace();
        }
    }
}

/**
 * <pre>
 * Sub menu file to load all MIDI files in a directory into the database.
 * </pre>
 */
static void menuLoadDirectory() {
    System.out.println();
    System.out.print("Please enter a directory to load the files from>");
    String input = readLine();
    if (input == null) {
        System.err.println("Error reading file name.");
    }
    else {
        loadDirectoryToDatabase(input, database);
    }
}

/**
 * <pre>
 * Sub menu file to load a MelodyRetrieval class and attach it to the
 * database.
 * </pre>
 */
```

```
static void menuLoadMelodyRetrieval() {
    System.out.println();
    System.out.println("------------------- select menu -------------" +
    "------");
    System.out.println(" 1) ParsonsMelodyRetrieval (Denys Parsons)");
    System.out.println(" 2) CubyHumMelodyRetrieval (Steffen Pauws)");
    System.out.println(" 3) MotifMelodyRetrieval with AlwaysTrueMotifComp" +
    "arator (Christian Datzko)");
    System.out.println(" 4) MotifMelodyRetrieval with IdentityMotifCompar" +
    "ator (Christian Datzko)");
    System.out.println(" 5) MotifMelodyRetrieval with CubyHumMotifCompara" +
    "tor (Christian Datzko)");
    String input = readLine();
    int action;
    if (input == null)
        action = 0;
    else {
        try {
            action = Integer.parseInt(input);
        }
        catch (NumberFormatException e) {
            action = 0;
        }
        switch (action) {
            case 1 :
                if (melodyRetrievalsCount < melodyRetrievals.length) {
                    melodyRetrievals[melodyRetrievalsCount] =
                        new ParsonsMelodyRetrieval(database);
                    melodyRetrievalsCount++;
                }
                break;
            case 2:
                if (melodyRetrievalsCount < melodyRetrievals.length) {
                    melodyRetrievals[melodyRetrievalsCount] =
                        new CubyHumMelodyRetrieval(database);
                    melodyRetrievalsCount++;
                }
                break;
            case 3:
                if (melodyRetrievalsCount < melodyRetrievals.length) {
                    melodyRetrievals[melodyRetrievalsCount] =
                        new MotifMelodyRetrieval(database,
                                new AlwaysTrueMotifComparator());
                    melodyRetrievalsCount++;
                }
                break;
            case 4:
                if (melodyRetrievalsCount < melodyRetrievals.length) {
                    melodyRetrievals[melodyRetrievalsCount] =
                        new MotifMelodyRetrieval(database,
                                new IdentityMotifComparator());
                    melodyRetrievalsCount++;
                }
                break;
            case 5:
                if (melodyRetrievalsCount < melodyRetrievals.length) {
                    melodyRetrievals[melodyRetrievalsCount] =
                        new MotifMelodyRetrieval(database,
                                new CubyHumMotifComparator());
                    melodyRetrievalsCount++;
                }
                break;
            default :
                break;
        }
```

```
    }
}

/**
 * <pre>
 * Sub menu which prints out all the pieces in the database at the moment.
 * </pre>
 */
static void menuPrintIndexDatabase() {
    System.out.println();
    System.out.println("The database contains the following pieces:");
    PieceDatabase.PieceEnum enum = database.getFirst();
    int i = 0;
    while (enum != null) {
        System.out.println(++i + " " + enum.getPiece().getName());
        enum = enum.getNext();
    }
}

/**
 * <pre>
 * Sub menu which loads a MIDI file and searches for it in the database
 * using all already loaded MelodyRetrieval classes.
 * </pre>
 */
static void menuSearchForMIDI() {
    System.out.println();
    System.out.print("Please enter a melody file to search for>");
    String input = readLine();
    if (input == null) {
        System.err.println("Error reading file name.");
    }
    else {
        MidiReader midiReader = new MidiReader();
        Piece piece = null;
        if (!input.startsWith("file:/"))
            input = "file:/" + input;
        try {
            piece = midiReader.getWork(new URL(input));
        } catch (MalformedURLException e) {
            System.err.println("Could not load " + input);
            e.printStackTrace();
        }
        if (piece != null) {
            for (int i = 0; i < melodyRetrievalsCount; i++) {
                System.out.println(melodyRetrievals[i].getClass().
                        getName() + ":");
                MelodyRetrieval.PieceScorePair[] returnedPieces =
                    melodyRetrievals[i].searchForPairs(piece);
                for (int j = 0; j < returnedPieces.length; j++) {
                    System.out.println(returnedPieces[j].score + " " +
                            returnedPieces[j].piece.getName());
                }
            }
        }
    }
}


/**
 * <pre>
 * Sub menu to do MelodyRetrieval-specific tasks.
 * </pre>
 */
static void menuMelodyRetrievalSpecific() {
    System.out.println();
```

```
System.out.println("------------------- select menu -------------" +
"------");
System.out.println(" 1) ParsonsMelodyRetrieval (Denys Parsons)");
System.out.println(" 2) CubyHumMelodyRetrieval (Steffen Pauws)");
System.out.println(" 3) MotifMelodyRetrieval (Christian Datzko)");
String input = readLine();
int action;
if (input == null)
    action = 0;
else {
    try {
        action = Integer.parseInt(input);
    }
    catch (NumberFormatException e) {
        action = 0;
    }
    switch (action) {
        case 1 :
            System.out.println();
            System.out.println("------------------- select menu -----" +
            "--------------");
            System.out.println(" 1) printIndexDatabase()");
            System.out.println(" 2) generateIndex()");
            String input2 = readLine();
            int action2;
            if (input2 == null)
                action2 = 0;
            else {
                try {
                    action2 = Integer.parseInt(input2);
                }
                catch (NumberFormatException e) {
                    action2 = 0;
                }
                switch (action2) {
                    case 1 :
                        System.out.println("1) printIndexDatabase()");
                        simpleMelodyRetrievalprintIndexDatabase();
                        break;
                    case 2:
                        System.out.println("2) generateIndex()");
                        simpleMelodyRetrievalGenerateIndex();
                        break;
                    default :
                        break;
                }
            }
            break;
        case 2:
            System.out.println();
            System.out.println("------------------- select menu -----" +
            "--------------");
            System.out.println(" 1) generateMelodyRepresentation()");
            System.out.println(" 2) getDurations()");
            System.out.println(" 3) compareMelodies() (with debugging" +
            " information)");
            System.out.println(" 4) show differences");
            String input3 = readLine();
            int action3;
            if (input3 == null)
                action3 = 0;
            else {
                try {
                    action3 = Integer.parseInt(input3);
                }
                catch (NumberFormatException e) {
```

```
            action3 = 0;
        }
        switch (action3) {
            case 1 :
                System.out.println("1) generateMelodyRepresen" +
                "tation()");
                cubyHumGenerateMelodyRepresentation();
                break;
            case 2:
                System.out.println("2) getDurations()");
                cubyHumGetDurations();
                break;
            case 3:
                System.out.println("3) compareMelodies() (wit" +
                "h debugging information)");
                cubyHumCompareMelodies();
                break;
            case 4:
                System.out.println("4) show differences");
                cubyHumShowDifferences();
                break;
            default :
                break;
        }
    }
    break;
case 3:
    System.out.println();
    System.out.println("------------------- select menu -----" +
    "-------------");
    System.out.println(" 1) segmentPiece()");
    System.out.println(" 2) segmentPiece() + classifySegments");
    System.out.println(" 3) encodeContur()");
    System.out.println(" 4) decodeContur()");
    System.out.println(" 5) toggle debug of all MotifMelodyRe" +
    "trieval classes");
    System.out.println(" 6) toggle withWeighting of all Motif" +
    "MelodyRetrieval classes");
    System.out.println(" 7) print current motifCounter");
    String input4 = readLine();
    int action4;
    if (input4 == null)
        action4 = 0;
    else {
        try {
            action4 = Integer.parseInt(input4);
        }
        catch (NumberFormatException e) {
            action4 = 0;
        }
        switch (action4) {
            case 1 :
                System.out.println(" 1) segmentPiece()");
                motifMelodyRetrievalSegmentPiece();
                break;
            case 2:
                System.out.println(" 2) segmentPiece() + class" +
                "ifySegments");
                motifMelodyRetrievalClassifySegments();
                break;
            case 3:
                System.out.println(" 3) encodeContur()");
                motifMelodyRetrievalEncodeContur();
                break;
            case 4:
                System.out.println(" 4) decodeContur()");
```

```
                            motifMelodyRetrievalDecodeContur();
                            break;
                        case 5:
                            System.out.println(" 5) toggle debug of all M" +
                            "otifMelodyRetrieval classes");
                            motifMelodyRetrievalToggleDebug();
                            break;
                        case 6:
                            System.out.println(" 6) toggle withWeighting " +
                            "of all MotifMelodyRetrieval classes");
                            motifMelodyRetrievalToggleWithWeighting();
                            break;
                        case 7:
                            System.out.println(" 7) print current motifCo" +
                            "unter");
                            motifMelodyRetrievalPrintMotifCounter();
                            break;
                        default :
                            break;
                    }
                }
                break;
            default :
                break;
        }

    }
}

/**
 * <pre>
 * Sub method to perform ParsonsMelodyRetrieval.printIndexDatabase().
 * </pre>
 */
static void simpleMelodyRetrievalprintIndexDatabase() {
    ParsonsMelodyRetrieval mr = new ParsonsMelodyRetrieval(database);
    mr.printIndexDatabase();
}

/**
 * <pre>
 * Sub method to perform ParsonsMelodyRetrieval.generateIndex().
 * </pre>
 */
static void simpleMelodyRetrievalGenerateIndex() {
    ParsonsMelodyRetrieval mr = new ParsonsMelodyRetrieval();
    System.out.println();
    System.out.print("Please enter a melody file>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading file name.");
    }
    else {
        Piece piece = loadPiece(inputString);
        if (piece != null) {
            System.out.println(mr.generateIndex(piece));
        }
    }
}

/**
 * <pre>
 * Sub method to perform CubyHumMelodyRetrieval.
 * generateMelodyRepresentation().
 * </pre>
 */
```

```
static void cubyHumGenerateMelodyRepresentation() {
    System.out.println();
    System.out.print("Please enter a melody file>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading file name.");
    }
    else {
        MidiReader midiReader = new MidiReader();
        Piece piece = null;
        if (!inputString.startsWith("file:/"))
            inputString = "file:/" + inputString;
        try {
            piece = midiReader.getWork(new URL(inputString));
        } catch (MalformedURLException e) {
            System.err.println("Could not load " + inputString);
            e.printStackTrace();
        }
        if (piece != null) {
            CubyHumMelodyRetrieval mr = new CubyHumMelodyRetrieval();
            int[] rep = mr.generateMelodyRepresentation(piece);
            for (int i = 0; i < rep.length; i++)
                System.out.print(rep[i] + " ");
            System.out.println();
        }
    }
}

/**
 * <pre>
 * Sub method to perform CubyHumMelodyRepresentation.getDurations().
 * </pre>
 */
static void cubyHumGetDurations() {
    System.out.println();
    System.out.print("Please enter a melody file>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading file name.");
    }
    else {
        MidiReader midiReader = new MidiReader();
        Piece piece = null;
        if (!inputString.startsWith("file:/"))
            inputString = "file:/" + inputString;
        try {
            piece = midiReader.getWork(new URL(
                    inputString));
        } catch (MalformedURLException e) {
            System.err.println("Could not load " +
                    inputString);
            e.printStackTrace();
        }
        if (piece != null) {
            CubyHumMelodyRetrieval mr =
                new CubyHumMelodyRetrieval();
            long[] rep = mr.getDurations(piece);
            for (int i = 0; i < rep.length; i++)
                System.out.print(rep[i] + " ");
            System.out.println();
        }
    }
}

/**
 * <pre>
```

```
     * Sub method to perform CubyHumMelodyRetrieval.compareMelodies().
     * </pre>
     */
    static void cubyHumCompareMelodies() {
        System.out.println();
        System.out.print("Please enter a melody file to search for>");
        String inputString = readLine();
        if (inputString == null) {
            System.err.println("Error reading file name.");
        }
        else {
            Piece piece1 = loadPiece(inputString);
            if (piece1 != null) {
                System.out.println();
                System.out.print("Please enter a melody file to search in>");
                inputString = readLine();
                if (inputString == null) {
                    System.err.println("Error reading file name.");
                }
                else {
                    Piece piece2 = loadPiece(inputString);
                    if (piece2 != null) {
                        CubyHumMelodyRetrieval mr =
                            new CubyHumMelodyRetrieval();
                        mr.setDebug(true);
                        mr.compareMelodies(
                                mr.generateMelodyRepresentation(piece1),
                                mr.getDurations(piece1),
                                mr.generateMelodyRepresentation(piece2),
                                mr.getDurations(piece2));
                    }
                }
            }
        }

    }

    /**
     * <pre>
     * Sub menu to perform CubyHumHelper.showDifferences().
     * </pre>
     */
    static void cubyHumShowDifferences() {
        System.out.println();
        System.out.print("Please enter a melody file to search for>");
        String inputString = readLine();
        if (inputString == null) {
            System.err.println("Error reading file name.");
        }
        else {
            Piece piece1 = loadPiece(inputString);
            if (piece1 != null) {
                System.out.println();
                System.out.print("Please enter a melody file to search in>");
                inputString = readLine();
                if (inputString == null) {
                    System.err.println("Error reading file name.");
                }
                else {
                    Piece piece2 = loadPiece(inputString);
                    if (piece2 != null) {
                        CubyHumHelper mr = new CubyHumHelper();
                        mr.showDifferences(piece1, piece2);
                    }
                }
            }
```

```
    }
}

/**
 * <pre>
 * Sub menu to perform MotifMelodyRetrieval.segmentPiece().
 * </pre>
 */
static void motifMelodyRetrievalSegmentPiece() {
    System.out.println();
    System.out.print("Please enter a melody file to segment>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading file name.");
    }
    else {
        Piece piece = loadPiece(inputString);
        if (piece != null) {
            Segmentation segmentation =
                MotifMelodyRetrieval.segmentPiece(piece);
            Iterator iter = segmentation.iterator();
            int i = 1;
            while (iter.hasNext()) {
                NoteList noteList = (NoteList) iter.next();
                System.out.println("The notes of Segment #" + i++ + ":");
                Iterator iter2 = noteList.iterator();
                while (iter2.hasNext()) {
                    Object note = (Object) iter2.next();
                    if (note instanceof Note) {
                        System.out.println(" " + ((Note) note).
                                getPerformanceNote().getPitch() + " " +
                                ((Note) note).getPerformanceNote().
                                getDuration());
                    }
                }
            }
        }
    }
}

/**
 * <pre>
 * Sub menu to perform MotifMelodyRetrieval.classifySegments().
 * </pre>
 */
static void motifMelodyRetrievalClassifySegments() {
    System.out.println();
    System.out.print("Please enter a melody file to segment and classify>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading file name.");
    }
    else {
        Piece piece = loadPiece(inputString);
        if (piece != null) {
            Segmentation segmentation =
                MotifMelodyRetrieval.segmentPiece(piece);
            Iterator iter = segmentation.iterator();
            int i = 1;
            while (iter.hasNext()) {
                NoteList noteList = (NoteList) iter.next();
                System.out.print("The class of Segment #" + i++ + ": ");
                int classification =
                    MotifMelodyRetrieval.classify(noteList);
                System.out.println(classification + " " +
                        MotifMelodyRetrieval.decodeContur(classification));
```

```
            }
            System.out.println("The Map containing the classes:");
            Map map = MotifMelodyRetrieval.classifySegments(segmentation);
            iter = map.values().iterator();
            while (iter.hasNext()) {
                MotifMelodyRetrieval.ClassCounter cc =
                    (MotifMelodyRetrieval.ClassCounter) iter.next();
                System.out.println(cc.getCounter() + "x " + cc.hashCode() +
                        " " + MotifMelodyRetrieval.
                        decodeContur(cc.hashCode()));
            }
        }
    }
}

/**
 * <pre>
 * Sub menu to perform MotifMelodyRetrieval.encodeContur().
 * </pre>
 */
static void motifMelodyRetrievalEncodeContur() {
    System.out.println();
    System.out.print("Please enter a melody contur to encode>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading string.");
    }
    else {
        try {
            System.out.println(MotifMelodyRetrieval.encodeContur(
                    inputString));
        } catch (Exception e) {
            System.out.println("There has been an error encoding the cont" +
            "ur.");
        }
    }
}

/**
 * <pre>
 * Sub menu to perform MotifMelodyRetrieval.decodeContur().
 * </pre>
 */
static void motifMelodyRetrievalDecodeContur() {
    System.out.println();
    System.out.print("Please enter a melody contur to decode>");
    String inputString = readLine();
    if (inputString == null) {
        System.err.println("Error reading string.");
    }
    else {
        try {
            System.out.println(MotifMelodyRetrieval.
                    decodeContur(Integer.parseInt(inputString)));
        } catch (Exception e) {
            System.out.println("There has been an error encoding the cont" +
            "ur.");
        }
    }
}

/**
 * <pre>
 * Sub menu to toggle all debugs of all loaded motifMelodyRetrieval classes.
 * </pre>
 */
```

```
    static void motifMelodyRetrievalToggleDebug() {
        for (int i = 0; i < maxMelodyRetrievals; i++) {
            if (melodyRetrievals[i] instanceof MotifMelodyRetrieval)
                ((MotifMelodyRetrieval) melodyRetrievals[i]).setDebug(
                        !((MotifMelodyRetrieval)melodyRetrievals[i]).isDebug());

        }
    }

    /**
     * <pre>
     * Sub menu to toggle all withWeightings of all loaded motifMelodyRetrieval
     * classes.
     * </pre>
     */
    static void motifMelodyRetrievalToggleWithWeighting() {
        for (int i = 0; i < maxMelodyRetrievals; i++) {
            if (melodyRetrievals[i] instanceof MotifMelodyRetrieval)
                ((MotifMelodyRetrieval) melodyRetrievals[i]).setWithWeighting(
                        !((MotifMelodyRetrieval)melodyRetrievals[i]).
                        isWithWeighting());

        }
    }

    /**
     * <pre>
     * Sub menu to print out the current MotifCounter
     * </pre>
     */
    static void motifMelodyRetrievalPrintMotifCounter() {
        for (int i = 0; i < maxMelodyRetrievals; i++) {
            if (melodyRetrievals[i] instanceof MotifMelodyRetrieval) {
                MotifMelodyRetrieval mr =
                    (MotifMelodyRetrieval) melodyRetrievals[i];
                mr.printMotifCounter();
            }
        }
    }

    /**
     * <pre>
     * This method starts the main menu which performs basic MelodyRetrieval
     * operations on different MelodyRetrieval classes.
     * </pre>
     *
     * @param args ignored
     */
    public static void main(String[] args) {
        // Who am I?
        System.out.println("MelodyRetrievalUI: Simple user interface for " +
        "the package ch.datzko.melodyRetrieval.");
        // call main menu
        mainMenu();
    }
}
```

A.13 PieceDatabase.java

```
/*
 * Created on 24.11.2003
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.Piece;


/**
 * <pre>
 * Generic database for Pieces.
 *
 * This class is not (yet) thread-safe!
 * </pre>
 *
 * @author Christian Datzko
 */
public class PieceDatabase {

    /**
     * <pre>
     * Creates an empty piece database.
     * </pre>
     */
    public PieceDatabase() {
        first = null;
        firstListener = null;
    }

    /**
     * <pre>
     * Small enumeration class to save the pieces.
     *
     * An object is deleted <=> piece == null.
     * </pre>
     *
     * @author Christian Datzko
     */
    public class PieceEnum {
        /**
         * <pre>
         * Creates a new PieceEnum.
         * </pre>
         * @param newpiece the piece for the PieceEnum. Can be null although it
         * is not recommended
         */
        public PieceEnum(Piece newpiece) {
            next = null;
            piece = newpiece;
        }

        private PieceEnum next;

        /**
         * <pre>
         * Returns the next PieceEnum.
         * </pre>
         * @return the next PieceEnum
         */
        public PieceEnum getNext() {
            return next;
        }
```

```
    /**
     * <pre>
     * Sets the next PieceEnum.
     * </pre>
     * @param newnext either to add a new next PieceEnum or to get rid of
     * the next PieceEnum
     */
    public void setNext(PieceEnum newnext) {
        next = newnext;
    }

    private Piece piece;

    /**
     * <pre>
     * Gets the piece of this PieceEnum.
     * </pre>
     * @return the piece in this PieceEnum. Can be null
     */
    public Piece getPiece() {
        return piece;
    }

    /**
     * <pre>
     * Sets the piece of this PieceEnum.
     * </pre>
     * @param newpiece is either the new piece or null to delete a piece
     */
    public void setPiece(Piece newpiece) {
        piece = newpiece;
    }
}

/**
 * <pre>
 * Small enumeration class to save the listeners:
 * </pre>
 *
 * @author Christian Datzko
 */
private class ChangeListenerEnum {


    /**
     * <pre>
     * Creates a new ChangeListenerEnum.
     * </pre>
     * @param newChangeListener the PieceDatabaseChangeListener to store
     */
    public ChangeListenerEnum(
            PieceDatabaseChangeListener newChangeListener) {
        next = null;
        changeListener = newChangeListener;
    }

    /**
     * <pre>
     * The next ChangeListenerEnum
     * </pre>
     */
    private ChangeListenerEnum next;

    /**
     * <pre>
     * Get the next ChangeListenerEnum.
```

```
         * </pre>
         *
         * @return the next ChangeListenerEnum
         */
        public ChangeListenerEnum getNext() {
            return next;
        }

        /**
         * <pre>
         * Set the next ChangeListenerEnum.
         * </pre>
         *
         * @param newChangeListener the new next ChangeListenerEnum
         */
        public void setNext(ChangeListenerEnum newChangeListener) {
            next = newChangeListener;
        }

        /**
         * <pre>
         * The next PieceDatabaseChangeListener that is saved in this
         * ChangeListenerEnum.
         * </pre>
         */
        private PieceDatabaseChangeListener changeListener;

        /**
         * <pre>
         * Get the PieceDatabaseChangeListener in this ChangeListenerEnum.
         * </pre>
         *
         * @return the PieceDatabaseChangeListener
         */
        public PieceDatabaseChangeListener getChangeListener() {
            return changeListener;
        }
    }

    /**
     * <pre>
     * The first Piece in the piece enumeration.
     * </pre>
     */
    private PieceEnum first;

    /**
     * <pre>
     * Returns the first piece in the piece enumeration.
     * </pre>
     */
    public PieceEnum getFirst() {
        return first;
    }

    /**
     * <pre>
     * The first PieceDatabaseChangeListener.
     * </pre>
     */
    private ChangeListenerEnum firstListener;

    /**
     * <pre>
     * Inserts a piece into the database. Notifies all the ChangeListeners.
     * </pre>
```

```
 * @param piece the piece to insert into the database
 */
public void insertPiece(Piece piece) {
    // insert piece
    int i = 1;
    if (first == null)
        first = new PieceEnum(piece);
    else {
        i++;
        PieceEnum t = first;
        while (t.getNext() != null) {
            t = t.getNext();
            i++;
        }
        t.setNext(new PieceEnum(piece));
    }
    // notify all the ChangeListeners
    ChangeListenerEnum e = firstListener;
    while (e != null) {
        e.getChangeListener().insertPiece(piece, i);
        e = e.getNext();
    }
}

/**
 * <pre>
 * A piece is deleted simply by setting it's reference pointer to null.
 * </pre>
 *
 * @param index which piece to delete
 */
public void deletePiece(int index) {
    // delete the piece
    PieceEnum t = first;
    for (int i = 1; i <= index; i++) {
        if (t != null)
            t = t.getNext();
    }
    if (t != null)
        t.setPiece(null);
    // notify all the ChangeListeners
    ChangeListenerEnum e = firstListener;
    while (e != null) {
        e.getChangeListener().deletePiece(index);
        e = e.getNext();
    }
}

/**
 * <pre>
 * Returns the number of pieces in the piece database.
 * </pre>
 * @return the number of pieces in the piece database
 */
public int getNumberOfPieces() {
    int i = 0;
    PieceEnum t = first;
    while (t != null) {
        i++;
        t = t.getNext();
    }
    return i;
}

/**
 * <pre>
```

```
     * Gets the index^th piece of the piece database.
     * </pre>
     * @param index which piece to return
     * @return the index^th piece or null if none it there
     */
    public Piece getPiece(int index) {
        if (index < 1) return null;
        else {
            int i = index;
            PieceEnum t = first;
            while ((i > 1) && (t != null)) {
                t = t.getNext();
                i--;
            }
            if ((i == 1) && (t != null))
                return t.getPiece();
            else
                return null;
        }
    }

    /**
     * <pre>
     * Adds a listener to the database listener list who will be called when
     * the database has been changed (for reindexing).
     * </pre>
     *
     * @param listener The listener to add to the database
     */
    public void addPieceDatabaseChangeListener(
            PieceDatabaseChangeListener listener) {
        if (firstListener == null) firstListener = new ChangeListenerEnum(
                listener);
        else {
            ChangeListenerEnum t = firstListener;
            while (t.getNext() != null)
                t = t.getNext();
            t.setNext(new ChangeListenerEnum(listener));
        }
    }

    /**
     * <pre>
     * Deletes _all_ occurrances of the listener in the database listener list.
     * </pre>
     *
     * @param listener the listener to delete from the database listener list
     */
    public void deletePieceDatabaseChangeListener(
            PieceDatabaseChangeListener listener) {
        while ((firstListener != null)
                && (firstListener.getChangeListener() == listener)) {
            firstListener = firstListener.getNext();
        }
        ChangeListenerEnum t = firstListener;
        if (t != null) {
            while ((t.getNext() != null)) {
                if (t.getNext().getChangeListener() == listener)
                    t.setNext(t.getNext().getNext());
                else
                    t = t.getNext();
            }
        }
    }
}
```

A.14 PieceDatabaseChangeListener.java

```
/*
 * Created on 24.11.2003
 */
package ch.datzko.melodyRetrieval;

import de.uos.fmt.musitech.data.structure.Piece;

/**
 * <pre>
 * Interface for listeners for the PieceDatabase. Called whenever the database
 * is changed so that indices can be updated.
 * </pre>
 *
 * @author Christian Datzko
 */
public interface PieceDatabaseChangeListener {
    /**
     * <pre>
     * The database has been changed: a piece has been added. A link to the
     * piece for indexing is supplied, also the index number in the database.
     * </pre>
     *
     * @param piece the piece that has been added
     * @param newIndex the given new index number
     */
    public void insertPiece(Piece piece, int newIndex);

    /**
     * <pre>
     * The database has been changed: a piece has been deleted. The index number
     * has been supplied.
     * </pre>
     *
     * @param index the index number of the deleted piece
     */
    public void deletePiece(int index);
}
```

B Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst habe und mich lediglich der angegebenen Hilfsmittel bedient habe.

Ort, Datum, Unterschrift