# Einsatz der Editierdistanz für einen Musikerkennungsalgorithmus


Mattia Brand, M22a
Betreut durch Stefan Rothe

Gymnasium Kirchenfeld (MN)
Bern, 2021

# Inhaltsverzeichnis

# 1. Einleitung und Ziele

Es kann des Öfteren geschehen, dass einem plötzlich ein Stück in den Sinn kommt, zu dem man den Namen nicht weiss, sei es, weil man es irgendwo gehört hat, es sich nun als Ohrwurm eingenistet hat oder weil es einem einfach so in den Sinn kommt. Den Namen des Stücks herauszufinden kann sich als höchst schwierig herausstellen. Oftmals muss man darauf hoffen, dass man dem Stück ein anderes Mal wiederbegegnet und dadurch dann den Namen erfährt, oder es muss darauf gehofft werden, dass es jemand kennt. Ein solches Problem kann gut mithilfe eines Computers gelöst werden. Man lässt die gesuchte Melodie mit einer Datenbank abgleichen und findet so das passende Lied. Beispiele für solche Programme wären SoundHound oder die “What’s this Song” Funktion von Google, welche mit künstlicher Intelligenz (KI) funktionieren (Shazam gehört nicht zu dieser Kategorie, da es Audiospuren vergleicht und nicht wie hier Melodien). Die oben beschriebene Situation hat mich dazu gebracht, dieses Thema als meine Maturaarbeit zu wählen, zudem ist eines meiner Hobbies das Klavierspielen, was mich zusätzlich in diese Richtung gestossen hat. Die Idee ist ein Programm zu entwickeln, welches das gesuchte oder ein Teil des gesuchten Stückes mit einer bereits vorhandenen Datenbank abgleicht und als Rückgabewert dasjenige nennt, welches die höchste Ähnlichkeit zum eingegeben Stück hat. Um eine Analyse durchführen zu können, muss das Stück in einem ersten Schritt in eine digitale Form umgewandelt werden, dafür nutzen wir das MIDI-Dateiformat. Damit die Stücke einander gegenübergestellt werden können, brauchen wir einen Algorithmus, der Sequenzen von Zeichen oder Zahlen miteinander vergleichen kann. Algorithmen für Worterkennung in Korrekturprogrammen funktionieren auf eine sehr ähnliche Art. Ein solcher Algorithmus ist der Levenshtein-Algorithmus, der zwar ursprünglich für die Worterkennung benutzt wurde, sich aber auch für dieses Musikprojekt eignet. Die beiden Aspekte, also das Lesen der Eingabe-Dateien und das Vergleichen mit der Datenbank wird mittels eines Python-Programmes durchgeführt. In dieser Maturaarbeit wird ein Einblick in das Entwickeln und Schreiben eines solchen Programmes gegeben, zudem werden die wichtigen Grundlagen erklärt.

# 2. Grundlagen

## 2.1. Musiktheorie

In der westlichen Musik unterscheiden wir zwischen den Ganzton- und Halbton-Intervallen. Das Halbton-Intervall ist das kleinstmögliche Intervall in der westlichen Musik und es bezeichnet den Abstand zwischen einer Klaviertaste und der nächsten, wobei nicht nur die weissen, sondern auch die schwarzen Tasten beachtet werden. Somit besteht eine Oktave (das Intervall zum Beispiel zwischen C1 und C2 resp. in Abb. 1 zwischen zwei «c») aus 13 Halbtonschritten. Ein Ganzton setzt sich ganz einfach aus zwei Halbtönen zusammen (ein Halbton wird manchmal auch kleine Sekunde und ein Ganzton grosse Sekunde bezeichnet). Da die kleinste Einheit in unserem Musiksystem der Halbton ist, wird dieser auch als Einheit in dem Programm verwendet.

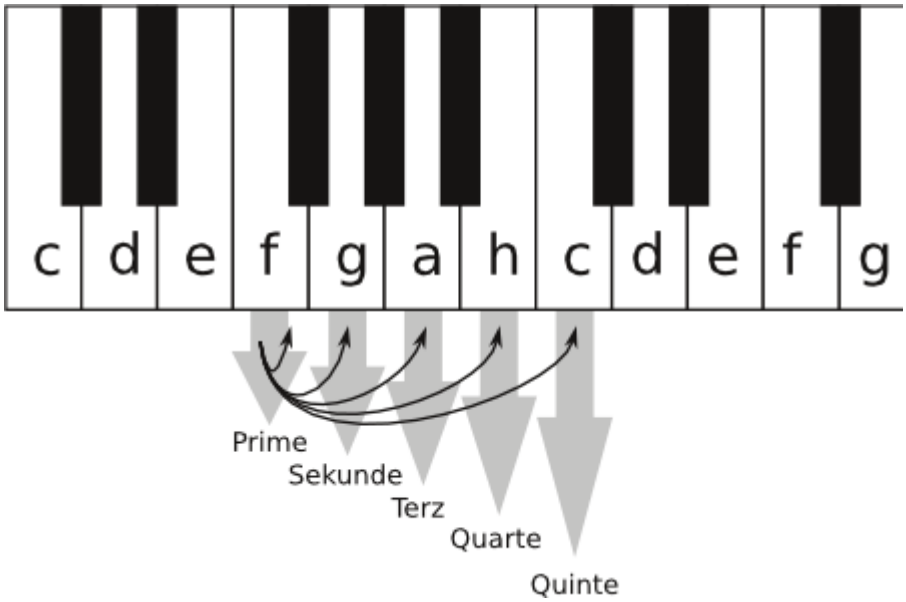


*Abbildung 1*

Nebstdem ist auch wichtig zu wissen, dass der Grossteil der Menschen nicht zwischen den absoluten Tönen unterscheiden kann, also den Unterschied zum Beispiel zwischen einem C3 und einem D3 nicht ohne Referenzton hören kann. Die Fähigkeit, absolute Töne erkennen zu können, nennt man “Absolutes Gehör” oder auf Englisch: “Perfect Pitch”, diese Fähigkeit kann typischerweise nicht erlernt werden, man wird damit geboren oder eben nicht. Was man sich hingegen aneignen kann, ist die Fähigkeit, Intervalle unterscheiden und ausmachen zu können. Hiermit ist gemeint, dass, wenn einem zwei Töne vorgespielt werden, man das Intervall ziemlich einfach ausmachen kann. Da wir also grundsätzlich nur Intervalle hören können, ist es naheliegend, dass wir uns auch nur an Intervalle erinnern können. Folglich können wir die genaue Tonlage (absolute Höhe) von Melodien nicht wiedergeben. Aus diesem Grund wird im Programm auch nur in Intervallen gearbeitet,

beziehungsweise es werden die einzelnen Intervalle der Melodien verglichen, nicht die absoluten Töne.

## 2.2. MIDI-Fileformat

Musik wird grundsätzlich schriftlich notiert, also auf Notenblättern. In diesen stehen Informationen über wie und was gespielt werden soll. Auf Notenblättern befinden sich die einzelnen Noten typischerweise auf fünf Linien, wobei jede Position auf oder zwischen den Linien für unterschiedliche weisse Tasten stehen. Die Positionen weiter oben korrespondieren mit einem höheren Ton, beziehungsweise mit einer weissen Taste weiter rechts auf dem Klavier. Um die schwarzen Tasten zu notieren, benutzen wir Versetzungs- beziehungsweise Vorzeichen wie zum Beispiel:

Das Kreuz: ♯

*Abbildung 2*

Das Be: ♭

*Abbildung 3*

Auf den Noten findet man zudem Informationen über die Rhythmik (dargestellt mit der Form der einzelnen Noten, siehe Abbildung 4) und der Dynamik.

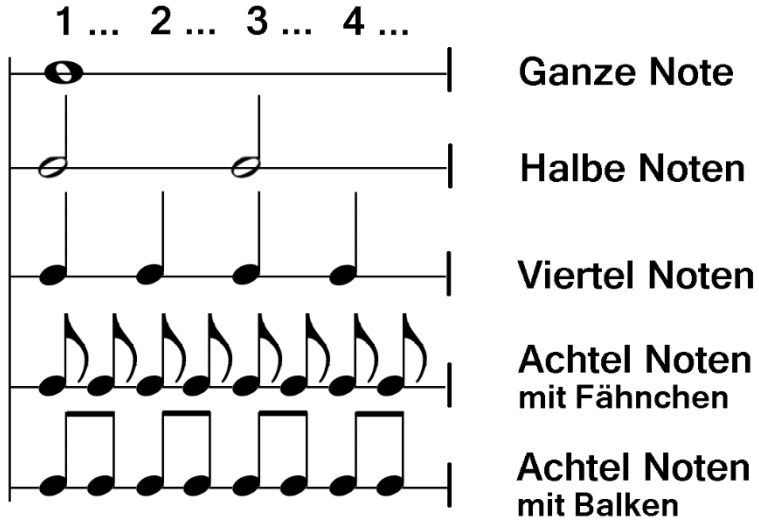


*Abbildung 4*

Wenn wir nun Musik digital notieren wollen, benutzen wir das MIDI Protokoll. Das MIDI Protokoll ist der technische Standard, um musikalische Steuerinformationen zu kommunizieren und zu speichern, das heisst anstatt Gespieltes als Audio zu sichern, wird es

als Anleitung, wie man das Gespielte auf einem Keyboard spielen kann, gespeichert. Es speichert also im Prinzip die Knopfdrücke des Keyboards.

Es wurde erstmals von Roland Gründer Ikutaro Kakashi 1981 vorgeschlagen, einen Standard für die Synchronisierung elektronischer Musik zu definieren.

*"Roland founder Ikutaro Kakehashi felt the lack of standardization was limiting the growth of the electronic music industry."* (Wikipedia/MIDI, 2021)

Dave Smith, Präsident von Sequential Circuits, entwickelte also genau so eine Schnittstelle basierend auf Roland's DCB (Digital Control Bus). Ursprünglich hiess die Schnittstelle USI für Universal Synthesizer Interface, man einigte sich schlussendlich aber auf MIDI für Musical Instrument Digital Interface, welches noch bis heute die Standardschnittstelle für elektronische Musik ist. Im Januar 2020 wurde das MIDI 2.0 präsentiert, ein Update des Standard MIDI, welches zukünftig dessen Rolle übernehmen soll.

MIDI-Dateien bestehen aus drei Schichten: Datei, Track und Kanal. In der ersten Schicht finden wir die grundsätzlichen Informationen über das Stück wie Name, Künstler, Copyright-Info und Format. In der zweiten Schicht findet man je nach Format der Datei einen oder mehrere Tracks, diese enthalten zum Beispiel eine Beschreibung des Inhalts, Tracklänge, Instrumentname und ID des Tracks. In der dritten Schicht finden wir die Kanäle. Das MIDI-Protokoll läuft auf 16 Kanälen, die unabhängig voneinander angesteuert werden können. Die Kanäle werden zum Beispiel dafür gebraucht, mehrere Klänge parallel anzusteuern.

Eine MIDI-Datei ist immer in einem der folgenden drei Formate aufgebaut: Format 0, für Dateien, die einen multi-Kanal Track enthalten, Format 1 für mehrere parallel verlaufende Tracks und Format 2 für mehrere Tracks, die unabhängig voneinander sind (wie zum Beispiel ein Album mit mehreren Stücken).

Auf der Ebene der Tracks finden wir dann auch die eigentliche Musik. Diese wird in Events abgespeichert wie 'Note ON'- und 'Note OFF' Events, die markieren, wann eine Taste gedrückt wird und wann sie losgelassen wird, oder Program-Change-Events, die den Klang eines Kanals ändern.

MIDI-Files sind in Chunks aufgeteilt, jeder Chunk fängt mit 4 ASCII Zeichen an, die bezeichnen, um welche Art von Chunk es sich handelt, gefolgt von 32 bits (most significant byte first), die die Länge des folgenden Chunks übermitteln.

Es gibt zwei Arten von Chunks, Header-Chunks und Track-Chunks (gekennzeichnet dadurch, dass sie mit den 4 ASCII Zeichen "MThd" für Header-Chunks respektive "MTrk" für die Track-Chunks beginnen), wobei die Datei immer mit einem Header-Chunk beginnt, welcher Informationen wie das schon vorher erwähnte Format und die Anzahl Track-Chunks enthält.

In Track-Chunks wird dann die eigentliche Komposition gespeichert. Sie enthalten, wie schon vorher erwähnt, Events, welche wiederum aus dem Zeitpunkt in Ticks seit dem ersten

Event und dem eigentlichen MIDI-Event, auch MIDI-Message genannt, bestehen. Es gibt viele verschiedene MIDI-Messages, aber für das Programm brauchen wir nur die Note ON, Note OFF und Program-Change-Message. Die Note ON/OFF Events enthalten je zwei Werte zwischen 0 und 127, der eine steht dafür, welche Note gespielt oder losgelassen wurde und der andere für die Anschlagsdynamik (Velocity) und eine Kanalnummer (0-15). Das Program-Change-Event enthält einen Wert, der für den Klang steht, der zugeordnet wird, sowie einen Wert für die Kanalnummer.

## 2.3. Levenshtein-Distanz

Die Levenshtein-Distanz wurde 1965 von Vladimir Levenshtein, nach welchem sie auch benannt wurde, erfunden. Sie wird gebraucht, um zwei Folgen, wie zum Beispiel Zahlen- oder Buchstabenfolgen (Strings), zu vergleichen. Sie steht dafür, wie viele Edits man benötigt, um von der einen Folge zur anderen zu gelangen (Editierdistanz). Als Edit wird dabei jedes Einfügen, Löschen und Ersetzen von Zeichen definiert.

Es gibt verschiedene Möglichkeiten zur Berechnung der Distanz. Diejenige, die in diesem Programm genutzt wurde, ist die iterative Variante mit ganzer Matrix, die hier näher betrachtet wird.

Bevor wir mit der Rechnung beginnen, müssen die verschiedenen Parameter definiert werden. Die Parameter sind: Insertions-, Löschungs- und Substitutionspreis. Diese sind dazu da, zu entscheiden, wie viel wert die verschiedenen Aktionen in diesem spezifischem Applikationsfall haben. Wenn wir wollen, dass das Löschen eines Zeichens mehr Distanz zwischen den beiden Folgen schafft, also die beiden Folgen unterschiedlicher macht als das Hinzufügen, dann setzen wir einfach den Löschungspreis höher als den Insertionspreis; das gleiche gilt auch für die Substitution. Im Normalfall werden aber alle Preise einfach gleich eins gesetzt.

Wir definieren eine zweidimensionale Matrix M, deren Höhe und Breite den Längen der beiden Ausgangsfolgen plus eins entsprechen und deren Felder alle 0 enthalten. Nun werden die Felder der Matrix ausgefüllt:

Es wird in der Ecke oben links gestartet und in der obersten Zeile nach rechts gearbeitet. Jedes Feld soll den Wert des vorherigen Feldes plus den Löschungspreis/Insertionspreis erhalten, wobei das erste Feld null bleibt. Das gleiche wird nun in der ersten Spalte von oben nach unten wiederholt, das erste Feld bleibt also null und die darauffolgenden ergeben dann immer den Wert des vorherigen Feldes plus den Insertionspreis/Löschungspreis. Wichtig ist dabei, dass, ob man nun den Insertions- oder Löschungspreis benutzt, davon abhängt, von welcher Zeichenfolge man auf die andere kommen will. Die Verwechslung spielt in dem Fall dieser Anwendung keine Rolle, da die Preise hier alle gleich eins sind.

An dem Beispiel von Katze und Platz gezeigt:

Matrix M in Gelb

↓Eingabewort 1

| | | K | A | T | Z | E |
|---|---|---|---|---|---|---|
| | 0 | 1 (0+1) | 2 (1+1) | 3 (2+1) | 4 (3+1) | 5 (4+1) |
| P | 1 (0+1) | 0 | 0 | 0 | 0 | 0 |
| L | 2 (1+1) | 0 | 0 | 0 | 0 | 0 |
| A | 3 (2+1) | 0 | 0 | 0 | 0 | 0 |
| T | 4 (3+1) | 0 | 0 | 0 | 0 | 0 |
| Z | 5 (4+1) | 0 | 0 | 0 | 0 | 0 |

← Eingabewort 2

*Abbildung 5*

Nach diesem ersten Schritt wird wie folgt fortgefahren:

Wir starten in der Ecke oben links der Felder, die wir noch nicht ausgefüllt haben (im oberen Abbild in orange markiert) und vergleichen die beiden dazugehörigen Zeichen. Im Falle des Feldes oben links in orange sind das “P” und “K”. Wenn die Buchstaben gleich sind, vergleichen wir diese Werte:

**Den Wert von dem Feld links daneben = + 1**

**Den Wert des Feldes oben dran = + 1**

**Den Wert des Feldes diagonal (oben-links) dazu = 0**

Wir würden nun den kleinsten dieser drei Werte in unser momentanes oranges Feld eintragen, das wäre 0. Wenn die Buchstaben hingegen nicht gleich sind, wie das hier ja der Fall ist, nehmen wir nicht den Wert des Feldes diagonal dazu, sondern den Wert des Feldes diagonal dazu plus den Substitutionspreis (in diesem Fall 1) und tragen wiederum den kleinsten Wert in unser momentanes oranges Feld ein.

Mathematisch ausgedrückt:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j) + 1 \\ \operatorname{lev}_{a,b}(i,j-1) + 1 \\ \operatorname{lev}_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise.} \end{cases}$$

*Abbildung 6*

wobei (i,j) die Koordinaten der Matrix repräsentieren und a,b die beiden Zeichenfolgen.

Wir wiederholen diese Schritte für alle Felder von links oben nach rechts unten. Wenn die Levenshtein-Matrix ausgefüllt ist, können wir einfach den Wert in der Ecke unten-rechts (in blau markiert) nehmen und dieser steht dann für die eigentliche Levenshtein-Distanz.

Die ausgefüllte Matrix sähe dann für dieses Beispiel so aus:

| | | K | A | T | Z | E |
|---|---|---|---|---|---|---|
| | 0 | 1 (0+1) | 2 (1+1) | 3 (2+1) | 4 (3+1) | 5 (4+1) |
| P | 1 (0+1) | 1 | 2 | 3 | 4 | 5 |
| L | 2 (1+1) | 2 | 2 | 3 | 4 | 5 |
| A | 3 (2+1) | 3 | 2 | 3 | 4 | 5 |
| T | 4 (3+1) | 4 | 3 | 2 | 3 | 4 |
| Z | 5 (4+1) | 5 | 4 | 3 | 2 | 3 |

*Abbildung 7*

Wir erhalten also schlussendlich eine Levenshtein-Distanz von 3, was heisst, um von dem einen String auf den anderen zu kommen brauchen wir 3 Edits. Wir können mit der Matrix Methode auch die benötigten Edits zurück tracken. Um dies zu machen, müssen wir zurückverfolgen, von wo die Werte in den Feldern eigentlich kommen.

Um das Verfahren verständlicher zu machen, nutzen wir dazu eine Matrix mit Richtungspfeilen, die den Herkunftsort des Wertes des jeweiligen Feldes indizieren:

| | | K | A | T | Z | E |
|---|---|---|---|---|---|---|
| | 0 | ← 1 (+1) | ← 2 (+1) | ← 3(+1) | ← 4(+1) | ← 5 (+1) |
| P | ↑ 1 (0+1) | ↖ 1(+1) | ← 2(+1) | ← 3(+1) | ← 4(+1) | ← 5(+1) |
| L | ↑ 2 (1+1) | ↑ 2(+1) | ← 2(+1) | ← 3(+1) | ← 4(+1) | ← 5(+1) |
| A | ↑ 3 (2+1) | ↑ 3(+1) | ↖ 2 | ← 3(+1) | ← 4(+1) | ← 5(+1) |
| T | ↑ 4 (3+1) | ↑ 4(+1) | ↑ 3(+1) | ↖ 2 | ← 3(+1) | ← 4(+1) |
| Z | ↑ 5 (4+1) | ↑ 5(+1) | ↑ 4(+1) | ↑ 3(+1) | ↖ 2 | ← 3(+1) |

*Abbildung 8*

Die Richtung zum Herkunftsfeld zeigt uns, ob die Aktion nun für eine Substitution, Insertion oder Löschung steht.

↖ = Substitution

← = Insertion

↑ = Löschung

Um an die konkreten Edits zu kommen, folgen wir den Richtungspfeilen von der Ecke unten-recht aus und notieren alle Aktionen (inklusive den dazugehörigen Buchstaben), die dem Wert eins hinzufügen (in diesem Fall die, die "(+1)" enthalten). Der Weg ist in Abb. x blau und die Edits hellblau markiert.

In diesem Beispiel haben wir:

Feld ("Z"/"E") Insertion, Feld ("L"/"K") Löschung, Feld ("P"/"K") Substitution.

Was diese Aktionen nun bedeuten ist:

Um von dem String "Platz" auf "Katze" zu kommen, muss man mindestens drei Edits vornehmen, nämlich die Insertion des "E"s am Ende von "Platz", die Löschung von "L" in "Platz" und die Substitution von "P" mit "K".

# 3. Aufbau des Programms und dessen Module

Da die Grundlagen zu dem Thema nun gesetzt sind, wird jetzt das eigentliche Programm und dessen Aufbau näher erklärt. Das Programm besteht aus einem Hauptskript (Main.py), drei selbst-geschriebenen Modulen (LD.py, LDLoop.py und MidiEd.py) und einem externen Modul namens Mido.

Das Hauptskript ist dafür verantwortlich, die Dateien der Datenbank zu sammeln und diese dann einzeln den anderen Skripts zu übermitteln. Es sammelt am Schluss auch die Daten und verarbeitet diese zu einer sortierten und verständlichen Ausgabe.

Das Mido Modul ist objektorientiert programmiert. Um ein MIDI-Datei Objekt zu erstellen, benutzen wir die Syntax: MidiFile(filename). Der filename ist natürlich Platzhalter für den relativen Filepath. Wenn wir dann das Objekt erstellt haben, müssen wir die Tracks extrahieren. Die Tracks in der Datei kann man mit MidiFile.tracks adressieren. MidiFile.tracks gibt dann ein Array von track-Objekten aus, die man wiederum in message-Objekte aufteilen kann. Um herauszufinden, um welche Art message es sich handelt, nutzen wir message.type. Bei Note ON und Note OFF Events können wir dann mit message.note die Note erhalten.

Es wurde in dem Programm auf das Nutzen eines externen Modules für die Levenshtein Distanz verzichtet, da ursprünglich eine radikale Änderung des Algorithmus vorgesehen war, die schlussendlich doch nicht nötig war.

# 4. Beschreibung des Programm

## 4.1. Funktionsweise

In diesem Kapitel wird die genaue Funktionsweise des Programmes erklärt. Das Programm benötigt 6 verschiedene Dateien: Eine Input-Datei im MIDI-Format, einen Datenbank-Ordner, der eine beliebige Anzahl Midi-Dateien enthält, das Main.py Skript und die drei bereits erwähnten Module. Das Verfahren startet im Main.py Skript. Zuerst werden die wichtigen Variablen definiert, sowie die Liste der Endresultate (distances), der filepath der Datenbank, der der Input-Datei und diverse Einstellungen für die Parallelisierung des Programs (Multiprocessing), wie die Anzahl Prozesse (also die Anzahl parallel verlaufender Stränge/Threads), die Liste, in der dann die Prozess-Objekte enthalten sind und die Liste, die die Daten der verschiedenen Prozesse schlussendlich sammelt (mpQueue).

Sobald wir das getan haben, iniziieren wir die Prozesse. Sie werden einer nach dem anderen gestartet und fangen an, die Funktion “partialDB()” abzuarbeiten. In dieser Funktion findet das Sammeln der eigentlichen Dateien der Datenbank statt. Die Datenbank wird in n Teile gespalten, wobei n die Anzahl Prozesse darstellt. Die Prozesse sammeln dann ihren Teil der Datenbank und konvertieren diesen von einer Datei im Midi-Format zu Python-Listen, die die Intervalle des Stücks enthalten. Dies wird mit der Funktion MidiEd.getTracks() und findDeltas() vollendet (die Funktion wird später noch behandelt). Jeder Prozess hat also jetzt

eine Liste (tracks), die alle Tracks und deren Kanäle der Midi-Dateien separat in Intervall-Form enthält und eine Liste (inDeltas), die die Intervalle des Input-Stücks enthält. Darauf werden mit ldl.distanceAll() (das später noch behandelt wird) die Distanzen des Input-Tracks zu den Tracks in der Trackliste berechnet und in die mpQueue gesendet. In der Main Funktion wird während diesen ganzen Vorgängen auf die Resultate der Prozesse gewartet. Die erhaltenen Resultate werden dann in die Distances-Liste geschrieben. Die Distances-Liste enthält Elemente, die wiederum den Namen der Midi-Datei, die Tracknummer und die Übereinstimmung mit dem Input-Track in Prozent enthalten. Sobald der Main-Prozess von allen Prozessen ein Resultat hat, beendet er die Prozesse. Wenn man dies nicht tun würde, würden die Prozesse noch auf dem Computer laufen und Arbeitsspeicher und Leistung des Prozessors beanspruchen, ohne dass sie wirklich etwas machen. Diese “kaputten” Prozesse nennt man Zombie-Prozesse. Am Schluss wird die Distances-Liste noch nach Übereinstimmung der verschiedenen Elemente sortiert und dann ausgegeben.

### Zu den Funktionen MidiEd.getTracks() und MidiEd.findDeltas()

Das MidiEd (Midi Editor) Modul ist der Teil des Programmes, der die Midi-Dateien zu Intervall-Arrays macht. Dies geschieht so:

Das MidiEd Modul besteht aus zwei Funktionen: findDeltas(), und getTracks(). Der erste Schritt vom Midi zum Intervall ist die getTracks() Funktion. Sie extrahiert alle Tracks von der Datei und liest die Events in den Tracks aus. Daraus werden alle Note ON und Program-Change Events aussortiert. Dann wird von den Note ON Events die Note rausgeschrieben und je nach Kanalnummer in die zur Kanalnummer zugehörigen Liste geschrieben. Aus den Program-Change Events wird der Klang gespeichert und ebenfalls in die zur Kanalnummer zugehörigen Liste geschrieben. Am Schluss haben wir je eine Liste aus absoluten Notenwerten pro Track/Kanal, die wir dann zurückgeben.

In einem zweiten Schritt wird aus den absoluten Notenwerten die Liste der Intervalle erstellt. Dies geschieht ganz einfach, indem wir mit der Funktion findDeltas() von dem jeweiligen Element der Eingabeliste das vorherige Element subtrahieren und das Resultat in eine neue Liste schreiben. In diese neue Liste geben wir nicht nur das Intervall Array, sondern auch den filepath der MIDI-Datei, Track-Nummer, Kanalnummer und Klangnummer (die wir im vorherigen Schritt mit den Program-Change Events erhalten haben).

### Zu der Funktion ldl.distanceAll() bzw. LDLoop.distanceAll()

Diese Funktion befindet sich im LDLoop (Levenshtein-Distance Loop) Modul, welches dafür verantwortlich ist, die Levenshtein-Distanz auf verschiedene Windows zu iterieren. Das heisst, die Input-Liste wird nicht einfach mit der ganzen Liste eines Vergleichtracks verglichen, sondern nur mit einem Teil davon, der gleich gross ist wie die Input-Liste selbst. Dieser Teil wird dann Element für Element durch den Vergleichstrack geshiftet, so dass jeder Teil des Tracks mit der Input-Liste verglichen wurde. Das Modul besteht aus einer einzigen Funktion distanceAll(), die fünf Parameter erfordert: dataIn, searchIn, subCost, delCost, insCost. Der Parameter dataIn ist eine Liste, die die Arrays der Vergleichstracks enthält, searchIn ist das Array der Input-Liste und sub-, del- und insCost sind die Substitutions-,

Löschungs- und Insertionspreise für den Levenshtein Algorithmus. In der Funktion wird über jeden Track in der dataIn-Liste iteriert und auf diesen wiederum die verschiedenen Windows angewandt. Dazu wird immer eine temporäre Liste mit dem Teil des Tracks innerhalb des Windows erstellt und darauf der Levenshtein Algorithmus angewandt. Das Resultat ist dann ein Array, welches filepath/Name des Stücks, Track-Nummer und Übereinstimmung mit dem Eingabetrack in Prozent beinhaltet. Die Prozente werden mit der folgenden Formel berechnet:

$$(1 - \frac{Distanz}{Fenstergr\ddot{o}sse}) * 100$$

Am Schluss wird aus allen Resultaten der Tracks je das Beste, also dasjenige mit dem höchsten Übereinstimmungswert, in eine neue Liste (distancesBest) geschrieben und zurückgegeben.

## 4.2. Probleme

Beim Schreiben und Entwickeln des Programmes sind einige Probleme aufgetreten. Einige kleine Probleme, wie Bugs in dem Programm oder nicht funktionierende Module, die man einfach oder relativ einfach reparieren konnte, aber auch ein paar schwerwiegendere, für die das Programm erweitert werden musste. So ein Problem war zum Beispiel, dass der Levenshtein Algorithmus nicht dafür gemacht ist, kleine Schnippchen eines Strings in einem grossen String zu finden, sondern um zwei ähnlich grosse zu vergleichen. Deshalb musste schlussendlich die Window-Methode angewandt werden. Diese ist viel leistungsintensiver und dauert deshalb auch viel länger. Dieses Problem wurde mit der Parallelisierung, dem Multiprocessing, gelöst, welches aber nicht ganz einfach zu implementieren war. Dies aus mehreren Gründen. Als Erstes musste man darauf achten, dass die Prozesse einzig und allein ihre Aufgabe und keine andere ausführen, als Zweites, dass die Resultate korrekt gesammelt werden, so dass nicht das Resultat eines Prozesses vom Resultat eines anderen überschrieben wird und als Drittes, dass keine Zombie-Prozesse generiert werden. Nachdem aber diese Hindernisse überkommen waren, gab es eigentlich keine Probleme mehr. Alles in allem lief das Entwickeln ziemlich reibungslos und wie anfangs geplant.

# 5. Mögliche Erweiterungen

## 5.1. Rhythmus und Verzierungen

Das Produkt funktioniert zwar soweit mit zufriedenstellender Genaugkeit, so dass man meistens das richtige Stück ausgegeben bekommt, es könnte jedoch genauer sein. In diesem Abschnitt werden ein paar Methoden für das Verbessern der Genauigkeit erklärt.
Die erste Methode wäre das Vergleichen der Rhythmen der Stücke. In der Musik ist Rhythmus essentiell und vermittelt auch typischerweise einen grossen Teil der Nachricht, der Ambiance und der Emotion des Stücks. Der Rhythmus kann in manchen Fällen sogar das Hauptmerkmal des Stücks sein. In dieser Version des Programmes ist der Rhythmus noch nicht mit einbezogen, was, wie ich vermute, der Genauigkeit des Programmes schaden kann. Ein simpler Weg, um den Rhythmus bei der Berechnung der Übereinstimmung ebenfalls zu integrieren, wäre es, ein leicht abgeändertes 'Melodie-Verfahren' zu verwenden. Dieses würde auch mittels Levenshtein-Distanz rechnen, würde jedoch anstatt die Intervalle zwischen den Noten die Multiplikanden von einer Zeitdifferenz zwischen zwei Noten auf die nächste nutzten. Diese würden dann wiederum durch den Levenshtein-Algorithmus geschickt werden und deren Levenshtein-Distanz mit in die finalen Übereinstimmungs-Prozentuale eingerechnet werden.

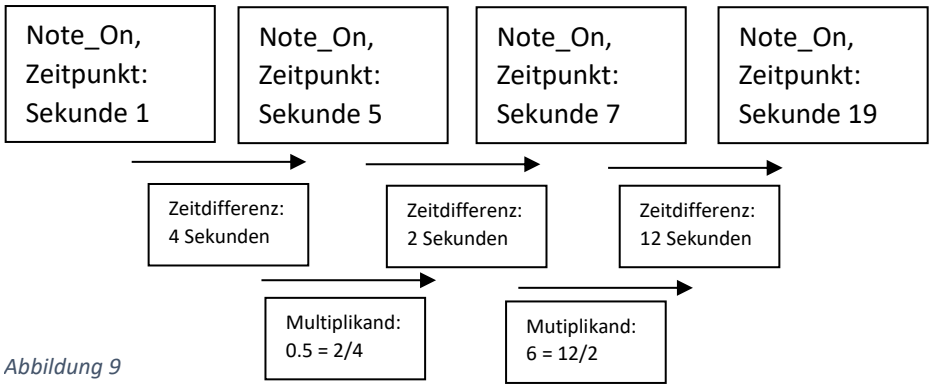


*Abbildung 9*

Wichtig ist beim Rhythmus noch zu bedenken, dass der kleinste Fehler beim Einspielen des Eingabestücks dazu führen kann, dass der Multiplikand als ganz falsch angesehen wird, da 1.00 nicht gleich 1.01 ist. Deshalb wendet man einen in der Musikproduktion “Quantisierung” genannten Prozess an. Dieser Prozess stellt sicher, dass alle Note On Events in einem Track sich an musikalisch sinnvollen Orten befinden. Bei der Quantisierung wird quasi ein Raster erstellt, dessen Abstände einem $2^n$-stel der musikalischen Grundeinheit für Rhythmus, dem Schlag (Beat), entspricht. Die Noten die sich dann nicht auf besagtem Raster befinden, werden auf das Raster verschoben.

Zusätzlich wird die Genauigkeit dadurch limitiert, dass wenn sich in der Datenbank eine Note mehr befindet als im Eingabetrack  oder umgekehrt gerade zwei Intervalle falsch gezählt werden und nicht nur einer oder sogar gar keiner.

Dieses Problem könnte man mit einer Erweiterung des Levenshtein-Algorithmus lösen. Man müsste einen zusätzlichen Prozess haben, der nicht nur die einzelnen Intervalle vergleicht, sondern auch zu einem gewissen Ausmass die Summen aufeinanderfolgender Intervalle.

## 5.2. Gesangserkennung

Zum Zeitpunkt des Schreibens dieses Textes ist eine Gesangerkennungs-Erweiterung, die es dem Nutzer ermöglicht, anstelle einer MIDI-Datei eine Audiodatei einzugeben, in der Entwicklung. Diese Gesangserkennung soll auf Basis des Python Moduls “crepe” funktionieren, welches die Frequenz des Gesangs erkennt, diese in Noten umwandelt und in eine MIDI-Datei schreibt, welche man wiederum in das jetzige Programm eingeben könnte.

# 6. Resultate

Für das Testen des Programs wurde eine Datenbank von 30 bekannten Songs benutzt. Die Input-Dateien wurden mit einem Keyboard eingespielt und es wurden einerseits Stücke eingespielt, von denen ich vor dem Einspielen die Original-Audioversion des Stücks gehört habe und andererseits solche, die ich schon lange nicht mehr gehört hatte. Die Ausgaben wurden dann in eine Textdatei geschrieben und gespeichert.

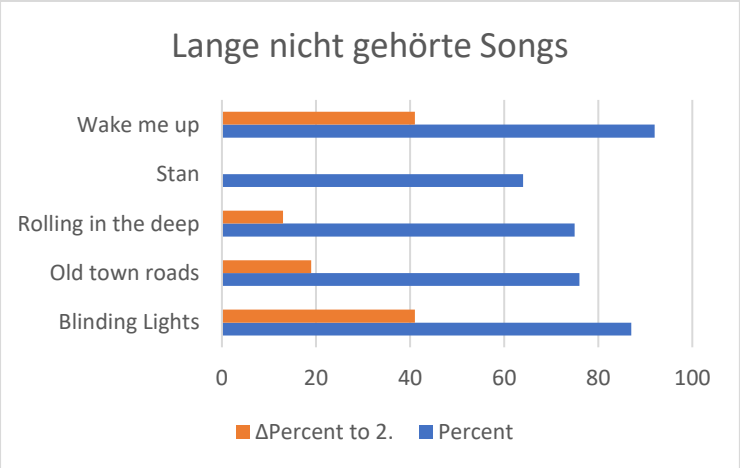


*Abbildung 10*

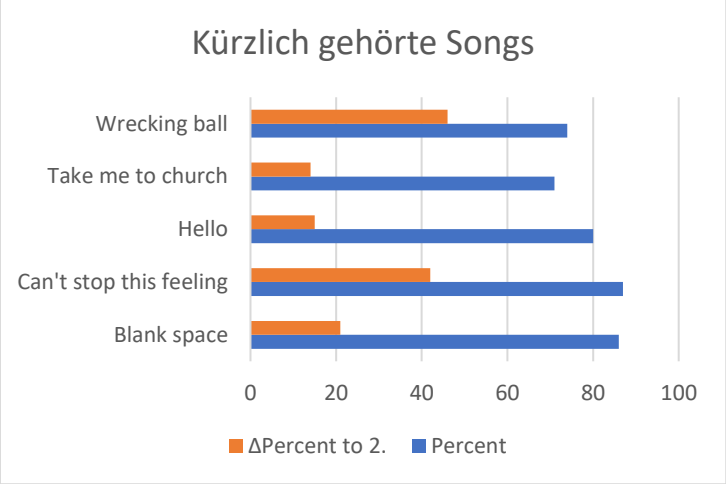


*Abbildung 11*

Die orangen Balken repräsentieren die Differenz zwischen dem erstbesten Resultat und dem zweitbesten und die blauen Balken den absoluten Wert des besten Resultates (welches im diesen Tests immer das eigentlich eingespielte Lied war, was heisst, dass das Programm funktioniert). Zu beachten ist, dass bei “Stan” die Differenz = 0 ist, also hat in diesem Falle das zweitbeste Resultat die gleiche Punktzahl wie das beste.

## 7. Fazit

Die Arbeit ist aus ausführlicher Einarbeitung in diverse Themen und aus viel Testen, Debuggen und Verändern enstanden. Sie hat mir gezeigt, dass die Anwendung eines simplen Algorithmus bzw. eines simplen Konzeptes auch mit komplexen Mustern funktionieren kann und diese recht gut vergleichen kann.
Ich bin sehr zufrieden mit den Resultaten. Das Programm funktioniert etwa so, wie ich mir das vorgestellt hatte. Obwohl es die Resultate mehr oder weniger via Brute-Force findet und deswegen ziemlich lange braucht, findet die Suche je nach Leistung des Geräts doch in einer angemessenen Zeitspanne statt. Das Programm kann und wird noch verbessert werden, wie in Kapitel «5. Mögliche Erweiterungen» auch schon erwähnt. Anfangs hatte ich zwar Schwierigkeiten bei der Implementation der MIDI-Dateien, da ich ein Modul benutzte, das nicht mit allen MIDI-Dateien funktioniert. Ich habe das Modul gewechselt und das neue Modul scheint bis jetzt mit allen MIDI-Dateien klarzukommen. Ich kam am Ende der Arbeit ein wenig in Zeitstress, da ich auf eine Planung meiner Arbeitseinteilung mehr oder weniger verzichtet hatte. Dies war eindeutig ein Fehler und ich würde dies in Zukunft nicht nochmal so machen.

## Danksagung

Ich bedanke mich bei allen, die mich bei der Arbeit unterstützt und mich motiviert haben. Spezieller Dank geht an meine Eltern, besonders meinen Vater, der mir mit seinem Wissen über digitale Musik ausgeholfen hat und mich ermunterte, einige Themen etwas ausführlicher zu erklären. Ich bedanke mich auch bei Ian, der ebenfalls noch einen Blick auf meine Arbeit geworfen hat und bei meinem Betreuer Herrn Rothe, der beim Entwickeln der Arbeit stets hilfsbereit war.

# Literaturverzeichnis

Die Midi Testdateien in der Datenbank kommen von:

https://freemidi.org/

*Mido Documentation*. (2021, Oktober). From mido.readthedocs: https://mido.readthedocs.io/en/latest/

*Notenwerte.* (2021, Oktober). From noten-lesen-lernen.de: https://noten-lesen-lernen.de/wissen-2-takt-notenwerte-pausenwerte/

*Standard MIDI-Fileformat*. (2021, Oktober 14). From Music Mcgill: http://www.music.mcgill.ca/~ich/classes/mumt306/StandardMIDIfileformat.html

*Wikipedia/Levenshtein-Distance*. (2021, Oktober 14). From Wikipedia: https://en.wikipedia.org/wiki/Levenshtein_distance

*Wikipedia/MIDI*. (2021, Oktober 14). From Wikipedia: https://en.wikipedia.org/wiki/MIDI

# Abbildungsverzeichnis

*Abbildung 1: Musikanalyse.* (2021, Oktober). From musikanalyse.net: https://musikanalyse.net/tutorials/intervalle-bestimmen/

*Abbildung 2: Bogenbalance Kreuz.* (2021, Oktober). From bogenbalance.de: https://bogenbalance.de/wp-content/uploads/2018/01/Bildschirmfoto-2016-01-02-um-09.12.36.png

*Abbildung 3: Bogenbalance Be.* (2021, Oktober). From bogenbalance.de: https://bogenbalance.de/wp-content/uploads/2018/01/Bildschirmfoto-2016-01-02-um-09.12.19.png

*Abbildung 4*: *Notenwerte.* (2021, Oktober). From noten-lesen-lernen.de: https://noten-lesen-lernen.de/wissen-2-takt-notenwerte-pausenwerte/

*Abbildung 5: Tabelle erstellt von Mattia Brand Oktober 2021*

*Abbildung 6: Medium Levenshtein-Formel.* (2021, Oktober). From medium.com: https://miro.medium.com/max/1400/1*o9k-pcrM-4NUrMNAqQbH9A.png

*Abbildung 7: Tabelle erstellt von Mattia Brand Oktober 2021*

*Abbildung 8: Tabelle erstellt von Mattia Brand Oktober 2021*

*Abbildung 9: Grafik erstellt von Mattia Brand Oktober 2021*

*Abbildung 10: Grafik erstellt von Mattia Brand Oktober 2021*

*Abbildung 11: Grafik erstellt von Mattia Brand Oktober 2021*

# Elektronischer Anhang

Die Dateien befinden sich auf GitHub:

https://github.com/Matdio/Musikerkennungsalgorithmus

Das Programm wurde ausschliesslich in Windows 10 in der virtuellen Entwicklungsumgebung Thonny getestet.

Thonny: Version 3.2.7

https://thonny.org/

Python Version: 3.7.7

https://www.python.org/

Midi Editor: MidiEditor 3.3.0

https://www.midieditor.org/

## Selbstständigkeitserklärung

Ich bestätige hiermit, dass ich, Mattia Brand, die Maturaarbeit “Einsatz der Editierdistanz für einen Musikerkennungsalgorithmus” selber verfasst habe und dass alle genutzten externen Bilder und Hilfsmittel deklariert worden sind.

Ort und Datum

Unterschrift