Bachelorarbeit

Erweiterung eines web-basierten Noteneditors um eine Funktion zum Vergleichen zweier Musikstücke

eingereicht von
Martin Unold

eingereicht bei
Prof. Dr.-Ing. Herbert Göttler

Fachbereich Physik, Mathematik und Informatik
der Johannes-Gutenberg-Universität Mainz

Juli 2010

Danksagung

Nicht nur für die Vergabe und Betreuung dieser Arbeit möchte ich Herrn Prof. Dr.-Ing. Herbert Göttler herzlich danken. Bereits während meines Studiums hat er mir in den Vorlesungen und im Rahmen meiner Tätigkeit als wissenschaftliche Hilfskraft wertvolle Einblicke in die Welt der Informatik gewährt.

Herrn Dr. Ludger Martin möchte ich ebenfalls für die gute Betreuung danken. Außerdem für die in den Vorlesungen zur Webentwicklung vermittelten Fähigkeiten, die bei der Programmierarbeit unersetzlich gewesen sind.

Bei den Vorarbeitern Thorsten Essig, Leo Ries und Stanislav Velychko bedanke ich mich für eine Grundlage, auf der man gut aufbauen konnte.

Thorsten Essig hat sich bereits während meines Studiums als guter Betreuer meiner Übungsgruppen ausgezeichnet.

Leo Ries war mir beim Einstieg in den Quellcode behilflich.

Ganz besonders möchte ich Herrn Konrad Georgi danken, der stets bereit war seine Zeit für mich zu opfern und meinen Horizont auf musikalischem Gebiet erweitert hat. Er hat mir die Problemstellung gut nähergebracht und so eine spannende interdisziplinäre Arbeit ermöglicht.

Inhaltsverzeichnis

Einleitung

Seit dem Jahr 2008 wurde von Studenten des Fachbereichs Informatik an der Johannes-Gutenberg-Universität ein webbasierter Musiknoteneditor entwickelt und weiterentwickelt. Diese Arbeit stellt einen weiteren Teil der Entwicklungsarbeit an diesem Editor dar. Dabei wird nun die Möglichkeit geboten in diesem Editor zwei Musikstücke mit einander zu vergleichen.

Thorsten Essig begann in seiner Diplomarbeit [Ess08] mit der Erstentwicklung eines webbasierten Musiknoteneditors. Dabei ging es nicht darum etablierten Editoren Konkurrenz zu machen. Vielmehr sollte eine Anwendung geschaffen werden, die ohne Installation auf allen Rechnern funktioniert. Es wurde grundlegende Funktionalität implementiert, die ein Musiknoteneditor unbedingt haben sollte. Im Laufe der Arbeit stellte sich ein interessantes Anwendungsgebiet heraus.

In seinem letzten Kapitel [Ess08] (Kapitel 8.3, S.82ff) hat Thorsten Essig bereits einen Ausblick auf mögliche Erweiterungen gegeben. Auch Leo Ries hat in [Rie09] (Kapitel 7.4.1) diese Erweiterung mit Bezug auf die neueren Entwicklungen des Musiknoteneditors beschrieben. Im Rahmen dieser Arbeit wurde der Musiknoteneditor um die Funktionalität des Vergleichs zweier Musikstücke erweitert. Andere in [Ess08] (Kapitel 8.3.1) erwähnte Verbesserungsmöglichkeiten wurden bereits von anderen Studenten realisiert.

Im März 2009 erweiterte Leo Ries in [Leo09] den Editor um Artikulationszeichen und Haltebögen. Später ergänzte Stanislav Velychko in [Vel09] noch Legatobögen, Triolen, 32stel-Noten und -Pausen sowie Undo- und Redo-Funktionalität.

Thorsten Essig hat in seiner Diplomarbeit auch auf andere Möglichkeiten zur Erweiterung hingewiesen. Es können also noch weitere Arbeiten angefertigt werden, die diese realisieren. Im Anhang finden sich Anregungen und Tipps dafür, im letzten Kapitel eine Zusammenfassung der offen gebliebenen Weiterentwicklungen. Auch für den hier vorgestellten Vergleich zweier Musikstücke gibt es seitens des Fachbereichs Musik noch Wünsche zum Ausbau. Auf diese wird am Ende der Arbeit näher eingegangen.

Die wesentliche Aufgabe dieser Arbeit war die Implementierung einer Funktion zum Vergleichen zweier Musikstücke im beschriebenen webbasierten Musiknoteneditor.

Neben einem akademischen Interesse an einem solchen Vergleich, findet dieser Anwendung im Fachbereich Musik bei der Korrektur von Notendiktaten. Das Programm soll dabei Studenten auf die Prüfung zur Gehörbildung vorbereiten. Die Prüfungsleistung besteht dort darin ein Musikstück zu höhren und als Notensatz wiederzugeben. Beim Üben solcher Notendiktate kann ein Musikstudent versuchen das, was er glaubt gehört zu haben, in den Noteneditor einzugeben. Eine Musterdatei, die das richtige Musikstück enthält, kann anschließend zum Vergleich geladen werden und das Programm zeigt dann an wie gut der Versuch des Studenten war, wo Fehler auftreten und welche Art von Fehlern besonders häufig vorkam. Dadurch wird es erleichtert Akzente beim Üben zu setzen und die eigentliche Prüfung erst dann anzutreten, wenn bereits beim Vorüben genügend richtig erkannt wurde. Allerdings soll das Programm keine Prüfungen abnehmen, sondern lediglich bei der Vorbereitung unterstützen. Wie sich im Laufe der Arbeit noch zeigen wird, kann ein Vergleichs-Algorithmus nicht das leisten, was ein Experte auf diesem Gebiet vorraussetzen würde. Anregungen zum Thema habe ich in [Mus] gefunden.

Im ersten Kapitel wird in die Problemstellung eingeführt. Es werden einfache und kompliziertere Beispiel-Korrekturen vorgeführt und es wird erläutert, warum es keine „perfekte“ sondern nur eine geeignete Lösung geben kann. Außerdem wird erklärt, wie man an die für den Vergleich relevanten Daten sowohl aus der Musik-XML-Datei als auch aus der internen Score-Datenstruktur von JavaScript gelangen kann und wie dies umgesetzt wurde.

Im zweiten Kapitel wird die erste Idee einer Vorgehensweise zum Vergleich zweier Musikstücke vorgestellt. Dabei werden beide Musikstücke in ein Raster von 32-stel-Noten überführt und eine möglichst gute Übereinstimmung durch lokales Verschieben des Rasters gesucht. Dieser Ansatz hat sich jedoch im Laufe der Arbeit als unvorteilhaft herausgestellt.

Ein besseres Verfahren wird in Kapitel drei gezeigt. Es handelt sich um den Levenshtein-Algorithmus, der die einzelnen Noten paarweise miteinander vergleicht. Auf den ersten Blick scheint dies ungeeigneter, weil kleine Veränderungen an den Notationselementen hart bestraft werden und dadurch keine Übereinstimmung zustande kommt. Ein großer Vorteil ist jedoch die Praxistauglichkeit. Dies wird dort und im letzten Kapitel genauer erläutert.

Im vierten Kapitel wird dann gezeigt wie die Ergebnisse des Vergleichs visualisiert werden und welche Veränderungen am Quellcode vorgenommen wurden auch außerhalb der eigenen Aufgabenstellung. Die Noten werden entsprechend der Art des Fehlers farblich markiert und eine Anzeige gibt dem Benutzer Auskunft darüber wie gut das Notendiktat war und welche Fehler besonders häufig aufgetreten sind.

Zum Schluss werden im fünften Kapitel einige Beispieleingaben gegeben und es wird vorgeführt, wie der Algorithmus sie verarbeitet. Es wird erörtert wie praxistauglich das Entwickelte ist und es wird diskutiert wo noch Verbesserungen angebracht werden könnten und welche Grenzen das Verfahren hat.

Im Anhang finden sich eine Bedienungsanleitung und eine Anleitung zur Weiterentwicklung des Noteneditors, die als Einstiegshilfe für diejenigen dienen soll, die den Noteneditor erweitern möchten.

Diese Arbeit ist also in etwa nach dem chronologischen Vorgehen bei der Benutzung des Programms aufgebaut. Man kann sich das an folgendem Schaubild verdeutlichen.

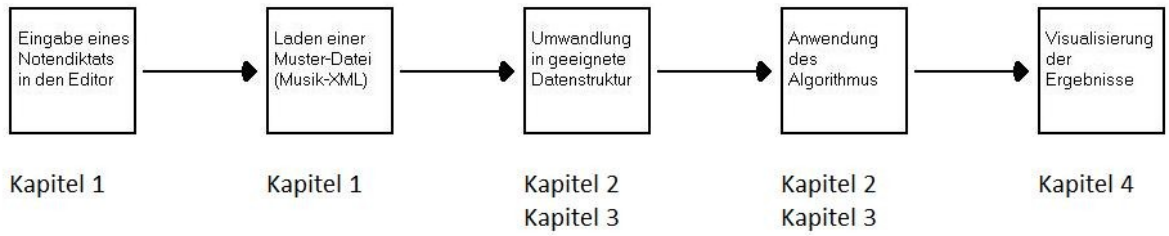


Abbildung 0.1 – Aufbau der Anwendung

Zu dieser Arbeit gehört eine CD-ROM mit dem programmierten Code. Dort ist außerdem eine elektronische Version dieser Arbeit zu finden.

1. XML-Vergleich

Dieses Kapitel beschreibt die generelle Problemstellung, die grundlegende Herangehensweise und die Gewinnung der für den Algorithmus relevanten Daten aus den vom Musiknoteneditor gegebenen Datenstrukturen.

Problemstellung

Zum Musiknoteneditor wird eine Funktion hinzugefügt, die es ermöglichen soll eine Eingabe mit einem hinterlegten Muster abzugleichen. Dies wird in erster Linie Anwendung finden bei Musikstudenten zur Vorbereitung auf die Prüfung zur Gehörbildung. Die korrekte Wiedergabe von Notendiktaten soll dabei überprüft werden. Dazu kann eine Eingabe in den Musiknoteneditor gemacht werden, die dann mit der hinterlegten Musterdatei verglichen wird. Es soll dann angezeigt werden wie gelungen das Notendiktat war. Das Thema lässt sich also auf eine einfache Frage reduzieren:

Gegeben sind zwei Musikstücke. Wie sehr und vor allem worin unterscheiden sie sich?

Um genauer zu verstehen, was diese Frage eigentlich bedeutet und worin die Schwierigkeiten bei der Ermittlung von Unterschieden bestehen, betrachte man sich ein paar Beispiele.

Beispiel 1.1

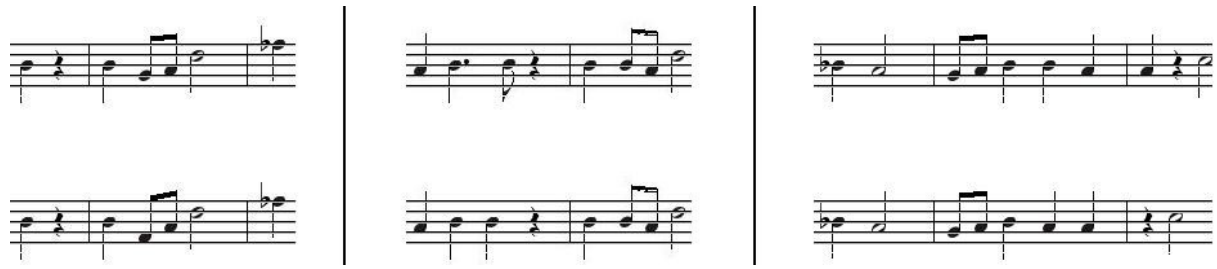


Abbildung 1.1 – Einfache einzelne Fehler

Hier sind Ausschnitte von Musikstücken abgebildet, bei denen die Unterschiede leicht zu erkennen sind. In der oberen Zeile ist das Muster und in der unteren der Versuch zu sehen. Im ersten Beispiel ist eine Tonhöhe falsch gesetzt worden, im zweiten Beispiel der Rhythmus. Im dritten Beispiel wurde eine Note vergessen.

Es gibt noch ein paar weitere Arten von möglichen Fehlern, bei denen sich nur ein Notationselement unterscheidet. Diese sind einfach zu ermitteln. Würden sich die beiden zu vergleichenden Musikstücke nur in einem dieser Fehler unterscheiden und ansonsten übereinstimmen, würde es genügen ein Programm zu schreiben, das die Musikstücke von vorne nach hinten durchläuft bis es auf den Fehler stößt. Dann könnte man leicht die Art des Fehlers bestimmen und das Problem wäre gelöst.

Auch bei mehreren einzelnen gut isolierten Fehlern dieser Art könnte man noch mit einem solchen Verfahren auskommen.

Die Schwierigkeit beginnt erst mit der Kombination mehrerer Fehler. Besonders ein Verrutschen der Tonhöhe ist schwierig zu erkennen. Wenn dann sogar rhythmisches Verrutschen kombiniert mit weiteren Fehlern hinzukommt, ist selbst für den menschlichen Korrektor das Entdecken von richtigen Elementen schwierig. Der menschliche Korrektor kann allerdings leicht erkennen, ab wann eine grob falsche Passage vorbei ist. Folgen später wieder längere korrekt erkannte Passagen, kann er nach eigenem Ermessen geeignete Wiederaufsetzpunkte finden und von dort weiter korrigieren.

Beispiel 1.2

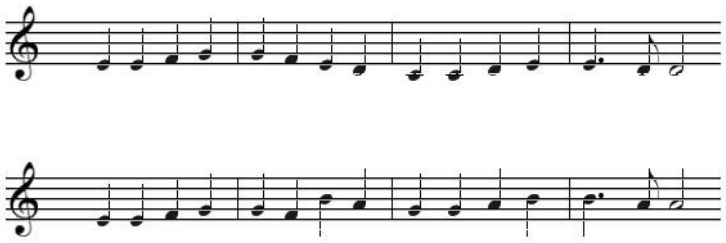


Abbildung 1.2 – Verrutschen in der Tonhöhe

Hier könnte ein Intervall falsch gehört worden sein und die übrigen falsch platzierten Töne sind lediglich Folgefehler. Ein Programm, das ausschließlich einzelne Noten miteinander vergleicht und nicht auf den Gesamtkontext achtet, würde das Ende des Stücks als falsch markieren, obwohl der geschulte menschliche Korrektor eine Verschiebung erkennen würde. Aufgrund dieser Beobachtungen ist die erste Idee zur Implementierung eines Algorithmus durch Rasterung entstanden.

Richtig problematisch wird es bei der Korrektur von Kombinationen aus richtigen und falschen Passagen, wie in dem folgenden Beispiel dargestellt.

Beispiel 1.3

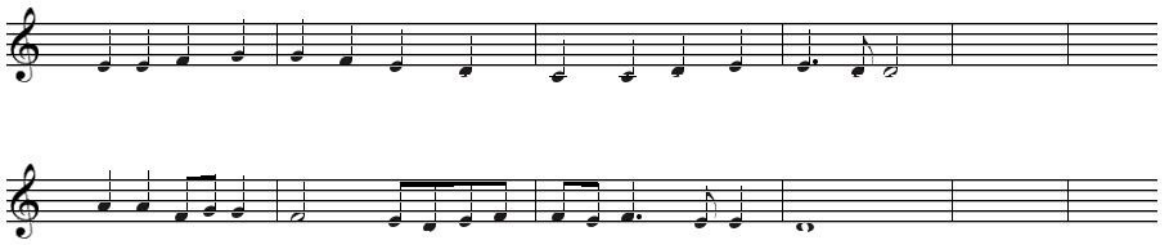


Abbildung 1.3 – Komplexe Fehlersituation

Selbst für den geübten menschlichen Korrektor ist hier eine Zuordnung sehr schwierig und nur nach gewissen Ermessensspielräumen möglich. Ein Computerprogramm kann hingegen nur nach eindeutigen und wohldefinierten Vorschriften handeln und korrigieren. Diese sinnvoll zu wählen wird die Aufgabe im Folgenden sein.

Doch zunächst sind die für den Algorithmus relevanten Informationen zu ermitteln. Welche Informationen werden überhaupt benötigt und wie kommt man an sie heran? Welche Datenstruktur dann zum Vergleichen verwendet werden soll, hängt vom verwendeten Algorithmus ab.

Es ist die aktuelle Eingabe in den Musiknoteneditor zu ermitteln. Dies wird nun erläutert. Anschließend folgt die Betrachtung der Muster-Datei. Diese ist im Musik-XML-Format hinterlegt und wird in den Editor geladen.

Score-Datei lesen

Die in den Editor eingegebenen Noten und Notationselemente werden mit JavaScript verarbeitet und in der internen Datenstruktur Score gespeichert. Vereinfacht dargestellt sieht diese wie folgt aus. Weitere Informationen dazu finden sich in [Ess08] (Kapitel 5) und im Anhang zu dieser Arbeit.

Abbildung 1.4 – Vereinfachte Datenstruktur (aus [Ess08] Abbildung 5.1)

Ein Stück wird in `score.js`, ein Takt in `measure.js`, eine Stimme in `part.js` und eine Note in `event.js` intern repräsentiert.

Die zum Vergleich wichtigen Informationen befinden sich innerhalb der Stimmen in den jeweiligen Layern. Die Klasse, die eine Note oder eine Pause repräsentiert, beinhaltet dann alle wichtigen Informationen, die zum Vergleich benötigt werden. Dazu gehören Art, Dauer, Akzente und Artikulation sowie Bögen.

Da der Algorithmus zum Vergleichen der Musikstücke ebenfalls in JavaScript geschrieben werden soll, können die Informationen an dieser Stelle einfach ausgelesen werden.

Musik-XML-Datei lesen

Das Muster ist als XML-Datei hinterlegt. Die Struktur von Musik-XML wird ausführlich in [Ess08] (Kapitel 3.2) beschrieben. An dieser Stelle genügt es die für den Vergleich relevanten Tags herauszufinden.

```
<score-timewise version="1.1">
    <identification> ... </identification>
    <part-list>
        <score-part id="PartID">
            <part-name>
                PartName
            </part-name>
        </score-part>
        ...
    </part-list>
```

Typischerweise beginnt eine Musik-XML-Datei mit dieser Beschreibung, die jedoch für den Vergleich der Musikstücke unerheblich ist. Sie können daher beim Laden weggelassen werden.

Anschließend folgen die wichtigen Informationen über die Noten, wobei der erste Takt separat betrachtet werden muss, da dort Informationen über generelle Einteilungen und Einstellungen der Takte zu finden sind.

```
<measure number="0">
    <part id="PartID">
        <attributes>
            <divisions>
                k
            </divisions>
            <time>
                beats
            </time>
            <clef>
                sign / line
            </clef>
            ...
        </attributes>
        s.u.
    </part>
</measure>
```

Die wichtigsten Informationen sind sicherlich Tonhöhe und Tondauer. Diese können aus `pitch` und `duration` entnommen werden. Weitere Notationselemente sind in `type` vorzufinden.

```
i = 1,2,3,...
<measure number="i">
    <part id="PartID">
        <note>
            <pitch>
                <step>
                    C / D / E / F / G / A / B
                </step>
                <octave>
                    0 .. 7
                </octave>
            </pitch>
            <duration>
                1 ..
            </duration>
            <type>
                ...
            </type>
            <stem> ... </stem>
        </note>
        ...
    </part>
    ...
</measure>
</score-timewise>
```

Das Einlesen dieser Datei erfolgt mittels php, denn JavaScript ist nicht in der Lage Dateien auf der Festplatte zu lesen. Das Vorgehen beim Laden der Muster-Datei geschieht analog zum Laden einer beliebigen Musik-XML-Datei in den Editor. Es sei daher an dieser Stelle auf [Ess08] (Kapitel 5.6, 5.7 und Kapitel 7.1) verwiesen, die die Vorgehensweise zum Laden einer solchen Datei und die dabei besonderen Konzepte beschreiben.

Vergleichs-Algorithmen

Sind die relevanten Daten der Musikstücke ermittelt und eingelesen worden, sollen sie miteinander verglichen werden. Bei der Korrektur von Notendiktaten mit der Hand ist es dem Korrektor überlassen, wie Fehler zu bewerten und zu erkennen sind. Dabei hält er sich an gewisse Grundsätze, wie einfache Fehler zu behandeln sind. Mit wachsender Erfahrung entwickelt sich ein Gespür für häufig gemachte Fehler und ihre entsprechende Handhabung. Die meisten Fehler können also in irgendein einfaches Muster eingeordnet werden. Die Zahl dieser häufig auftretenden Fehlermuster ist endlich. Schwierig wird es, wenn Fehler auftreten, die in keines dieser Muster passen. Dann wird sich der menschliche Korrektor ein neues Muster einfallen lassen und, falls später wieder ein Fehler dieser Art auftaucht, auf die gleiche Weise korrigieren.

Einem Computerprogramm ist dies jedoch unmöglich. Es muss auf alle Arten von Fehlern gefasst sein und irgendeine Reaktion, ein Korrekturverhalten, durchführen. Deshalb wird die Frage nach dem Unterschied zwischen zwei Musikstücken nicht nach klaren und wohldefinierten Kriterien zu beantworten sein können. Es muss eine Heuristik gefunden werden, die typische und einfache Fehler gut erkennt, bei komplizierteren Fehlern aber dennoch sinnvoll korrigiert. Eine formale Beschreibung der Aufgabenstellung ist also nicht möglich. Es kann lediglich eine angemessene Praxistauglichkeit nachgewisen werden.

Formuliert werden können allerdings Algorithmen und die Ergebnisse von Algorithmen können auch formalisiert werden. In dieser Arbeit werden zwei Algorithmen vorgestellt, die die Unterschiede zwischen zwei Musikstücken herausfinden sollen. Es wird dabei deutlich, dass der zweite Algorithmus sich in einigen Fällen besser an die intuitive Vorstellung von Fehlern bei Notendiktaten anlehnt. Deshalb ist der erste Algorithmus für die Praxis eher unbrauchbar. Man kann ihn aber nicht als „falsch“, sondern nur als „ungeeignet“ bezeichnen.

Der erste Algorithmus versucht die Notenzeilen und die Tonhöhen in ein Raster zu bringen. Alle Takte werden in sehr kleine 32stel-Takte unterteilt und die Notenlinien stellen das vertikale Raster dar. Dadurch erstrecken sich einzelne Noten über mehrere Takte, also Gitterpunkte des Rasters auf der gleichen horizontalen Ebene. Nun wird durch möglichst wenig Auftrennen und Verschieben versucht die beiden Stücke passend übereinander zu legen.

Der zweite Algorithmus vergleicht die einzelnen Noten miteinander und vernachlässigt dabei ihre genaue Position. Die Musikstücke werden als Worte, Sätze oder Zahlenreihen aufgefasst, die in ihrer Buchstaben- bzw. Ziffernfolge möglichst gut übereinstimmen sollen. Dabei wird nur paarweise verglichen und keine größeren zusammenhängenden Teile der Musikstücke in Übereinstimmung gebracht.

Am Ende der jeweiligen Kapitel und im Fazit werden Vor- und Nachteile der Algorithmen und die Eingnung für die Praxis diskutiert.

Notationen und Konventionen

Zum erleichterten Einstieg in die Beschreibung der Algorithmen sei hier bereits erwähnt, dass in der Datenstruktur, die zum Vergleich dienen soll, stets „template“ das Muster aus der XML-Datei und „try“ die Eingabe in den Noteneditor bezeichnet. Pseudocode für nur ein Musikstück wird stets mittels „try“ angegeben.

Außerdem wird nun davon ausgegangen, dass alle Noten mit zugehöriger Ornemantik und Pausen sowohl vom Muster als auch vom Versuch vorliegen. Dabei werden für das Erstellen von Pseudocode folgende Bezeichnungen verwendet.

`Note n` → Bezeichnet eine Variable n vom Typ „Note“, der die eingelesenen Informationen enthält, auf die durch einfache Befehle zugegriffen werden kann. Für Pausen gibt es keine Tonhöhe bzw. sie enthält eine pausenspezifische Konstante.

`n.duration` = Dauer einer Note (z.B. Achtel, Viertel, punktierte Halbe)

`n.position` = Rhythmische Position der Note relativ zum Anfang des Musikstücks

`n.heigth` = Tonhöhe relativ zur Mittelline (h1), Halbtonschritte können mit Kommazahlen angegeben werden

`n.specials` = besondere Notationselemente an dieser Note (z.B. staccato, accent)

2. Vergleich durch Rasterung

In diesem Kapitel wird ein erster Algorithmus zum Vergleich von Musikstücken vorgestellt. Dieser erstellt zunächst für Muster und Versuch je ein Raster aus der kleinstmöglichen Taktung (bei dem aktuellen Musiknoteneditor sind dies 32stel-Noten) in der Vertikalen und Halbtonschritten in der Horizontalen. Die Musikstücke werden dann in die entsprechenden Raster eingefügt. Nun wird das eine Raster so verschoben, dass eine möglichst lange Übereinstimmung erzielt wird ohne dafür zu stark zu verschieben und zu zerschneiden.

Datenstruktur

Zunächst wird das Konzept anhand nur eines Musikstücks erklärt. Später sollen beide Musikstücke in diese Datenstruktur übertragen werden. Die Rasterung könnte zum Beispiel folgendermaßen aussehen.

Beispiel 2.1

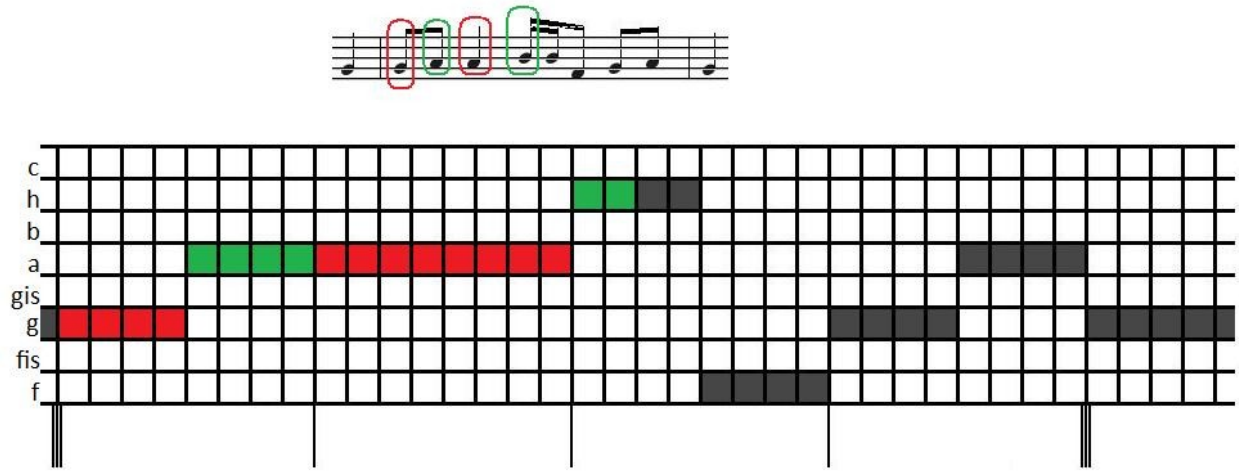


Abbildung 2.1 – Einfaches Raster-Schema

In vertikaler Raster-Richtung wird die Dauer und rhythmische Platzierung der Noten eingetragen, in horizontaler Raster-Richtung die Tonhöhe. Die Datenstruktur ist also ein zweidimensionales Array mit boolschen Einträgen, die angeben, ob zu diesem Zeitpunkt ein Ton einer gewissen Tonhöhe erklingt.

Dummerweise werden auf diese Art keine neuen Anschläge einer Note erkannt. Außerdem werden weitere Notationselemente wie Ornemantik nicht dargestellt. Dies lässt sich jedoch beheben, indem statt boolschen Einträgen Zahlen genommen werden, die einen neuen Ton oder die Ornemantik anzeigen. 0 könnte beispielsweise für einen gehaltenen Ton stehen, 1 für einen neu angeschlagenen Ton ohne besondere Merkmale und weitere Zahlen für gewisse Betonungselemente.

Beispiel 2.2

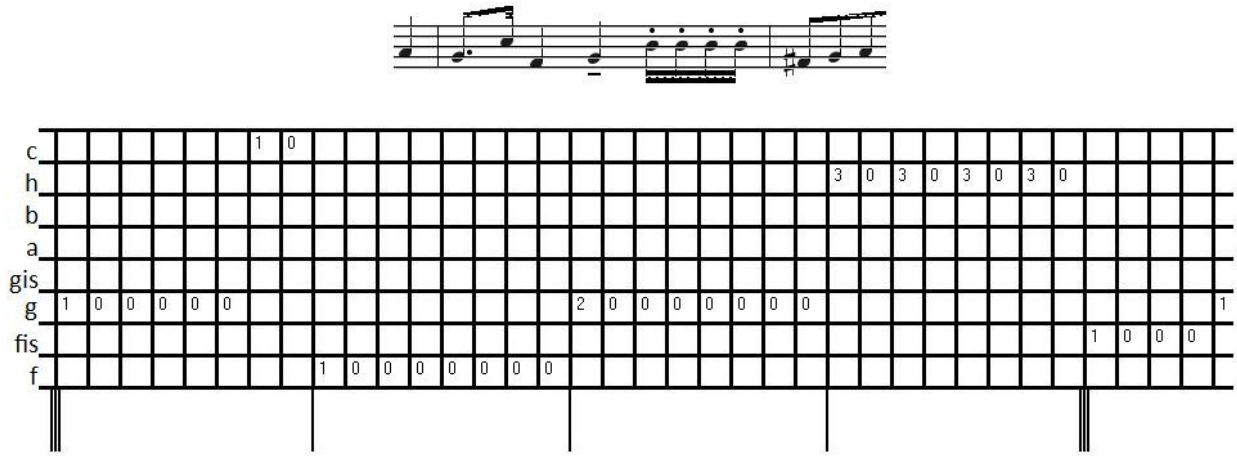


Abbildung 2.2 – Raster-Schema mit unterschiedlichen Einträgen

Auf diese Weise können auch sehr einfach mehrstimmige Stücke in die Datenstruktur übertragen werden. Dabei werden manche Spalten mehrfach belegt.

Man sieht bereits jetzt, dass die Datenstruktur sehr dünn besetzt ist. Daher könnte man statt eines Gitters auch für jeden horizontalen Rasterpunkt, also für jeden 32stel-Schlag, eine Liste von erklingenden Tönen speichern. Dies würde den Speicherbedarf erheblich senken. Zum Verständnis des Algorithmus ist es jedoch hilfreich die zweidimensionale Struktur (nur für die Erklärung) zu behalten.

Pseudocode:

```
for all (i,j) {
     try[i][j] = empty;
}
for all (Note n) {
     for (i=0; i<n.duration; i++) {
          if (i==0) {
               try[n.position][n.heigth] = n.specials;
          }
          else {
               try[n.position+i][n.heigth] = 0;
          }
     }
}
return (try);
```

Dieser Code erstellt die Raster-Datenstruktur. Es wird ein leeres zweidimensionales Gitter erzeugt, das dann zu jedem Zeitpunkt mit den erklingenden und neu angeschlagenen Tönen gefüllt wird.

Algorithmus

Das Ziel ist zu erreichen, dass beide Raster durch möglichst wenig Zerschneidung, Verschiebung und Ersetzung in einander überführt werden.

Gegeben seien zwei Musikstücke `template` und `try` in der gerade eingeführten Datenstruktur.

Definition 2.3

Aus dem zweidimensionalen Array *try* kann ein Teilarray entnommen werden.

$$try(i_{1,}i_L, j_{1,}j_L)=\begin{pmatrix} try[i_1][j_1] & \cdots & try[i_1][j_1+j_L] \\ \vdots & & \vdots \\ try[i_1+i_L][j_1] & \cdots & try[i_1+i_L][j_1+j_L] \end{pmatrix}$$

Es gilt also: $try(i_{1,}i_{2,}j_{1,}j_2)[i][j]=try[i_1+i][j_1+j]$

Definition 2.4

Ein Teilarray $try(i_{1,}i_L, j_{1,}j_L)$ zusammen mit Indizes (i_T, j_T) des Musters heißt maximale Übereinstimmung, falls gilt:

$$template(i_T, i_L, j_T, j_L) = try(i_{1,}i_L, j_{1,}j_L)$$

Und für alle (i_{L2}, j_{L2}) mit $template(i_{T2}, i_{L2}, j_{T2}, j_{L2})=try(i_{1,}i_{L2}, j_{1,}j_{L2})$ ist $i_{L2}\leq i_L$.

Der Algorithmus versucht eine maximale Übereinstimmung zu finden. Gibt es mehrere, so ist diejenige auszuwählen, die den geringsten vertikalen Abstand (Höhenunterschied) fordert. Er zerschneidet die Musikstücke anschließend links und rechts dieser Übereinstimmung. Diese werden jeweils miteinander verglichen. Es wird wieder eine maximale Übereinstimmung gesucht. Dies wird iteriert bis die verbliebenen Noten nicht mehr in Übereinstimmung gebracht werden können, weil eines der beiden Arrays leer ist.

Beispiel 2.5

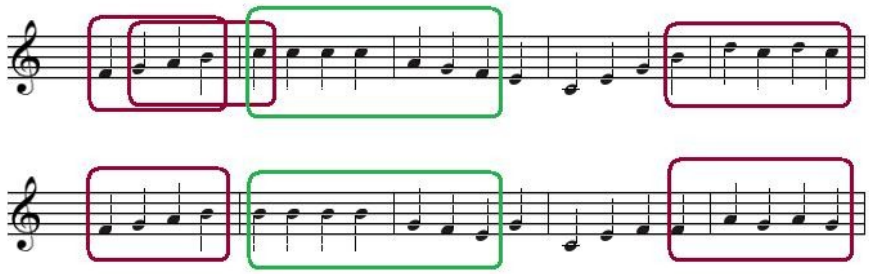


Abbildung 2.3 – verschiedene Übereinstimmungen

Die hier abgebildeten Musikstücke haben einige Bereiche, in denen sie übereinstimmen. Manchmal ist eine Verschiebung in der Tonhöhe nötig. Exemplarisch sind ein paar Bereiche, die übereinstimmen, abgebildet. Die maximale Übereinstimmung befindet sich in der Mitte mit 7 übereinstimmenden Tönen.

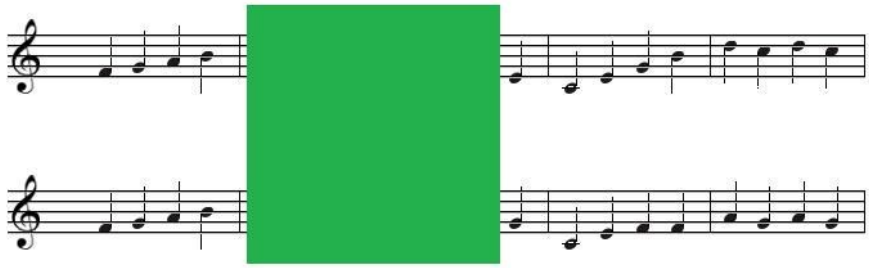


Abbildung 2.4 – Entfernen der maximalen Übereinstimmung

Der Bereich der maximalen Übereinstimmung wird als korrekt markiert, wobei die Abweichung in der Tonhöhe auch vermerkt wird. Er wird gelöscht und das Verfahren beginnt mit den beiden verbleibenden Bereichen erneut.

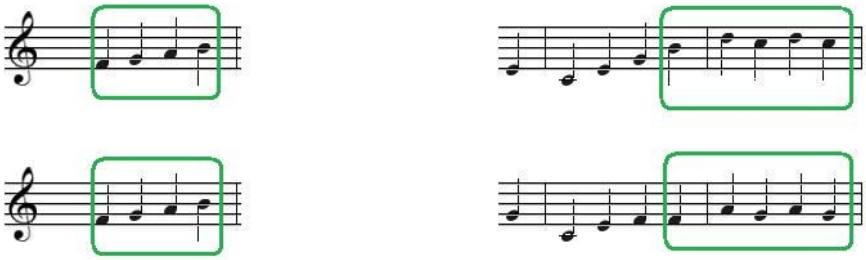


Abbildung 2.5 – Erneutes Ermitteln der maximalen Übereinstimmung

Die maximalen Übereinstimmungen werden wieder markiert, rechts mit Verschiebung in der Tonhöhe, und die Rekursion wird erneut aufgerufen.

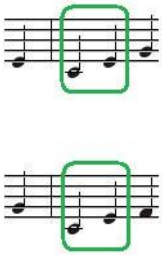


Abbildung 2.6 – Letzte Ermittlung der maximalen Übereinstimmung

Auch einzelne Töne können in Übereinstimmung gebracht werden. Nur falls mindestens eine der beiden Notenlinien leer ist, wird die Rekursion abgebrochen. Im angegebenen Beispiel liefert der Algorithmus ein Resultat, das auf den ersten Blick auch vernünftig erscheint.

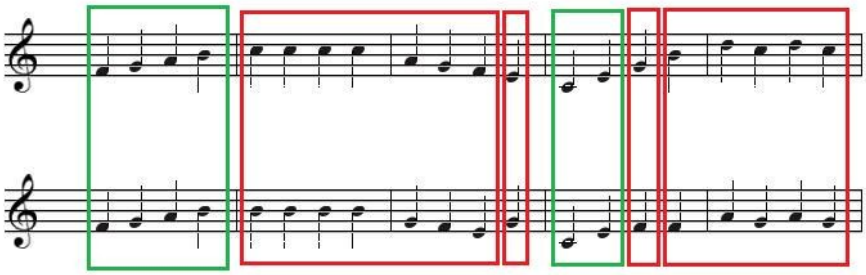


Abbildung 2.7 – Resultat des Vergleichs durch Rasterung

Im folgenden Pseudocode wird eine Funktion `getMaxSync` benutzt, die eine maximale Übereinstimmung berechnet. Rückgabe sind jedoch nur die i-Komponenten. Diese Übereinstimmung wird dann als richtig bzw. um Höhenunterschied verschoben markiert. Außerdem bezeichnen `try(.,.,.,.)` und `template(.,.,.,.)` das gemäß Definition 2.3 eingeführte Teilarray. Dabei gibt maxI bzw. maxJ an, dass das Teilarray bis zum Ende gehen soll. Nicht in Übereinstimmung zu bringende Teile werden als falsch markiert.

Pseudocode:

```
compare(try,template) {
     if (try == empty || template == empty) return;
     (i1,iL,iT) = getMaxSync(try,template);
     compare(try(0,i1,0,maxJ),template(0,iT,0,maxJ));
     compare(try(i1+iL,maxI,0,maxJ),template(iT+iL,maxI,0,maxJ));
}
```

Resultate

Dieser Algorithmus liefert in vielen Fällen brauchbare Ergebnisse. Die einfachen Fehler aus Beispiel 1.1 und Beispiel 1.2 können gut erkannt werden. Auch für Beispiel 1.3, für das es keine „perfekte“ Korrektur gibt, liefert er ein annehmbares Resultat.

Probleme tauchen dann auf, wenn sich Muster wiederholen.

Beispiel 2.6

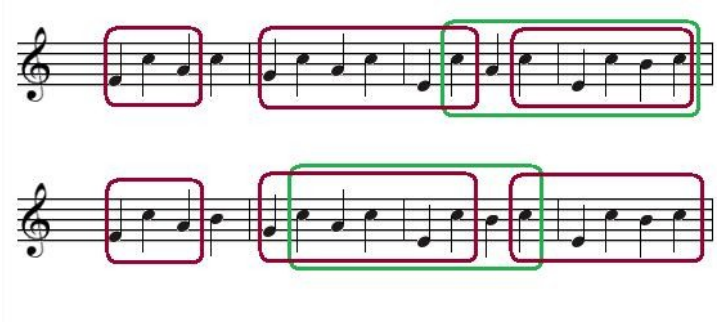


Abbildung 2.8 – Problematik der Rasterung

Hier ist die maximale Übereinstimmung durch einen Bereich gegeben, der stark verschoben werden musste. Der Algorithmus würde also große Teile als falsch markieren. Dabei ist einzusehen, dass lediglich zwei Noten eine leicht verschobene Tonhöhe haben.

Um dem entgegenzuwirken, kann man bei der Ermittlung der maximalen Übereinstimmung große Verschiebungen stärker bestrafen.

Definition 2.7

Ein Teilarray $try(i_{1,}i_L, j_{1,}j_L)$ zusammen mit Indizes (i_T, j_T) des Musters heißt maximale Übereinstimmung, falls gilt:

$$template(i_T, i_L, j_T, j_L) = try(i_{1,}i_L, j_{1,}j_L)$$

Und für alle (i_{L2}, j_{L2}) mit $template(i_{T2}, i_{L2}, j_{T2}, j_{L2}) = try(i_{1,}i_{L2}, j_{1,}j_{L2})$ ist $i_{L2} \leq i_L$.

Verwendet man statt der maximalen die optimale Übereinstimmung, taucht dieses Problem nicht mehr auf.

Der im nächsten Kapitel vorgestellte Levenshtein-Algorithmus verwendet keine Rasterung. Die Bestrafung von Verschiebungen steht bei ihm im Vordergrund. Ein ausführlicher Vergleich dieser beiden Algorithmen folgt in Kapitel 3. An dieser Stelle wird lediglich noch die Laufzeit des Verfahrens durch Rasterung ermittelt.

Satz 2.8

Sei n die Länge des kürzeren, m die Länge des längeren Musikstücks im 32stel-Raster.
Laufzeit von *getMaxSync*: $O(n^2 m)$

Beweis:

Jeder Rasterpunkt auf beiden Seiten könnte der Beginn einer optimalen Übereinstimmung sein.
Daher müssen beide Musikstücke durchlaufen werden ($O(nm)$).
In jedem Schritt muss dann die Länge der Übereinstimmung ermittelt werden ($O(n)$).

Die Laufzeit zur Bestimmung der optimalen Übereinstimmung ist die gleiche, da lediglich die Verschiebung bei der Berechnung addiert werden muss.

Die hier ermittelte Laufzeit gilt nur für den schlimmsten Fall. Stimmen beide Musikstücke überein, ist lediglich ein einmaliges Durchlaufen nötig und die Laufzeit ist linear. Doch in ungünstigen Fällen kann der komplette Algorithmus sehr große Laufzeiten haben.

Die Laufzeit des Verfahrens durch Rasterung insgesamt liegt, wenn man beide Musikstücke als etwa gleich lang vorraussetzt, wegen des doppelt rekursiven Aufrufs bei $O(n^5)$!

3. Levenshtein-Algorithmus

Der im Folgenden vorgestellte Algorithmus ist etwas allgemeiner formuliert als der klassische Levenshtein-Algorithmus. Die Idee der Vorgehensweise ist jedoch stark an ihn angelehnt. Für weiterführende Informationen siehe [Lev].

Levenshtein-Distanz

Ziel ist es zwei Worte in einander zu überführen und dabei möglichst wenige oder kostengünstige Operationen durchzuführen. Doch zunächst bedarf es dazu einer formalen Definition.

Definition 3.1

Ein Alphabet Σ ist eine endliche Menge von unterscheidbaren Symbolen (Buchstaben).
Eine endliche Folge von Buchstaben $w=(w_1, w_2, ..., w_n)$, wobei $w_i \in \Sigma$ für $i \in \{1,2, ..., n\}$, $n \in \mathbb{N}$, heißt Wort über dem Alphabet Σ. n ist die Länge des Wortes.
Mit λ wird das leere Wort der Länge 0 bezeichnet.
Zur vereinfachten notation bezeichne $w^k=(w_1, w_2, ..., w_k)$ für $k \leq n$ das Teilwort von w mit den ersten k Buchstaben.

Definition 3.2

Der Levenshtein-Abstand zweier Worte $v=(v_1, ..., v_n)$ und $w=(w_1, ..., w_m)$ der Längen n und m ist definiert durch folgende rekursive Vorschrift:

$$d(\lambda, \lambda)=0 \ , \ d(v, \lambda)=0 \ , \ d(\lambda, w)=0$$
$$d(v, w)=min(d(v^{n-1}, w)+T, d(v, w^{m-1})+T, d(v^{n-1}, w^{m-1})+e(v_n, w_m))$$

Der Levenshtein-Abstand soll angeben wie sehr sich zwei Worte unterscheiden. Dabei stehen dem Nutzer dieser Metrik Parameter zur Verfügung, um zu gewichten welche Operationen zwei Worte wie unterschiedlich machen. Der Parameter T gibt die Kosten für das Löschen oder das Hinzufügen eines Buchstabens an. Die Funktion e gibt an wie teuer Ersetzungen von Buchstaben sind. Diese Funktion sollte auch eine Metrik sein, damit d zu einer Metrik wird. Zu beachten ist außerdem, dass die Parameter nicht völlig unabhängig voneinander gewählt werden können. Ist 2T < e(.,.), so ist es immer vorteilhafter statt einer Ersetzungs-Operation eine Lösch- und eine Einfüge-Operation durchzuführen. Wie die Parameter genau gewählt werden sollten, hängt von der Problemstellung ab. Dies wird später diskutiert. In der klassischen Variante des Levenshtein-Algorithmus ist T = 1 und e konstant auf 1 gesetzt. In der Tat macht es auch Sinn immer T = 1 zu setzen, denn ist e eine Metrik und T eine Konstante, dann kann man T' = 1 und e' = e/T setzen. Der daraus resultierende Levenshtein-Abstand ist dann gerade d' = d/T. Man erhält also eine Metrik, die zwar andere Zahlenwerte für Abstände ausgibt, die relativen Entfernungen von Worten bleiben jedoch erhalten. Es wird also im Folgenden ohne den Parameter T gearbeitet. Dennoch muss die eben erwähnte Forderung eingehalten werden.

Regel 3.3

$e(a,b) \leq 2$ für alle $a, b \in \Sigma$.

Später wird der Levenshtein-Abstand noch um eine Option erweitert, die ermöglichen soll Verschiebungen in der Tonhöhe zu erkennen. Die nun folgenden Beweise werden ohne diese Erweiterung geführt, die Idee dabei kann jedoch auch mit ihr durchgegangen werden.

Satz 3.4

Der Levenshtein-Abstand $d: \Sigma^n \times \Sigma^m \to \mathbb{R}_0^+$ ist eine Metrik, falls $e: \Sigma \times \Sigma \to \mathbb{R}_0^+$ eine Metrik ist.

Beweis:

Zu zeigen sind:
1. $d(v,w) = 0 \Leftrightarrow v = w$
2. $d(v,w) = d(w,v)$
3. $d(v,w) \leq d(v,u) + d(u,w)$

Es seien v ein Wort der Länge n, w ein Wort der Länge m und u ein Wort der Länge l.
Die Punkte werden induktiv nach n gezeigt.

1. IA: $d(\lambda, w) = m \Rightarrow (d(\lambda, w) = 0 \Leftrightarrow w = \lambda)$
IS: $d(v,w) = 0 \Leftrightarrow d(v,w) = d(v^{n-1}, w^{m-1}) + e(v_n, w_m) = 0$, da Addition von $T = 1$ größer null wäre
$\Leftrightarrow (v^{n-1} = w^{m-1}$ nach IV, $v_n = w_n$ da e eine Metrik ist$) \Leftrightarrow v = w$

2. IA: $d(\lambda, w) = m = d(w, \lambda)$
IS: Es gibt drei Möglichkeiten, wie $d(v,w)$ und damit auch $d(w,v)$ rekursiv gegeben sind:
i. $d(v,w) = d(v^{n-1}, w^{m-1}) + e(v_n, w_m) = d(w^{m-1}, v^{n-1}) + e(w_m, v_n) = d(w,v)$
ii. $d(v,w) = d(v, w^{m-1}) + 1$. Gilt dies, so kann die IV nicht unmittelbar angewendet werden.
In diesem Fall ist eine weitere Induktion über m zu führen, es wird also erneut bei IS begonnen bis i. oder iii. eintritt oder $m = 0$. In letzterem Fall gilt: $d(v,w) = d(v,\lambda) = n = d(\lambda, v) = d(w,v)$
iii. $d(v,w) = d(v^{n-1}, w) + 1 = d(w, v^{n-1}) + 1 = d(w,v)$ analog zu ii.

3. IA: $d(\lambda, w) = m = l + (m-l) \leq l + |m-l| \leq d(\lambda, u) + d(u, w)$
IS:
Es bedarf einiger Indizes und einer langen Kette von Induktionen diesen Beweis formal auszuführen.
Man macht sich die Dreiecksungleichung leicht klar, indem man die Überführungen betrachtet.
Diese geben an auf welche Weise die Worte ineinander überführt werden können.
Wendet man v nach u und u nach w an, so sind die Gesamtkosten gleich der Summe.
Es könnte aber auch eine direktere Überführung geben.

Der Levenshtein-Abstand erfüllt eine weitere Eingenschaft, die bei der Argumentation zur Praxiseignung sehr wesentlich ist. Sie wird in Satz 3.12 gezeigt.

Levenshtein für Musikstücke

Der Levenshtein-Algorithmus soll den Levenshtein-Abstand zweier Worte berechnen. Er gibt dabei außerdem Auskunft darüber welche Operationen durchzuführen sind, um mit möglichst kostengünstigen Operationen auszukommen.

Bei der Implementierung bietet es sich an die Levenshtein-Distanz mit Hilfe eines zweidimensionalen Tableaus zu beschreiben. So kann man leicht das rekursive Minimum bestimmen.

Pseudocode:

```
for (i=0;i<=n;i++) {
     d[i][0] = i;
}
for (j=1;j<=n;j++) {
     d[0][j] = j;
}
for (i=1; i<=n; i++) {
     for (j=1; j<=m; j++) {
          dist[i][j] = min(d[i-1][j]+1,d[i][j-1]+1,
                           d[i-1][j-1]+e(v[i],w[j]));
     }
}
return (d);
```

Im zentralen Schritt, bei der Bildung des Minimums, kommt es darauf an, wo das Minimum angenommen wird.

`dist[i][j] = d[i-1][j] + 1` : ein Buchstabe ist in v zu viel, kostet 1 „Strafpunkt“

`dist[i][j] = d[i][j-1] + 1` : ein Buchstabe ist in w zu viel, kostet 1 „Strafpunkt“

`dist[i][j] = d[i-1][j-1] + e(v[i],w[j])` : ein Buchstabe wird ersetzt.

Zur eigentlichen Distanz muss die Art der Ersetzung gespeichert werden, mit deren Hilfe am Ende die Überführung der Worte ineinander angegeben werden kann.

Beispiel 3.5

Wähle $\Sigma = \{0,1,2,\ldots 9\}$ und $e(x,y) = \frac{|x-y|}{5}$

Seien $v = (0,6,1,3,1,3,9,2,3,2,8,6) \in \Sigma^n$, $w = (0,0,5,2,2,2,3,9,2,3,5,3,4) \in \Sigma^m$

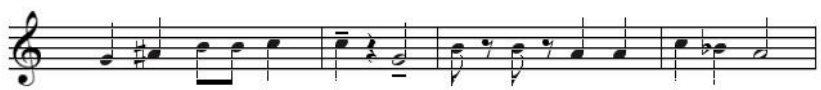
		0	**0**	**5**	**2**	**2**	**2**	**3**	**9**	**2**	**3**	**5**	**3**	**4**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	1	0	1	2	3	4	5	6	7	8	9	10	11	12
6	2	1	1.2	1.2	2.2	3.2	4.2	5.2	6.2	7.2	8.2	9.2	10.2	11.2
1	3	2	1.2	2	1.4	2.4	3.4	4.4	5.4	6.4	7.4	8.4	9.4	10.4
3	4	3	2.2	1.6	2.2	1.6	2.6	3.4	4.4	5.4	6.4	7.4	8.4	9.4
1	5	4	3.2	2.6	1.8	2.4	1.8	2.8	3.8	4.6	5.6	6.6	7.6	8.6
3	6	5	4.2	3.6	2.8	2	2.6	1.8	2.8	3.8	4.6	5.6	6.6	7.6
9	7	6	5.2	4.6	3.8	3	3.4	2.8	1.8	2.8	3.8	4.8	5.8	6.8
2	8	7	6.2	5.6	4.6	3.8	3	3.6	2.8	1.8	2.8	3.8	4.8	5.8
3	9	8	7.2	6.6	5.6	4.8	4	3	3.8	2.8	1.8	2.8	3.8	4.8
2	10	9	8.2	7.6	6.6	5.6	4.8	4	4.8	3.8	2.8	2.4	3	4
8	11	10	9.2	8.6	7.6	6.6	5.8	5	4.2	4.8	3.8	3.4	3.4	3.8
6	12	11	10.2	9.4	8.6	7.6	6.8	6	5.2	5	4.8	4	4	3.8

Überführung:

0	**0**	**5**	**2**	**2**	**2**	**3**	**9**	**2**	**3**	**5**	**3**	**4**
0	1	0.2	0.2	0.2	0.2	0	0	0	0	0.6	1	0.4
0	**-**	**6**	**1**	**3**	**1**	**3**	**9**	**2**	**3**	**2**	**8**	**6**

Nun soll der Levenshtein-Algorithmus auf das Problem der Fehlerermittlung auf ein Musikstück angewendet werden. Die Notationselemente sind dort die Buchstaben. Die Tondauer kann wieder in 32stel-Raster angegeben werden, die Tonhöhe relativ zur Mittelline. Weitere Musikalische Mittel können als Zusätze angefügt werden.

Beispiel 3.6



8/-2	8/-1k	4/0	4/0	8/1	8/1L	8/p	16/-2L	4/0	4/p	4/0	4/p	8/-1	8/-1	8/1	8/0b	16/-1

Abbildung 3.1 – Beispiel zur Konvertierung in Levenshtein-Datenstruktur

Bevor man nun die Levenshtein-Distanz anwendet, muss man sich Gedanken darüber machen, auf welche Weise die Metrik e sinnvollerweise zu wählen ist. Durch e soll die Art des Fehlers angegeben werden. Es kann ein falscher Rhythmus vorliegen oder eine falsche Tonhöhe. Es können enharmonische Verwechslungen auftreten oder fehlerhafte Ornemantik gesetzt sein. Und diese Fehler können auch kombiniert werden. Als Orientierung dient dabei der „Strafpunkt“ von 1 für ein Weglassen oder Hinzufügen einer Note.

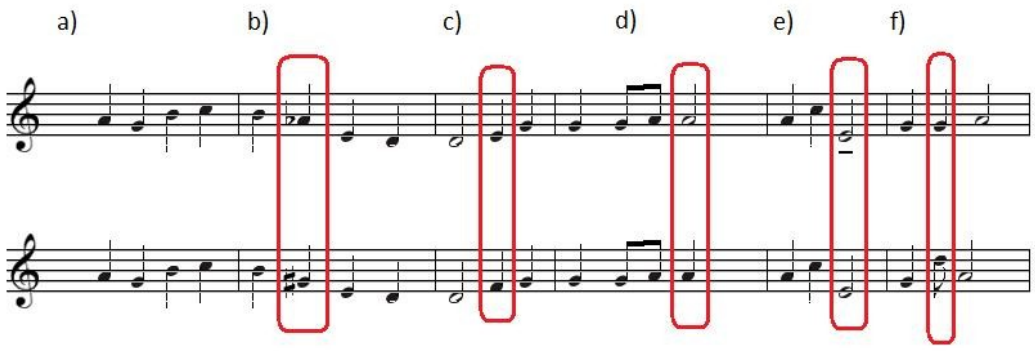


Abbildung 3.2 – Verschiedene Fehlerarten

Art des Fehlers	**Wert von e**
a) Kein Fehler	0
b) Enharmonische Verwechslung	0.02
c) Falsche Tonhöhe	Unterschied / 5, maximal 2
d) Falscher Rhythmus	Unterschied / 10, maximal 2
e) Falsche Ornemantik	0.05
f) Sonstige Fehler	maximal 2

Fehlerkombinationen erhalten die Summe der entsprechenden Werte, maximal jedoch 2.

Verbesserung durch Intervall-Erkennung

Zur verbesserten Fehlererkennung ist es häufig sinnvoller nicht nur einzelne Noten zu betrachten, sondern einen gewissen Bereich. Das Verrutschen in der Tonhöhe wird bisher noch nicht behandelt. Das folgende Beispiel zeigt die Problematik. Zur übersichtlicheren Gestaltung sind nur die Tonhöhen, nicht die Tondauern eingetragen.

Beispiel 3.7

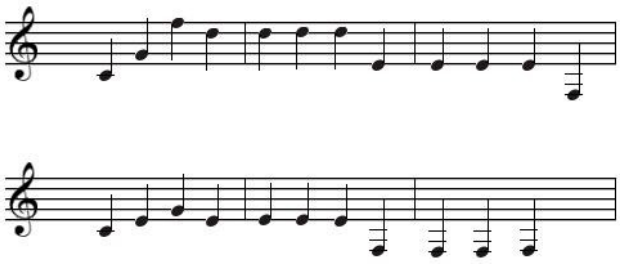


Abbildung 3.3 – Levenshtein ohne Intervallerkennung

		-6	**-2**	**4**	**2**	**2**	**2**	**2**	**-4**	**-4**	**-4**	**-4**	**-10**
	0	1	2	3	4	5	6	7	8	9	10	11	12
-6	1	0	1	2	3	4	5	6	7	8	9	10	11
-4	2	1	0.4	1.4	2.4	3.4	4.4	5.4	6	7	8	9	10
-2	3	2	1	1	2	3	4	5	5.8	6.8	7.8	8.8	9.8
-4	4	3	2	2	2.2	3.2	4.2	5.2	5	5.8	6.8	7.8	8.8
-4	5	4	3	3	3.2	3.4	4.4	5.4	5.2	5	5.8	6.8	7.8
-4	6	5	4	4	4.2	4.4	4.6	5.6	5.4	5.2	5	5.8	6.8
-4	7	6	5	5	5.2	5.4	5.6	5.8	5.6	5.4	5.2	5	6
-10	8	7	6	6	6.2	6.4	6.6	6.8	6.6	6.4	6.2	6	5
-10	9	8	7	7	7.2	7.4	7.6	7.8	7.6	7.4	7.2	7	6
-10	10	9	8	8	8.2	8.4	8.6	8.8	8.6	8.4	8.2	8	7
-10	11	10	9	9	9.2	9.4	9.6	9.8	9.6	9.4	9.2	9	8

Überführung:

-6	**-2**	**4**	**2**	**2**	**2**	**2**	**-4**	**-4**	**-4**	**-4**				**-10**
0	0.4	0.6	1	1	1	1	0	0	0	0	1	1	1	0
-6	**-4**	**-2**					**-4**	**-4**	**-4**	**-4**	**-10**	**-10**	**-10**	**-10**

Betrachtet man sich die eingegebenen Noten, könnte man auch interpretieren, dass am Anfang zwei Fehler gemacht wurden, die Intervallerkennung aber für den Rest des Stücks korrekt war und der letzte Ton vergessen wurde. Der bisherige Algorithmus interpretiert aber einige weggelassene und hinzugefügte Noten.

Dagegen kann eine Erweiterung der Levenshtein-Distanz und damit des Levenshtein-Algorithmus sorgen. Bisher gibt es nur die Möglichkeit eine Note zu entfernen, eine Note hinzuzufügen und eine Note zu ersetzen mit Fehlergewichtung e. Zusätzlich wird nun die richtige Intervallerkennung eingefügt.

Definition 3.8

Der erweiterte Levenshtein-Abstand zweier Worte $v=(v_1,...,v_n)$ und $w=(w_1,...,w_m)$ der Längen n und m ist definiert durch folgende rekursive Vorschrift:

$$d(\lambda,\lambda)=0\ ,\ d(v,\lambda)=0\ ,\ d(\lambda,w)=0$$

$$d(v,w)=$$
$$min(d(v^{n-1},w)+T,d(v,w^{m-1})+T,d(v^{n-1},w^{m-1})+e(v_n,w_m),d(v^{n-1},w^{m-1})+f(v_{n-1},v_n,w_{m-1},w_m))$$

$$f:\Sigma\times\Sigma\times\Sigma\times\Sigma\to\mathbb{R},\ f(v_0,v_1,w_0,w_1)=\begin{cases} f_0, \text{ falls } v_1-v_0=w_1-w_0 \\ \infty \text{ sonst} \end{cases}$$

Die Subtraktion ist im Sinne von Tonhöhen zu verstehen. f_0 gibt die Fehlerpunkte an.

In Beispiel 3.9 wird jetzt nur noch ein Fehler erkannt und die Möglichkeiten einer alternativen, nicht geeigneten Korrektur werden verringert. Im folgenden Beispiel wurde der Intervall-Fehlerpunkt auf 0.1 gesetzt.

Beispiel 3.9

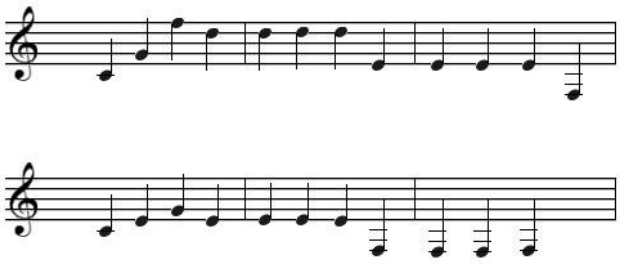


Abbildung 3.4 – Levenshtein mit Intervallerkennung

		-6	**-2**	**4**	**2**	**2**	**2**	**2**	**-4**	**-4**	**-4**	**-4**	**-10**
	0	1	2	3	4	5	6	7	8	9	10	11	12
-6	1	0	1	2	3	4	5	6	7	8	9	10	11
-4	2	1	0.4	1.4	2.4	3.4	4.4	5.4	6	7	8	9	10
-2	3	2	1	1	2	3	4	5	5.8	6.8	7.8	8.8	9.8
-4	4	3	2	2	1.1	2.1	3.1	4.1	5	5.8	6.8	7.8	8.8
-4	5	4	3	3	2.1	1.2	2.2	3.2	4.2	5	5.8	6.8	7.8
-4	6	5	4	4	3.1	2.2	1.3	2.3	3.3	4.3	5	5.8	6.8
-4	7	6	5	5	4.1	3.2	2.3	1.4	2.4	3.4	4.4	5	6
-10	8	7	6	6	5.1	4.2	3.3	2.4	1.5	2.5	3.5	4.5	5
-10	9	8	7	7	6.1	5.2	4.3	3.4	2.5	1.6	2.6	3.6	4.6
-10	10	9	8	8	7.1	6.2	5.3	4.4	3.5	2.6	1.7	2.7	3.7
-10	11	10	9	9	8.1	7.2	6.3	5.4	4.5	3.6	2.7	1.8	2.7

Überführung:

-6	**-2**	**4**	**2**	**2**	**2**	**2**	**-4**	**-4**	**-4**	**-4**	**-10**
0	0.4	0.6	0.1	0.1	0.1	0.1	0.1	0.1	0.1	1	0
-6	**-4**	**-2**	**-4**	**-4**	**-4**	**-4**	**-10**	**-10**	**-10**		**-10**

Die eigentliche Implementierung erfolgt dann in etwa nach folgendem Code. Die Matrix `dist` enthält den tatsächlichen Levenshtein-Abstand, die Matrix `ref` enthält die Art des Fehlers, also wo das Minimum bei der Berechnung der Levenshtein-Distanz angenommen wurde.

Pseudocode:

```
v = try;
w = template;
dist[0][0] = 0;
ref[0][0] = „fertig“;
for (i=1;i<=n;i++) {
     dist[i][0] = i;
     ref[i][0] = „Note zu viel“;
}
for (j=1;j<=n;j++) {
     dist[0][j] = j;
     ref[0][j] = „Note zu wenig“;
}
for (i=1; i<=n; i++) {
     for (j=1; j<=m; j++) {
          if (i==1 || j==1) {
               dist[i][j] =
                    min(d[i-1][j]    + 1,
                         d[i][j-1]   + 1,
                         d[i-1][j-1] + e(v[i],w[j]));
               ref[i][j] =
                    (min == d[i-1][j]+1 ? „Note zu viel“ :
                    (min == d[i][j-1]+1 ? „Note zu wenig“ :
                    (min == d[i-1][j-1]+e ? eRef(.,.) )))
          }
          else {
               dist[i][j] =
                    min(d[i-1][j]    + 1,
                         d[i][j-1]   + 1,
                         d[i-1][j-1] + e(v[i],w[j]),
                         d[i-1][j-1] + f(v[i-1],v[i],w[j-1],w[j]));
               ref[i][j] =
                    (min == d[i-1][j]+1 ? „Note zu viel“ :
                    (min == d[i][j-1]+1 ? „Note zu wenig“ :
                    (min == d[i-1][j-1]+e ? eRef(.,.) :
                    (min == d[i-1][j-1]+f ? fRef(.,.,.,.) ))))
          }
     }
}
return (dist,ref);
```

Die Funktionen `eRef` und `fRef` liefern die Art des Fehlers zurück. In der oben aufgeführten Tabelle der Werte von e entsprechen die Werte von `eRef` gerade der Legende links, also der Art der Fehler. Da `f` nur zwei Werte annehmen kann, nimmt auch `fRef` nur zwei Werte an, nämlich „`Intervall korrekt erkannt`“ und „`Intervall nicht korrekt erkannt`“. Eine ausführliche Erklärung, wie der Algorithmus genau operiert, wird anhand eines kleinen Beispiels in Kapitel 4 gegeben.

Vergleich mit Rasterung

Im vorangegangenen Kapitel wurde ein erster Algorithmus beschrieben, der Notendiktate korrigieren soll. Dabei wurden die Musikstücke in Raster eingeteilt. Der Levenshtein-Algorithmus vergleicht nur einzelne Noten miteinander.

Satz 3.10

Sei n die Anzahl der Noten im kürzeren, m die Anzahl der Noten im längeren Musikstück.
Laufzeit des vorgestellten Levenshtein-Algorithmus: $O(nm)$

Beweis:

Im Pseudocode findet man zwei ineinander geschachtelte Schleifen, die jeweils über die Länge der Musikstücke gehen.

Setzt man voraus, dass beide Musikstücke etwa gleich lang sind, hat man eine quadratische Laufzeit. Wohlgemerkt handelt es sich dabei um die Anzahl der Noten, nicht um die Länge in einem 32stel-Raster. Bei sehr langen Musikstücken mit einigen Fehlern ist der Levenshtein-Algorithmus also wesentlich effizienter.

Ein weiterer entscheidender Vorteil dieses Verfahrens ist, dass es direkt die Art der Fehler mitliefert. Da das Programm vorwiegend Anwendung bei der Korrektur von Notendiktaten findet und es dort in erster Linie hilfreich ist zu wissen wie falsch das Diktat war und an welchen Stellen noch geübt werden muss, ist dies sehr günstig. Es ist also möglich mit Hilfe des Levenshtein-Algorithmus direkt eine Anzeige zu generieren, die diese Resultate des Notendiktats unmittelbar bereitstellt. Beim Verfahren durch Rasterung müsste man im Anschluss an die Fehlerermittlung noch die Art der Fehler bestimmen.

Doch ein geeignetes Verfahren zeichnet sich in erster Linie durch vernünftige Ergebnisse aus. In Kapitel 5 werden Beispieleingaben für den Levenshtein-Algorithmus gemacht und gleichzeitig auch die Grenzen des Verfahrens aufgezeigt. Das folgende Beispiel soll zeigen, wo die Unterschiede bei den Resultaten der beiden Verfahren liegen.

Beispiel 3.11

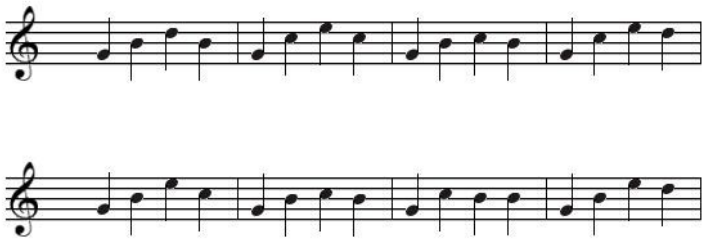


Abbildung 3.5 – Vergleich zwischen Levenshtein und Rasterung (1)

Wendet man die Rasterung auf diese Musikstücke an, wird direkt im ersten Schritt das längste zusammenhängende Muster ermittelt. Trotz verschiebung ist es die optimale Übereinstimmung. Aber ein menschlicher Korrektor würde vermutlich die einzelnen Unterschiede in den Tonhöhen anstreichen.

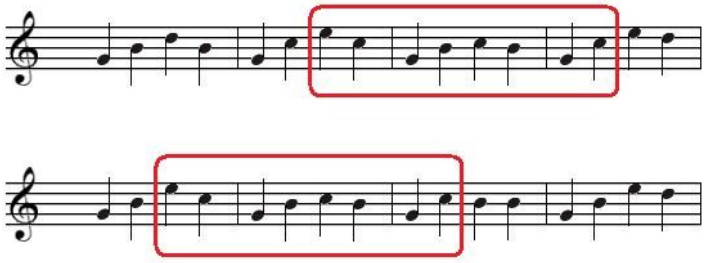


Abbildung 3.6 – Vergleich zwischen Levenshtein und Rasterung (2)

Am Ende liefert der Vergleich durch Rasterung ein Resultat, das eher mäßig zufriedenstellt.

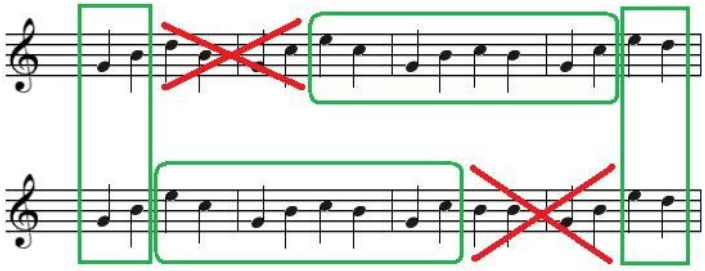


Abbildung 3.7 – Vergleich zwischen Levenshtein und Rasterung (3)

Der Levenshtein-Algorithmus wird hingegen alle Noten mit falschen Tonhöhen als falsch markieren, die Rhythmusstruktur jedoch beibehalten.

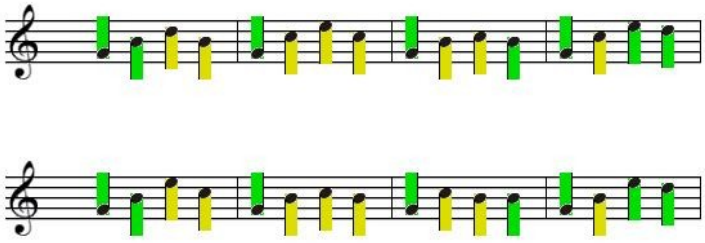


Abbildung 3.8 – Vergleich zwischen Levenshtein und Rasterung (Levenshtein)

Ein Satz zur Theorie der Levenshtein-Distanz folgt an dieser Stelle, da mit Hilfe des Tableaus die Beweisidee leicht nachvollzogen werden kann.

Satz 3.12

Seien $v \in \Sigma^n$ und $w \in \Sigma^m$. Teilt man v und w in zwei Teilworte auf, wobei die Längen der ersten Teilworte k_v und k_w sein sollen, also $v = (v', v'')$ und $w = (w', w'')$ mit $v' = v^{k_v}$ und $w' = w^{k_w}$. Dann gilt:

$$d(v', w') + d(v'', w'') \geq d(v, w)$$

Beweis:

Die rekursive Vorschrift des Levenshtein-Abstands bildet ein Minimum über vier Zahlen. Diese Zahlen hängen von den bisher berechneten Abständen ab. Bei der Berechnung von $d(v, w)$ stehen in jedem Schritt alle um eins kleineren Teilworte zur Verfügung. Bei der Berechnung von $d(v', w')$ und $d(v'', w'')$ stehen jedoch keine Kombinationen von Buchstaben aus v' und w'' sowie Kombinationen aus v'' und w' zur Verfügung. Da jedoch stets minimiert wird, muss $d(v, w)$ mindestens so klein sein wie $d(v', w') + d(v'', w'')$.

Die folgende Skizze veranschaulicht den Beweis.

		w'	**w''**
	0	...	...
v'	...	d(v',w')	Nicht zur Verfügung
v''	...	Nicht zur Verfügung	d(v',w') + d(v'',w'')

Wenn diese Eigenschaft für zwei Teilworte gilt, dann gilt sie auch für beliebig viele Teilworte. Insbesondere auch dann, wenn die Teilworte so gewählt sind, dass genau an den Taktgrenzen geteilt wurde. Das bedeutet, dass der Levenshtein-Algorithmus ein mindestens genau so gutes Verfahren darstellt, wie wenn man die Takte paarweise vergleichen würde. Dadurch ist eine Rasterung im Levenshtein-Algorithmus implizit gegeben.

4. Veränderungen am Noteneditor

In diesem Kapitel wird die Visualisierung der Ergebnisse des Vergleichs im Noteneditor beschrieben. Es wird außerdem noch kurz auf die Umsetzung des Levenshtein-Algorithmus und die Eingabemaske eingegangen.

Visualisierung der Resultate

Analog zum Vorgehen beim Laden einer Datei in den Musiknoteneditor erfolgt das Laden des Musters. Ein einheitliches und intuitiv bedienbares GUI-Konzept ist dadurch gegeben. Nach dem Laden der Datei durch php-Funktionen erfolgt der Vergleichs-Algorithmus ausschließlich mittels JavaScript. Das eingegebene Notendiktat wird also nicht an den Server übermittelt. Auch die Visualisierung der Resultate und die Bewertung wurden in JavaScript implementiert.

Das Muster wird in die erste Notenzeile geladen, der ursprünglich eingegebene Versuch wird nach unten verschoben. Anschließend werden die Noten je nach Art des Fehlers farblich markiert. Der Levenshtein-Algorithmus liefert Resultate der Form `dist` und `ref`. Aus `ref` wird die Art des Fehlers gelesen und dadurch die Farbe bestimmt, die eine Note erhält.

Pseudocode:

```
i = try.length;
j = template.length;
while (i > 0 || j > 0) {
     if (ref[i][j] == „Note zu viel“) {
          i--;
          markiere(try[i],rot);
     }
     else if (ref[i][j] == „Note zu wenig“) {
          j--;
          markiere(template[j],rot);
     }
     else {
          i--; j--;
          markiere(try[i], Farbe gemäß ref[i][j]);
          markiere(template[j], Farbe gemäß ref[i][j]);
     }
}
```

Ermittlung der Bewertung

Der Levenshtein-Algorithmus liefert Resultate der Form `dist` und `ref`. Daraus lässt sich eine Gesamtbewertung erstellen. Nach Regel 3.3 ist eine obere Schranke für den Gesamtfehler die Summe der Längen der beiden Musikstücke. Dann ist eine Bewertung möglich, die angibt wieviel Prozent des Stücks richtig waren.

Pseudocode:

```
n = try.length; m = template.length;
GesBewertung = (n+m - dist[n][m]) / (n+m);
```

Man beachte, dass es bei einer Berechnung in dieser Form sehr schwierig ist eine Bewertung von 0% zu erreichen. Denn sobald eine Note mit irgendeiner anderen übereinstimmt, egal an welcher Position, wird der Levenshtein-Abstand nicht maximal.

Weiterhin beachte man, dass für die Berechnung der Bewertung die Fehlermaße von e und f verwendet werden. Prinzipiell wäre auch denkbar eine andere Gewichtung für verschiedene Fehlerarten vorzunehmen. Setzt man allerdings die eigene Gewichtung bereits in den Levenshtein-Algorithmus ein, interpretiert dieser die Unterschiede in den Musikstücken stets so, dass eine möglichst günstige Fehlerkombination gewählt wird. Dies entspricht auch gängigen Konventionen, im Zweifel für den Angeklagten.

Eine weitere Ausgabe erfolgt durch eine Prozentuale Angabe der Art der Fehler. Da diese durch den Levenshtein-Algorithmus bereits ermittelt wird, ist lediglich eine Aufsummierung notwendig.

Pseudocode:

```
i = try.length; j = template.length;
for all (Art) {
     fehler[Art] = 0;
}
while (i > 0 || j > 0) {
     if (ref[i][j] != „richtig“) {
          fehler[ref[i][j]]++;
     }
     if (ref[i][j] == „Note zu viel“) i--;
     else if (ref[i][j] == „Note zu wenig“) j--;
     else {
          i--; j--;
     }
}
fehlersumme = 0;
for all (Art) {
     fehlersumme += fehler[Art];
}
for all (Art) {
     fehler[Art] /= fehlersumme;
}
```

Sowohl die Gesamtbewertung als auch die Aufteilung nach der Art der Fehler werden nach der Ausführung des Vergleichs im Noteneditor angezeigt. Dabei wird ähnlich vorgegangen wie beim Dialog zum Öffnen von Dateien.

Weitere Änderungen am Musiknoteneditor in Bezug auf die Wartbarkeit und Erweiterbarkeit werden in Kapitel 5 und im Anhang B beschrieben.

Ausführliches Beispiel

Eingabe des Versuchs und Laden des Musters

Der Versuch wird in den Editor eingegeben, das Muster ist als Datei hinterlegt und wird geladen. In diesem Beispiel nehmen wir an sie sehen wie folgt aus.

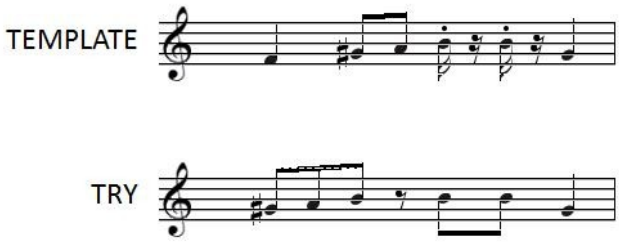


Abbildung 4.1 – Ausführliches Beispiel (1)

1. Umwandlung in Vergleichs-Datenstruktur

Beide Musikstücke werden umgewandelt.

TEMPLATE	8/-3	4/-2k	4/-1	2/0s	2/p	2/0s	2/p	8/-2

TRY	4/-2k	4/-1	4/0	4/p	4/0	4/0	8/-2

2. Anwendung von Levenshtein

2a. Raster wird Initialisiert

		8/-3	**4/-2k**	**4/-1**	**2/0s**	**2/p**	**2/0s**	**2/p**	**8/-2**
	0	1	2	3	4	5	6	7	8
4/-2k	1								
4/-1	2								
4/0	3								
4/p	4								
4/0	5								
4/0	6								
8/-2	7								

2b. Erste Zeile und Spalte werden noch ohne `f`, also ohne Intervallerkennung ermittelt

dist		**8/-3**	**4/-2k**	**4/-1**	**2/0s**	**2/p**	**2/0s**	**2/p**	**8/-2**
	0	1	2	3	4	5	6	7	8
4/-2k	1	0.7	1	2	3	4	5	6	7
4/-1	2	1.7							
4/0	3	2.7							
4/p	4	3.7							
4/0	5	4.7							
4/0	6	5.7							
8/-2	7	6.2							

```
dist[1][1] = min(dist[0][1] + 1, dist[1][0] + 1,
                 dist[0][0] + e( 4/-2k , 8/-3 ) )
           = dist[0][0] + ( 0.4 + 0.3 ) = 0 + 0.7 = 0.7

dist[1][2] = min(dist[0][2] + 1, dist[1][1] + 1,
                 dist[0][1] + e( 4/-2k , 4/-2k ) )
           = dist[0][1] + ( 0.0 + 0.0 ) = 1 + 0.0 = 1

dist[2][1] = min(dist[1][1] + 1, dist[2][0] + 1,
                 dist[1][0] + e( 4/-1 , 8/-3 ) )
           = dist[1][1] + 1 = 1.7
```

Denn `dist[1][0] + e( 4/-1 , 8/-3 ) = 1 + ( 0.4 + 0.4 ) = 1.8 > 1.7`

2c. Auch die Referenz, die Art des Fehlers, muss gespeichert werden (H = Höhe, D = Dauer)

ref		**8/-3**	**4/-2k**	**4/-1**	**2/0s**	**2/p**	**2/0s**	**2/p**	**8/-2**
	fertig	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.
4/-2k	zu viel	H D	richtig	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.
4/-1	zu viel	zu viel							
4/0	zu viel	zu viel							
4/p	zu viel	zu viel							
4/0	zu viel	zu viel							
4/0	zu viel	zu viel							
8/-2	zu viel	H							

2d. Die übrigen Einträge rekursiv bestimmen unter Beachtung von `f`.

dist		**8/-3**	**4/-2k**	**4/-1**	**2/0s**	**2/p**	**2/0s**	**2/p**	**8/-2**
	0	1	2	3	4	5	6	7	8
4/-2k	1	0.7	1	2	3	4	5	6	7
4/-1	2	1.7	0.8	1	2	3	4	5	6
4/0	3	2.7	1.8	1	1.25	2.25	3.25	4.25	5.25
4/p	4	3.7	2.8	2	2.25	1.45	2.45	3.45	4.45
4/0	5	4.7	3.8	3	2.25	2.45	1.7	2.7	3.7
4/0	6	5.7	4.8	4	3.25	3.45	2.7	3.7	3.5
8/-2	7	6.2	5.8	5	4.25	4.45	3.7	4.7	3.7

2e. Analog werden die Fehlerarten ermittelt (A = Artikulation)

ref		**8/-3**	**4/-2k**	**4/-1**	**2/0s**	**2/p**	**2/0s**	**2/p**	**8/-2**
	fertig	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.
4/-2k	zu viel	H D	richtig	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.
4/-1	zu viel	zu viel	H	richtig	zu wen.	zu wen.	zu wen.	zu wen.	zu wen.
4/0	zu viel	zu viel	zu viel	H	D A	zu wen.	zu wen.	zu wen.	zu wen.
4/p	zu viel	zu viel	zu viel	zu viel	zu viel	D	zu wen.	zu wen.	zu wen.
4/0	zu viel	zu viel	zu viel	zu viel	zu viel	zu viel	D A	zu wen.	zu wen.
4/0	zu viel	zu viel	zu viel	zu viel	zu viel	zu viel	zu viel	zu viel	H D
8/-2	zu viel	H	zu viel	zu viel	zu viel	zu viel	zu viel	zu viel	richtig

2f. Die Bewertung wird ermittelt

```
GesBewertung =
      (n+m - dist[n][m]) / (n+m) = (7+8 - 3.7) / (7+8) = 11.3/15
```

Gesamtbeurteilung liegt also bei 75,3%. Man beachte, dass diese Bewertung von der Wahl der Parameter in `e` und `f` abhängt. Will man für Eingaben dieser Art, die einige Fehler aufweisen, eine schlechtere Bewertung ermittelt bekommen, muss man mehr Fehlerpunkte einstellen.

Es kann außerdem eine Artbezogene Darstellung der Fehler ermittelt werden.

```
fehler[„Note zu wenig“] = 2
fehler[„Note zu viel“] = 1
fehler[„Dauer“] = 3
fehler[„Artikulation“] = 2

fehlersumme = 8
```

Es sind also anteilig 25% Noten/Pausen-Auslassungs-Fehler, 12.5% Noten/Pausen-Überschuss-Fehler, 37.5% Fehler in der Dauer und 25% Artikulations-Fehler.

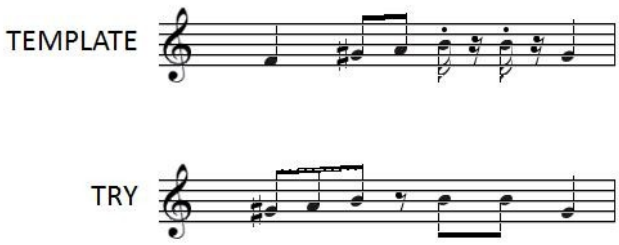


Abbildung 4.2 – Ausführliches Beispiel (2)

Vergleicht man diese Resultate mit der Eingabe, kann man dies als vernünftig interpretieren. Echte Praxistauglichkeit kann jedoch nur mit realistischen Mustern und Eingaben gezeigt werden.

5. Fazit und Resultate

Dieses Kapitel beschreibt die Eignung des Programms zur Anwendung. Es werden Beispiele gegeben, wie mögliche Eingaben und Resultate aussehen könnten. Es wird ein Ausblick auf mögliche Erweiterungen gegeben, die zukünftige Arbeiten betreffen könnten. Außerdem werden Verbesserungsvorschläge für das in dieser Arbeit entwickelte Verfahren aufgezeigt.

Beispieleingaben

Beispiel 5.1

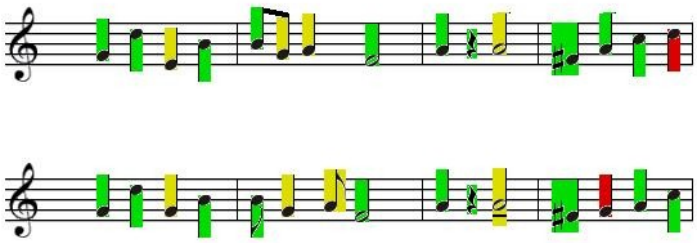


Abbildung 5.1 – Levenshtein-Anwendung auf einfache Fehler

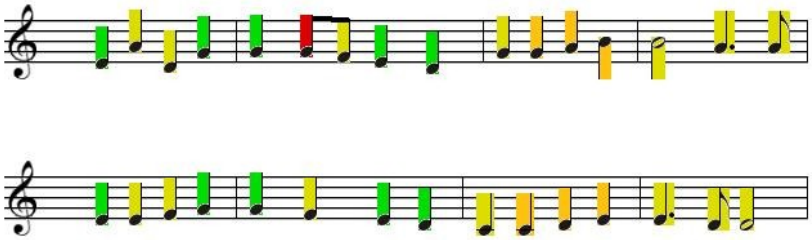


Abbildung 5.2 – Levenshtein-Anwendung auf komplexe Fehler

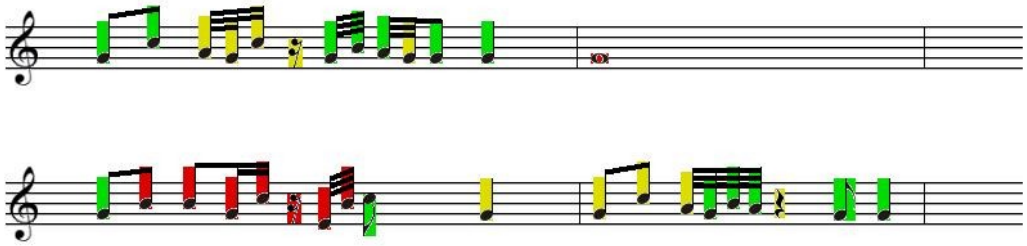


Abbildung 5.3 – Grenzen von Levenshtein

Für einfache gut isolierte Fehler liefert der Levenshtein-Algorithmus die erwarteten Ergebnisse (Abbildung 5.1). Auch bei komplexeren Fehlern konstruiert er Korrekturen, die sinnvoll erscheinen (Abbildung 5.2). Schwach zeigt sich das Verfahren nur bei sehr vielen und unübersichtlichen Diktaten, in denen gravierende Fehler begangen werden (Abbildung 5.3). Da solche Fälle jedoch sehr konstruiert sind, kann man sie in der Praxis vernachlässigen. Eine Justierung der Parameter könnte ebenfalls Abhilfe schaffen. In diesem Punkt muss man das Programm erst erproben, um wirklich fundierte Entscheidungen diesbezüglich treffen zu können. Die Beispiele zeigen jedoch, dass das Programm für den ersten Einstieg als Unterstützung beim Trainieren des Gehörs geeignet ist. Außerdem zeigen Satz 3.4 und Satz 3.12 auch die theoretische Eignung.

Verbesserung des Levenshtein-Verfahrens

Eine Idee zur Verbesserung der Resultate ist die Kombination der beiden in dieser Arbeit vorgestellten Verfahren. Beim Vergleich durch Rasterung wird eine Übereinstimmung nur bei absoluter Gleichheit akzeptiert. Stattdessen könnte der Levenshtein-Algorithmus verwendet werden und die Länge der Übereinstimmung abzüglich der Levenshtein-Distanz maximiert werden. Dadurch würden auch Muster mit kleinen Fehlern zwischendurch erkannt werden. Außerdem würde dieses Verfahren die Art des Fehlers auch unmittelbar bestimmen. Dies würde allerdings die Laufzeit weiter erhöhen und die Nachteile der Rasterung nicht vollständig eleminieren.

Der folgende Pseudocode benutzt die Implementierung der Levenshtein-Distanz `dist`. Im Übrigen folgt er der Vorgehensweise aus Kapitel 2.

Pseudocode:

```
compare(try,template) {
     if (try == empty || template == empty) return;
     (i1,iL,iT) = getLevenshteinSync(try,template);
     compare(try(0,i1,0,maxJ),template(0,iT,0,maxJ));
     compare(try(i1+iL,maxI,0,maxJ),template(iT+iL,maxI,0,maxJ));
}

getLevenshteinSync(try,template) {
     for (i1 = 0; i1 < try.length; i1++) {
          for (iT = 0; iT < template.length; iT++) {
               for (iL = 1; iL < try.length; iL++) {
                    syncCandidat = (i1,iL,iT);
                    syncCandidatDist = iL -
                    dist(try(i1,iL,0,maxJ),template(iT,iL,0,maxJ));
               }
          }
     }
     return(max(syncCandidatDist) ? syncCandidat);
}
```

Stattdessen könnten Bereiche anderer Art in die Berechnung der Levenshtein-Distanz einbezogen werden. Bei den Beispieleingaben wurde gezeigt, dass besonders Bereiche mit sehr vielen Notationselementen problematisch sind.

Beispiel 5.2

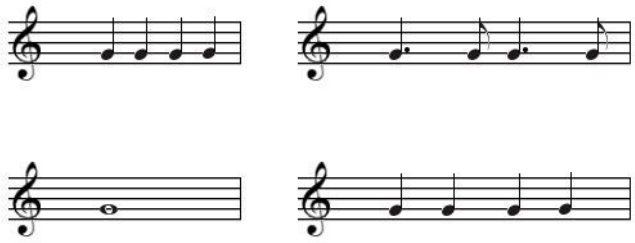


Abbildung 5.4 – Verbesserung von Levenshtein

Im linken Fall könnte man als Vorschrift zur Minimumbildung folgendes nehmen:

$$d(v,w)=min(\ldots,d(v^{n-1},w^{m-4})+g(v_n,w_{m-3},w_{m-2},w_{m-1},w_m))$$

Im rechten Fall könnte man als Vorschrift zur Minimumbildung folgendes nehmen:

$$d(v,w)=min(\ldots,d(v^{n-2},w^{m-2})+h(v_{n-1},v_n,w_{m-1},w_m))$$

Dabei sind `g` und `h` ähnlich wie `f` angelegt. Sie nehmen nur falls der oben abgebildete Fall eintritt einen kostanten Wert an, ansonsten den Wert unendlich.

Man kann dadurch einen Raster-Effekt erzielen und dessen Vorteile nutzen. Prinzipiell ist die Levenshtein-Distanz noch um andere Arten von Fehlern erweiterbar. Man muss allerdings beachten, dass für jede solche Erweiterung auch die Anfälligkeit gegen ungewolltes Ausnutzen gewisser Regeln steigt. Besonders problematisch ist dies, wenn die Strafpunkte vergleichsweise gering ausfallen.

Weiterentwicklungen am Noteneditor

Da das Projekt rund um den Musiknoteneditor schon von einigen Studenten bearbeitet wurde und vorraussichtlich auch noch weitere Arbeiten folgen werden, ist es sehr wichtig sauber zu arbeiten und sich an gewisse Spielregeln zu halten. Eine aussagekräftige Kommentierung und strukturierter Code sind absolut notwendig, um das Programmierte für andere und auch für sich selbst in mittlerer Zukunft verständlich zu machen. Auch wenn manchmal Effizienz und Laufzeit unter einer ordentlichen Programmierung leiden, ist diese letztendlich entscheidend für das Gelingen des Projekts. Die im Anhang befindliche Hilfe zum Einstieg in das Projekt soll auch auf Richtlinien und Regeln hinweisen.

Dokumentierung zum Quelltext geschieht ohnehin meist in Form der Abschlussarbeiten. Oft ist dort jedoch die direkte Auseinandersetzung mit JavaScript- und php-Dateien nicht vorhanden. Daher ist es sinnvoll, dass am Anfang jeder Datei eine kurze Erklärung vorhanden ist, die den Nutzen dieser Datei beschreibt. Wichtig dabei ist auch eine klare Kapselung von Inhalten. Zusammengehöriger Quelltext sollte jedoch in die selbe Datei kommen. Man lese hierzu auch das Schlusswort in [Vel09] (Kapitel 9.5).

Das Herausarbeiten von Fehlern gestaltet sich oft leichter, wenn es eine Liste von vorhandenen Fehlern gibt. Eine nicht zum Programm, aber zum Quelltext gehörende Liste solcher Fehler ist nun dabei. Diese darf und soll um andere gefundene Fehler ergänzt und natürlich auch von Fehlern befreit werden, wenn diese im Code behoben wurden. Außerdem ist es sinnvoll eine aktuelle Liste von implementierter und noch nicht implementierter, aber gewünschter, Funktionalität zu führen. In den vorangegangenen Arbeiten wurden im Ausblick gute Ideen für Weiterentwicklungen vorgestellt und manchmal auch Lösungsansätze beschrieben.

Die aktuell noch offenen Erweiterungen werden hier nochmal kurz vorgestellt.

Setzen von Vorschlagsnoten

Vorschlagsnoten sind Noten, die extrem kurz erklingen sollen. Sie haben keine Dauer, daher würde sich bei Hinzufügen einer Vorschlagsnote das Taktgefüge nicht ändern. Sie werden als kleine Noten vor anderen Noten dargestellt. Ansätze zur Vorgehensweise sind in [Ess08] (Kapitel 8.3.1) beschrieben.

Auswählen verschiedener Taktstriche

Es gibt ein paar Arten von Taktstrichen, die Wiederholungen oder neu beginnende Abschnitte anzeigen sollen. Weitere Informationen finden sich in [Ess08] (Kapitel 8.3.1).

Setzen von Akkorden

Bisher ist lediglich die Möglichkeit gegeben mittels verschiedener Layer gewisse Töne gleichzeitig erklingen zu lassen. Das Setzen von Akkorden in einer Notenzeile ist bisher nicht möglich. Auch das direkte Setzen von Akkorden ist nicht implementiert.

Setzen von n-olen

Es wurde bereits Funktionalität zum Erstellen von Triolen entwickelt. Diese kann um weitere n-olen erweitert werden, indem die bestehenden Klassen verwendet werden.

Setzen weiterer Ornemantik

Es wurden bereits einige Artikulationszeichen zum Musiknoteneditor hinzugefügt. Die Möglichkeiten der Notation sind damit aber noch nicht ausgeschöpft.

Abspiel-Funktion

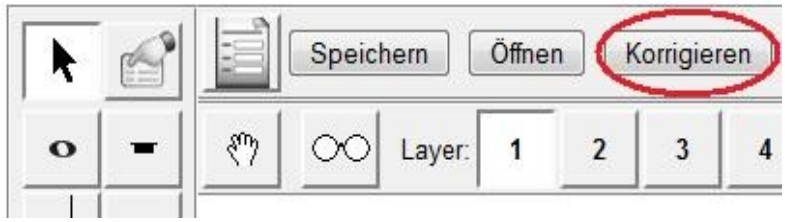
Eingaben in den Noteneditor sollten auch abgespielt werden können. Weitere Informationen dazu sind in [Rie09] (Kapitel 7.4.2) zu finden.

Verknüpfung mit anderen Programmen

Eine Erweiterung, die eng mit dieser Arbeit verknüpft ist und sehr hohe Priorität genießt, ist diese. Der Fachbereich Musik wünscht sich, dass das Programm mit anderen verknüpft werden soll. Musikstudenten verwenden bereits jetzt zur Vorbereitung auf die Gehörbildungs-Prüfung ein Programm, das jedoch eher auf rhythmisches Notenlesen wertlegt. Herr Georgi wäre zu einer Fortsetzung der Arbeit an diesem Thema sicher begeistert und würde als kompetenter Ansprechpartner zur Verfügung stehen.

A. Bedienungsanleitung

Ein Klick auf den Korrigieren-Button öffnet einen Dialog, um eine Datei zu wählen, die als Muster dienen soll.



Der Algorithmus vergleicht die Eingabe (wird nach unten verschoben) mit dem Muster (erscheint oben).

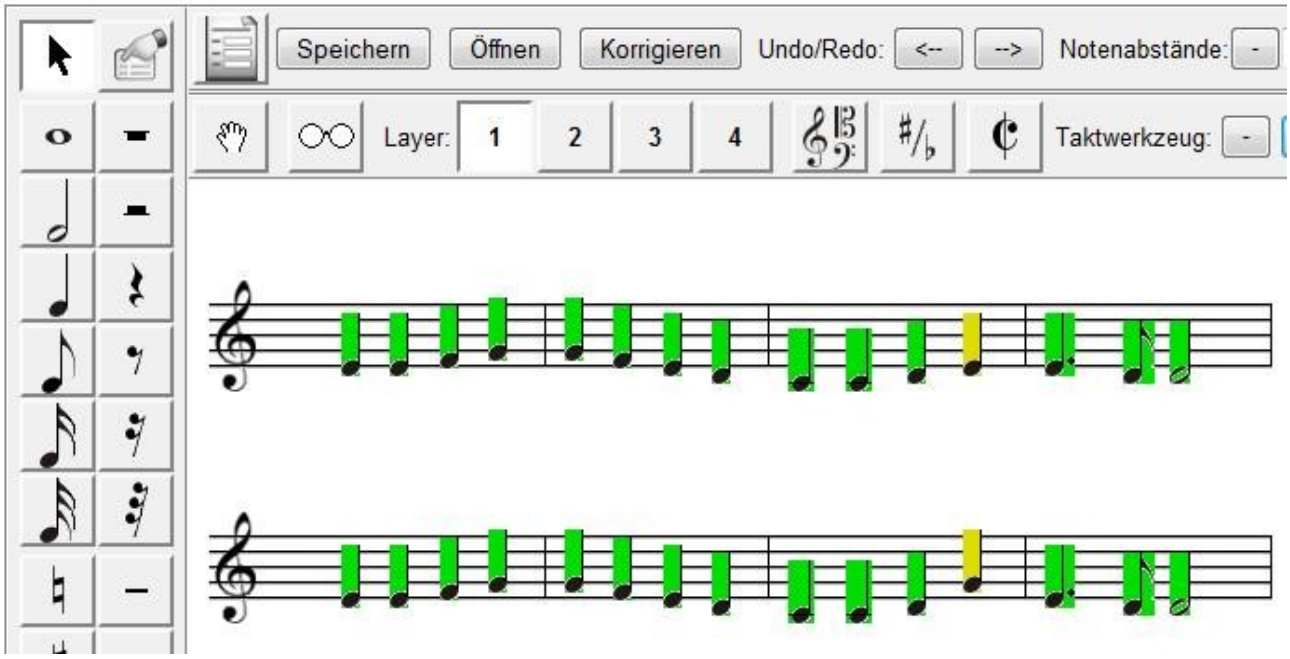


Abbildung A.0 / A.1 / A.2 – Bedienungsanleitung

Die Farbgebung bleibt bei anschließender Bearbeitung erhalten. Um die ursprüngliche Eingabe wiederherzustellen, empfiehlt es sich vorher zu speichern. Durch Laden oder Zurücksetzen wird die Farbgebung wieder entfernt.

Dieses Resultat kann auch gespeichert werden; auf die gleiche Weise, wie auch andere Musikstücke gespeichert werden.

Die anschließend erscheinende Anzeige, die Auskunft über die Art der Fehler gibt, stellt eine zusammengefasste Bewertung dar. Sie dient dem Benutzer lediglich als Hilfestellung, damit er nicht selbst die Fehler zusammenzählen muss, sondern schnell und übersichtlich die Qualität der Leistung dargestellt bekommt.

B. Anleitung zur Weiterentwicklung

Der Einstieg in den von anderen erstellten Quelltext fällt nicht immer leicht. Dieser Abschnitt soll denjenigen helfen, die Weiterentwicklungen durchführen möchten, aber nicht wissen wo sie anfangen sollen. Dabei sollte nicht nur die Struktur des Quellcodes verstanden werden, sondern vor allem bei der anstehenden Programmierarbeit gewisse Konventionen befolgt werden, die in Regeln und Richtlinien zusammengefasst sind.

Regeln und Richtlinien

Viele dieser Regeln scheinen trivial und längst bekannt. Doch ein Blick in den Code zeigt, dass sie nicht immer befolgt wurden. Dies geschieht häufig unter Zeitdruck oder wegen unsauberem Arbeiten. Die Lesbarkeit des Codes ist allerdings für alle weiteren Entwickler und auch für den aktuellen Entwickler enorm wichtig. Deshalb sollten diese Regeln wirklich beherzigt werden.

Jede Datei sollte klar von anderen abgegrenzt sein. Funktionalität sollte gekapselt werden. Zu Beginn jeder Datei sollte kurz ihr Zweck erklärt werden.

Kommentierung fördert das Verständnis von komplizierten Codeblöcken. Gerne darf auch Bezug auf Kapitel oder Textpassagen der schriftlichen Arbeit genommen werden. Zu viel Kommentierung oder aussagelose Kommentierung behindert dagegen eher.

Der Quellcode sollte durch Einrückungen seine Struktur erkennen lassen. Nach einer öffnenden geschweiften Klammer wird weiter eingerückt, nach einer schließenden geschweiften Klammer wird weniger weit eingerückt.

Mit der Zeit haben sich nicht nur Ideen zur Weiterentwicklung, sondern auch einige häufig auftretende Fehler eingeschlichen. Das Erstellen und Pflegen einer Liste über alle Fehler im Noteneditor hilft bei der Bekämpfung.

Werden neue Fehler entdeckt, sollten sie dort eingetragen werden. Werden Fehler vollständig oder teilweise behoben, sollte das dort auch vermerkt werden. Auch Ideen zur Fehlerbehebung sind bereits hilfreich und sollten notiert werden.

Einstieg in den Quellcode

Bekommt man ein Problem und eine große Menge an Programmcode vorgesetzt, ist es oft schwierig das Problem in den Kontext einzuordnen.

Zunächst ist es hilfreich sich mit der Struktur der Notationselemente vertraut zu machen. Ein Musikstück wird durch `score.js` repräsentiert. Darin sind geschachtelt Takte, Stimmen, Layer und Noten enthalten. Außer die Layer haben alle ihre eigene Datei, in denen die Eigenschaften enthalten sind.

Handelt es sich um eine Erweiterung an unmittelbarer Funktionalität, ist es sinnvoll sich die Hauptdatei `editor.js` anzusehen. Sie trägt die Funktionalität, um elementare Notationselemente hinzuzufügen, zu löschen, rückgängig zu machen, etc. Diese Datei ist nicht gut gekapselt, es wäre besser die Funktionalitäten aufzuteilen.

In `in_out.js` befindet sich die Funktionalität zum Speichern und Laden. Diese wird allerdings, da der Zugriff auf die Festplatte nur mit php möglich ist, von anderen Dateien aus aufgerufen.

Wichtig für das Verständnis des Quelltextes ist eine kurze Einarbeitung in DOM-Strukturen, dabei kann die Seite [W3C] hilfreich sein.

Es ist außerdem hilfreich die Arbeit von Thorsten Essig zu lesen. Diese beschreibt die ursprünglichen und elementaren Funktionen des Noteneditors. Allgemein sind auch die Passagen zum Ausblick der anderen Arbeiten empfehlenswert und natürlich geben diese auch Auskunft über die dort implementierten Erweiterungen.

Die Entwickler können außerdem auch persönlich befragt werden. Ich stehe gerne auch als Einstiegshilfe zur Verfügung.

Literaturverzeichnis

[Ess08] Thorsten Essig
Diplomarbeit: Webbasiertes Erstellen und Editieren von Notentext
August 2008

[Rie09] Leo Ries
Bachelorarbeit: Erweiterung eines webbasierten Noteneditors um Artikulationszeichen und Haltebögen
März 2009

[Vel09] Stanislav Velychko
Bachelorarbeit: Erweiterung eines Web-basierten Noteneditors um Legatobögen, Triolen, 32-tel Noten und Pausen sowie Undo- bzw. Redo-Funktionalität
Juli 2009

[Lev] Effiziente Levenshtein-Implementierungsansätze
http://www.levenshtein.de
Stand: Juli 2010

[Mus] Gehörbildung online
http://www.musikwissenschaften.de/interaktiv/gb/basis.htm
Stand: Juli

[W3C] Document Object Model
http://www.w3.org/DOM
Stand: Juli 2010

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen Hilfsmittel und Quellen als die im Literaturverzeichnis angegebenen benutzt habe. Alle Stellen, die wörtlich oder dem Sinn nach aus veröffentlichten Quellen stammen, sind als solche kenntlich gemacht worten. Alle Quelltexte oder Ausschnitte davon sowie Abbildungen sind von mir erstellt oder mit einem entsprechenden Hinweis versehen worden. Ich versichere auch, dass diese Arbeit weder vollständig noch auszugsweise und auch nicht in ähnlicher Form für eine andere Prüfung verwendet wurde.

Martin Unold
14. Juli 2010