KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Melodische transformatie en evaluatie van muziek

Elias Moons

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

Promotor:
Prof. dr. D. De Schreye

Assessoren:
Prof. dr. ir. T. Schrijvers
Prof. dr. M.-F. Moens

Begeleider:
Ir. V. Nys

Academiejaar 2015 – 2016

Voorwoord

TODO: iedereen bedanken...

Elias Moons

Inhoudsopgave

Samenvatting

Deze masterproef beschrijft methodes om melodielijnen van een muziekstuk te transformeren tot nieuwe melodielijnen. Er gaat ook aandacht uit naar een referentiekader waarin deze transformaties geëvalueerd kunnen worden. Tot slot wordt er gekeken naar wanneer bepaalde transformaties nuttig kunnen zijn om de consonantie van een muziekstuk te verhogen en hoe verschillende transformaties efficiënt gecombineerd kunnen worden. Om dit te verwezenlijken ontwikkelden we een algoritme dat gebaseerd is op de principes van *dynamic programming*. Dit algoritme zal, gegeven een aantal mogelijke transformaties en een melodielijn, de best mogelijke getransformeerde melodielijn teruggeven volgens het gedefiniëerde referentiemodel.

Lijst van figuren en tabellen

Lijst van figuren

Lijst van tabellen

Hoofdstuk 1

Inleiding

1.1 Probleemstelling

Een van de meest voorkomende problemen voor muzikanten is de zogenaamde *writer's block*. Dit fenomeen waarbij je als muzikant merkt dat je de hele tijd op hetzelfde melodietje uitkomt en maar niet met iets nieuw kan komen kan zeer frustrerend zijn. Daarom zou het handig zijn als er een tool zou bestaan die je als het ware de inspiratie kan geven die je nodig hebt om deze *writer's block* te doorbreken. Een mogelijke oplossing voor dit probleem wordt in deze thesis onderzocht. Hier gaat het dan om het transformeren van reeds gekende melodieën tot nieuwe melodieën.

1.2 Vergelijking met voorgaand onderzoek

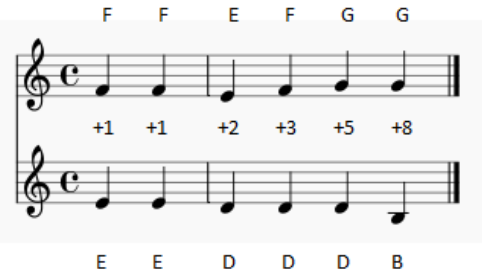
Onderzoek naar het bekomen van nieuwe melodieën waarbij gebruik gemaakt wordt van een computer is niet nieuw [10] [11]. Al gedurende tientallen jaren is er veel werk geleverd op het vlak van muziekgeneratie, waarbij het de bedoeling is om een muziekstuk te genereren waarbij vertrokken wordt van een lege partituur [2]. Dit wordt gedaan rekening houdend met de regels uit de muziektheorie en vaak probeert men er ook een vorm van repetitiviteit en herkenning in te brengen zoals vaak ook het geval is in hedendaagse muziek.

Een groot probleem dat echter altijd terugkwam was dat het heel moeilijk was om een goede verhouding te vinden tussen twee heel belangrijke eigenschappen van de muziek, namelijk verwachting en verrassing van de luisteraar [3]. Een muziek luisteraar heeft namelijk een bepaald verwachtingspatroon voor het onmiddellijke vervolg van een melodie. Dit verwachtingspatroon strookt vaak met de regels van de muziektheorie. Wanneer deze regels dan weer te nauwgezet gevolgd worden, wordt het muziekstuk als saai ervaren, er is dus nood aan een zekere verrassing in de melodie. Te veel onverwachte wendingen worden dan weer als frustrerend gezien, kortom, het is van cruciaal belang om een goed evenwicht te vinden tussen deze twee waarden. Dit blijkt zeer moeilijk te verwezenlijken vanuit het standpunt van muziekgeneratie.

1.3 Doelstelling

In deze thesis wordt een onderzoek gevoerd naar de generatie van nieuwe melodieën vertrekkende van originele melodieën. Deze originele melodieën worden bij aanname verondersteld te voldoen aan de eisen van verrassing en verwachting. In deze thesis worden specifiek melodische transformaties onderzocht. Deze transformaties zullen gebaseerd zijn op wiskundige reeksen. Hierbij gaat men voor elke noot in de oorspronkelijke melodie de toonhoogte aanpassen volgens een bepaald patroon. Als voorbeeld om het concept te verduidelijken kan figuur 1.1 dienen. In deze figuur wordt met behulp van de rij van Fibonacci een originele melodie getransformeerd tot een nieuwe.

Er wordt getracht zo een transformatie te zoeken zodat de nieuw bekomen melodie nog steeds consonant is (nog steeds goed klinkt). Er zal ook onderzocht worden onder welke omstandigheden een transformatie al dan niet een consonante melodie oplevert. Om te bepalen of een nieuwe melodie goed klinkt is er ook nood aan een objectieve beoordeling van melodieën. Er wordt dus ook aandacht besteed aan het opstellen van een model dat hiervoor kan dienen. Een model dat zo goed mogelijk kan beoordelen of een gegeven melodie al dan niet, en in welke mate, consonant is. Het voordeel van het werken met transformaties in plaats van met generatie is dat men op deze manier nog een deel van de eigenheid van het originele werk kan behouden en het bekomen werk hierdoor ook minder artificieel zal overkomen bij de luisteraar.



Figuur 1.1: Melodische transformatie m.b.v. rij van Fibonacci: Noten op de bovenste notenbalk liggen respectievelijk 1,1,2,3,5,8 halve tonen hoger dan op de onderste notenbalk.

1.4 Assumpties en beperkingen

Op een onderdeel van hoofdstuk 3 na, behandelt deze masterproef enkel muziek met precies één melodielijn. Meerstemmige muziek waarbij meerdere noten tegelijkertijd gespeeld kunnen worden, wordt in deze thesis dus niet behandeld. Verder worden alle testen uitgevoerd op een corpus (Essencorpus [9]) bestaande uit muziekstukken van het folk genre.

Tot slot zal ook telkens wanneer een transformatie uitgevoerd is, er vanuit gegaan worden dat het originele muziekstuk telkens voldoet aan de algemene voorwaarden waaraan een muziekstuk moet voldoen. In dat geval kan de score van dit muziekstuk in het voorgestelde model telkens ook als referentie dienen voor de getransformeerde versie.

1.5 Overzicht van de tekst

In het eerstvolgende hoofdstuk, hoofdstuk 2 wordt een korte inleiding gegeven tot de muziektheorie. Hierin worden enkel deze elementen behandeld die relevant zijn voor de rest van deze thesis. Vervolgens zal er een besproken worden hoe een bepaald muziekstuk objectief kan beoordeeld worden, dit zal gebeuren in hoofdstuk 3. In hoofdstuk 4 worden verschillende melodische transformaties toegelicht en met elkaar vergeleken. Vervolgens zal hoofdstuk 5 twee algoritmes beschrijven. Deze algoritmes kunnen gebruikt worden om gegeven een aantal toegestane transformaties en een oorspronkelijke melodielijn, de meest waarschijnlijke getransformeerde melodielijn terug te geven. Hierna zullen een aantal experimenten, alsook hun resultaten besproken worden in hoofdstuk 6. Tot slot wordt er in hoofdstuk 7 nog teruggekeken op het geleverde onderzoek in een samenvattend besluit. Er wordt ook nog beschreven welk verder onderzoek zeker nog interessant zou kunnen zijn binnen dit onderwerp.

Hoofdstuk 2

Muzikale achtergrond

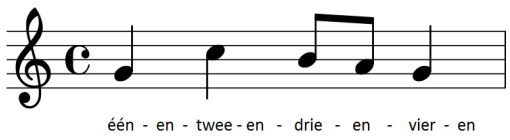
Aangezien deze thesis zal handelen over melodische transformaties wordt hieronder in het kort info gegeven over elementaire begrippen rond ritme en melodie van een muziekstuk en hoe deze voorgesteld kunnen worden. Voor lezers met een voorkennis in de muziekwereld zal dit onderdeel waarschijnlijk redundant zijn en kan er bijgevolg ook meteen overgegaan worden naar het volgende hoofdstuk.

2.1 Ritme

In deze masterproef worden enkel melodische transformaties behandeld, waarbij het ritme ongewijzigd blijft. De tegenhangers hiervan zijn de ritmische transformaties[10], die in deze masterproef niet behandeld worden. Toch is het zeker nuttig om ook een zekere voorkennis te hebben van de betekenis van ritme in een muziekstuk. Melodie en ritme van een muziekstuk gaan namelijk hand in hand. Een basiskennis van ritmische begrippen zal dus zeker ook nuttig zijn voor het begrijpen van bepaalde melodische fenomenen.

2.1.1 Tijdssignaturen

De tijdssignatuur geeft de ritmische structuur weer van het muziekstuk. Deze tijddssignatuur wordt weergegeven aan het begin van de notenbalk en ziet eruit als een breuk zonder streepje. Een voorbeeld hiervan is de vaak gebruikte $\frac{3}{4}$ (of 'drie vierden'). In deze voorstelling staat het onderste getal voor welke nootlengte overeen komt met een tel. Het bovenste getal geeft weer hoeveel tellen in een maat voorkomen [8]. In het voorbeeld van de signatuur $\frac{3}{4}$ komt dit over met 4 tellen van lengte $\frac{1}{4}$. Voor de specifieke signatuur $\frac{4}{4}$ (of 'vier vierden') heeft men nog een andere notatie, namelijk de letter C. Deze letter komt het woord *common time*, aangezien deze signatuur zo typisch en veelvoorkomend is in moderne Westerse muziek. Figuur 2.1 geeft het gebruik van deze notatie weer. De figuur illustreert ook de 4 verschillende tellen van deze maatsoort.



Figuur 2.1: Maat met voortekening van *common time* tijdssignatuur.

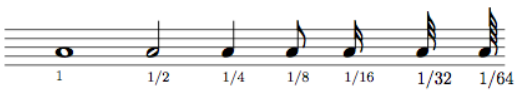
2.1.2 Tempo

Een volgend belangrijk onderdeel dat het ritme van een muziekstuk bepaalt is het tempo. Het tempo van een stuk bepaalt namelijk de duur van de verschillende noten in het muziekstuk. Het temp wordt meestal uitgedrukt in aantal tellen per minuut. De lengte van een tel zelf wordt dan weer bepaald door de tijdssignatuur. Zo betekent bijvoorbeeld een tempo van 60 tellen voor een muziekstuk met signatuur $\frac{3}{4}$ dat elke tel, ofwel elke kwartnoot, precies één seconde duurt.

2.1.3 Nootlengtes

De nootlengte is het laatste element dat de absolute duur van een noot zal bepalen. De nootlengte geeft de relatieve lengte van een noot weer ten opzichte van het tempo en de tijdssignatuur. De nootlengte wordt uitgedrukt door een breuk. Deze breuk heeft als relatieve lengte zijn verhouding tot de lengte van een tel. Zo zal een $\frac{1}{4}$-noot in $\frac{3}{4}$ één tel duren, en zal een $\frac{1}{8}$-noot in $\frac{3}{4}$ twee tellen duren.

Een noot van hele lengte wordt aangeduid met een hol bolletje. Een noot van halve lengte wordt aangeduid met een half bolletje met een streep aan de rechterkant. Een $\frac{1}{4}$-noot wordt aangeduid zoals een halve noot maar dan met een vol bolletje. Een $\frac{1}{8}$-noot wordt voorgesteld door een vierde noot met een streepje vanboven. Vanaf een $\frac{1}{16}$-noot wordt er dan een streepje vanboven bijgezet telkens de nootlengte gehalveerd wordt. Deze notatie wordt geïllustreerd in figuur **??**. Door het gebruik van verbindingstekens(waarbij de nieuwe nootlengte de som is van de lengtes van de twee noten die verbonden zijn) kan men dan eender welke nootlengte bekomen die men maar wenst.



Figuur 2.2: Illustratie van een aantal veelvoorkomende nootlengtes.

2.2 Melodie

Naast ritme is het andere fundamentele bestanddeel van een muziekstuk de melodie. Waar het ritme de structuur van een muziekstuk weergeeft, is de melodie het

A	B	C	D	E	F	G
la	si	do	re	mi	fa	sol

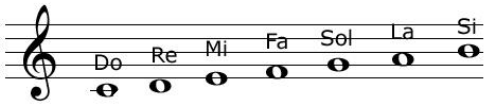
TABEL 2.1: Opsomming van de toonhoogtes in 2 verschillende benamingen.

voornaamste bestanddeel van de muziek dat een gevoel meegeeft aan het stuk. De melodie geeft een toonhoogte of frequentie mee aan elke noot. Aangezien deze thesis handelt over melodische transformaties is het van belang te weten wat het begrip "melodie" precies inhoudt. Het is namelijk dat gedeelte van een muziekstuk waarop een transformatie zal toegepast worden. Het ritme van een muziekstuk wordt in deze thesis onveranderd gelaten.

2.2.1 Toonhoogte

Toonhoogte kan in het algemeen op twee manieren voorgesteld worden. Een eerste manier is fysisch waarbij elke toon met een bepaalde frequentie overeenkomt, een andere manier is meer muziek-theoretisch, waarbij de toonhoogte als discreet beschouwd wordt.

In deze masterproef zal met de laatste voorstellingswijze gewerkt worden. Dit omdat we noten willen transformeren naar nieuwe noten en niet frequenties naar nieuwe frequenties (die dan niet overeen komen met een noot en bijgevolg zeer waarschijnlijk niet goed gaan klinken in het geheel). Deze toonhoogte wordt dan voorgesteld door een naam. Er zijn twee naamgevingen die vaak gebruikt worden. Eerst is er de naamgeving waarbij de letters A t.e.m. G gebruikt worden. De andere naamgeving maakt gebruik van de woorden do – re – mi – fa – sol – la – si, de relatie tussen deze 2 naamgevingen wordt weergegeven in tabel 2.1. Noten kunnen uiteraard ook op een notenbalk weergegeven worden, zie figuur 2.3 voor een visuele weergave. Hoe hoger de noot op de notenbalk, hoe hoger haar frequentie.



FIGUUR 2.3: Noten do t.e.m. si op een notenbalk.

2.2.2 Octaaf en onderverdeling in toonhoogtes

Een eenvoudige definitie van een octaaf is het interval tussen een gegeven toonhoogte en de toonhoogte met het dubbele van de frequentie van de eerste. Deze noten worden als zeer gelijkaardig ervaren en krijgen daarom dezelfde benaming (bijvoorbeeld beide een A). Hierdoor gaat men vaak in muzieknotatie wanneer men een bepaalde noot benoemt, ook aangeven tot welk octaaf de noot behoort (want een bepaalde noot komt met meerdere toonhoogtes overeen). Dit doet men door in subscript de index

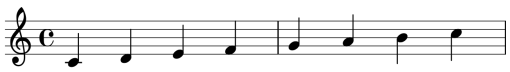
van het octaaf weer te geven, een A (of la) in het vierde octaaf wordt dan weergegeven door A_4.

Een octaaf wordt opgedeeld in 12 toonhoogtes. De afstand tussen elk paar van 2 opeenvolgende toonhoogtes wordt een halve toon genoemd. Hiervan zijn er slechts 7 tonen die ook deel uitmaken van de toonladder. De andere 5 noten worden voorgesteld op de notenbalk relatief ten opzichte van een van de nabijgelegen tonen die slechts een halve toon hier vanaf ligt door het gebruik van een kruis (♯) of een mol(♭). Een kruis dient om aan te duiden dat de noot die bedoeld wordt een halve toon hoger is dan de noot die ervoor staat. Een mol gebruikt men om aan te duiden dat de noot die bedoeld wordt een halve toon lager is dan de noot die net voor het molteken staat.

2.2.3 Toonladders en toonaarden

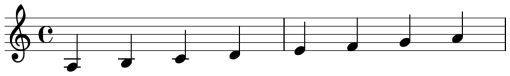
Zoals reeds vermeld werd in het vorige deel, bestaat een octaaf uit 12 tonen waarvan er slechts 7 tot de eigenlijke toonladder behoren. De noten binnen een toonladder worden bepaald door de intervallen tussen de noten. In het algemeen zijn er twee grote onderverdelingen om een toonladder te construeren.

Een eerste resultaat wordt de “grote toonladder” genoemd. deze wordt bepaald door de opeenvolging van stappen 1-1-$\frac{1}{2}$-1-1-1-$\frac{1}{2}$. Het meeste elementaire voorbeeld van een toonladder die hieraan voldoet is de toonladder van “Do-Groot” waarbij volgende noten voldoen aan de opgegeven intervalafstanden: “do - re - mi - fa - sol - la - si - do”, zoals weergegeven in figuur 2.4.



Figuur 2.4: Toonladder van “Do Groot”

Een tweede mogelijkheid is de zogenaamde “kleine toonladder”. In deze toonladder komen de intervalafstanden overeen met 1-$\frac{1}{2}$-1-1-$\frac{1}{2}$-1-1. Een concreet voorbeeld hiervan is de toonladder van “La-Klein” waarbij deze noten voldoen aan de voorwaarden: “la - si - do - re - mi - fa - sol - la”, deze wordt weergegeven in figuur 2.5.



Figuur 2.5: Toonladder van “La Klein”

Deze toonladder is de kleine versie van de toonladder van Do groot, aangezien ze dezelfde noten gebruikt, de toonladder is als het ware een verschuiving van die van Do groot. Het verschil tussen beide toonladders is dus de functie van elke noot in de bijhorende toonaarden. Iets wiskundiger verwoord is er ook een verschil van frequentie waarin bepaalde noten voorkomen in muziekstukken horende bij een kleine of grote toonladder [11]. Van de kleine toonladders zijn ook nog een harmonische en melodische versie, die nog andere afstanden hanteren, maar aangezien deze niet zo vaak gebruikt worden, zou het te ver leiden deze ook te bespreken.

Het feit of een toonladder groot of klein is gaat vaak ook de sfeer bepalen van het muziekstukje. Zo zal een muziekstuk dat geschreven is in een grote toonladder eerder een vrolijk karakter hebben. Dit terwijl een stukje dat geschreven is in een kleine toonladder eerder een droeviger karakter zal hebben.

Niet alleen de toonladder zelf, ook de noten in de toonladder hebben hun functie. Zo is bijvoorbeeld de eerste noot van een toonladder een rustpunt waarop het muziekstuk vaak beëindigd wordt. De vijfde noot wordt de dominant genoemd en is ook sterk vertegenwoordigd in het muziekstuk. Deze noot creëert namelijk spanning. Vaak wordt deze spanning dan ook opgelost door een overgang naar de eerste noot, als rustpunt.

Hoofdstuk 3

Objectieve beoordeling van een muziekstuk

Om melodische transformaties te kunnen beoordelen is er nood aan een framework dat kan voorspellen of een gegeven melodie al dan niet goed klinkt. In dit hoofdstuk worden twee dergelijke modellen besproken die een bepaalde consonantiescore (maat voor het goed klinken van een muziekstuk) gaan toekennen aan een muziekstuk.

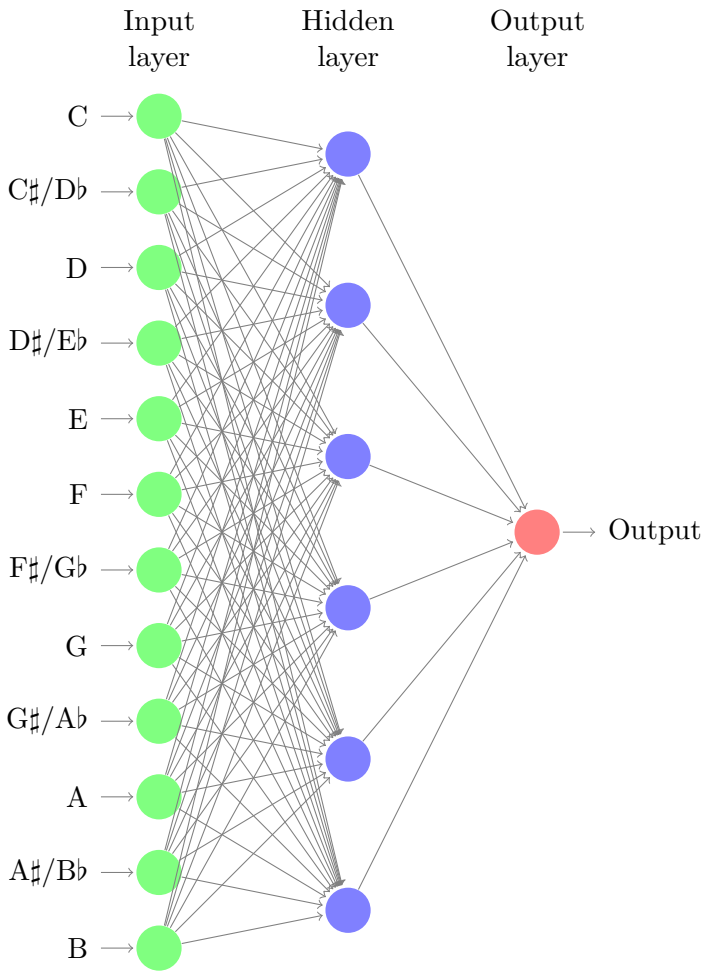
Allereerst wordt een aanpak besproken die gebruik maakt van een artificieel neuraal netwerk [7]. Deze methode is speciaal gemaakt voor het beoordelen van polyfone muziek, waarbij er meerdere noten tegelijk klinken. Polyfone muziek wordt in verdere hoofdstukken van deze masterproef niet meer besproken. Het eerste onderdeel van dit hoofdstuk zal bijgevolg dus weinig invloed hebben op het vervolg van de tekst. De methode is echter wel onderzocht geweest en leverde enkele interessante resultaten op waardoor de methode toch even in het kort vermeld wordt.

Als tweede methode wordt een zogenaamd 'RPK-Model' besproken dat gebaseerd is op een gelijknamig model uit [11]. Dit model werkt op muziek met een enkele melodielijn en zal ook het model zijn dat als verificator gebruikt zal worden bij de experimenten in het vervolg van deze masterproef.

3.1 Neuraal netwerk

In [3] wordt er een gedeelte gewijd aan het meten van de consonantie van akkoorden (samenklank van meerdere noten). Dit gebeurt kort samengevat door het trainen van een (feed forward) neuraal netwerk[14] op zogenaamde tweeklanken (twee noten die tegelijkertijd gespeeld worden). Het neuraal netwerk maakt dan de veralgemening naar hogere orde samenklanken.

Het neuraal netwerk dat gemaakt werd als verificator bestond uit drie lagen. een invoerlaag, een uitvoerlaag en dan nog een verborgen laag. De invoerlaag bestond uit 12 neurons die elk voor een van de 12 verschillende noten staan. Twee noten die een octaaf uit elkaar liggen (en muziektechnisch dezelfde naam krijgen) zullen dus ook op eenzelfde neuron afgebeeld worden. Deze 12 input neurons krijgen een '1' als input wanneer er een noot in het akkoord zit dat afgebeeld wordt op die neuron. In het



Figuur 3.1: Neuraal netwerk met 12 input nodes (1 voor elke noot), 6 hidden nodes en 1 output node.

andere geval krijgt deze een '0' als input. Vervolgens komen we op een hidden layer terecht die volledig geconnecteerd is en uit 6 neurons bestaat. Tot slot komen we nog bij de output layer uit die uit slechts 1 neuron bestaat. Deze neuron gaat een getal tussen 0 en 1 teruggeven, hoe dichter het getal bij 1 ligt hoe zekerder het neuraal netwerk is dat het ingegeven akkoord consonant is. Hoe dichter bij 0 hoe zekerder hij is dat het akkoord dissonant (tegengestelde van consonant) is. De lay-out van dit netwerk is weergegeven in figuur 3.1.

Dit netwerk werd getraind op alle mogelijke tweeklanken gebruik makend van de *backpropagation of error*-methode [6]. Voor elke mogelijke tweeklank werd afhankelijk van de afstand (in halve tonen) tussen de twee noten bepaald of ze goed samen klinken of niet. Dit volgens de regels van de muziektheorie. Deze regels komen overeen met de verhoudingen van de frequenties van de twee invoernoten. Hoe kleiner

de getallen in de vereenvoudigde breuk van de frequenties, hoe beter het akkoord klinkt.

Na het trainen van het neuraal netwerk kan dit gebruikt worden om meerstemmige muziek te gaan beoordelen. Op elk tijdstip waarop er noten gespeeld worden kan men deze noten als input in het neuraal netwerk steken. De uitvoer van het netwerk geeft dan een maat voor de consonantie terug. Als we dit doen voor elk tijdstip in het muziekstuk waarop er noten gespeeld worden en we nemen dan het gemiddelde over al deze tijdstippen krijgen we een maat voor de gemiddelde consonantie van het hele muziekstuk.

Nadat het neuraal netwerk getraind werd, bleek het neuraal netwerk goed te generaliseren naar akkoorden met meer dan 2 noten. Dit wil zeggen dat akkoorden van meer dan twee noten die muziektheoretisch 'goed' zouden moeten klinken ook door het netwerk als dusdanig beoordeeld werden. Het netwerk kon dus dienen als een soort van alternatieve voorstelling van de regels van de muziektheorie wat de samenklank van akkoorden betreft. Het probleem lag er echter in dat de klassieke werken van Bach en Mozart waarop getest werd zelf helemaal niet zo consonant waren als oorspronkelijk gedacht. Dit was in die mate het geval dat vaak tot een derde van de akkoorden in zo een stuk als dissonant (tegengestelde van consonant) bestempeld werden. Deze akkoorden blijken ook daadwerkelijk dissonant te zijn. Het is slechts door de context, de noten die net voor het akkoord gespeeld worden, dat deze akkoorden in die stukken toch niet als dissonant ervaren worden door de luisteraar. Het neuraal netwerk dat hier gebruikt werd houdt echter geen rekening met deze context.

Dit leidt er toe dat dit neuraal netwerk niet gebruikt kan worden als verificator, aangezien zelfs muziekstukken van Bach en Mozart door de verificator niet als consonant herkend worden. Het is daarentegen wel nog maar eens een bevestiging van het genie van componisten als Bach en Mozart, de kunst ligt het niet in het volgen van de regels maar in het weten wanneer en hoe de regels gebroken mogen worden.

De broncode die geschreven werd voor het trainen van dit neuraal netwerk is beschikbaar in appendix A.1. Er zijn een aantal parameters die ingesteld kunnen worden in dit algoritme. Allereerst is er het aantal neuronen in de verborgen laag. Er kan ook een waarde voor de *learning rate* opgegeven worden en dan zijn er nog twee parameters die een bias kunnen definiëren.

3.2 Keuze tussen polyfone en monofone muziek

In deze thesis wordt gewerkt met transformaties op een muziektheoretische manier (en dus niet fysisch met frequenties). Daarom is wordt er gebruik gemaakt van muziekstukken die in het MusicXML[1] formaat beschikbaar zijn. Het nadeel van deze keuze is dat er slechts een beperkt aantal muziekstukken beschikbaar is. Dit in tegenstelling tot bijvoorbeeld het MIDI-formaat[4] waarbij dit niet het geval is.

De polyfone(meerstemmige) muziek die beschikbaar is in het gewenste formaat bestaat eigenlijk bijna uitsluitend uit klassieke muziek. Verder is er ook nog een

redelijk groot corpus beschikbaar van folkmuziek, het zogenaamde Essencorpus[9]. Deze muziek is wel monofoon (eenstemmig).

Aangezien de methode met het neuraal netwerk die beschreven staat in bovenstaand onderdeel niet voldoende werkte voor de meerstemmige stukken moest er een andere weg ingeslagen worden. Ofwel verder werken met meerstemmige muziek maar met een ander framework dat de context in rekening brengt. Ofwel de focus verleggen naar eenstemmige muziek. Gezien het grootste deel van de meerstemmige muziekstukken klassieke stukken zijn, is de keuze gemaakt om over te schakelen op eenstemmige muziek. Dit omdat de complexiteit van het aanpassen van muziekstukken met meerdere lijnen veel hoger is dan die van muziekstukken met slechts een instrument. Ook omdat klassieke muziekstukken veel delicater zijn om te behandelen dan stukken folkmuziek die enkel uit een melodielijn bestaan. Alle transformaties die zullen besproken worden in de rest van deze masterproef zullen dus ook uit een enkele melodielijn bestaan.

3.3 RPK-model

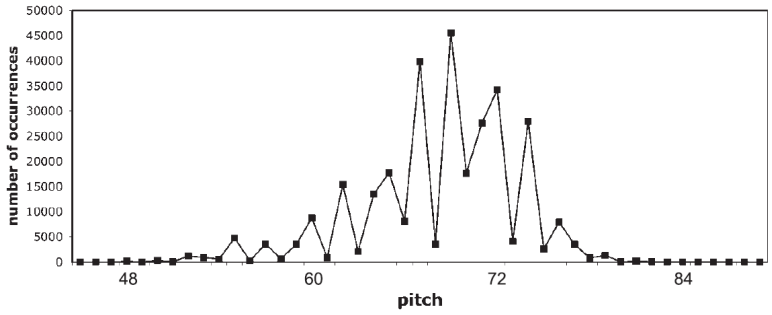
Aangezien er nood was aan een framework voor het beoordelen van muziekstukken van slechts een enkele melodielijn werd er uiteindelijk uitgekomen bij het zogenaamde RPK-model. Dit model wordt beschreven in [11]. Het model dat besproken gaat worden in dit onderdeel en dus ook gebruikt werd in de rest van het onderzoek is gebaseerd op het model uit dit boek. Er werden een aantal vereenvoudigingen gedaan gebaseerd op extra data die beschikbaar is in het MusicXML formaat. Dit RPK-model gaat dus uit van een enkele lijn melodie (er wordt slechts een noot tegelijkertijd gespeeld).

De waarschijnlijkheid van een bepaalde noot in een muziekstuk wordt gekenmerkt door de combinatie van 3 kenmerken waar het model naar genoemd is.

Allereerst is er de zogenaamde *range*, dit is de afwijking tot een centrale toonhoogte. Deze centrale toonhoogte is de noot die in het midden ligt van de distributie van alle noten uit alle muziekstukken van het beschikbare corpus. Globaal gezien hebben noten die dicht bij die centrum liggen een grotere waarschijnlijkheid tot voorkomen dan noten die verder van dit centrum verwijderd zijn. Dit wordt geïllustreerd in figuur 3.2, waarbij elke noot voorgesteld wordt door een geheel getal. De centrale C (do) heeft als waarde 60 gekregen. Een eenheid in deze schaal komt overeen met een halve toon, de volgende C krijgt dus als waarde 72 omdat het verschil tussen de deze twee noten 12 halve tonen is.

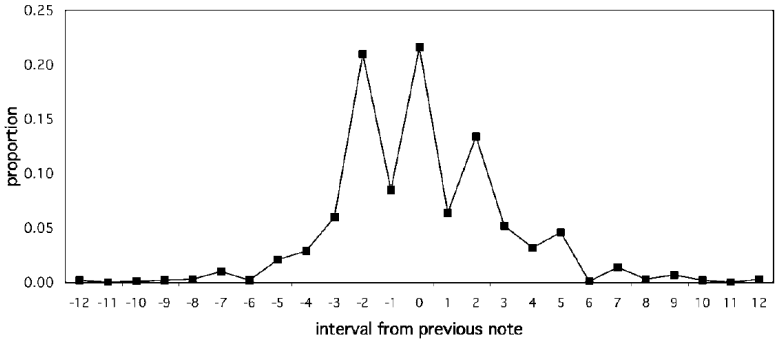
Ook nog belangrijk om op te merken is dat in dit model een melodielijn als een opeenvolging van noten beschouwd wordt, zonder informatie over het ritme. Het ritme van een muziekstuk zal dus ook geen invloed hebben op de score die het RPK-model hieraan geeft.

Verder is er ook nog de *proximity*, dit heeft te maken met de relatieve afstand tot de voorgaande noot. Bepaalde groottes van sprongen zijn veel waarschijnlijker dan andere. Een sprong met een terts (3 of 4 halve tonen) of een kwint (7 halve tonen) komt bijvoorbeeld veel vaker voor dan een sprong van een sext(8 of 9 halve tonen).



FIGUUR 3.2: Distributie van alle noten in het Essencorpus.

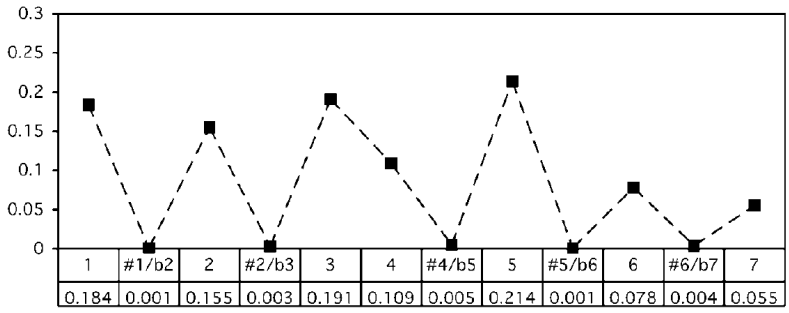
In het algemeen zijn kleine sprongen veel waarschijnlijker dan grotere. Figuur 3.3 geeft de frequenties weer waarmee bepaalde intervallen tussen opeenvolgende noten voorkomen in het Essencorpus.



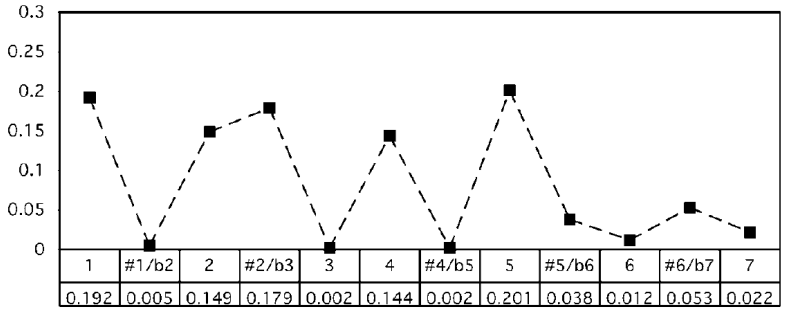
FIGUUR 3.3: Proporties van voorkomen van alle intervallen van opeenvolgende noten in het Essencorpus.

Ten slotte wordt er ook nog rekening gehouden met de *key*, ofwel de toonaard van het muziekstuk. Afhankelijk van de toonaard zijn bepaalde noten waarschijnlijker om voor te komen dan andere. Zo is de grondtoon van een toonladder bijvoorbeeld altijd sterk aanwezig terwijl de noot die een halve toon hoger ligt quasi nooit zal voorkomen in het muziekstuk. Ook is er een verschil in profiel tussen majeur en mineur toonaarden, hier wordt ook rekening mee gehouden. Dit wordt ook geïllustreerd in figuren 3.4 en 3.5 die respectievelijk voor stukken die in grote en kleine toonaarden staan de distributies van noten uit het Essencorpus weergeeft.

De *range*- en *proximity*-waarden worden gemodelleerd door een normaal verdeling rond respectievelijk de centrale en de vorige noot uit het muziekstuk. De *key*-waarde van een noot wordt bepaald aan de hand van de frequentie van voorkomen van deze zijn nootfunctie in alle muziekstukken van het corpus.



Figuur 3.4: Proporties van voorkomen in het Essencorpus van nootfuncties in grote toonladder.



Figuur 3.5: Proporties van voorkomen in het Essencorpus van nootfuncties in kleine toonladder.

Bepaling score van een volledige melodielijn

Nu kan voor elke noot in een melodielijn bepaald worden wat zijn waarschijnlijkheid is. Dit gebeurt door het product te nemen van zijn drie verschillende probabiliteitsscores (*range*, *proximity* en *key*) en dit te normaliseren over alle mogelijke toonhoogtes. De score van een volledige melodielijn staat dan gelijk aan het gemiddelde score over al zijn noten. De score van een melodielijn geeft dus weer wat de gemiddelde waarschijnlijkheid is van een noot in die melodielijn volgens het RPK-model.

De score van een melodielijn kan berekend worden door het product te nemen van de scores van elke noot in het stuk en hier dan het meetkundig gemiddelde van te nemen. Aangezien dit product kan leiden tot zeer kleine waarden voor de waarschijnlijkheid wordt in de plaats daarvan gerekend met de logaritmes van deze probabiliteiten. De probabiliteitsscore van een noot wordt dus voorgesteld door het logaritme van zijn echte probabiliteit. De score van een volledige melodielijn is nu dus het rekenkundig gemiddelde van de individuele scores van alle noten in het muziekstuk. Dit is computationeel interessanter.

De broncode die gebruikt werd om deze scores te berekenen wordt weergegeven in appendix A.2. De score die in deze code berekend wordt voor een muziekstuk is de

som van de logaritmen van de probabiliteiten van alle noten in het muziekstuk. Door deze waarde te delen door het aantal noten in het muziekstuk kan een gemiddeld logaritme van de probabiliteit bekomen worden.

Beoordeling van een muziekstuk

Men zou snel kunnen stellen dat voor het beoordelen van een muziekstuk met het RPK-model er gewoon gekeken zal worden naar de score die het model geeft aan dat stuk en dat een hogere score dan logischerwijs overeenkomt met een (theoretisch gezien) beter muziekstuk. Er moet echter ook rekening gehouden worden met de beperkingen van het RPK-model.

Aangezien dit model afhankelijk is van de drie besproken parameters zal een ideale melodielijn voor dit model bestaan uit een opeenvolging van noten van telkens dezelfde toonhoogte (zodat de afstand tot de vorige noot telkens 0 is), waarvan deze noot zeer waarschijnlijk is in de toonaard en dicht bij de centrale toonhoogte ligt. Het RPK-model geeft dus hoge scores aan stukken die zich dicht rond de centrale noot afspelen en kleine sprongen vertonen tussen opeenvolgende noten. Dit zijn echter niet de meest interessante muziekstukken om naar te luisteren.

Vandaar dat bij het beoordelen van een transformatie er gaat gekeken worden naar het verschil in scores tussen het originele muziekstuk en zijn getransformeerde versie. Hierbij zal een transformatie als beter bestempeld worden dan een andere wanneer hij gemiddeld gezien minder afwijking van score oplevert tussen de getransformeerde melodie en het originele muziekstuk. De reden dat dit zo gebeurt is omdat de originele stukken telkens verondersteld worden van goede kwaliteit te zijn.

Een sterke verbetering in score zou dan betekenen dat het stuk waarschijnlijk minder interessant is geworden (minder grote sprongen, meer voorspelbaarheid). Een sterke verlaging van de score zou betekenen dat het stuk te onvoorspelbaar geworden is en een grotere kans heeft om als frustrerend ervaren te worden door de luisteraar. Vandaar dat de score van het originele stuk telkens als een soort van ‘gouden verhouding’ zal bekeken worden horende bij het ritme van dat stuk (dat onveranderd zal blijven na melodische transformatie) tussen voorspelbaarheid en verrassing.

Hoofdstuk 4

Melodische transformatie

4.1 Inleiding

In dit hoofdstuk zullen verschillende melodische transformaties besproken worden met hun voor- en nadelen. Deze transformaties zullen werken op melodielijnen die monofoon zijn. Deze melodielijnen zullen behandeld worden als een opeenvolging van noten (toonhoogtes) en het ritme zal na transformatie gewoon behouden blijven. Enkel de toonhoogte van een noot zal aangepast worden. Het ritme zal bij de transformaties die hier besproken staan ook geen invloed hebben op de transformatie van een noot.

4.1.1 Fibonacci

De transformaties die hier als voorbeeld gegeven worden zullen vaak in een zekere vorm de rij van Fibonacci[5] bevatten. Dit komt omdat in een vroeg stadium van het onderzoek de vraag bestond of transformaties die voortgaan op de rij van Fibonacci een beter resultaat zouden bieden dan willekeurige transformaties. Dit omdat de rij van Fibonacci in de natuur zo nadrukkelijk aanwezig is dat ook redelijkerwijs de vraag gesteld zou kunnen worden of deze in de muziekwereld zo een invloed zou kunnen hebben. Dit bleek moeilijk hard te maken. Aangezien de resultaten ook zeker niet slechter waren en omdat het ook onderdeel van het onderzoek was, zijn deze transformaties gebaseerd op de rij van Fibonacci vaak gebruikt ter illustratie.

4.1.2 Beschrijving van een transformatie

De melodische transformaties die besproken worden in dit hoofdstuk gaan telkens beschreven worden aan de hand van een tabel. Deze tabel zal telkens een mapping van 8 waarden bevatten. de tweede rij zal altijd waarden tussen -5 en +6 bevatten die als betekenis hebben met welke waarde (namelijk de hoeveelheid halve tonen) een bepaalde noot verhoogd of verlaagd moet worden.

Welke noot met welke hoeveelheid getransformeerd moet worden wordt dan telkens weergegeven via een waarde op de bovenste rij. Deze rij is ook telkens cyclisch modulo 8. Wanneer bijvoorbeeld zoals in tabel 4.1 de index van de noot weergegeven

Index (mod 8)	0	1	2	3	4	5	6	7
Verhoging	1	1	2	3	5	-4	1	-3

Tabel 4.1: Transformatie afhankelijk van de positie van de noot.

wordt op de bovenste rij, dan zal de noot op positie 4 met 5 halve tonen verhoogd worden door de transformatie. Maar ook de noot op positie 12 zal met 5 halve tonen verhoogd worden omdat 12 ook 4 geeft als rest na deling door 8.

Afronding naar de toonaard

In hoofdstuk 3.3 werd reeds het RPK-model besproken. Dit model wordt gebruikt ter evaluatie van de transformaties. Bij de bespreking van dit model was een van de drie belangrijke kenmerken van een noot om de probabiliteit te bepalen de *key*. Noten die in de toonaard voorkomen zijn zo veel waarschijnlijker om voor te komen dan noten die niet in de toonaard voorkomen. Deze noten, die niet in de toonaard voorkomen, klinken over het algemeen ook vals voor de luisteraar.

Daarom is ervoor gekozen om na uitvoer van de transformaties nog een afronding door te voeren. Deze afronding bestaat erin om na de transformatie van een noot in de melodielijn, indien deze niet tot de toonaard behoort (en enkel dan), te verhogen of verlagen met een halve toon. De afronding zal zijn naar die noot van de twee die de hoogste probabiliteit heeft volgens het RPK-model. Deze twee noten zullen ook telkens beide wel tot de toonaard behoren.

De transformaties zelf zullen theoretisch geen rekening houden met deze afronding. Met andere woorden, dit zal niet expliciet vermeld worden in de beschrijvingen van de transformaties. Het is echter wel belangrijk te weten dat dit wel degelijk altijd gebeurt. Vandaar dat het soms ook kan zijn dat het in een illustratie lijkt alsof een transformatie een halve toon the hoog of te laag uitgevoerd is voor een bepaalde noot. Dit ligt dan aan de zonet besproken afronding.

4.2 Afbeelding afhankelijk van de positie

4.2.1 Beschrijving transformatie

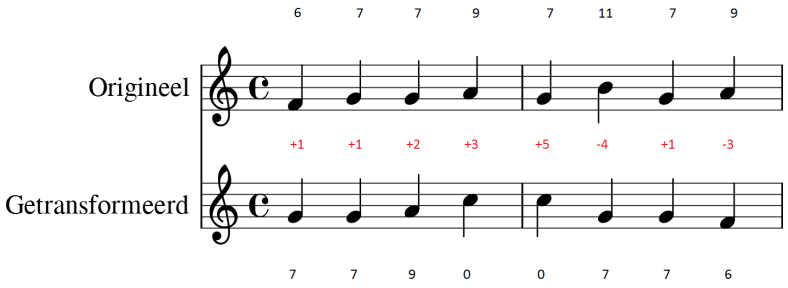
Een eerste, voor de hand liggende melodische transformatie is er eentje die een noot in een muziekstuk gaat transformeren enkel naargelang zijn positie in de melodielijn. De transformatie die ter illustratie dient van dit concept wordt beschreven in tabel 4.1. Zo zal de noot op positie 6 door deze transformatie met 1 halve toon verhoogd worden. De noot op positie 13 zal met 4 halve tonen verlaagd worden.

Voorbeeld

In figuur 4.1 wordt ter illustratie deze transformatie toegepast op een korte melodielijn. De bovenste lijn geeft de originele melodie weer, de onderste lijn geeft het resultaat weer na transformatie. Bij de twee melodielijnen staat bij elke noot telkens ook zijn

representatie in halve tonen (modulo 12). Dit zodat het voor de lezer makkelijker te volgen is hoe de transformatie precies verloopt.

Tussen de twee notenbalken wordt aangegeven welke sprong de transformatie oplegt aan de melodielijn. Zo zal het verschil in getalwaarde tussen overeenkomstige noten op de bovenste en de onderste notenbalk telkens gelijk zijn aan de waarde die hier aangegeven staat. Merk op dat die voor de tweede en zevende noot in dit voorbeeld niet het geval is. Hier lijkt het verschil tussen de noten in de bovenste en onderste lijn telkens een halve toon kleiner dan deze zou moeten zijn. Dit komt door de afronding besproken in onderdeel 4.1.2 aangezien de noot met getalwaarde 8 (G♯/A♭) niet tot de toonaard van het muziekstuk (Do groot) behoort.



FIGUUR 4.1: Voorbeeld van toepassing transformatie afhankelijk van positie.

Bespreking transformatie

Het grootste voordeel van deze transformatie is dat hij zeer eenvoudig uit te voeren en te begrijpen is. Enkel de positie van de noot is van belang met betrekking tot naar welke noot deze getransformeerd zal worden. Een groot nadeel van deze transformatie is dus ook dat deze totaal geen rekening houdt met de eigenlijke toonhoogte van de noot die getransformeerd gaat worden. Er wordt ook geen rekening gehouden met de context van de noot. En als er in het originele stuk patronen zitten zullen die ook nooit herkend en gelijk getransformeerd worden tenzij in het zeer specifieke geval dat deze telkens mooi op een veelvoud van 8 noten van elkaar voorkomen.

Deze transformatie zal dus ook verder niet veel gebruikt worden. Het kan wel interessant zijn om deze transformatie te zien als een soort van *baseline* voor een willekeurige transformatie, om dan andere transformaties mee te kunnen vergelijken. Ook kan het interessant zijn na te gaan of er veel verschil zit tussen zo een transformaties waarin grote sprongen zitten ten opzichte van transformaties waarin vooral kleinere sprongen zitten (aangezien deze het originele stuk minder zullen gaan vervormen).

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	5	-4	1	-3	1	1	2	3

Tabel 4.2: Transformatie afhankelijk van de afstand van de huidige noot tot de vorige noot na transformatie.

4.3 Afbeelding afhankelijk van de afstand ten opzichte van vorige noot

Beschrijving transformatie

Een andere melodische transformatie is er een die een noot in een muziekstuk gaat transformeren naargelang zijn afstand ten opzichte van de vorige noot in het muziekstuk. Hierbij zal er gekeken worden naar de afstand van de noot op positie x in de originele melodielijn ten opzichte van de noot op positie $x - 1$ in de nieuwe melodielijn (dit is dus de getransformeerde waarde van de vorige noot in het originele muziekstuk). De eerste noot van eender welk muziekstuk wordt in deze transformatie behouden omdat deze noot geen voorgaande noot heeft.

Wat ook speciaal is aan deze transformatie is dat de verhoging die weergegeven wordt in de transformatietabel toegepast wordt in de tegengestelde richting van waar de huidige noot in de originele melodie staat ten opzichte van de vorige noot na transformatie. De transformatie die ter illustratie dient van dit concept wordt beschreven in tabel 4.2.

Zo zal een noot die 2 halve tonen hoger ligt dan de getransformeerde waarde van de vorige noot met 1 halve toon verlaagd worden. Een noot die 6 halve tonen lager ligt dan de getransformeerde waarde van de vorige noot zal met 2 tonen verhoogd worden. Tot slot zal een noot die 1 halve toon lager ligt dan de getransformeerde toonhoogte van de vorige noot nog eens met 4 halve tonen verder verlaagd worden (omdat de waarde in de tabel negatief is).

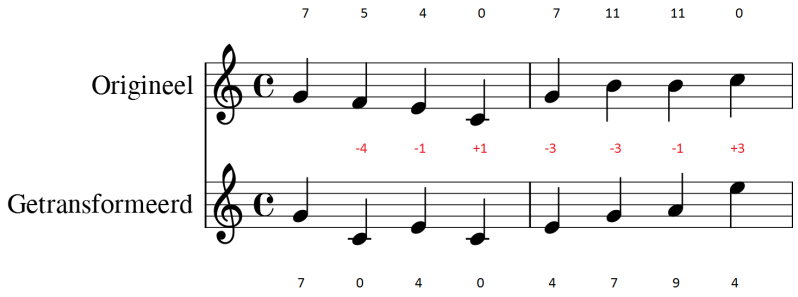
Voorbeeld

In figuur 4.2 wordt een voorbeeld gegeven van een korte melodielijn waarop deze transformatie wordt toegepast. Voor de eerste noot wordt er geen transformatie weergegeven, dat is omdat deze ook geen vorige noot heeft en dus niet getransformeerd wordt.

Als we dan bijvoorbeeld naar de tweede noot kijken dan heeft deze getalwaarde 5, de vorige noot uit de getransformeerde melodie heeft waarde 7 dus het verschil is -2. De absolute waarde is 2 waardoor de sprong als grootte -4 heeft Aangezien het verschil negatief is moet deze waarde bij die van de noot opgeteld worden want we willen een sprong in de richting van de vorige noot. In dit geval zal de sprong toch de andere richting uit gaan aangezien de noot in de tabel zelf negatief is. Zo komen we normaal gezien uit op een noot met getalwaarde 1. Deze noot ligt niet in de toonaard en omdat de noot met getalwaarde 0 (die de grondtoon is van de

toonaard) waarschijnlijker is dan die met waarde 2 wordt de noot met waarde 0 als getransformeerde noot gekozen.

Als we dan verder gaan naar de volgende noot merken we dat het verschil met de getransformeerde van de voorgaande noot 4 halve tonen is. Een absolute waarde van 4 voor het verschil geeft aanleiding tot een sprong van grootte 1. aangezien het verschil positief is en de sprong in tegengestelde richting uitgevoerd moet worden, zal de sprong als waarde -1 hebben. En zo komen we na afronding bij een noot met getalwaarde 4 uit.



FIGUUR 4.2: Voorbeeld van toepassing van de transformatie afhankelijk van het interval ten opzichte van de vorige noot.

Bespreking transformatie

Het interessante aan de transformatie die hier beschreven werd is dat deze de noten niet als losstaand beschouwt, maar ook de context gedeeltelijk in rekening brengt. Dit betekent dat eenzelfde noot naar zeer veel verschillende noten kan getransformeerd worden afhankelijk van de voorgaande noot.

Een voordeel van deze transformatie is ook dat als er een zekere repetitiviteit in het originele muziekstuk zit, deze bijna altijd ook in het getransformeerde stuk zal voorkomen (enkel de noot die net voor zo een patroon voorkomt kan nog een extra invloed hebben). Dit maakt dat de structuur van het originele stuk nog iets meer bewaard blijft.

Een nadeel van deze transformatie is dat deze wel nog steeds blind is voor de rest van de context. Ook de toonhoogte van de noot zelf heeft geen invloed op de grootte en richting van de transformatie op deze noot.

Hoofdstuk 5

Transformaties combineren

In dit hoofdstuk worden twee algoritmes behandeld die varianten zijn van elkaar. Beide algoritmes transformeren een gegeven muziekstuk tot een nieuw muziekstuk zodat dat de totale consonantiescore zo hoog mogelijk is. Dit gebeurt door een aantal toegelaten operaties op het oorspronkelijke muziekstuk. Buiten de originele melodielijn zullen ook een aantal verschillende transformaties (zoals die beschreven zijn in onderdeel 4.3) meegegeven worden aan het algoritme. Deze transformaties zullen door het algoritme gebruikt mogen worden om het originele stuk te transformeren naar een nieuw stuk met een hogere consonantiescore (De score die gebruikt wordt, is deze beschreven in onderdeel 3.3, volgens het 'RPK-Model').

Voor elke noot van de melodielijn heeft het algoritme de keuze om ofwel de noot te behouden, ofwel deze te transformeren conform een van de transformaties die meegegeven werd aan het algoritme. Dit algoritme wordt beschreven in onderdeel 5.1.

In gedeelte 5.2 wordt een algoritme beschreven dat hetzelfde doel heeft maar voldoet aan een extra voorwaarde: een transformatie mag enkel toegepast worden indien dit gebeurt op een minimum aantal opeenvolgende noten van het muziekstuk. Deze extra voorwaarde gaat het *overfitten* tegen, indien deze er niet zou zijn convergeert het algoritme te snel naar een uitgevlakte melodielijn.

Opmerking

Een eerste doel van deze algoritmen is om te deze gaan gebruiken om muziekstukken te gaan creëren die goed zullen klinken. Er wordt nadrukkelijk geprobeerd om die melodielijn te vinden waarnaar getransformeerd kan worden die een zo hoog mogelijke consonantiescore heeft volgens het RPK-model. Wat het algoritme zal doen door de consonantie te verhogen, is ervoor zorgen dat het bekomen muziekstuk normaal gezien niet slecht zou mogen klinken aangezien verondersteld wordt dat het origineel goed klinkt en de score enkel verhoogt wordt. Er werd echter reeds aangehaald dat muziekstukken met een zeer hoge consonantiescore vaak niet interessant klinken (veel dezelfde noten en heel kleine sprongen tussen opeenvolgende noten). Het is ook belangrijk dit in het achterhoofd te houden.

Een tweede opzet van deze algoritmen is om achteraf te kunnen bepalen hoe afhankelijk de consonantiescore zal zijn van het aantal transformaties dat toegepast wordt op het muziekstuk, wat de eventuele minimum transformatielengte als invloed heeft en hoe het aantal beschikbare transformaties de consonantiescore bepaalt.

5.1 Beste sequentie

5.1.1 Doelstelling

Het algoritme dat hier beschreven wordt is afhankelijk van twee parameters. De eerste parameter is een originele melodielijn. De tweede parameter is een verzameling van transformaties (gedefinieerd zoals in onderdeel 4.3), die gebruikt mogen worden door het algoritme. Het doel is nu om aan de hand van enkel deze transformaties de originele melodielijn te transformeren tot een nieuwe melodielijn met een zo hoog mogelijke consonantiescore. Dit moet gebeuren in een enkele *pass* over de melodielijn. Hierbij heeft het algoritme voor elke noot in het muziekstuk twee mogelijke keuzes.

Een eerste mogelijkheid is dat de noot niet getransformeerd wordt en identiek overgenomen wordt in de nieuwe melodielijn. De tweede mogelijkheid bestaat erin dat de noot getransformeerd mag worden, maar dan enkel volgens een van de beschikbare transformaties.

5.1.2 Idee van het algoritme

Bereik van een transformatie

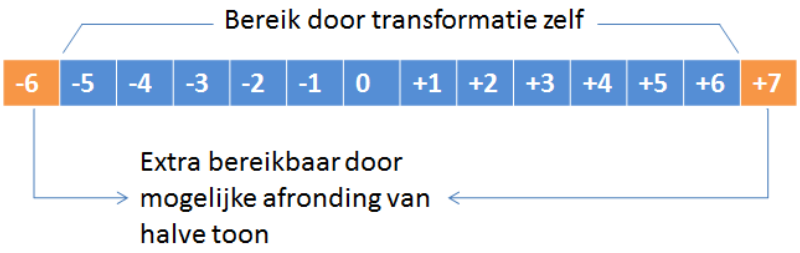
Een belangrijke observatie is dat eender welke noot in het muziekstukje zelf op slechts maximaal 14 verschillende noten kan afgebeeld worden. Elke noot kan namelijk door transformatie enkel afgebeeld worden op een noot die maximaal 5 halve tonen lager ligt dan de oorspronkelijke noot en ook maximaal 6 halve tonen hoger ligt dan de originele noot (dit is een deel van de definitie van de soort transformaties die gebruikt worden in het algoritme, zie hoofdstuk 4.3).

Er is echter nog een speciaal geval waarbij een transformatie tot een van de twee extreme sprongen zou leiden (dus exact -5 of +6) en deze noot dan ook nog eens geen deel zou uitmaken van de toonaard. In dat geval is het mogelijk dat de noot nog een halve toon verder afgerond wordt om terug tot op een noot te komen die in de toonaard ligt. Dit wordt ook geïllustreerd in figuur 5.1.

Elke noot kan dus theoretisch gezien (na afronding) afgebeeld worden op eender welke noot die maximaal 6 halve tonen lager en maximaal 7 halve tonen hoger ligt dat zichzelf. Dit zijn in totaal 14 verschillende mogelijkheden.

Hoog niveau idee van het algoritme

Het belangrijkste idee van het algoritme is gebaseerd op de principes van *dynamic programming* [13]. Toegepast op dit algoritme komt het er op neer dat achtereenvolgens voor elke noot in het muziekstuk het beste pad (en bijhorende beste score) bijgehouden zal worden voor elk van de 14 mogelijke toonhoogtes die deze noot kan



FIGUUR 5.1: Illustratie van de 14 mogelijkheden om een noot in het muziekstuk te transformeren.

aannemen in de nieuwe melodielijn. En enkel op deze optimale paden zal verder gerekend worden om te bepalen wat de beste paden zijn tot de 14 mogelijke toonhoogtes die de volgende noot in het muziekstuk kan aannemen. Dit wordt dan verder herhaald tot alle noten van de originele melodielijn overlopen zijn.

5.1.3 Werking van het algoritme

In dit onderdeel zal de werking van het algoritme beschreven worden. Als extra referentie voor de lezer is de broncode bijgevoegd in appendix A.3. Tijdens de uitvoer van het algoritme zal achtereenvolgens elke noot in het muziekstuk overlopen worden. Telkens zal voor elk van de noten waar de beschouwde noot naartoe getransformeerd kan worden enkel de probabiliteit bijgehouden worden van het pad dat eindigt op deze noot en het meest waarschijnlijk is.

Notatie

Voor het beschrijven van het algoritme worden een aantal notaties en functies beschreven. Allereerst wordt volgende notatie ingevoerd:

$\mathcal{T}$: Set van alle transformaties die beschikbaar zijn voor het algoritme, in deze set zit ook telkens de nultransformatie die een noot nooit verandert.
$\mathcal{P}$: Set van alle noten waar de vorige beschouwde noot naartoe getransformeerd kan worden.
$\mathcal{C}$: Set van alle noten waar de huidige noot naartoe getransformeerd kan worden.
AN: Het aantal noten in het muziekstuk.
x: Noot uit de originele melodielijn die beschouwd wordt als huidige noot in het algoritme.

De functie $transform$ geeft weer dat een vorige noot $p \in \mathcal{P}$ de huidige noot x in de originele melodie onder transformatie $t \in \mathcal{T}$ afbeeldt naar noot $c \in \mathcal{C}$.

$$transform(p, x, t) = c$$

De functie *proxProb* geeft het logaritme terug van de probabiliteit gegeven door de *proximity* parameter van het RPK-model. Als voor de overgang van noot $p \in \mathcal{P}$ naar noot $c \in \mathcal{C}$, de afstandsprobabiliteit tussen deze twee noten gelijk is aan d, dan geeft de functie *proxProb* de volgende waarde terug

$$proxProb(p, c) = log(d)$$

De functie *posProb* geeft het logaritme terug van de probabiliteit van een noot die enkel afhangt van de toonhoogte van de noot zelf. Deze probabiliteiten komen overeen met de *range* en de *key* parameters uit het RPK-model. Deze probabiliteit is dus onafhankelijk van de voorgaande noot. stel dat voor een noot $c \in \mathcal{C}$ de probabiliteit door de *range* parameter gegeven wordt door r. Stel ook dat de probabiliteit bepaald door de *key* gegeven wordt door k. Nu wordt de totale *posProb* gegeven door:

$$posProb(c) = log(r) + log(k)$$

Maximale probabiliteit

$prob(c)$ geeft het logaritme van de probabiliteit weer van het meest waarschijnlijke pad dat eindigt op deze noot $c \in \mathcal{C}$.

Wanneer de eerste noot van het muziekstuk beschouwd wordt en x dus de waarde heeft van deze eerste noot, zal de probabiliteit van het pad dat eindigt op deze noot gelijk zijn aan 1. De probabiliteit van alle andere noten is gelijk aan 0. Dit komt omdat de eerst noot van een muziekstuk nooit getransformeerd wordt. Aangezien het algoritme werkt met de logaritmen van deze probabiliteiten zullen de waarden respectievelijk op 0 en $-\infty$ gezet worden. Wanneer x gelijk is aan de eerste noot van het originele muziekstuk geldt dus:

$$\forall c \in \mathcal{C} : \begin{cases} prob(c) = 0 & \text{if } c == x \\ prob(c) = -\infty & \text{if } c! = x \end{cases}$$

Voor alle mogelijke noten op andere posities van het muziekstuk geldt dat:

$$\forall c \in \mathcal{C} : prob(c) = max(prob(p) + proxProb(p, c) + posProb(c)) \\ |\forall p \in \mathcal{P} : \exists t \in \mathcal{T} : transform(p, x, t) = c \quad (5.1)$$

Door deze functie achtereenvolgens toe te passen voor alle noten van het muziekstuk kan er bepaald worden wat de probabiliteit is van het beste pad, gebruik makend van de gegeven transformaties uit $\mathcal{T}$. Dit zal namelijk gelijk zijn aan de hoogste probabiliteit van alle noten $c \in \mathcal{C}$ in de laatste stap van het algoritme.

Optimale pad

We zijn natuurlijk niet enkel geïnteresseerd in de probabiliteit van het optimale pad, maar ook dit pad zelf, want dit gaat de gevonden melodie weergeven. Daarom wordt er voor de elke combinatie van een positie in de melodie en de mogelijke noten die op die positie kunnen voorkomen, bijgehouden waarvan het optimale pad kwam dat uitkwam bij die exacte noot op die positie. Om deze paden voor te stellen wordt de functie *path* gebruikt. Meer algemeen, stel we zijn op positie i in het algoritme (we beschouwen dus de i-de noot van het muziekstuk), dan zal wanneer het optimale pad dat eindigt op noot n komt van een zekere m, het volgende gelden:

$$path(i, n) = m$$

Wanneer het algoritme voor alle mogelijke noten op elke positie in de melodie de maximale probabiliteit bepaald heeft kan het optimale pad gevonden worden. Dit gebeurt zoals beschreven in algoritme 2.

Algorithm 1 Optimaal pad

$note = argmax_{c \in \mathcal{C}}(prob(c))$
$Path.addToFront(note)$
for $(i = AN : i \geq 2 : i = i - 1)$ **do**
    $note = path(i, note)$
    $Path.addToFront(note)$
end for
return $Path$

Intern in het algoritme worden al deze *path* relaties opgeslagen in een $(n \times 14)$-*matrix*. voor elk van de n posities in de melodie worden voor alle 14 noten waar theoretisch naar getransformeerd kan worden, de index bijgehouden van de noot op de vorige positie waarvan het optimale pad kwam. Een voorbeeld van hoe hier een pad uit gereconstrueerd kan worden, is afgebeeld in figuur 5.2. Voor twee verschillende noten op positie i in het muziekstuk wordt het optimale pad om tot die noot te geraken afgebeeld. De waarden die in de vakjes staan zijn telkens indexen die verwijzen naar een noot op de vorige positie.

5.1.4 Performantie en geheugencomplexiteit

De eerste parameter waarvan het algoritme afhankelijk is, is de lengte van de melodielijn of met andere woorden het aantal noten (AN) in de invoer. Ook het aantal

Index		(i-3)	(i-2)	(i-1)	i	
0	...				5	...
1	...				6	...
2	...				4	...
3	...				2	...
4	...			7	5	...
5	...	3			7	...
6	...		5		3	...
7	...	8	7	6	8	...
8	...				6	...
9	...				11	...
10	...				7	...
11	...				10	...
12	...				7	...
13	...				8	...

Figuur 5.2: Illustratie van de reconstructie van het optimale pad voor 2 verschillende noten op positie i.

beschikbare transformaties (AT) heeft een invloed (in het voorbeeld beschreven in appendix A.3 wordt gebruik gemaakt van slechts 2 transformaties, maar het algoritme werkt voor eender welk aantal transformaties dat gedefinieerd wordt).

Tijd

De snelheid van het algoritme is lineair afhankelijk van beide van deze parameters. Indien de lengte van het originele melodietje met een factor f omhoog gaat en de rest constant blijft dan gaat ook het aantal stappen in het algoritme met een factor f omhoog. Het rekenwerk per stap blijft echter gelijk. Wanneer het aantal transformaties met een factor t omhoog gaat en de rest constant blijft, dan zal het aantal stappen onveranderd blijven. Het rekenwerk per stap gaat wel met een factor t omhoog (op het constante rekenwerk per stap van het 'niet transformeren' na).

$$\underline{\text{Tijd:}}\ \mathcal{O}(AN \times AT)$$

Geheugen

De hoeveelheid geheugen die nodig is voor de uitvoer van het algoritme is lineair afhankelijk van de lengte van de originele melodie. Dit komt omdat de *matrix* array als een van zijn dimensies deze lengte heeft. Wanneer de lengte van het originele stukje met een factor f omhoog gaat, zal de grootte van deze array dus ook met een factor f omhoog gaan. De grootte van de andere twee gebruikte arrays(*past* en *current*) is onveranderlijk ten opzichte van die lengte. De transformaties zelf moeten natuurlijk ook opgeslagen worden, waardoor de het geheugengebruik ook lineair afhankelijk is van het aantal transformaties. maar in het totale geheugengebruik is de *matrix*-array dominant en opzichte van de opslag van de respresentaties van de transformaties, aangezien deze zo veel groter is. Het aantal gebruikte transformaties heeft dus weinig effect op het geheugengebruik van het algoritme. En in het algemeen kan er dus veronderstel worden dan het aantal gebruikte transformaties het geheugengebruik niet merkbaar beïnvloedt.

Geheugen: $\mathcal{O}(AN)$

5.2 Beste sequentie met minimum transformatie lengte

5.2.1 Doelstelling

Het algoritme dat in dit onderdeel beschreven wordt heeft dezelfde doelstelling als het algoritme beschreven in onderdeel 5.1. Enkel wordt dit algoritme aan nog een extra restrictie onderworpen.

Zo zal dit algoritme afhankelijk zijn van drie parameters. De eerste twee parameters zijn een originele melodielijn en een aantal toegelaten transformaties. Als extra parameter is dit algoritme nog afhankelijk van een opgegeven minimum transformatie lengte.

Dit betekent dat het algoritme enkel een deel van de originele melodielijn mag transformeren als het voor minstens dit opgegeven aantal opeenvolgende noten dezelfde transformatie uitvoert.

5.2.2 Idee van het algoritme

Hoog niveau idee van het algoritme en notatie

Ook nu zal er gebruik gemaakt worden van de principes van *dynamic programming*. Dit zal enkel op een verschillende manier gebeuren dan bij het vorige algoritme, aangezien de opgegeven minimumlengte verhindert om eenzelfde data representatie te gebruiken. Ook zullen dezelfde regels gelden als in het overeenkomstige onderdeel uit 5.1 wat het bereik van een tranformatie betreft.

Het idee van het algoritme bestaat erin om elke noot van het muziekstukje chronologisch te overlopen. Bij elke noot uit het originele stuk zijn er dan een aantal mogelijkheden om het beste pad te bepalen dat eindigt op een van de 14 mogelijke noten waarnaar de originele noot getransformeerd kan worden.

Als in het vervolg van de tekst gesproken wordt over een “geldig pad”, dan wordt hier een pad mee bedoeld dat de regels van de minimumlengte voor transformatie respecteert.

Verschillende mogelijkheden om tot een optimaal pad te komen

In dit onderdeel wordt een intuïtieve beschrijving gegeven van hoe het optimale pad tot op een bepaalde positie in het muziekstuk kan berekend worden. Dit, gebruik makend van de optimale paden en bijhorende probabiliteiten van de noten op vorige posities in het muziekstuk.

Een eerste mogelijkheid voor een optimaal pad is de uitbreiding van eender welk optimaal pad dat eindigt op een van de noten waar de vorige noot uit het muziekstuk naar getransformeerd kan worden.

Een tweede mogelijkheid is het uitbreiden van een optimaal pad dat geldig is, eindigt op de vorige noot en eindigt met transformatie f uit te breiden met dezelfde transformatie f.

Tot slot is er ook nog de mogelijkheid om eender welk optimaal en geldig pad dat een lengte ML korter is dan het huidige uit te breiden met ML keer dezelfde transformatie. Op deze manier kunnen alle optimale paden bekomen worden die aan de vooropgestelde eisen voldoen.

5.2.3 Werking van het algoritme

In dit onderdeel zal de werking van het algoritme beschreven worden. Als extra referentie voor de lezer is de broncode bijgevoegd in appendix A.4. Tijdens de uitvoer van het algoritme zal achtereenvolgens elke noot in het muziekstuk overlopen worden. Nu zal voor elke transformatie apart bijgehouden worden wat de optimale paden en probabiliteiten zijn voor alle verschillende noten waar de huidige noot naartoe getransformeerd kan worden en die eindigen met die specifieke transformatie.

Notatie

Voor het beschrijven van het algoritme worden een aantal notaties en functies beschreven. Allereerst wordt volgende notatie ingevoerd:

$\mathcal{T}$: Set van alle transformaties die beschikbaar zijn voor het algoritme, in deze set zit ook telkens de nultransformatie die een noot nooit verandert.
nt: De nultransformatie die ook in $\mathcal{T}$ zit.
ML: De minimum transformatie lengte.
$\mathcal{P}_{\mathcal{ML}}$: Set van alle noten waar de noot die ML posities voor de huidige noot naartoe getransformeerd kan worden.
$\mathcal{P}$: Set van alle noten waar de vorige beschouwde noot naartoe getransformeerd kan worden.
$\mathcal{C}$: Set van alle noten waar de huidige noot naartoe getransformeerd kan worden.

AN: Het aantal noten in het muziekstuk.
x: Noot uit de originele melodielijn die beschouwd wordt als huidige noot in het algoritme.
i: De index van de noot die momenteel beschouwd wordt in het algoritme.
OL: Geordende lijst van de noten in de originele melodie.

De functie $transform$ geeft weer dat een vorige noot $p \in \mathcal{P}$ de huidige noot x in de originele melodie onder transformatie $t \in \mathcal{T}$ afbeeldt naar noot $c \in \mathcal{C}$.

$$transform(p, x, t) = c$$

De functie *proxProb* geeft het logaritme terug van de probabiliteit gegeven door de *proximity* parameter van het RPK-model. Als voor de overgang van noot $p \in \mathcal{P}$ naar noot $c \in \mathcal{C}$, de afstandsprobabiliteit tussen deze twee noten gelijk is aan d, dan geeft de functie *proxProb* de volgende waarde terug

$$proxProb(p, c) = log(d)$$

De functie *posProb* geeft het logaritme terug van de probabiliteit van een noot die enkel afhangt van de toonhoogte van de noot zelf. Deze probabiliteiten komen overeen met de *range* en de *key* parameters uit het RPK-model. Deze probabiliteit is dus onafhankelijk van de voorgaande noot. stel dat voor een noot $c \in \mathcal{C}$ de probabiliteit door de *range* parameter gegeven wordt door r. Stel ook dat de probabiliteit bepaald door de *key* gegeven wordt door k. Nu wordt de totale *posProb* gegeven door:

$$posProb(c) = log(r) + log(k)$$

Tot slot is er ook nog een algoritme *extendPath* dat berekent wat het logaritme van de probabiliteit is van de uitbreiding van een pad. Deze uitbreiding moet van lengte l zijn en volgens een meegegeven transformatie $t \in \mathcal{T}$ bekomen worden. Dit algoritme krijgt ook het pad *path* waarop deze uitbreiding moet uitgevoerd worden mee. Tot slot wordt aan dit algoritme wordt ook een lijst *list* van opeenvolgende noten meegegeven, dit zijn de noten van het originele muziekstuk waarop deze transformatie toegepast moet worden. Dit algoritme geeft niet alleen het logaritme van de probabiliteit van deze uitbreiding mee, maar ook het volledige pad dat na uitbreiding bekomen wordt.

Optimale paden voor verschillende transformaties

De reden dat paden die eindigen op verschillende transformaties apart bijgehouden worden heeft twee redenen.

Algorithm 2 $extendPath(l, t, Path, list)$

$Prob = 0$
$p = Path.last$
for $(i = 1 : i \leq l : i = i + 1)$ **do**
  $x = list[i]$
  $c = transform(p, x, t)$
  $Path.add(c)$
  $Prob = Prob + proxProb(p, c) + posProb(c)$
  $p = c$
end for
return $(Prob, Path)$

Ten eerste zal, wanneer het algoritme op voorhand weet dat een bepaald pad geldig is (voldoet aan de voorwaarden voor minimum transformatie lengte) en eindigt op een zeker transformatie t, weten dat dit pad enkel door dezelfde transformatie t kan uitgebreid worden naar enkel de volgende noot. Voor andere transformaties zal minstens ML keer die transformatie moeten toegepast worden. Met deze voorstelling is het meteen duidelijk welke transformatie ook voor een noot mag uitgevoerd worden en welke voor minstens ML volgende noten moet uitgevoerd worden om terug een geldig pad te bekomen.

Ten tweede heeft het ook te maken met de extra eis van de minimum transformatie lengte. Indien alle optimale paden zouden bijgehouden worden met een *path* relatie zoals in het vorige algoritme, die onafhankelijk zou zijn van de transformatie waarop geëindigd werd, dan kan tijdens de uitvoer van het algoritme de regel van de minimum transformatie lengte geschonden worden. Een voorbeeld hiervan wordt hieronder gegeven:

Stel dat: $ML = 2$
$path(3, W_1) = V_1$
$path(2, V_1) = U_1$
$path(3, W_2) = V_2$
$path(2, V_2) = U_2$
Dan zijn momenteel de optimale paden voor W_1 en W_2, respectievelijk $U_1 - V_1 - W_1$ en $U_2 - V_2 - W_2$.
Stel verder dat het optimale pad voor W_1 bekomen werd door tweemaal een transformatie $t_1 \in \mathcal{T}$ toe te passen en dat het optimale pad voor W_2 bekomen werd door tweemaal $t_2 \in \mathcal{T}$ toe te passen.
Beide paden zijn momenteel dus geldig onder de voorwaarde van de minimum transformatie lengte.
Stel nu dat er in het verdere verloop van het algoritme nog een beter pad gevonden wordt dat eindigt in W_2 en dat verloopt volgens $U_2 - V_1 - W_2$. Stel ook dat dit pad gevonden werd door het tweemaal toepassen van $t_3 \in \mathcal{T}$. Nu wordt de *path* functie aangepast zodat deze de juiste waarden teruggeeft.

Nu geldt:
$path(3, W_1) = V_1$
$path(2, V_1) = U_2$
$path(3, W_2) = V_1$
$path(2, V_2) = U_2$
Dan zijn nu de optimale paden voor W_1 en W_2, respectievelijk $U_2 - V_1 - W_1$ en $U_2 - V_1 - W_2$.
Het optimale pad van W_1 is nu geen geldig optimaal pad meer aangezien het door het achtereenvolgens toepassen van telkens een maal de transformatie $t3$ en $t1$ niet meer voldoet aan de voorwaarden van de minimum transformatie lengte.

Maximale probabiliteit

$prob(c, t)$ geeft het logaritme van de probabiliteit weer van het meest waarschijnlijke pad dat eindigt op deze noot $c \in \mathcal{C}$. Deze noot (en zelfs minstens de laatste ML noten) moet in dit geval bekomen zijn door gebruik van transformatie $t \in \mathcal{T}$.

Wanneer de eerste noot van het muziekstuk beschouwd wordt en x dus de waarde heeft van deze eerste noot, zal de probabiliteit van het pad dat eindigt op deze noot gelijk zijn aan 1. De probabiliteit van alle andere noten is gelijk aan 0. Dit komt omdat de eerste noot van een muziekstuk nooit getransformeerd wordt. Dit zal enkel het geval zijn voor de nultransformatie. Voor alle andere transformaties zal eender welke noot een kans 0 hebben om deel uit te maken van het optimale pad aangezien er geen enkel optimaal pad kan zijn dat eindigt op een transformatie als dat pad maar een noot kan bevatten. Aangezien het algoritme werkt met de logaritmen van deze probabiliteiten zullen de waarden respectievelijk op 0 en $-\infty$ gezet worden. Wanneer x gelijk is aan de eerste noot van het originele muziekstuk en transformatie $t \in \mathcal{T}$ beschouwd wordt geldt dan:

$$\forall c \in \mathcal{C}, t \in \mathcal{T} \begin{cases} prob(c,t) = 0 & \text{if } c == x \ \& \ t = nt \\ prob(c,t) = -\infty & \text{if } c == x \ \& \ t! = nt \\ prob(c,t) = -\infty & \text{if } c! = x \end{cases}$$

Voor alle andere noten en transformaties op alle andere posities in het muziekstuk geldt:

$$\begin{aligned} \forall t \in \mathcal{T} : ProbSame(c,t) = max(&prob(p,t) \\ &+ extendPath(1, t, path(i-1, p, t), OL[i]).prob) \\ &| \forall p \in \mathcal{P} : transform(p, x, t) = c \end{aligned} \quad (5.2)$$

$$\forall t \in \mathcal{T} : ProbDifferent(c,t) = max(prob(p_{ML},t) + extendPath(i-ML,t,path(i-ML,p_{ML},t1).prob, OL[(i-ML+1)..i]) | \forall p_{ML} \in \mathcal{P}_{\mathcal{ML}}, \forall t1 \in \mathcal{T} : t1! = t, extendPath(i-ML,t,path(i-ML,p_{ML},t1).path.last = c \quad (5.3)$$

$$\forall c \in \mathcal{C}, t \in \mathcal{T} : prob(c,t) = max(ProbSame(c,t), ProbDifferent(c,t)) \quad (5.4)$$

Door deze functie achtereenvolgens toe te passen voor alle noten van het muziekstuk kan er bepaald worden wat de probabiliteit is van het beste pad, gebruik makend van de gegeven transformaties uit $\mathcal{T}$ rekening houdend met de minimum transformatie lengte ML. Dit zal namelijk gelijk zijn aan de hoogste probabiliteit van de combinatie van alle noten $c \in \mathcal{C}$ en transformaties $t \in \mathcal{T}$ in de laatste stap van het algoritme.

Optimale pad

$path(i,n,t)$ stelt het volledige pad voor dat optimaal is onder de gegeven voorwaarden en dat eindigt bij noot n op positie i en door te eindigen met transformatie $t \in \mathcal{T}$. Initieel zijn al deze paden volledig leeg, enkel het pad dat eindigt op de nultransformatie en op de noot die gelijk is aan de startnoot bestaat uit exact deze noot. Telkens er een pad gevonden wordt dat eindigt op noot n en met transformatie $t \in \mathcal{T}$, dan zal niet enkel de $prob(n,t)$ waarde aangepast worden, maar ook $path(i,n,t)$ zal het nieuwe beste pad bevatten voor de combinatie van deze parameters.

Aangezien volledige beste paden bijgehouden worden moet er geen werk verricht worden om het beste pad te bepalen. Er moet enkel dat pad gekozen worden van lengte AN dat de hoogste probabiliteit oplevert. Voor de combinatie van alle noten $c \in \mathcal{C}$ en transformaties $t \in \mathcal{T}$ is er zo een optimaal pad dat hier een kandidaat voor is. Van al deze kandidaten zal het pad met de hoogste probabiliteit het optimale pad zijn voor het geheel.

$$optPath = argmax_{c \in \mathcal{C}, t \in \mathcal{T}}(prob(path(AN,c,t)))$$

5.2.4 Performantie en geheugencomplexiteit

In dit algoritme zijn er drie parameters waar rekening mee dient gehouden te worden bij het bespreken van de tijds- en geheugencomplexiteit. Deze drie parameters zijn het aantal noten in de originele melodielijn (AN), het aantal beschikbare transformaties (AT) en de minimum transformatie lengte (ML).

Tijd

De performantie van het algoritme is lineair afhankelijk van het aantal noten. Dit komt doordat voor elke noot in het oorspronkelijk muziekstuk een stap in het algoritme moet uitgevoerd worden. Het rekenwerk per stap verandert niet wanneer de lengte van het stuk verandert.

De snelheid van het algoritme is kwadratisch afhankelijk van het aantal toegestane transformaties. Dit komt omdat voor elke stap in het algoritme we voor elke transformatie gaan proberen een optimaal pad te vinden dat kan vertrekken van bij tussenoplossingen horende bij alle andere transformaties. Het aantal stappen in het algoritme verandert niet wanneer het aantal transformaties verandert. In totaal geeft dit dus een kwadratische afhankelijkheid.

Tot slot is de snelheid van het algoritme ook afhankelijk van de minimum transformatie lengte volgens $\mathcal{O}(ML \times (AN - ML))$. zowel het aantal stappen in het algoritme als het aantal berekende paden per stap zal onveranderd blijven wanneer deze parameter verandert. Het extra rekenwerk dat gecreëerd wordt door het moeten uitbreiden van paden die ML korter zijn dan de huidige lengte is lineair afhankelijk van de grootte van ML. Maar het aantal paden dat zo berekend moet worden is lineair afhankelijk van $(AN - ML)$. Hierdoor zal de totale rekentijd afhankelijk zijn van het product van deze twee.

$$\underline{\text{Tijd:}}\ \mathcal{O}(AN \times ML \times (AN - ML) \times AT^2)$$

Geheugen

De totale opslagcapaciteit die noodzakelijk is voor de uitvoer van het algoritme is ten eerste lineair afhankelijk van de lengte van de invoer sequentie van noten. Voor alle mogelijke transformaties zullen namelijk optimale paden bijgehouden worden en de lengte van deze paden is altijd van dezelfde grootte-orde als de lengte van het originele.

Ten tweede is het geheugengebruik ook lineair afhankelijk van de minimum transformatie lengte. Er worden namelijk optimale sub paden bijgehouden in het algoritme voor de laatste ML beschouwde noten. De lengte van deze paden is ook elk van dezelfde grootte-orde.

Tot slot is er nog het aantal transformaties. Ook deze parameter zal een lineaire invloed hebben op het geheugengebruik. Dit aangezien voor elke beschikbare transformatie even veel optimale paden bijgehouden worden die eindigen op deze transformatie.

$$\underline{\text{Geheugen:}}\ \mathcal{O}(AN \times ML \times AT)$$

Hoofdstuk 6

Experimenten en resultaten

In dit hoofdstuk worden de belangrijkste experimenten besproken die uitgevoerd in het verloop van deze masterproef. Deze hebben zowel betrekking op de besproken melodische transformaties, de algoritmes om deze transformaties te combineren en ook het RPK-model dat gebruikt werd om deze transformaties te evalueren.
De resultaten worden meestal weergegeven op een plot waarbij op de y as de gemiddelde probabiliteit staat. Deze probabiliteit staat voor de gemiddelde probabiliteit van voorkomen van een noot (gegeven de vorige noot) volgens het RPK-model in alle muziekstukken waarop er getest werd. Als er dus in dit hoofdstuk gesproken wordt over een gemiddelde probabiliteit van een bepaald muziekstuk dan betekent dit de gemiddelde probabiliteit van een noot in dit muziekstuk.

6.1 Melodische transformaties

6.1.1 Vergelijking van de twee melodische transformaties

Beschrijving experiment

Dit experiment heeft als opzet om de twee verschillende soorten transformaties die besproken werden in hoofdstuk 4 te vergelijken in performantie. De eerste transformatie gaat een noot in de melodielijn transformeren naargelang zijn positie in de notensequentie (deze transformatie noemen we in het vervolg van die onderdeel T1). De tweede transformatie gaat een noot transformeren enkel op basis van zijn (melodische) afstand ten opzichte van de vorige noot in de melodielijn (deze transformatie noemen we in de rest van dit onderdeel T2). De werking van deze twee transformaties staat beschreven respectievelijk in hoofdstukken 4.2 en 4.3.

Voor dit experiment zullen voor 50 testgevallen telkens 10 verschillende willekeurige voorkomens voor beide transformatie getest worden. Zo een willekeurige transformatie wordt bekomen door 8 opeenvolgende getallen tussen -5 en +6 (beide grenzen inclusief) willekeurig te genereren. Noem deze 8 getallen in volgorde X_i met $0 \leq i \leq 7$. Voor T1 levert dit dan de transformatie op beschreven in tabel 6.1, voor T2 levert dit de transformatie beschreven in tabel 6.2.

Pos (mod 8)	0	1	2	3	4	5	6	7
Verhoging	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7

Tabel 6.1: Willekeurige transformatie volgens T1, gebruikt in het experiment van onderdeel 6.1.1.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7

Tabel 6.2: Willekeurige transformatie volgens T2, gebruikt in het experiment van onderdeel 6.1.1.

Transformatie	Consonantiescore	Probabiliteit(%)
Oiginele melodie	-2.36	9.42
T1	-3.72	2.43
T2	-3.19	4.12

Tabel 6.3: Resulaten van experiment 6.1.1. Gemiddelde consonantiescores voor de twee soorten transformaties en de originele melodielijnen die getranformeerd werden. De twee geteste soorten van transformaties staan beschreven in onderdelen 4.2(T1) en 4.3(T2).

Voor elk van de 50 testgevallen worden zo dus 20 transformaties gegenereerd (10 voor T1 en 10 voor T2). Over deze 50 testgevallen en 10 transformaties per transformatie soort wordt dan een gemiddelde consonantiescore berekend. De exponentiële van deze consonantie score levert dan voor beide transformatie soorten een gemiddelde probabiliteit van een getransformeerd muziekstuk.

Resultaten

In tabel 6.3 staan de resultaten van dit experiment weergegeven. Er valt meteen op dat beide transformaties de originele melodielijnen gemiddeld gezien transformeren naar een nieuwe melodielijn met een minder goede consonantiescore. Er is ook een duidelijk verschil merkbaar in kwaliteit van de transformaties, T2 scoort merkbaar beter dan T1. Dit is ook logisch aangezien T1 eigenlijk niets van informatie over het muziekstuk in rekening brengt, buiten zijn absolute positie (modulo 8). T2 brengt de afstand tot de vorige noot in rekening wat er toe kan leiden dat in sommige gevallen de intervallen tussen opeenvolgende noten relatief kleiner gehouden wordt dan bij T1, wat consistenter betere scores kan opleveren volgens het RPK-model.

Positie (mod 8)	0	1	2	3	4	5	6	7
Verhoging	1	1	2	3	5	-4	1	-3

TABEL 6.4: Fibonacci transformatie volgens T1 gebruikt in het experiment van onderdeel 6.1.2.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	1	1	2	3	5	-4	1	-3

TABEL 6.5: Fibonacci transformatie volgens T2 gebruikt in het experiment van onderdeel 6.1.2.

Transformatie	Consonantiescore	Probabiliteit(%)
Oiginele melodie	-2.36	9.42
T1	-2.92	5.42
T2	-2.63	7.21

TABEL 6.6: Resulaten van experiment 6.1.2. Gemiddelde consonantiescores voor de twee Fibonacci transformaties en de originele melodielijnen die getranformeerd werden.

6.1.2 Vergelijking Fibonacci transformatie met gemiddelde transformatie

Beschrijving experiment

Het doel van dit experiment is om na te gaan of een transformatie, die gebaseerd is op de rij van Fibonacci een beter resultaat geeft dan een gemiddelde transformatie. De reden dat er specifiek getest wordt op de rij van Fibonacci is omdat deze bekende rij in zoveel verschillende onderdelen van de natuur te herkennen valt dat het in zeker zin niet onlogisch zou zijn, moest deze ook in de muziek onder bepaalde vormen voorkomen. Voor beide soorten van transformaties die beschreven staan in onderdelen 4.2 en 4.3 (en die we in het kort respectievelijk T1 en T2 noemen), wordt zo een transformatie gecreëerd die gebaseerd is op de rij van fibonacci. Deze transformaties worden beschreven door tabellen 6.4 en 6.5. Hierbij is elk getal op de onderste rij telkens de gelijk aan het overeenkomstig element uit de rij van fibonacci modulo 12, waarbij de waarde van elke sprong tussen -5 en +6 ligt.

Nu wordt op dezelfde 50 testgevallen als waarop getest werd in experiment 6.1.1, deze twee transformaties toegepast. De gemiddelde consonantiescores die deze twee Fibonacci transformaties opleveren kunnen nu vergeleken worden met het gemiddelde algemene geval voor de twee soorten transformaties.

Resultaten

Tabel 6.6 geeft de resultaten van de Fibonacci transformaties weer. In tabel 6.3 staan de resultaten van dezelfde soorten transformaties maar dan met willekeurige

sprongen in plaats van sprongen volgens de rij van Fibonacci. Er valt op dat voor beide transformatie soorten, de transformatie volgens de rij van Fibonacci merkbaar beter scoort dan een gemiddelde transformatie van zijn soort. Een reden hiervoor is dat de sprongen in de Fibonacci transformaties redelijk klein zijn (bijvoorbeeld drie keer op acht een sprong van slechts een halve toon). Hierdoor zal het originele muziekstuk minder aangepast worden en de score dus ook niet zo veel verslechteren dan na een gemiddelde transformatie.

Voor de Fibonacci transformatie volgens T2 kan zelfs gezegd worden dat deze zeer goed scoort, aangezien de consonantiescore die gemiddeld bekomen wordt met deze transformatie zelfs in de buurt ligt van de originele melodielijn. De reden hiervoor is dat er bij deze transformatie veel dezelfde noten na elkaar gecreëerd worden (wat goede scores geeft voor het RPK-model aangezien de *proximity* maximaal is). Dit komt doordat sprongen in dit algoritme in tegengestelde richting uitgevoerd worden als de afstand ten opzichte van de vorige noot. En afstanden van grootte 1, 2 en 3 worden bijvoorbeeld allemaal gecounterd door een sprong van dezelfde grootte in tegengestelde richting. Hierdoor zullen noten die op zo een afstand van elkaar zitten telkens op dezelfde noot afgebeeld worden.

Er kan dus besloten worden dat deze Fibonacci transformaties iets betere resultaten leveren dan een gemiddelde transformatie (wat de consonantiescore betreft). Dit betekent echter niet automatisch dat de muziekstukken ook beter klinken want bijvoorbeeld bij de Fibonacci transformatie volgens tabel 6.5 zullen sequenties gecreëerd worden die veel dezelfde noten bevatten, wat over het algemeen niet als interessante muziek beschouwd wordt.

6.2 RPK-model

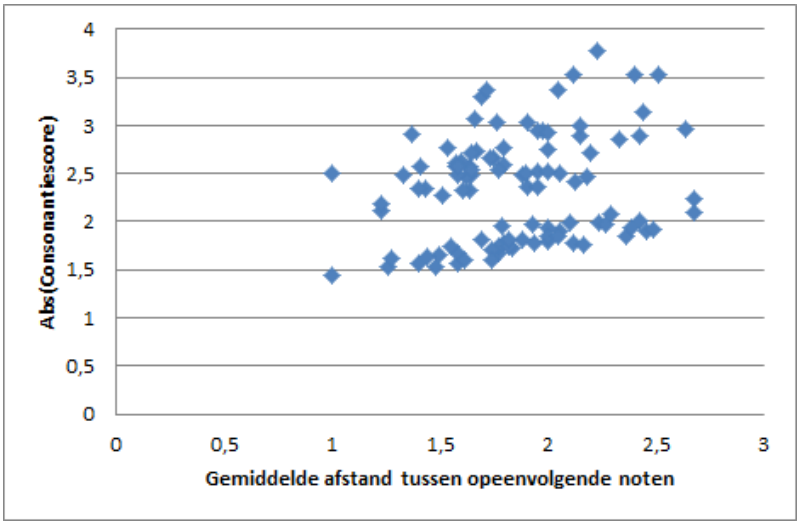
6.2.1 Afhankelijkheid score RPK-Model en gemiddelde afstand tussen opeenvolgende noten

Beschrijving experiment

De bedoeling van dit experiment is om na te gaan hoe sterk het verband is tussen de consonantiescore van een muziekstuk en de gemiddelde afstand tussen opeenvolgende noten in datzelfde muziekstuk. De reden dat deze test wordt gedaan is om dat de afstand tot de vorige noot een belangrijke parameter is in de berekening van de consonantiescore volgens het RPK-model. Indien er een sterke afhankelijkheid zou zijn dan kan de gemiddelde afstand tot opeenvolgende noten in een muziekstuk als een soort eerste voorspeller dienen voor de consonantiescore die computationeel veel minder zwaar is dan de berekening van de consonantiescore zelf.

Voor dit experiment wordt voor 100 muziekstukken uit het Essencorpus de consonantiescore berekend alsook de gemiddelde afstand tussen twee opeenvolgende noten. Hierna wordt de absolute waarde van de consonantiescore uitgezet ten opzichte van de gemiddelde afstand.

Resultaten



FIGUUR 6.1: Resultaten van het experiment uitgevoerd in deel 6.2.1. Absolute waarde van de consonantiescore uitgezet t.o.v. de gemiddelde afstand in halve tonen tussen opeenvolgende noten in het muziekstuk.

De resulaten van dit experiment staan weergegeven in figuur 6.1. op de y-as staat de absolute waarde van de consonantiescore gegeven, een lagere score komt hierbij overeen met een hogere probabiliteit van het muziekstuk volgens het RPK-model. Op de figuur is er een verband zichtbaar waarbij een hogere gemiddelde afstand tussen opeenvolgende noten gemiddeled gezien ook leidt tot een hogere absolute waarde voor de consonantiescore (en dus een lagere probabiliteit voor het muziekstuk). Het verband is echter niet sterk genoeg om de afstand tussen opeenvolgende noten als nuttige voorspeller te zien voor de consonantiescore van een muziekstuk.

6.2.2 Afhankelijkheid score RPK-Model en afstand ten opzichte van gemiddelde notendistributie

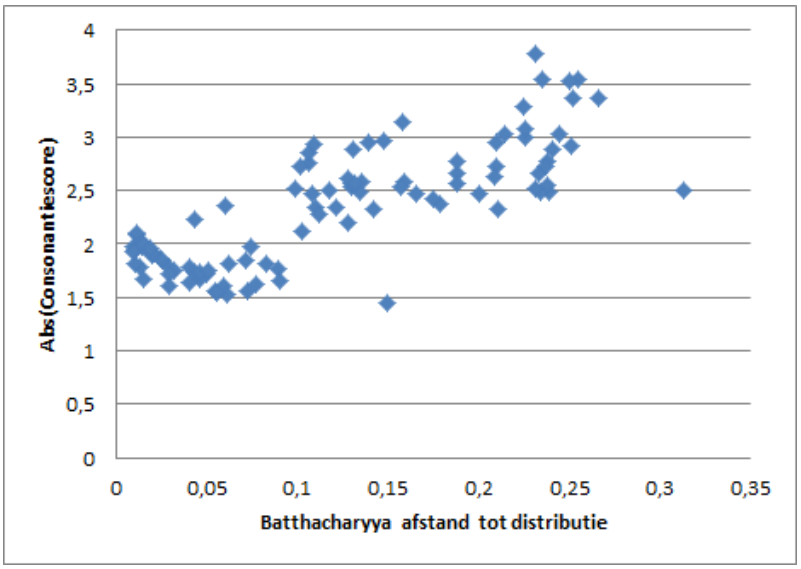
Beschrijving experiment

De bedoeling van dit experiment is om na te gaan hoe sterk het verband is tussen de consonantiescore van een muziekstuk en de afwijking van de notendistributie in het muziekstuk ten opzichte van de gemiddelde notendistributie in het Essencorpus. In figuren 3.4 en 3.5 werden reeds de distributies van alle noten in het Essencorpus weergegeven naargelang hun nootfunctie in de toonaard en naargelang de toonaard groot of klein is. Voor elk muziekstuk kan zo ook een distributie berekend worden en de afstand tot deze gemiddelde distributie zou een voorspeller kunnen zijn voor de consonantiescore van het muziekstuk.

Voor dit experiment wordt voor 100 muziekstukken uit het Essencorpus de consonantiescore berekend alsook de afstand tussen de notendistributie van het

muziekstuk en de notendistributie van het Essencorpus. De afstand tussen deze twee distributies is de zogenaamnde Bhattacharyya afstand [12].

Resultaten



Figuur 6.2: Resultaten van het experiment uitgevoerd in deel 6.2.2. Absolute waarde van de consonantiescore uitgezet t.o.v. de afstand tot de gemiddelde notendistributie in het Essencorpus.

De resultaten van dit experiment staan weergegeven in figuur 6.2. op de y-as staat weer de absolute waarde van de consonantiescore, op de x-as de Bhattacharyya afstand van de notendistributie van het huidige stuk tot de gemiddelde distributie van het Essencorpus. Er is een duidelijk verband merkbaar tussen deze twee waarden. Een stijging van de Bhattacharyya afstand zorgt gemiddeld gezien voor een stijging voor de absolute waarde van de consonantiescore (en dus een verlaging van de probabiliteit van het muziekstukje volgens het RPK-model).

6.2.3 Afhankelijkheid score RPK-Model en combinatie metrieken uit experimenten 6.2.1 en 6.2.2

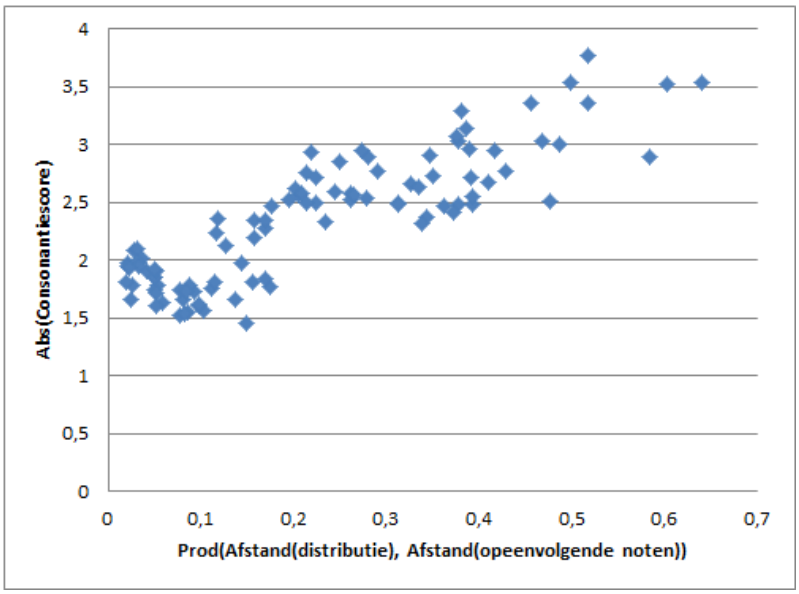
Beschrijving experiment

In experimenten 6.2.1 en 6.2.2 werd al duidelijk dat er wel degelijk een verband merkbaar is tussen de consonantiescore en de twee daar besproken metrieken. Het voordeel van deze metrieken was is ze computationeel minder veel minder zwaar zijn dan de berekening van de consonantiescore zelf. De afhankelijkheid was echter telkens niet voldoende om echt als een goede voorspeller beschouwd te worden. Het doel van dit experiment is om te kijken of een combinatie van deze twee metrieken

(die beide toch een zeker afhankelijkheid vertonen) misschien tot betere resultaten kan leiden.

In dit experiment wordt voor 100 muziekstukken uit het Essencorpus eerst de consonantiescore bereken. Vervolgens wordt ook het product van de afstand van zijn notendistributie ten opzichte van die van het Essencorpus en de gemiddelde afstand tussen opeenvolgende noten in het muziekstuk berekend.

Resultaten



FIGUUR 6.3: Resultaten van het experiment uitgevoerd in deel 6.2.3. Absolute waarde van de consonantiescore uitgezet t.o.v. het product van de afstand tot de gemiddelde notendistributie in het Essencorpus en de gemiddelde afstand tussen opeenvolgende noten in het muziekstuk.

De resultaten van dit experiment zijn zichtbaar op figuur 6.3. Op de y-as wordt de absolute waarde van de consonantiescore weergegeven. Op de x-as de waarde van het product van de twee afstandsmetrieken die besproken werden. Er is een duidelijk verband merkbaar waarbij een hogere waarde van het product gemiddeld gezien ook leidt tot een hogere absolute waarde van de consonantiescore (en dus een lagere probabiliteit van het muziekstuk).

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	5	-4	1	-3	1	1	2	3

Tabel 6.7: Transformatie gebruikt in het experiment van onderdeel 6.3.1.

6.3 Werking van de algoritmes voor het combineren van transformaties

6.3.1 Transformaties combineren: 1 transformatie, meerdere iteraties

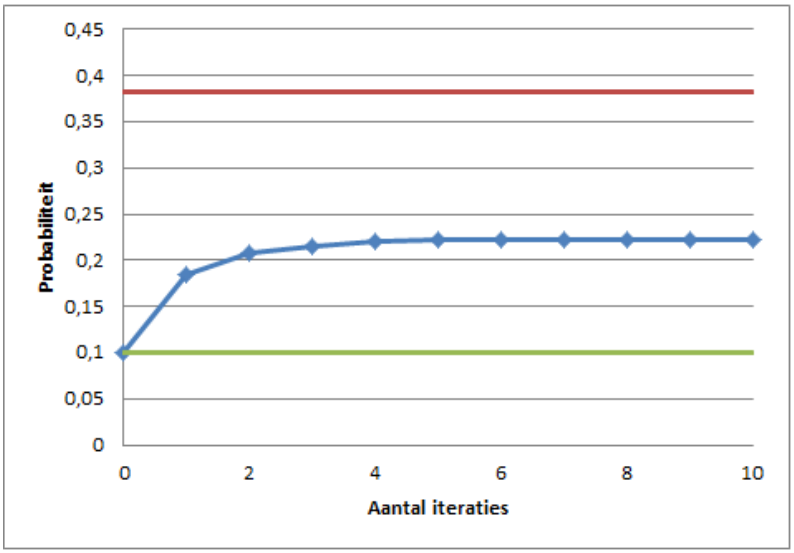
Beschrijving experiment

Dit experiment heeft betrekking tot het algoritme dat besproken werd in onderdeel 5.1. Dit experiment gaat nagaan wat de invloed is van het aantal iteraties van het aantal iteraties van dit algoritme op de consonantiescore van het totale muziekstuk. Er wordt in dit algoritme slechts gebruik gemaakt van een transformatie. Deze gebruikte transformatie wordt weergegeven in tabel 6.7.

Deze test wordt uitgevoerd op 100 muziekstukken uit het Essencorpus. De gemiddelde probabiliteit van de originele stukken wordt berekend alsook de gemiddelde probabiliteit van het muziekstuk dat optimaal is volgens het RPK-model in de toonaard van de 100 stukken. Nu kan er gekeken worden naar hoe snel de consonantiescore zich gaat verplaatsten van die van het originele naar die van de theoretisch best mogelijke volgens het model afhankelijk van het aantal iteraties dat het algoritme wordt uitgevoerd.

Resultaten

In figuur 6.4 worden de restultaten weergegeven van dit experiment. De groene lijn op de figuur geeft de gemiddelde probabiliteit weer van alle melodieën waarop getest werd. De rode lijn geeft voor al deze melodieën de gemiddelde waarde mee voor het theoretisch beste muziekstuk dat volgens het RPK-model gemaakt kan worden in deze toonaard. Tot slot geeft de blauwe lijn de gemiddelde probabiliteit weer van een noot in de getransformeerde melodie, na toepassing van 1 tot 10 iteraties van de transformatie.
Er valt duidelijk op dat de eerste paar iteraties nog een redelijke verhoging van de probabiliteit teweeg brengt, maar dat na een vijftal iteraties gemiddeld gezien een soort van maximum bereikt is dat met deze transformatie kan bekomen worden. De reden dat deze waarde nog zo ver onder het theoretische maximum ligt, heeft er vooral mee te maken dat er maar 1 mogelijke transformatie is dat het algoritme mag gebruiken, hierdoor zijn er nog steeds maar een zeer beperkt aantal mogelijkheiden om een bepaalde noot te transformeren, en kunnen bijgevolg nog steeds de meeste noten niet bereikt worden vanuit eender welke noot.



FIGUUR 6.4: Resultaten van het experiment uitgevoerd in deel 6.3.1. De groene lijn staat voor de gemiddelde probabiliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabiliteit van de theoritisch beste melodielijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabiliteit van een noot weer na uitvoer van het algoritme na een verschillend aantal iteraties.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging transformatie 1	5	-4	1	-3	1	1	2	3
Verhoging transformatie 2	1	3	4	-5	-1	6	5	-1
Verhoging transformatie 3	1	4	5	-3	2	-1	1	0
Verhoging transformatie 4	4	6	-2	4	2	6	-4	2
Verhoging transformatie 5	-3	-2	3	-1	4	-3	2	-4

TABEL 6.8: Transformaties gebruikt in het experiment van onderdeel 6.3.2.

6.3.2 Transformaties combineren: meerdere transformaties, 1 iteratie

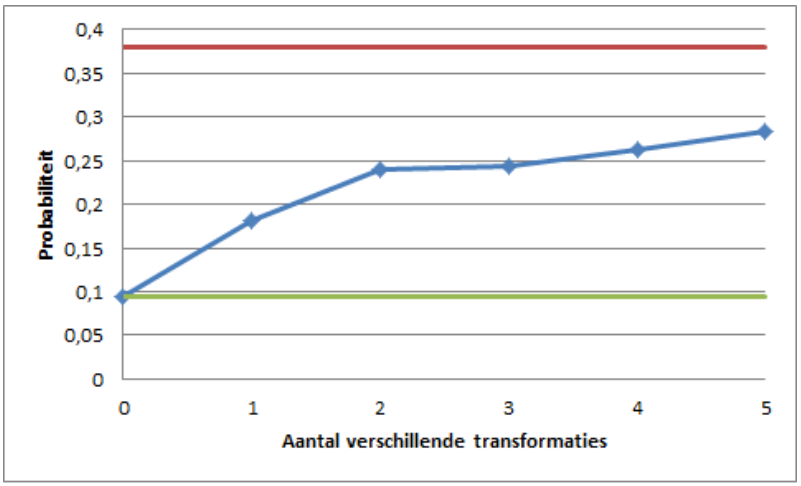
Beschrijving experiment

Dit experiment heeft betrekking tot het algoritme dat besproken werd in onderdeel 5.1. Het experiment gaat nagaan wat de invloed is van het aantal verschillende toegelaten transformaties op de consonantiescore van het totale muziekstuk. Zo zijn de vijf transformaties die gebruikt zullen worden weergegeven in tabel 6.8. Er wordt nu telkens slechts een iteratie van het algoritme uitgevoerd.

Deze test wordt uitgevoerd op 50 muziekstukken uit het Essencorpus. De gemiddelde probabiliteit van de originele stukken wordt berekend alsook de gemiddelde probabiliteit van het muziekstuk dat optimaal is volgens het RPK-model in de

toonaard van de 50 stukken. Nu kan er weer gekeken worden naar hoe snel de consonantiescore zich gaat verplaatsten van die van het originele naar die van de theoretisch best mogelijke volgens het model afhankelijk van het aantal transformaties dat aan het algoritme ter beschikking wordt gesteld. Het experiment wordt eerst uitgevoerd met slechts een mogelijke transformatie. Dit zal dan transformatie 1 uit de tabel zijn. Daarna wordt het experiment uitgevoerd met 2 mogelijke transformaties. Dit zullen dan de eerste twee transformaties uit de tabel zijn enz..

Resultaten



Figuur 6.5: Resultaten van het experiment uitgevoerd in deel 6.3.2. De groene lijn staat voor de gemiddelde probabiliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabiliteit van de theoritisch beste melodielijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabiliteit van een noot weer na uitvoer van het algoritme afhankelijk van het aantal transformaties dat ter beschikking gesteld was aan het algoritme.

In figuur 6.5 worden de restultaten weergegeven van dit experiment. Het is duidelijk dat een hoger aantal transformaties ook telkens een betere score teruggeeft. Als we de resultaten van dit experiment vergelijken met dat uit onderdeel 6.3.1, dan merken we op dat het aantal transformaties een grotere impact heeft op de probabiliteit dan het aantal iteraties. De grootste reden hiervoor is dat een extra transformaties er voor kan zorgen dat er voor elke een noot in het muziekstuk een extra mogelijke noot is waarnaar hij getransformeerd kan worden. Dit zorgt voor enorm veel extra mogelijkheden waardoor er hogere probabiliteiten kunnen bekomen worden dan in het geval waarbij het aantal iteraties verhoogd wordt in plaats van het aantal mogelijke transformaties.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging transformatie 1	5	-4	1	-3	1	1	2	3
Verhoging transformatie 2	1	3	4	-5	-1	6	5	-1

TABEL 6.9: Transformaties gebruikt in het experiment van onderdeel 6.3.3.

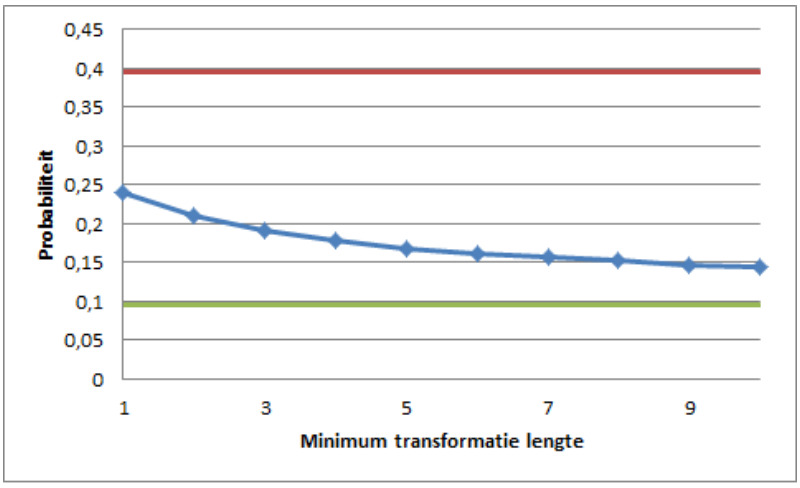
6.3.3 Transformaties combineren: minimum transformatie lengte

Beschrijving experiment

Dit experiment heeft betrekking tot het algoritme dat besproken werd in onderdeel 5.2. Het experiment gaat nagaan wat de invloed is van de minimum transformatie-lengte op de consonantiescore van het totale muziekstuk. Er wordt in dit experiment telkens gebruik gemaakt van 2 transformaties die beschreven zijn in tabel 6.9. Er wordt telkens slechts een iteratie van het algoritme uitgevoerd.

Dit experiment wordt uitgevoerd op 50 muziekstukken uit het Essencorpus en dit voor alle waarden van de minimum transformatie lengte tussen 1 en 10. Voor elk van deze 10 gevallen wordt de gemiddelde probabiliteit van het getransformeerde muziekstuk berekend.

Resultaten



FIGUUR 6.6: Resultaten van het experiment uitgevoerd in deel 6.3.3. De groene lijn staat voor de gemiddelde probabiliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabiliteit van de theoritisch beste melodielijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabiliteit van een noot weer na uitvoer van het algoritme afhankelijk van de minimum transformatie lengte.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	5	-4	1	-3	1	1	2	3

Tabel 6.10: Transformatie gebruikt in het experiment van onderdeel 6.3.4.

Figuur 6.6 geeft de resultaten weer van het experiment. Het is duidelijk dat een hogere minimum transformatie lengte leidt tot een kleinere verbetering van de consonantiescore, wat te verwachten was. De grafiek is ook monotoon dalen voor stijgende waarde van de minimum transformatie lengte. Dit moet ook zo zijn aangezien alle transformaties die geldig zijn voor een zekere transformatie lengte ook altijd geldig zijn voor alle kortere transformatie lengtes.

6.3.4 Transformaties combineren: Gelijkheid algoritmen voor transformatie lengte 1

Beschrijving experiment

Dit experiment is opgezet als extra test om het geloof in de juiste werking van de algoritmes beschreven in 5.1 en 5.2 te versterken. Aangezien de implementaties van deze twee algoritmen toch op een aantal vlakken (vooral de voorstelling van de paden) verschillen van elkaar, is het interessant om voor een minimum transformatie lengte van 1 eens te kijken of de twee algoritmes hetzelfde resultaat geven. Dit zou normaal gezien altijd het geval moeten zijn aangezien de twee algoritmes hetzelfde doel en dezelfde middelen hebben in het geval van een minimum transformatie lengte van 1. De transformatie waarvan gebruik gaat worden gemaakt staat beschreven in tabel 6.10.

Voor deze transformatie gaat het experiment zoals het beschreven staat in onderdel 6.3.1 herhaald worden maar dan ook voor het tweede algoritme. Dus voor een aantal iteraties van 1 tot en met 10 van het algoritme gaan de probabiliteiten die beide algoritmes opleveren voor dezelfde transformatie op dezelfde 100 testgevallen uit het Essencorpus vergeleken worden.

Resultaten

In tabel 6.11 staan de resultaten van dit experiment weergegeven. Beide algoritmen leveren dezelfde gemiddelde consonantiescore (logaritme van de gemiddelde probabiliteit) voor een muziekstukje na eenzelfde aantal iteraties gebruik makende van dezelfde transformatie. Dit versterkt de stelling dat de twee algoritmes wel degelijk werken zoals gewenst.

# iteraties	Algoritme 1	Algoritme 2
1	-1.69	-1.69
2	-1.57	-1.57
3	-1.54	-1.54
4	-1.51	-1.51
5	-1.50	-1.50
6	-1.50	-1.50
7	-1.50	-1.50
8	-1.50	-1.50
9	-1.50	-1.50
10	-1.50	-1.50

TABEL 6.11: Resulaten van experiment 6.3.4. Gemiddelde consonantiescores voor de twee algoritmen (logaritme van de probabiliteit) na een gegeven aantal iteraties. Algoritme 1 staat beschreven in 5.1, algoritme 2 is hetgene dat beschreven staat in 5.2.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging	1	1	2	3	5	-4	1	-3

TABEL 6.12: Transformatie gebruikt in het experiment van onderdeel 6.4.1.

6.4 Tijdscomplexiteit van de algoritmes voor het combineren van transformaties

6.4.1 Tijdsafhankelijkheid algoritme 5.2 van het aantal noten in de melodielijn

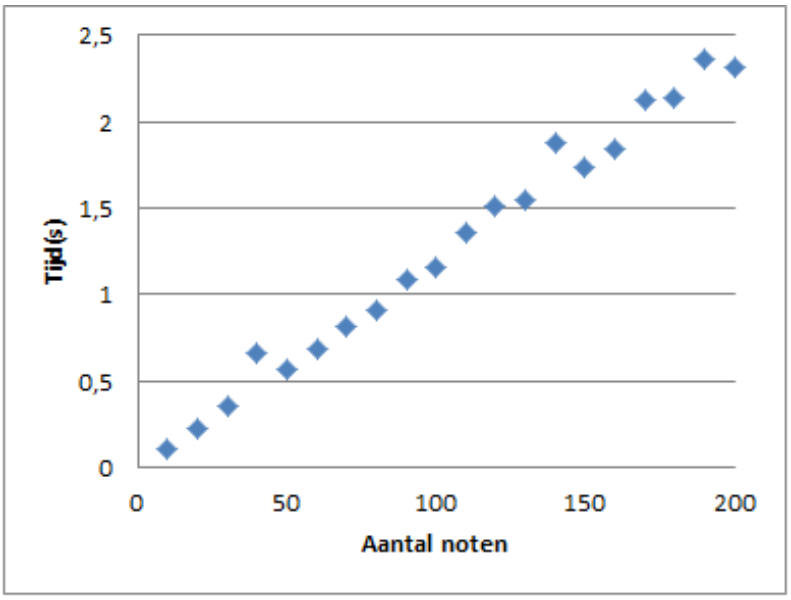
Beschrijving experiment

Van het algoritme beschreven in onderdeel 5.2 werd reeds beargumenteerd dat de uitvoertijd lineair afhankelijk is van het aantal noten van de originele melodielijn. De bedoeling is nu om experimenteel na te gaan of dit wel klopt. Om dit te testen wordt gebruik gemaakt van een transformatie die beschreven staat in tabel 6.12.

Voor 20 verschillende lengtes van melodielijnen (10 tot en met 200 met stapgrootte 10) wordt nu de uitvoertijd gemeten. Hierna kan de uitvoertijd van het algoritme uitgezet worden ten opzicht van de invoergrootte en zou er een lineair verband zichtbaar moeten zijn.

Resultaten

In figuur 6.7 worden de resultaten van dit experiment weergegeven in een scatter plot. Er is een duidelijk lineair verband zichtbaar tussen de grootte van de melodielijn en de uitvoertijd. Dit komt overeen met de afhankelijkheid die beredeneerd werd in hoofdstuk 5.2.



Figuur 6.7: Resultaten van het experiment uitgevoerd in deel 6.4.1. Tijdsduur van het algoritme in seconden t.o.v. het aantal noten in de originele melodie.

6.4.2 Tijdsafhankelijkheid algoritme 5.2 van aantal toegelaten transformaties

Beschrijving experiment

Dit experiment heeft betrekking op het algoritme beschreven in hoofdstuk 5.2. Er werd gesteld dat de uitvoertijd van dit algoritme evenredig is met het kwadraat van het aantal toegelaten transformaties. De bedoeling van dit experiment is om na te gaan of deze beredenering kan kloppen door voor verschillende aantallen transformaties te berekenen hoelang het algoritme nodig heeft om eenzelfde invoer te verwerken. De tien transformaties die gebruikt worden in dit experiment staan beschreven in tabel 6.13. Het maakt voor de snelheid va uitvoer van het algoritme geen verschil welke transformaties er meegegeven worden (10 dezelfde transformaties duren even lang om te verwerken dan 10 verschillende), maar voor de volledigheid zijn de transformaties hier toch weergegeven aangezien deze gebruikt werden tijdens het experiment.

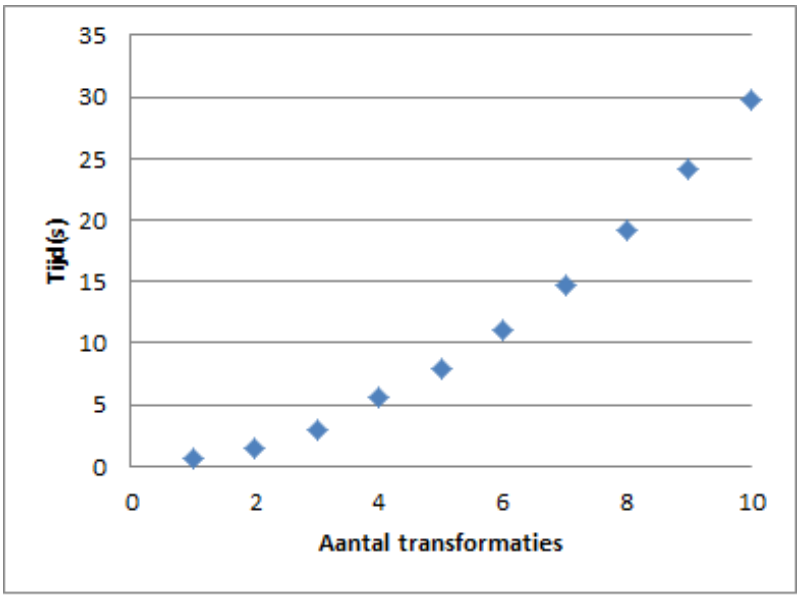
Voor eenzelfde melodielijn wordt 10 keer het algoritme uitgevoerd. De eerste keer heeft het algoritme enkel ‘transformatie’ 1 ter beschikking. De tweede keer heeft het algoritme zowel ‘transformatie 1’ als ‘transformatie 2’ ter beschikking, enz..

Resultaten

De resultaten van dit experiment staan weergegeven in een scatter plot in figuur 6.8. Op deze figuur lijkt een kwadratische afhankelijkheid tussen het aantal beschikbaren

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging transformatie 1	5	-4	1	-3	1	1	2	3
Verhoging transformatie 2	1	3	4	-5	-1	6	5	-1
Verhoging transformatie 3	1	4	5	-3	2	-1	1	0
Verhoging transformatie 4	4	6	-2	4	2	6	-4	2
Verhoging transformatie 5	-3	-2	3	-1	4	-3	2	-4
Verhoging transformatie 6	1	1	2	3	5	-4	1	-3
Verhoging transformatie 7	-1	6	5	-1	2	1	3	4
Verhoging transformatie 8	2	-1	1	0	3	1	4	5
Verhoging transformatie 9	2	6	-4	2	4	6	-2	4
Verhoging transformatie 10	4	-3	2	-4	-3	-2	3	-1

TABEL 6.13: Transformaties gebruikt in het experiment van onderdeel 6.4.2.



FIGUUR 6.8: Resultaten van het experiment uitgevoerd in deel 6.4.2. Tijdsduur van het algoritme in seconden t.o.v. het aantal transformaties dat het algoritme ter beschikking heeft.

transformaties en de uitvoertijd van het algoritme zichtbaar. Dit komt overeen met de beredeneerde afhankelijkheid van in hoofdstuk 5.2.

Diff (mod 8)	0	1	2	3	4	5	6	7
Verhoging transformatie 1	5	-4	1	-3	1	1	2	3
Verhoging transformatie 2	1	1	2	3	5	-4	1	-3

Tabel 6.14: Transformaties gebruikt in het experiment van onderdeel 6.4.3.

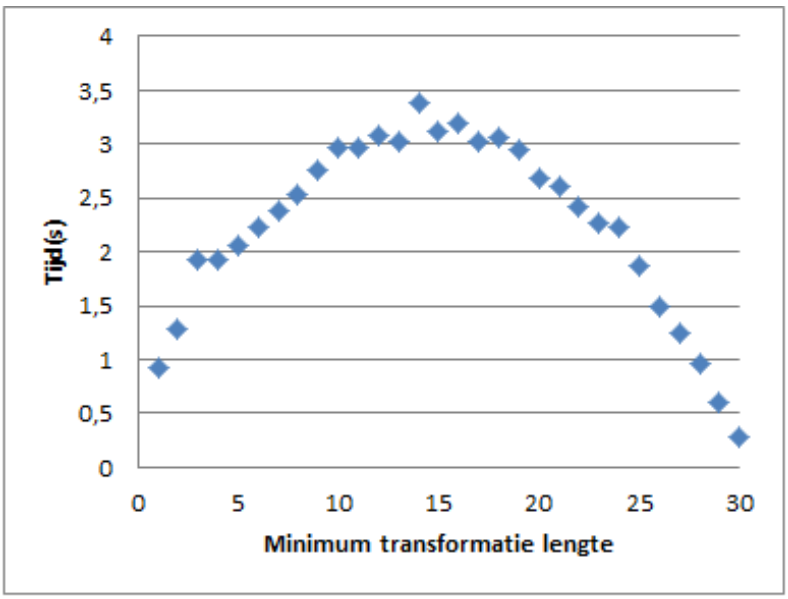
6.4.3 Tijdsafhankelijkheid algoritme 5.2 van minimum transformatie lengte

Beschrijving experiment

Van het algoritme beschreven in hoofdstuk 5.2 werd reeds beredeneerd dat de uitvoertijd van het algoritme afhankelijk is van $ML \times (AN - ML)$. Met ML, de minimum transformatie lengte en AN het aantal noten in de originele melodie. De bedoeling van dit experiment is om na te gaan of deze beredeneerde afhankelijkheid ook zichtbaar is in realiteit. Het algoritme kreeg tijdens dit experiment telkens twee transformaties ter beschikking, deze staan beschreven in tabel 6.14.

In dit experiment wordt telkens eenzelfde melodielijn met een lengte van 30 noten (dus AN=30) meegegeven aan het algoritme. En nu wordt voor 30 verschillende waarden van de minimum transformatie lengte (ML van 1 tot en met 30), het algoritme uitgevoerd.

Resultaten



Figuur 6.9: Resultaten van het experiment uitgevoerd in deel 6.4.3. Tijdsduur van het algoritme in seconden t.o.v. de opgelegde minimum transformatie lengte.

De resultaten van dit experiment zijn zichtbaar op de scatter plot van figuur 6.9. Dit experiment lijkt de stelling dat de uitvoertijd afhankelijk is van $ML \times (AN - ML)$ te staven.

Hoofdstuk 7

Besluit

7.1 Samenvatting

In deze masterproef zijn allereerst twee methoden besproken voor de evaluatie van muziekstukken. Het gebruik van een neuraal netwerk voor polyfone muziekstukken had zijn tekortkomingen door het negeren van de context van een noot. Dit probleem werd opgelost door het RPK-model dat geschikt is voor monofone muziekstukken. Dit RPK-model kon daarna gebruikt worden voor de melodische evaluatie van verschillende transformaties.
Er werden twee melodische transformaties besproken. Een van deze transformaties diende vooral als baseline en was zeer elementair en eenvoudig te implementeren. De andere transformatie dat zich baseert op de intervallen tussen opeenvolgende noten leverde betere resultaten (afwijking consonantiescore van de originele melodie ten opzichte van afwijking met een willekeurige transformatie is significant kleiner). De resultaten zijn, alhoewel ze duidelijk beter zijn dan in het willekeurige geval, wel nog steeds niet goed genoeg om te kunnen spreken van een goede transformatie.
Tot slot zijn er nog twee algoritmen ontworpen die zorgen voor het efficiënt combineren van verschillende transformaties. Het eerste van deze algoritmes kan gegeven een aantal transformaties en een originele melodielijn, deze melodielijn transformeren naar een nieuwe melodielijn met een zo hoog mogelijke consonantiescore. Het tweede algoritme verwezenlijkt hetzelfde doel, maar heeft als extra voorwaarde dat een transformatie, wanneer deze wordt uitgevoerd, meteen voor een minimum aantal opeenvolgende noten uitgevoerd moet worden. Van deze algoritmen werd de afhankelijkheid van de verschillende parameters getest en in kaart gebracht. Van het tweede algoritme werd ook de berekende tijdscomplexiteit geverifiëerd. Deze algoritmen kunnen dus gebruikt worden om zo efficiënt mogelijk (volgens het RPK-model) verschillende transformaties te combineren. Hierbij dient wel in het achterhoofd gehouden te worden dat het in deze algoritmen de bedoeling is de consonantiescore zo hoog mogelijk te krijgen. Dit leidt vaak niet tot interessante muziekstukken.

7.2 Verder onderzoek

In dit onderdeel worden nog enkele onderwerpen aangekaart die onderwerp zouden kunnen zijn voor verder onderzoek in dit onderzoeksdomein.

7.2.1 Uitbreiding context RPK-model

Het RPK-model brengt momenteel voor het berekenen van de probabiliteit van een muziekstuk voor elke noot, de voorafgaande noot in rekening. Het kan interessant zijn dit uit te breiden naar meerdere vorige noten aangezien deze, zij het in mindere mate, ook een invloed hebben op de waarschijnlijkheid van de huidige noot.
Momenteel wordt er ook enkel melodisch geëvalueerd, het ritme van het muziekstuk wordt namelijk genegeerd. Het zou ook interessant kunnen zijn, zelfs voor het louter evalueren van melodische transformaties, om ook het ritme in rekening te brengen. Men zou bijvoorbeeld kunnen verwachten dat hoe sneller de noten elkaar opvolgen hoe kleiner de gemiddelde sprongen in toonhoogte zullen zijn. Hierdoor zou de *proximity*-component mede afhankelijk van de ritmische afstand kunnen gemaakt worden in plaat van enkel van de melodische afstand.

7.2.2 Algoritme dat combineert met als doel het behouden van de consonantiescore

De algoritmen die ontworpen zijn in dit onderzoek hebben als doel om de consonantiescore van een melodielijn zo hoog mogelijk te krijgen door het combineren van verschillende transformaties. Dit levert echter geen interessante resultaten op in de praktijk. In het onderzoek is gesteld dat een transformatie als 'goed' beschouwd wordt wanneer deze de consonantiescore zou weining mogelijk zou veranderen. In dit opzicht is het interessant om een algoritme te maken dat als doel zou hebben om een gegeven melodielijn te transformeren, gebruik makende van de beschikbare transformaties, naar een nieuwe melodielijn wiens consonantiescore die van het origineel zo dicht mogelijk benadert. En als extra voorwaarde zou dan kunnen opgelegd worden dat het nieuwe muziekstukje op melodisch vlak zo veel mogelijk zou moeten afwijken van het origineel zodat dit niet meer herkenbaar zou zijn na transformatie.

Bijlagen

Bijlage A

Broncode

A.1 Neuraal netwerk trainer

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Random;
import java.util.Set;


public class Main {
        public static void main(String[] args) throws
           FileNotFoundException {
                new Main();
        }

        private static int NB_HIDDEN_NODES = 6;
        private static final double LEARNING_RATE = 0.35;
        private static double BIAS_1 = 0.0;
        private static double BIAS_2 = 0.0;

        private Random random;

        private double[][] weights1;
        private double[] weights2;

        public Main() throws FileNotFoundException {
                PrintWriter pw = new PrintWriter(new File("
                   weights.txt"));
                random = new Random();
```

```
            initializeWeights();
            Set<Sample> trainingData = getTrainingData()
                ;
            System.out.println("SIZE TRAININGDATA: " +
                trainingData.size());
            int count = 0;
            while(count++ < 100000) {
                    trainNetwork(trainingData);
                    if (count%10000 == 0) System.out.
                        println((count/10000) + " " + (
                        getSquaredError(trainingData)/
                        trainingData.size()));
            }
            printDiffs(trainingData);
            saveWeights(pw);
            pw.close();
    }


    private void saveWeights(PrintWriter pw) {
            pw.println("WEIGHTS LAYER 1:");
            for (int j = 0; j < NB_HIDDEN_NODES; j++) {
                    String line = "";
                    for (int i = 0; i < 12; i++) {
                            if (i != 0) line += " ";
                            line += (String.format("%.2f
                                ", weights1[i][j]));
                    }
                    pw.println(line);
            }

            pw.println();

            pw.println("WEIGHTS LAYER 2:");
            for (int j = 0; j < NB_HIDDEN_NODES; j++) {
                    pw.println(String.format("%.2f",
                        weights2[j]));
            }
    }


    private void printDiffs(Set<Sample> testData) {
            for (Sample sample: testData) {
                    System.out.println(sample.output + "
                        " + getNetworkValue(sample));
```

```
        }
    }


    private double getNetworkValue(Sample sample) {
        double[] input = sample.getInput();

        //calculation of result
        double[] outputsHiddenLayer = new double[
            NB_HIDDEN_NODES];
        for (int j = 0; j < NB_HIDDEN_NODES; j++) {
                double hiddenSum = BIAS_1;
                for (int i = 0; i < 12; i++) {
                        hiddenSum += input[i]*
                            weights1[i][j];
                }
                outputsHiddenLayer[j] = logicalValue
                    (hiddenSum);
        }
        double endSum = BIAS_2;
        for (int j = 0; j < NB_HIDDEN_NODES; j++) {
                endSum += outputsHiddenLayer[j]*
                    weights2[j];
        }
        double result = logicalValue(endSum);
        return result;
    }


    private double getSquaredError(Set<Sample> testData)
        {
        double squaredError = 0.0;
        for (Sample sample: testData) {
                double[] input = sample.getInput();
                double output = sample.getOutput();
                double[] outputsHiddenLayer = new
                    double[NB_HIDDEN_NODES];
                for (int j = 0; j < NB_HIDDEN_NODES;
                    j++) {
                        double hiddenSum = BIAS_1;
                        for (int i = 0; i < 12; i++)
                            {
                                hiddenSum += input[i
                                    ]*weights1[i][j];
                        }
```

```
                        outputsHiddenLayer[j] =
                            logicalValue(hiddenSum);
                    }
                    double endSum = BIAS_2;
                    for (int j = 0; j < NB_HIDDEN_NODES;
                        j++) {
                            endSum += outputsHiddenLayer
                                [j]*weights2[j];
                    }
                    double result = logicalValue(endSum)
                        ;
                    double error = Math.abs(output-
                        result);
                    squaredError += error*error;
            }
            return squaredError;
    }


    private void trainNetwork(Set<Sample> trainingData)
        {
            for (Sample sample: trainingData) {
                    double[] input = sample.getInput();
                    double output = sample.getOutput();

                    //calculation of result
                    double[] outputsHiddenLayer = new
                        double[NB_HIDDEN_NODES];
                    for (int j = 0; j < NB_HIDDEN_NODES;
                        j++) {
                            double hiddenSum = BIAS_1;
                            for (int i = 0; i < 12; i++)
                                {
                                    hiddenSum += input[i
                                        ]*weights1[i][j];
                            }
                            outputsHiddenLayer[j] =
                                logicalValue(hiddenSum);
                    }
                    double endSum = BIAS_2;
                    for (int j = 0; j < NB_HIDDEN_NODES;
                        j++) {
                            endSum += outputsHiddenLayer
                                [j]*weights2[j];
                    }
```

```
                    double result = logicalValue(endSum)
                        ;
                    double error = output-result;

                    double[] oldWeights2 = weights2.
                        clone();
                    //backpropagation of error
                    for (int j = 0; j < NB_HIDDEN_NODES;
                        j++) {
                        double diff = LEARNING_RATE
                            * error  * result * (1 -
                            result) *
                            outputsHiddenLayer[j];
                        weights2[j] += diff;
                    }
                    for (int j = 0; j < NB_HIDDEN_NODES;
                        j++) {
                        for (int i = 0; i < 12; i++)
                            {
                            double diff =
                                LEARNING_RATE *
                                error *
                                outputsHiddenLayer
                                [j] * (1 -
                                outputsHiddenLayer
                                [j]) * input[i] *
                                 oldWeights2[j];
                            weights1[i][j] +=
                                diff;
                        }
                    }
                }
            }


            private double logicalValue(double d) {
                return (1 / (1 + Math.exp(-1 * d)));
            }


            private Set<Sample> getTrainingData() {
                Set<Sample> data = new HashSet<Sample>();
                boolean[] goodDistance = new boolean[]{true,
                    false, false, true, true, true, false,
                    true, true, true, false, false};
```

```
                for (int i = 0; i < 12; i++) {
                        for (int j = i; j < 12; j++) {
                                double[] sampleInput = new
                                    double[12];
                                Arrays.fill(sampleInput,
                                    0.0);
                                sampleInput[i] = 1.0;
                                sampleInput[j] = 1.0;
                                double sampleOutput;
                                if (goodDistance[j-i]) {
                                        sampleOutput = 0.99;
                                } else {
                                        sampleOutput = 0.01;
                                }
                                Sample sample = new Sample(
                                    sampleInput, sampleOutput
                                    );
                                data.add(sample);
                        }
                }
                return data;
        }


        private class Sample{
                private double[] input;
                private double output;
                public Sample(double[] input, double output)
                    {
                        this.input = input;
                        this.output = output;
                }

                public double[] getInput() {
                        return input;
                }

                public double getOutput() {
                        return output;
                }
        }


        private void initializeWeights() {
                weights1 = new double[12][NB_HIDDEN_NODES];
```

```
                for (int i = 0; i < 12; i++) {
                        for (int j = 0; j < NB_HIDDEN_NODES;
                            j++) {
                                weights1[i][j] = random.
                                    nextDouble();
                        }
                }

                weights2 = new double[NB_HIDDEN_NODES];
                for (int i = 0; i < NB_HIDDEN_NODES; i++) {
                        weights2[i] = random.nextDouble();
                }
        }
}
```

A.2 RPK-model

```
__author__ = 'Elias'
import math

#probabilities of note functions in major and minor keys
key_major_prob = [0.184, 0.001, 0.155, 0.003, 0.191, 0.109,
    0.005, 0.214, 0.001, 0.078, 0.004, 0.055]
key_minor_prob = [0.192, 0.005, 0.149, 0.179, 0.002, 0.144,
    0.002, 0.201, 0.038, 0.012, 0.053, 0.022]

#which key, given amount of sharps +7
key_major = [11, 6, 1, 8, 3 ,10 ,5 ,0, 7, 2, 9, 4, 11, 6, 1]
key_minor = [8, 3, 10, 5, 0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10]

#calculat the sum of the probs in a given score
def score_val(score):
    flatscore = score.flat
    keySign = flatscore.getElementsByClass('KeySignature')
    keyIndex = keySign[0].sharps + 7
    #print('Sharps: ' + str(keyIndex))
    notes = flatscore.pitches
    int_notes = [0 for x in range(len(notes))]
    for i in range(len(notes)):
        int_notes[i] = int(pitch_number(notes[i].
            nameWithOctave))
    return calculate_probability(int_notes, keyIndex)
```

```
#calculate the sum of the probs of a series of notes in a
    given key
def calculate_probability(notes, keyIndex):
    key_maj = key_major[keyIndex]
    major_prob = calculate_prob(notes, key_maj, True)
    key_min = key_minor[keyIndex]
    minor_prob = calculate_prob(notes, key_min, False)
    return math.log(0.88*math.exp(major_prob) + 0.12*math.
        exp(minor_prob))

#calculate the sum of the probs of a series of notes in a
    given key and whether it is major or not
def calculate_prob(notes, key, major):
    p = normalized_prob_log_first(notes[0], key, major)
    for i in range(1, len(notes)):
        note = notes[i]
        p += normalized_prob_log(note, key, major,notes[i
            -1])
    return p

#calculate the log of the normalized probability of a the
    first note in a given key (no previous note)
def normalized_prob_log_first(note, key, major):
    prob_notes = [0 for x in range(60)]
    total = 0
    for i in range(0, 60):
        p = key_prob(i+38, key, major)*range_prob(i+38)
        prob_notes[i] = p
        total += p
    return math.log(prob_notes[note-38]/total)

#calculate the log of the normalized probability of a given
    note in a given key, and a previous note
def normalized_prob_log(note, key, major, previous):
    prob_notes = [0 for x in range(60)]
    total = 0
    for i in range(0, 60):
        p = key_prob(i+38, key, major)*range_prob(i+38)*
            proximity_prob(i+38, previous)
        prob_notes[i] = p
        total += p
    return math.log(prob_notes[note-38]/total)

#calculate the log of the proximity score of a given current
    note and previous note
```

```
def proximity_prob_log(note, previous):
    return math.log(proximity_prob(note, previous))

#calculate the proximity score of a given current note and
    previous note
def proximity_prob(note, previous):
    return normal_prob(previous, 7.2, note)

#calculate the probability of a given note in a given key
def note_prob_in_key(note, keyIndex):
    key_maj = key_major[keyIndex]
    major_prob = key_prob(note, key_maj, True)
    key_min = key_minor[keyIndex]
    minor_prob = key_prob(note, key_min, False)
    return (0.88*major_prob + 0.12*minor_prob)

#calculate the log of the probability of a given note, key
    and boolean to  indicate whether the scale is major or
    minor
def key_prob_log(note, key, major):
    return math.log(key_prob(note, key, major))

#calculate the key probability of a given note, key and
    boolean to indicate whether the scale is major or minor
def key_prob(note, key, major):
    index = (((note - key + 12) % 12) + 12) % 12
    if (major == True):
        return key_major_prob[index]
    else:
        return key_minor_prob[index]

#calculate the log of the prob of a given note in integer
    representation
def range_prob_log(note):
    return math.log(range_prob(note))

#calculate the range probability of a given note in integer
    representation
def range_prob(note):
    p = 0
    for x in range(38, 98):
        p += normal_prob(68, 13.2, x)*normal_prob(x, 29,
            note)
    return p
```

```
#mean mu, variance sigma_sq, value x
def normal_prob (mu, sigma_sq, x):
    return (1/(math.sqrt(sigma_sq*2*math.pi)))*math.exp(-(x-
        mu)*(x-mu)/(2*sigma_sq))

#integer value of given note in note+octave representation
def pitch_number(pitch):
    octave = pitch[len(pitch)-1:]
    note = pitch[:len(pitch)-1]
    val = (int(octave)+1)*12 + intValue(note)
    return str(val)

#integer values (modulo 12 )of all possible notes
def intValue(name):
    if (name == 'C'):
        return 0
    elif (name == 'C#'):
        return 1
    elif (name == 'D-'):
        return 1
    elif (name == 'D'):
        return 2
    elif (name == 'D#'):
        return 3
    elif (name == 'E-'):
        return 3
    elif (name == 'E'):
        return 4
    elif (name == 'E#'):
        return 5
    elif (name == 'F-'):
        return 4
    elif (name == 'F'):
        return 5
    elif (name == 'F#'):
        return 6
    elif (name == 'G-'):
        return 6
    elif (name == 'G'):
        return 7
    elif (name == 'G#'):
        return 8
    elif (name == 'A-'):
        return 8
    elif (name == 'A'):
```

```
        return 9
    elif (name == 'A#'):
         return 10
    elif (name == 'B-'):
         return 10
    elif (name == 'B'):
        return 11
    elif (name == 'B#'):
        return 0
    elif (name == 'C-'):
        return 11
    else:
        return -1
```

A.3 Beste sequentie

```
__author__ = 'Elias'
import RPK
import math

zero = [0, 0, 0, 0, 0, 0, 0, 0]
transformations = [[1, 1, 2, 3, 5, -4, 1, -3],[5, -4, 1, -3,
    1, 1, 2, 3]]

def bestTransformed(int_notes, keyIndex):
    #transformation from -6 to +7
    past = [-100, -100, -100, -100, -100, -100, 0, -100,
       -100, -100, -100, -100, -100, -100]
    current = [-100, -100, -100, -100, -100, -100, -100,
       -100, -100, -100, -100, -100, -100, -100]
    matrix = [[0 for x in range(len(past))] for y in range(
       len(int_notes))]
    for x in range(len(int_notes)-1):
        present_note = int_notes[x+1]

        #case of keeping original
        for y in range(len(past)):
            last_note = int_notes[x]+y-6
            diff = abs(present_note-last_note)
            new_note = present_note+(zero[(diff+8)%8])
            if current[new_note-present_note+6] < past[y]+
               RPK.proximity_prob_log(new_note, last_note):
```

```
                current[new_note-present_note+6] = past[y]+
                    RPK.proximity_prob_log(new_note,
                    last_note)
                matrix[x][new_note-present_note+6] = y

        #case of transform
        for t in range(len(transformations)):
            for y in range(len(past)):
                last_note = int_notes[x]+y-6
                diff = abs(present_note-last_note)
                new_note = -1;
                if (present_note-last_note <= 0):
                    new_note = present_note+(transformations
                        [t][(diff+8)%8])
                else:
                    new_note = present_note-(transformations
                        [t][(diff+8)%8])
                if RPK.note_prob_in_key(new_note, keyIndex)
                    < 0.02:
                    if (RPK.note_prob_in_key(new_note+1,
                        keyIndex) > RPK.note_prob_in_key(
                        new_note-1, keyIndex)):
                        new_note = new_note+1
                    else:
                        new_note = new_note-1
                if current[new_note-present_note+6] < past[y
                    ]+RPK.proximity_prob_log(new_note,
                    last_note):
                    current[new_note-present_note+6] = past[
                        y]+RPK.proximity_prob_log(new_note,
                        last_note)
                    matrix[x][new_note-present_note+6] = y

        #reinitialize arrays
        for y in range(len(past)):
            past[y] = current[y]+RPK.range_prob_log(
                present_note+y-6)+math.log(RPK.
                note_prob_in_key(present_note+y-6, keyIndex))
            current[y] = -10000000;

    #look at which endpoint had the most probable path and
        reconstruct that most probable path
    maxIndex = 0
    for x in range(len(past)):
        if (past[x] > past[maxIndex]):
```

```
            maxIndex = x
    reversed_list = [0 for x in range(len(int_notes))]
    reversed_list[0] = maxIndex
    for x in range(len(int_notes)-1):
        reversed_list[x+1] = matrix[len(int_notes)-x-2][
            maxIndex]
        maxIndex = reversed_list[x+1]
    ordered_list = [0 for x in range(len(int_notes))]
    for x in range(len(int_notes)):
        ordered_list[x] = reversed_list[len(int_notes)-x-1]
    new_int_notes = [0 for x in range(len(int_notes))]
    for x in range(len(int_notes)):
        new_int_notes[x] = int_notes[x]+ordered_list[x]-6
    return new_int_notes
```

A.4 Beste sequentie met minimum transformatie lengte

```
__author__ = 'Elias'
import RPK
import math

zero = [0, 0, 0, 0, 0, 0, 0, 0]
transformations = [[1, 1, 2, 3, 5, -4, 1, -3], [5, -4, 1,
    -3, 1, 1, 2, 3]]

#min amount of concatenated notes that need to be
    transformed.
min_length = 4
height = 14

def bestTransformed(int_notes, keyIndex):
    #transformation from -6 to +7
    #add new history to back of the array
    prob_keep_past = [[-100000, -100000, -100000, -100000,
        -100000, -100000, 0, -100000, -100000, -100000,
        -100000, -100000, -100000, -100000] for x in range(
        min_length)]
    prob_keep_current = [-100000 for y in range(height)]
    prob_transform_past = [[[-100000 for y in range(height)]
         for x in range(min_length)] for z in range(len(
        transformations))]
```

```
    prob_transform_current = [[-100000 for y in range(height
        )] for z in range(len(transformations))]

    path_end_on_keep_past = [[[] for x in range(height)] for
         y in range(min_length)]
    path_end_on_keep_current = [[] for x in range(height)]
    path_end_on_transform_past = [[[[] for x in range(height
        )] for y in range(min_length)] for z in range(len(
        transformations))]
    path_end_on_transform_current = [[[] for x in range(
        height)] for z in range(len(transformations))]

    for x in range(len(int_notes)-1):
         present_note = int_notes[x+1]

         #case of keeping original
         for y in range(height):
              last_note = int_notes[x]+y-6
              diff = abs(present_note-last_note)
              new_note = present_note+(zero[(diff+8)%8])
              #extending one that already ended on a keep
              if prob_keep_current[new_note-present_note+6] <
                  prob_keep_past[min_length-1][y]+RPK.
                  proximity_prob_log(new_note, last_note):
                   prob_keep_current[new_note-present_note+6] =
                        prob_keep_past[min_length-1][y]+RPK.
                       proximity_prob_log(new_note, last_note)
                   copy = list(path_end_on_keep_past[min_length
                       -1][y])
                   copy.append(y)
                   path_end_on_keep_current[new_note-
                       present_note+6] = copy
              #extending one that ended on a transform
              for z in range(len(transformations)):
                   if prob_keep_current[new_note-present_note
                       +6] < prob_transform_past[z][min_length
                       -1][y]+RPK.proximity_prob_log(new_note,
                       last_note):
                        prob_keep_current[new_note-present_note
                            +6] = prob_transform_past[z][
                            min_length-1][y]+RPK.
                            proximity_prob_log(new_note,
                            last_note)
                        copy = list(path_end_on_transform_past[z
                            ][min_length-1][y])
```

```
                copy.append(y)
                path_end_on_keep_current[new_note-
                    present_note+6] = copy

        #case of transform
        if x >= (min_length-1):
            for z in range(len(transformations)):
                #extending one that already ended on the
                    same transform
                for y in range(height):
                    last_note = int_notes[x]+y-6
                    diff = abs(present_note-last_note)
                    new_note = -1
                    if (present_note-last_note <= 0):
                        new_note = present_note+(
                            transformations[z][(diff+8)%8])
                    else:
                        new_note = present_note-(
                            transformations[z][(diff+8)%8])
                    if RPK.note_prob_in_key(new_note,
                        keyIndex) < 0.02:
                        if (RPK.note_prob_in_key(new_note+1,
                            keyIndex) > RPK.note_prob_in_key
                            (new_note-1, keyIndex)):
                            new_note = new_note+1
                        else:
                            new_note = new_note-1
                    if prob_transform_current[z][new_note-
                        present_note+6] < prob_transform_past
                        [z][min_length-1][y]+RPK.
                        proximity_prob_log(new_note,
                        last_note):
                        prob_transform_current[z][new_note-
                            present_note+6] =
                            prob_transform_past[z][min_length
                            -1][y]+RPK.proximity_prob_log(
                            new_note, last_note)
                        copy = list(
                            path_end_on_transform_past[z][
                            min_length-1][y])
                        copy.append(y)
                        path_end_on_transform_current[z][
                            new_note-present_note+6] = copy
```

```
                #extending one that ended on another
                    transform
                for q in range(len(transformations)):
                    if q==z:
                        continue
                    start_probs = list(prob_transform_past[q
                        ][0])
                    past_paths = [list(
                        path_end_on_transform_past[q][0][x])
                        for x in range(height)]
                    result = simulate_best_paths(start_probs
                        , transformations[z], min_length,
                        past_paths, x-(min_length-1),
                        keyIndex, int_notes)
                    end_probs = result[0]
                    end_paths = result[1]
                    for y in range(height):
                        if prob_transform_current[z][y] <
                            end_probs[y]:
                            prob_transform_current[z][y] =
                                end_probs[y]
                            path_end_on_transform_current[z
                                ][y] = list(end_paths[y])

                #extending one that ended on keep
                start_probs = list(prob_keep_past[0])
                past_paths = [list(path_end_on_keep_past[0][
                    x]) for x in range(height)]
                result = simulate_best_paths(start_probs,
                    transformations[z], min_length,
                    past_paths, x-(min_length-1), keyIndex,
                    int_notes)
                end_probs = result[0]
                end_paths = result[1]
                for y in range(height):
                    if prob_transform_current[z][y] <
                        end_probs[y]:
                        prob_transform_current[z][y] =
                            end_probs[y]
                        path_end_on_transform_current[z][y]
                            = list(end_paths[y])

        #reinitialize arrays
        for y in range(height):
            for q in range(min_length-1):
```

```
                prob_keep_past[q][y] = prob_keep_past[q+1][y
                    ]
                path_end_on_keep_past[q][y] =
                    path_end_on_keep_past[q+1][y]
                for z in range(len(transformations)):
                    prob_transform_past[z][q][y] =
                        prob_transform_past[z][q+1][y]
                    path_end_on_transform_past[z][q][y] =
                        path_end_on_transform_past[z][q+1][y]
            prob_keep_past[min_length-1][y] =
                prob_keep_current[y]+RPK.range_prob_log(
                present_note+y-6)+math.log(RPK.
                note_prob_in_key(present_note+y-6, keyIndex))
            prob_keep_current[y] = -100000
            for z in range(len(transformations)):
                prob_transform_past[z][min_length-1][y] =
                    prob_transform_current[z][y]+RPK.
                    range_prob_log(present_note+y-6)+math.log
                    (RPK.note_prob_in_key(present_note+y-6,
                    keyIndex))
                prob_transform_current[z][y] = -100000
            path_end_on_keep_past[min_length-1][y] =
                path_end_on_keep_current[y]
            path_end_on_keep_current[y] = []
            for z in range(len(transformations)):
                path_end_on_transform_past[z][min_length-1][
                    y] = path_end_on_transform_current[z][y]
                path_end_on_transform_current[z][y] = []

    #Return path with max probability
    maxIndexKeep = 0
    for x in range(height):
        if (prob_keep_past[min_length-1][x] > prob_keep_past
            [min_length-1][maxIndexKeep]):
            maxIndexKeep = x

    maxIndexTransform = 0
    maxIndexTypeTransform = 0
    for z in range(len(transformations)):
        for x in range(height):
            if (prob_transform_past[z][min_length-1][x] >
                prob_transform_past[maxIndexTypeTransform][
                min_length-1][maxIndexTransform]):
                maxIndexTransform = x
                maxIndexTypeTransform = z
```

```
    best_path = []
    if (prob_keep_past[min_length-1][maxIndexKeep] >
        prob_transform_past[maxIndexTypeTransform][min_length
        -1][maxIndexTransform]):
            best_path = list(path_end_on_keep_past[min_length
                -1][maxIndexKeep])
            best_path.append(maxIndexKeep)
    else:
            best_path = list(path_end_on_transform_past[
                maxIndexTypeTransform][min_length-1][
                maxIndexTransform])
            best_path.append(maxIndexTransform)

    new_int_notes = [0 for x in range(len(int_notes))]
    for x in range(len(int_notes)):
        new_int_notes[x] = int_notes[x]+best_path[x]-6
    return new_int_notes

    reversed_list = [0 for x in range(len(int_notes))]
    reversed_list[0] = maxIndex
    for x in range(len(int_notes)-1):
        reversed_list[x+1] = matrix[len(int_notes)-x-2][
            maxIndex]
        maxIndex = reversed_list[x+1]
    ordered_list = [0 for x in range(len(int_notes))]
    for x in range(len(int_notes)):
        ordered_list[x] = reversed_list[len(int_notes)-x-1]
    new_int_notes = [0 for x in range(len(int_notes))]
    for x in range(len(int_notes)):
        new_int_notes[x] = int_notes[x]+ordered_list[x]-6
    return new_int_notes

#return the best paths of given length, only using the
    tranform given by the array parameter
def simulate_best_paths(start_probs, transform_array, length
    , start_paths, start_note, keyIndex, int_notes):
    past_probs = list(start_probs)
    current_probs = [-100000 for x in range(height)]
    past_paths = [list(start_paths[x]) for x in range(height
        )]
    current_paths = [[] for x in range(height)]

    for x in range(length):
        present_note = int_notes[start_note+x+1]
```

```
        for y in range(height):
            last_note = int_notes[start_note+x]+y-6
            diff = abs(present_note-last_note)
            new_note = -1
            if (present_note-last_note <= 0):
                new_note = present_note+(transform_array[(
                    diff+8)%8])
            else:
                new_note = present_note-(transform_array[(
                    diff+8)%8])
            if RPK.note_prob_in_key(new_note, keyIndex) <
                0.02:
                if (RPK.note_prob_in_key(new_note+1,
                    keyIndex) > RPK.note_prob_in_key(new_note
                    -1, keyIndex)):
                    new_note = new_note+1
                else:
                    new_note = new_note-1
            #replace if better
            if current_probs[new_note-present_note+6] <
                past_probs[y]+RPK.proximity_prob_log(new_note
                , last_note):
                current_probs[new_note-present_note+6] =
                    past_probs[y]+RPK.proximity_prob_log(
                    new_note, last_note)
                copy = list(past_paths[y])
                copy.append(y)
                current_paths[new_note-present_note+6] =
                    copy

        for y in range(height):
            past_probs[y] = current_probs[y]+RPK.
                range_prob_log(present_note+y-6)+math.log(RPK
                .note_prob_in_key(present_note+y-6, keyIndex)
                )
            current_probs[y] = -100000
            past_paths[y] = current_paths[y]
            current_paths[y] = []

    last_note = int_notes[start_note+length]
    for y in range(height):
        past_probs[y] = past_probs[y]-RPK.range_prob_log(
            last_note+y-6)-math.log(RPK.note_prob_in_key(
            last_note+y-6, keyIndex))
```

```
    return (past_probs, past_paths)
```

Bijlage B

IEEE_Paper

Melodical Transformation and Evaluation of Music

Elias Moons
KU Leuven, 2016, Leuven, Belgium

This paper describes a method to melodically transform given musical pieces to new ones. There will also be presented a framework on how to evaluate these transformations and musical pieces in general. This framework will be used to calculate the efficiency of the transformations. At last an algorithm is presented that will efficiently combine different melodical transformations to acquire a new musical piece that is optimal under the proposed framework. The algorithm can only make use of a given set of transformations to perform this task.

1 Introduction

One of the most common problems for musicians these days is the *writer's block*. This phenomenon, where a musician momentarily lacks the inspiration to come with ideas for new melodies can be very frustrating. Therefore, it would be very useful to have a tool that gives the songwriter the inspiration that he/she needs. One possible solution to this problem will be explained in this paper. This solutions consists of transforming an original melody into a new one, using transformations.

A lot of research has already been done in the field of musical generation [1]. But the results of these experiments often led to musical pieces that were either musically correct but often less interesting to listen to (often had an 'artificial' sound), or less musically correct but too frustrating. Musical transformation might give an interesting out in this respect. Transforming an original melody, of which it is assumed to be interesting, will hopefully keep this interestingness after transformation. This paper discusses a particular set of transformations, called the melodical transformations. These transformations completely ignore the ryhtmical part of a musical piece and focus on the melody. These transformations will only have an effect on the pitch of the notes in the original melody. The counterpart of melodical transformations is rhythmical transformations. There has already been done research in this domain, with promising results which led to the belief that melodical transformations might work as well [2].

There will also be proposed a framework to judge these transformations and melodies in general. A good framework is necessary to be able to adequately judge these melodical transformations. At last an algorithm will be described which will combine different given transformations to transform a certain melody as efficiently as possible into a new one. This new melody should have the highest possible probability in the proposed framework.

2 Scope

Since research on the topic of musical transformation is still in its early days, it is too difficult to immediately start dealing with all kinds of problems at once. This paper therefore describes transformations only to transform melodies (only one note played at the same time), instead of full polyphonic musical pieces. Also the proposed framework will be built based on the assumption of single-voice problems. The transformation of polyphonic musical pieces is a lot more complex and is subject to further research.

The transformation, reference model and algorithm that will be described in this paper will be tested and based on the Essencorpus [3]. This corpus contains Folk songs in the MusicXML [4] format which makes it easy to use an internal representation of the music based on music theory with notes (instead of a physical one based on frequencies). It is also important to note that the framework that will be presented to assess the quality of musical pieces, will only consider the melody of a musical piece and ignore the rhythmical information in the piece. The reason for this is because the transformation and algorithm that are described also only reason in terms of the melody.

3 Framework for evaluating melodies

In this section, a framework is described that will be used to evaluate transformations later on. There are lots of important parameters that define the quality of a transformation. The framework that is proposed combines three of these important parameters.

3.1 Idea of the RPK-model

The RPK-model is a framework of which the ideas are described in [5]. This model is based on three important pa-

rameters, which are also the basis for the name of the model. For each of the notes in a melody, its probability is modelled as the product of these three parameters.

There is the *range*-parameter, which models the distance of a note to the average pitch in the corpus. Notes that have a closer distance to the average pitch, have a higher probability of occurrence. The distribution of notes in the Essencorpus is given in figure 1 to illustrate this.

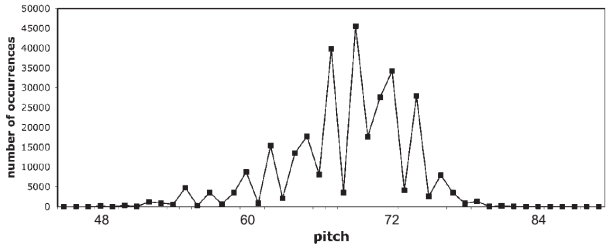


Fig. 1. Distribution of all notes in the Essencorpus. Middle C has value 60, the unit on the x-axis is a semitone.

The second parameter is the *proximity*. This models the distance of a note to the note that is played previous to this note. This is an important parameter because the context of a note (notes played in the immediate neighbourhood of that note) have an influence on the probability of occurrence of a certain note. Notes that differ less in pitch are for example more common to succeed each other than notes that have a bigger difference in pitch. In figure 2, the occurrences of all intervals between consecutive notes in the Essencorpus is illustrated.

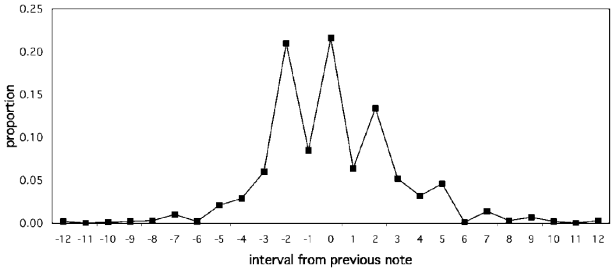


Fig. 2. Proportions of occurrence of all intervals between neighbouring notes in the Essencorpus. Difference expressed in amount of semitones.

At last, the *key*-parameter is introduced. This parameter models the probability of occurrence of a certain note in the key of the musical piece. This is crucial because there is a big difference in occurrence for different notes in the key. There is also a difference in the key-profile for major and minor scales. This becomes clear in figures 3 and 4 for major and minor scales respectively. Here the distribution of notes for all musical pieces in the Essencorpus is displayed.

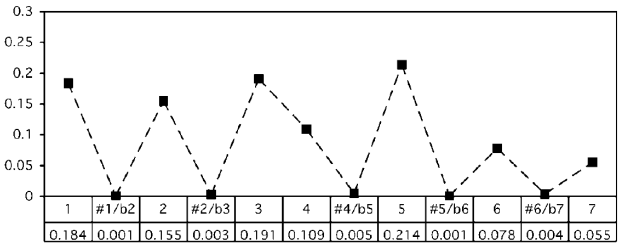


Fig. 3. Average distribution of notes in the Essencorpus for pieces that are in a major key.

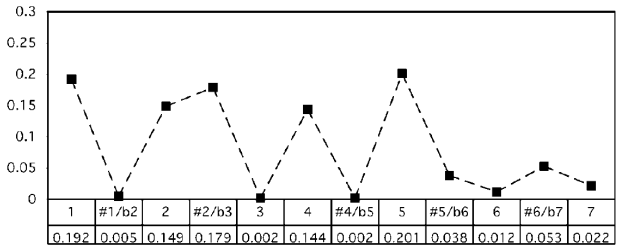


Fig. 4. Average distribution of notes in the Essencorpus for pieces that are in a minor key.

3.2 Using the RPK-model

We can model the probability of a note in a musical piece. Now we can define the probability of a given melody as the mean probability of all notes in that melody. By calculating the probability of a melody before and after a certain transformation, the quality of a given transformation can be assessed.

It is important to note that a transformation that gives better scores on average, is not necessarily a better transformation. The reasoning behind this is that musical pieces that score extremely high on the RPK-model are very boring pieces of music that have a lot of occurrences of the same note that lies close to the central pitch. Musical pieces with very low scores are on average frustrating to listen to, because they will probably consist of a lot of big 'jumps' in the melody.

The solution to this problem is to consider the score of an original melody as a sort of golden number for a melody on this rhythm (since the rhythm of a musical piece will not change under the melodical transformations described in this paper). Now a given transformation will be judged 'better' than another one if, one average, it produces results that in score differ less from the original.

4 Melodical Transformation

The melodical transformation that will be discussed in this paper is one that is dependent on the difference in semitones to the previous one. This means that only the difference in semitones to the previous note will decide which note a given note will transform to.

Diff (mod 8)	0	1	2	3	4	5	6	7
Addition	5	-4	1	-3	1	1	2	3

Table 1. Transformation dependent on the difference in semitones between the current and previous note.

4.1 Idea of the transformation

This difference in semitones (modulo 8) can be seen as an index of a function that gives back a number that represents that amount of semitones that will be added to the current note. An important aspect of the transformation is that this amount of semitones will be 'added' in the other direction of the distance to the previous note. This means that a positive value for addition always leads to a jump of pitch of the current note in the direction of the pitch of the precious note. A benefit of this directive jump is that differences in both directions are handled equally. A certain rising gap will lead to a lowering jump that is as big as a descending gap of equal size would generate in the opposite direction.

4.2 Example

As an example, table 1 is given. A difference in semitones of 6 downwards will for example lead to a rise in semitones of 2 of the current note using this transformation. In figure 5, an example is given of the application of this transformation on a given melody. It is also important to note that after applying the transformation on a note, if the resulting note is not part of the key of the piece (and only then), it will be transformed to its neighbour note (+1 of -1 semitone) which has the highest probability. This means that a jump of +4 semitones in the table can theoretically lead to a jump of +3, +4 or +5 semitones.

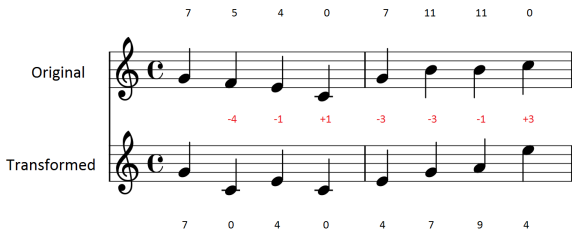


Fig. 5. Example of using transformation described in table 1.

4.3 Discussion

One of the benefits of this transformation that it is still a relatively simple one to use. The concept is easy and the applications doesn't require any complex or heavy computation. It is however, far from optimal since only a small part of the context of a given note (only the previous one) is taken into account. Results might be better if a bigger idea of context could be integrated into the transformation.

An interesting benefit of the transformation is that, although it does not explicitly searches for repetitiveness in the music, it almost always keeps it through the transformation. This means that the structure of the original musical piece is often kept, which can be interesting for giving the music a more natural feel that the listener is more used to.

5 Combining Transformations

In this section, an algorithm will be described that will combine different melodical transformations to transform an original melody into a new one. This new melody should have a probability, as judged by the RPK-model, that is as high as possible. To do this, for each note in an original melody, the algorithm has the choice between either keeping the note as it is, or alternatively to transform the note using one of the given transformations.

5.1 Different parameters

The algorithm is dependent on three different parameters. First, there is the input, the original melody on which the transformation should take place. The impact that the original melody has on the algorithm, performance wise, is firstly in its amount of notes (AN). Second, there are the different given transformations (or the amount of transformations (AN)) that the algorithm can use. The more different transformations, the more options the algorithm has to transform a given note. Lastly, there is also a 'minimum transformation length' (ML) parameter. This parameter states that if the algorithm wants to transform a certain part of the original melody using a certain transformation, it should do so for at least ML consecutive notes with the same transformation.

5.2 High level idea of the algorithm

The algorithm is based on the principles of *dynamic programming* [6]. The main idea of the algorithm is the following. Suppose we are at note n in the melody. This note on position n has a few different notes it can be transformed to (conform the given transformations). Of all the paths that end on such a note, we save the most probable one for each of those notes. There are only a few ways tot construct such a path.

First we can extend any optimal path that ends on note $n-1$ with just a 'non-transformation, this is always a possibility and will not violate any constraint. Second, for a given transformation $\mathcal{F}$, we can extend an optimal path of length $n-1$ that already ended on this transformation, with the same transformation. Third, for a given transformation $\mathcal{F}$, we can extend any optimal path that ends on note $(n-ML)$, with applying this transformation f on the ML next notes.

These three options capture all different ways an optimal path can be constructed. For the algorithm to work correctly, all the paths that are constructed or saved as optimal paths should also be 'valid' paths. This means that at any point in time all saved paths that consist of partially transformed parts, should have these parts transformed over a minimum length of ML notes with the same transformation.

To be able to keep up with all the bookkeeping, all the optimal paths for each transformation (also the 'non-transformation'), for each of the last ML last notes and for each of the notes these last ML notes can be transformed to, the optimal and valid path that ends on that position on that note with that transformation is saved. The same is done with the probabilities, which eases the computation of the probabilities of the different paths. Because the probabilities are stored separately, it is not necessary to recompute the RPK-probabilities by going over the whole stored path.

5.3 Performace and memory complexity

5.3.1 Performance

The performance time of this algorithm is linearly dependent on the amount of notes (AN). Because, if all other parameters are constant, the amount of work is constant for each timestep, and the amount of timesteps is linearly dependent on AN.

The speed of the algorithm is quadratically dependent on the amount of transformations (AT). The amount of work done in each time step is linearly dependent on AT and the work for each transformation again is linearly dependent on AT.

At last the algorithm is dependent on $ML \times (AN - ML)$. Since the amount of work in each time step to compute the extension of length ML to a path of length $(n - ML)$ in step n, is linearly dependent on ML. The amount of times such a path has to be calculated though is linearly dependent on $(AN - ML)$.

This gives a total time complexity for the algorithm of:

$$\underline{\text{Time:}}\ O(AN \times ML \times (AN - ML) \times AT^2)$$

5.3.2 Memory usage

The amount of memory used during the execution of the algorithm is linearly dependent on the amount of notes in the original melody. Since for all different transformations and possible notes that can be reached via transformation, a single optimal path will be saved. This length of this path is, of course, linearly dependent on the length of the original path.

The memory usage is also linearly dependent on the minimum transformation length. The minimum transformation length leads to the amount of optimal paths of past notes that have to be stored for each transformation. This amount is linearly dependent on ML.

At last, the memory usage is also linearly dependent on the amount of transformations given to the algorithm to use. For each of these transformations an array is made to store all the optimal paths that end on using this specific transformation.

This gives a total memory complexity for the algorithm of:

$$\underline{\text{Memory:}}\ O(AN \times ML \times AT)$$

6 Experiments and Results

The RPK-model that was proposed as a verifier for the musical transformations also has some disadvantages. One of which is that for each note in the musical piece, to compute its probability, the product of the three probability values for that note has to be normalized against all other possible notes for that position. This creates a lot of extra work just to be able to normalize at every note of the melody. An experiment was conducted to find a computationally more interesting predictor for this probability of the RPK-model.

Also, a few experiments have been carried out concerning the algorithm for combining transformations. In these experiments the influence of three parameters of the algorithm (minimum transformation length, the amount of iterations of the algorithm on the melody and the amount of different transformations) on the efficiency of the algorithm is checked.

It is interesting to know which parameters can deliver results that have a score that is still close to the original. Therefore in all the experiments the score of the melody that is found by the algorithm is compared to the score of the original melody as well as the score of the melody that is most probable in the given key. The closer the score stays to the original the better. If the score quickly converges in the direction of the theoretically optimal solution, then the parameter that is tested might, when overly used, lead to results that have too consonance high scores. This is not ideal since these melodies are often not that interesting, and on average contain a lot of the same notes.

When, in the results of the experiments the (average) probability of a musical piece is mentioned, this mean the average probability of a note in the musical piece. Which technically is the score that the RPK-model gives to the melody.

6.1 Predictor for RPK-model score

In this experiment, the combination of two different parameters that can be evaluated much more quickly than the RPK-score are tested on how their score for a musical piece compares tot that of the RPK-model itself.

The first parameter is the average distance in semitones between consecutive notes in the musical piece. One could assume that a smaller average distance leads to a higher probability of the melody. The second parameter is a distance of the distribution of notes in the melody, to the distribution of the notes in the whole of the Essencorpus. This distance is modeled as the Bhattacharyya distance [7].

For 100 musical pieces of the Essencorpus, two different values are calculated. The first one is the absolute value of the consonance score(logarithm of the probability) calculated by the RPK-model. The second one is the product of the average distance between successive notes and the Bhattacharyya distance of the note distribution to the average note distribution. The results of this experiment are shown in figure 6.

The results of this experiment show that there is a correlation between these two metrics. A higher value for the product of the average distance and the Bhattacharyya score,

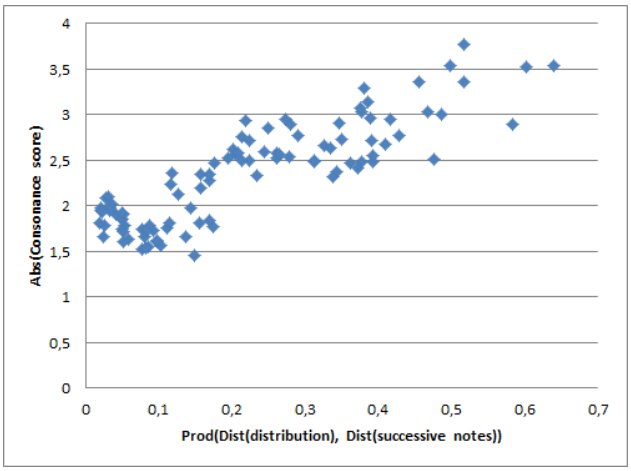


Fig. 6. Product of average distance between successive notes and Bhattacharyya distance of note distribution to average distribution on x-axis. Absolute value of the consonance score on the y-axis.

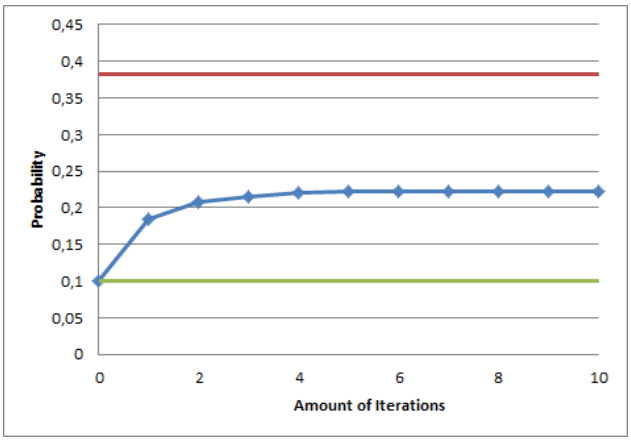


Fig. 7. Average probability of transformed melody after different amounts of iterations in blue. Average theoretical best probability in red. Average probability of the original melody in green.

on average, leads to a higher absolute value for the consonance score and vice versa. This means that for computationally very intensive programs this product metric might be used as a first predictor for the real RPK-value of a musical piece.

6.2 Amount of iterations of the algorithm

In this experiment, the influence of the amount of iterations of the algorithm on the probability of the resulting melody is checked. In this case the algorithm is successively applied a certain number of times to the result of the precious iteration. In this experiment, only the transformation described in table 1 can be used by the algorithm. The minimum transformation length is set to 1.

Now for 100 pieces of the Essencorpus, The average probability of these pieces is computed. Also the average probability of the theoretically best piece in the key of each of those pieces is calculated. And against those two values, for an amount of iterations between 1 and 10 the average scores are compared. The results of this experiment are visible in figure 7.

The figure shows that most of the improvement is made in the first step. After more than five steps there isn't even an improvement any more but at that point the probability of the transformed piece is still closer to that of the original piece than that of the theoretically best one. The reason for this is that in each iteration only the same transformation is available for the algorithm, so the amount of options the algorithm has is small and doesn't change over time in this experiment. This way, most of the improvement is logically in the first step of the algorithm and since the amount of options is limited, it is difficult to reach a score that is close to the theoretically best one.

6.3 Amount of different transformations

This experiment will test the influence of the amount of different transformations available to the algorithm on the probability of the resulting transformed melody.

Diff (mod 8)	0	1	2	3	4	5	6	7
Addition 1	5	-4	1	-3	1	1	2	3
Addition 2	1	3	4	-5	-1	6	5	-1
Addition 3	1	4	5	-3	2	-1	1	0
Addition 4	4	6	-2	4	2	6	-4	2
Addition 5	-3	-2	3	-1	4	-3	2	-4

Table 2. Transformations used in experiment 6.3.

during execution, the algorithm can use the transformations mentioned in table 2. Only one iteration of the algorithm is performed. The algorithms is executed for 5 different values of the amount of available transformations. First the algorithm only has transformation 1 available. Then the algorithm has transformation 1 and 2 that it can use, etc.. The experiment is carried out on 50 pieces of the Essencorpus and the average probability values are calculated for all 5 different values for the amount of transformations.

The results of this experiment are visualised in figure 8. This time the probability of the resulting piece keeps growing for a higher number of available transformations. More so than was the case with a higher number of iterations. This can be explained because the algorithm has more different options for each note and therefore can transform each note to more different target notes that on average lie closer to the optimal solution.

6.4 Minimum transformation length

In a last experiment the influence of the minimum transformation length on the probability of the transformed melody is measured. Now the algorithm can only transform a part of the original melody if that part is at least as long as the minimum transformation length and if every note of that part of the melody will be transformed via the same transfor-

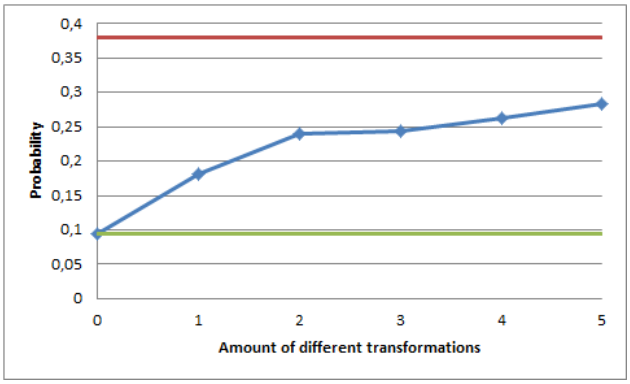


Fig. 8. Average probability of transformed melody for different amounts of available transformations in blue. Average theoretical best probability in red. Average probability of the original melody in green.

Diff (mod 8)	0	1	2	3	4	5	6	7
Addition 1	5	-4	1	-3	1	1	2	3
Addition 2	1	3	4	-5	-1	6	5	-1

Table 3. Transformations used in experiment 6.4.

mation.

During this experiment, the algorithm has two different transformations available which are described in table 3. Only one iteration of the algorithm will be performed each time. For 50 pieces of the Essencorpus, the average probability after transformation is measured for ten different values of the minimum transformation length parameter (1 to 10). This value is then again compared to the average theoretical best probability and the average probability of the original musical piece.

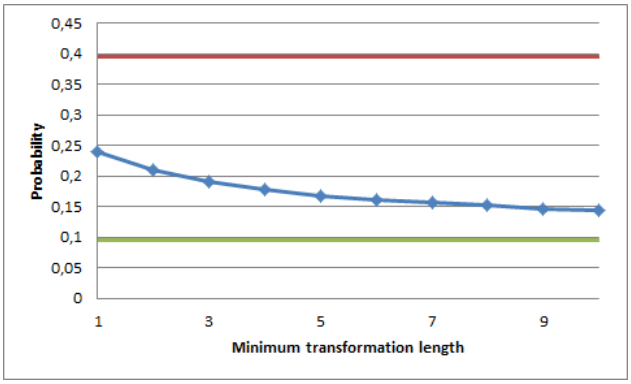


Fig. 9. Average probability of transformed melody for different amounts of minimum transformation lengths in blue. Average theoretical best probability in red. Average probability of the original melody in green.

The results of this experiment are shown in figure 9. Now the average probability of the transformed melody lowers with higher values for the minimum transformation length. This is logical since a higher value for the minimum transformation length constraints the problem more. This result is interesting since choosing a higher value for the minimum transformation length leads to a melody which has an average probability that is relatively close to the average probability of the original piece. One downside is that this is partially the result because some parts of the piece will probably not be transformed. Using a higher amount of different transformations can help against this problem but then also the average probability of the transformed piece will be higher. Depending on what is valued most, the ML parameter can be set accordingly.

7 Conclusions

In this paper, a melodical transformation is described that can transform a given melody into a new one. To judge this type of transformations or a combination of them, the RPK-model was introduced. This model serves as a reference model for musical pieces (of which only the melody is considered, not the rhythm). The model grants a probability to the melody based on the three described parameters that are important in the melodic aspect of music.

Because the RPK-model has some computational disadvantages, which are located in a normalization step for the probability of each note, a predictor has been constructed. This predictor is computationally more interesting and led to promising results which makes it a candidate to use as when a quick prediction of the PRK-probability is necessary for a relatively large piece of music.

Lastly, an algorithm was described which combines different transformations to transform an original melody into a new one with highest reachable probability. In experiments it became clear that, to get resulting melodies that have either a high probability or a probability close to the original, the different parameters of the model can tweaked to influence the average probability into the desired direction.

References

[1] Kroger, P., 2012. *Music for Geeks and Nerds*.
[2] Pattyn, T., 2015. “Op zoek naar inspiratie: Transformatie en evaluatie van muziek”. Master’s thesis, KU Leuven, Belgium.
[3] of Oxford Text Archive, U. Essen corpus of german folksong melodies. http://ota.ox.ac.uk/desc/1038.
[4] Musicxml. http://www.musicxml.com/.
[5] Temperley, D., 2007. *Music and Probability*. The MIT Press, Cambridge, Massachusetts.
[6] A. Lew, H. M., 2006. *Dynamic Programming*. Springer.
[7] Thacker, N. A., Aherne, F. J., and Rockett, P. I., 1998. The bhattacharyya metric as an absolute similarity measure for frequency coded data.

Bijlage C

Poster

Muziek Compositie via Melodische Transformaties

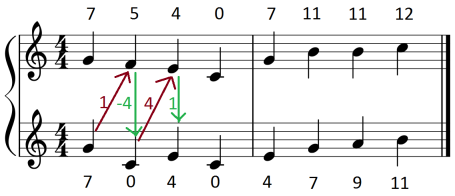
Probleemstelling

- Nood aan **Inspiratie** voor singer-songwriters met writers block
- Muziek generatie leidt vaak tot te 'Artificieel'-klinkende melodieën

→ (Melodische) muziek transformatie kan een uitweg bieden
→ Onder **welke omstandigheden** kan zo een transformatie werken?

Melodische Transformatie

- Sprong op basis van (absolute)**afstand t.o.v. vorige noot**
- Sprong in tegengestelde richting van positie t.o.v. vorige noot
- Altijd eindigen op **noot in juiste toonaard** (door afronding)
- Eenheid in halve tonen



Afstand	Sprong
0	5
1	-4
2	1
3	-3
4	1
5	1
6	2
7	3

Objectieve Beoordeling Melodie

RPK – Model [D. Temperley – 2007]
Probabiliteit van een noot in een notensequentie is het product van 3 factoren:

- R(ange): Probabiliteit van afstand toonhoogte t.o.v. centrale toonhoogte (normaal verdeeld)
- P(roximity): Probabiliteit van afstand in toonhoogte t.o.v. vorige noot (normaal verdeeld)
- K(ey) Profile: Probabiliteit van noot in gebruikte toonaard

Dit leidt tot een **consonantie-score**, een maat voor het goed klinken van een melodielijn

Gebruik transformatie voor verhoging consonantie melodie

Doel:

- Voor elke noot in de melodielijn: behoud deze noot of transformeer ze gebruik makend van één van de meegegeven transformaties
- Doe dit voor elke noot zodat de **probabiliteit** van het totale muziekstuk **zo hoog mogelijk** is
- Zorg dat het computationeel efficiënt is
- Laat toe een **minimum lengte (ML)** mee te geven die aangeeft dat transformaties enkel mogen toegepast worden op minstens ML opeenvolgende noten

Oplossing:

- Algoritme voornamelijk gebaseerd op technieken van dynamic programming
- **Tijd: O(AN x AT^2)** **Geheugen: O(AN x ML x AT)**
  - AN: aantal noten in sequentie
  - ML: minimum lengte transformatie
  - AT: aantal beschikbare transformaties

Algoritme:

Voor elke beschikbare transformatie:
- Houd tabel bij met beste pad dat eindigt met deze specifieke transformatie, en dit voor alle paden eindigend op een van de laatste ML beschouwde noten

Voor elke noot in muziekstuk:

- Voor elke mogelijke transformatie 2 opties:
  - Breid pad dat eindigt met deze transformatie uit gebruik makend van deze transformatie
  - Breid pad dat eindigt met andere transformatie en exact ML in lengte korter is uit met ML keer deze transformatie
  - Pad met hoogste probabiliteit wordt bijgehouden
- Ofwel geen transformatie toepassen:
  - Uitbreiding van eender welk pad dat eindigt op huidige noot zonder transformatie

KU LEUVEN
DTAI Declarative Languages & Artificial Intelligence

Student: Elias Moons
Begeleider: Vincent Nys
Promotor: Prof. De Schreye

Bibliografie

[1] Musicxml. URL: http://www.musicxml.com/.

[2] P. Kroger. *Music for Geeks and Nerds*. 2012.

[3] G. Loy. *Musimathics*. The MIT Press, Cambridge, Massachusetts, 2006.

[4] D. Marshall. Introduction to midi (musical instrument digital interface). URL: https://www.cs.cf.ac.uk/Dave/Multimedia/node155.html.

[5] W. MathWorld. Fibonacci number. URL: http://mathworld.wolfram.com/FibonacciNumber.html.

[6] M. Nielsen. Backpropagation. URL: http://neuralnetworksanddeeplearning.com/chap2.html, 2016.

[7] M. Nielsen. *Neural Networks and Deep Learning*. 2016.

[8] V. Nys. Noten lezen voor elektrische gitaristen: een expertsysteem voor het genereren van bladmuziek. Master's thesis, KU Leuven, Belgium, 2013.

[9] U. of Oxford Text Archive. Essen corpus of german folksong melodies. URL: http://ota.ox.ac.uk/desc/1038.

[10] T. Pattyn. Op zoek naar inspiratie: Transformatie en evaluatie van muziek. Master's thesis, KU Leuven, Belgium, 2015.

[11] D. Temperley. *Music and Probability*. The MIT Press, Cambridge, Massachusetts, 2007.

[12] N. A. Thacker, F. J. Aherne, and P. I. Rockett. The bhattacharyya metric as an absolute similarity measure for frequency coded data, 1998.

[13] Topcoder. Dynamic programming - from novice to advanced. URL: https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/.

[14] S. University. Feed-forward networks. URL: http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html.

Fiche masterproef

Student: Elias Moons

Titel: Melodische transformatie en evaluatie van muziek

Engelse titel: Melodical transformation en evaluation of music

UDC: 004.9

Korte inhoud:
Deze masterproef beschrijft methodes om melodielijnen van een muziekstuk te transformeren tot nieuwe melodielijnen. Er gaat ook aandacht uit naar een referentiekader waarin deze transformaties geëvalueerd kunnen worden. Tot slot wordt er gekeken naar wanneer bepaalde transformaties nuttig kunnen zijn om de consonantie van een muziekstuk te verhogen en hoe verschillende transformaties efficiënt gecombineerd kunnen worden. Om dit te verwezenlijken ontwikkelden we een algoritme dat gebaseerd is op de principes van *dynamic programming*. Dit algoritme zal, gegeven een aantal mogelijke transformaties en een melodielijn, de best mogelijke getransformeerde melodielijn teruggeven volgens het gedefiniëerde referentiemodel.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

Promotor: Prof. dr. D. De Schreye

Assessoren: Prof. dr. ir. T. Schrijvers
Prof. dr. M.-F. Moens

Begeleider: Ir. V. Nys