

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

Assessor:

Prof. Onbekend

Begeleider:

Ir. V. Nys

© Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail info@cs.kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Voorwoord

TODO: iedereen bedanken...

Elias Moons

Inhoudsopgave

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
1 Inleiding	1
1.1 Probleemstelling	1
1.2 Vergelijking met voorgaand onderzoek	1
1.3 Doelstelling	2
1.4 Assumpties en beperkingen	2
1.5 Overzicht van de tekst	3
2 Muzikale Achtergrond	5
2.1 Ritme	5
2.2 Melodie	6
3 Objectieve Beoordeling van een Muziekstuk	11
3.1 Neuraal Netwerk	11
3.2 Keuze tussen Polyfone en Monofone Muziek	13
3.3 RPK-Model	14
4 Melodische Transformatie	19
4.1 Inleiding	19
4.2 Afbeelding afhankelijk van positie	20
4.3 Afbeelding afhankelijk van afstand ten opzichte van vorige noot	22
5 Transformaties Combineren	25
5.1 Beste sequentie	26
5.2 Beste sequentie met minimum transformatie lengte	31
6 Experimenten en Resultaten	39
6.1 Experimenten met melodische transformaties	39
6.2 Experimenten met het RPK-model	42
6.3 Experimenten over de werking van de algoritmes voor het combineren van transformaties	46
6.4 Experimenten over de tijdscomplexiteit van de algoritmes voor het combineren van transformaties	51
7 Besluit	57

7.1	Eerste onderwerp in dit hoofdstuk	57
7.2	Figuren	57
7.3	Tabellen	57
7.4	Besluit van dit hoofdstuk	58
A	Broncode	61
A.1	Neuraal Netwerk Trainer	61
A.2	RPK-Model	67
A.3	Beste sequentie	71
A.4	Beste sequentie met minimum transformatie lengte	73
B	IEEE_Paper	81
C	Poster	83
	Bibliografie	85

Samenvatting

Deze masterproef beschrijft methodes om melodielijnen van een muziekstuk melodisch te transformeren to nieuwe melodielijnen. Verder wordt er ook aandacht besteed aan een referentiekader waarin deze transformaties geëvalueerd kunnen worden. Tot slot wordt er ook nog 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 is een algoritme beschreven dat gebaseerd is op de principes van *dynamic programming*. Dit algoritme zal, gegeven een aantal mogelijke transformaties en een melodilijn, de best mogelijke getransformeerde melodilijn teruggeven volgens het gedefiniëerde referentiemodel.

Lijst van figuren en tabellen

Lijst van figuren

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.	2
2.1	Maat met voortekening van <i>common time</i> tijdssignatuur.	6
2.2	Illustratie van een aantal veelvoorkomende nootlengtes.	6
2.3	Noten do t.e.m. si op een notenbalk.	7
2.4	Toonladder van “Do Groot”	9
2.5	Toonladder van “La Klein”	9
3.1	Neuraal netwerk met 12 input nodes (1 voor elke noot), 6 hidden nodes en 1 output.	12
3.2	Distributie van alle noten in het Essencorpus.	15
3.3	Properties van voorkomen van alle intervallen van opeenvolgende noten in het Essencorpus.	15
3.4	Properties van voorkomen in het Essencorpus van nootfuncties in grote toonladder.	16
3.5	Properties van voorkomen in het Essencorpus van nootfuncties in kleine toonladder.	16
4.1	Voorbeeld van toepassing transformatie afhankelijk van positie.	21
4.2	Voorbeeld van toepassing transformatie afhankelijk van interval ten opzichte van vorige noot.	23
5.1	Illustratie van de <i>matrix</i> -variabele. Illustratie van beste pad voor 2 verschillende toonhoogtes.	28
5.2	Illustratie van de <i>past</i> - en <i>current</i> variabelen tijdens een stap in het algoritme. De weergave is enkel voor de transformatie weergegeven in tabel 5.1. Dit is een voorbeeld voor het specifieke geval waarin de huidige noot in het originele stuk 3 halve tonen hoger ligt dan de vorige noot.	30
5.3	Illustratie van de variabelen die probabiliteiten opslaan met betrekking tot het ‘niet-transformeren’ van de laatste noot. Elk element in deze tabel stelt een (logaritme van een) probabilliteit voor.	34

5.4	Illustratie van de variabelen die probabiliteiten opslaan met betrekking tot het transformeren van de laatste noot. Elk element in deze tabel stelt een (logaritme van een) probabilliteit voor.	35
5.5	Illustratie van de variabelen die optimale (geldige) paden opslaan met betrekking tot het ‘niet-transformeren’ van de laatste noot. Elk element in deze tabel stelt een optimaal en geldig pad voor.	36
5.6	Illustratie van de variabelen die optimale (geldige) paden opslaan met betrekking tot het transformeren van de laatste noot. Elk element in deze tabel stelt een optimaal en geldig pad voor.	37
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.	43
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.	44
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.	45
6.4	Resultaten van het experiment uitgevoerd in deel 6.3.1. De groene lijn staat voor de gemiddelde probabilliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabilliteit van de theoretisch beste melodieliijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabilliteit van een noot weer na uitvoer van het algoritme na een verschillend aantal iteraties.	47
6.5	Resultaten van het experiment uitgevoerd in deel 6.3.2. De groene lijn staat voor de gemiddelde probabilliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabilliteit van de theoretisch beste melodieliijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabilliteit van een noot weer na uitvoer van het algoritme afhankelijk van het aantal transformaties dat ter beschikking gesteld was aan het algoritme.	48
6.6	Resultaten van het experiment uitgevoerd in deel 6.3.3. De groene lijn staat voor de gemiddelde probabilliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabilliteit van de theoretisch beste melodieliijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabilliteit van een noot weer na uitvoer van het algoritme afhankelijk van de minimum transformatie lengte.	49
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.	52
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.	53

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.	54
7.1	Het KU Leuven logo.	57

Lijst van tabellen

2.1	Opsomming van de toonhoogtes in 2 verschillende benamingen.	7
4.1	Transformatie afhankelijk van de positie van de noot.	20
4.2	Transformatie afhankelijk van de afstand van de huidige noot tot de vorige noot.	22
5.1	Voorstelling transformatie gebruik in figuur 5.2.	29
6.1	Willekeurige transformatie volgens T1, gebruikt in het experiment van onderdeel 6.1.1.	40
6.2	Willekeurige transformatie volgens T2, gebruikt in het experiment van onderdeel 6.1.1.	40
6.3	Resultaten van experiment 6.1.1. Gemiddelde consonantiescores voor de twee soorten transformaties en de originele melodielijnen die getransformeerd werden. De twee geteste soorten van transformaties staan beschreven in onderdelen 4.2(T1) en 4.3(T2).	40
6.4	Fibonacci transformatie volgens T1 gebruikt in het experiment van onderdeel 6.1.2.	41
6.5	Fibonacci transformatie volgens T2 gebruikt in het experiment van onderdeel 6.1.2.	41
6.6	Resultaten van experiment 6.1.2. Gemiddelde consonantiescores voor de twee Fibonacci transformaties en de originele melodielijnen die getransformeerd werden.	41
6.7	Transformatie gebruikt in het experiment van onderdeel 6.3.1.	46
6.8	Transformaties gebruikt in het experiment van onderdeel 6.3.2.	47
6.9	Transformaties gebruikt in het experiment van onderdeel 6.3.3.	49
6.10	Transformatie gebruikt in het experiment van onderdeel 6.3.4.	50
6.11	Resultaten 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.	51
6.12	Transformatie gebruikt in het experiment van onderdeel 6.4.1.	51
6.13	Transformaties gebruikt in het experiment van onderdeel 6.4.2.	53
6.14	Transformaties gebruikt in het experiment van onderdeel 6.4.3.	54
7.1	Een tabel zoals het niet moet.	58
7.2	Een tabel zoals het beter is.	58

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. 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. 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. Een muziekluisteraar 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. En dit blijkt zeer moeilijk te verwezenlijken vanuit een muziek generatie-standpunt.

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 deze 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 (Essen corpus [8]) 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, zodat zijn score in het

voorgestelde model telkens ook als referentie kan dienen voor zijn 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 melodieline, de meest waarschijnlijke getransformeerde melodieline 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. 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 tijdssignatuur wordt weergegeven aan het begin van de notenbalk en ziet er uit 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. 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.

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



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

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 note 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 voornaamste bestanddeel van de muziek dat een gevoel meegeweet 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

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.

‘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 fysische 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 (\sharp) of een mol(\flat). 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 $1-1-\frac{1}{2}-1-1-\frac{1}{2}$. 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. 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 ook weergegeven in figuur 2.5. 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. En iets wiskundiger verwoord is er ook een verschil van frequentie waarin bepaalde noten voorkomen in muziekstukken horende bij een kleine of grote toonladder [10]. Van de kleine toonladders zijn ook nog een harmonische en melodische versie, die nog andere afstanden gebruik, 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. Maar 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 nooit creëert namelijk spanning. Vaak wordt deze spanning dan ook opgelost door een overgang naar de eerste noot, als rustpunt.



FIGUUR 2.4: Toonladder van “Do Groot”



FIGUUR 2.5: Toonladder van “La Klein”

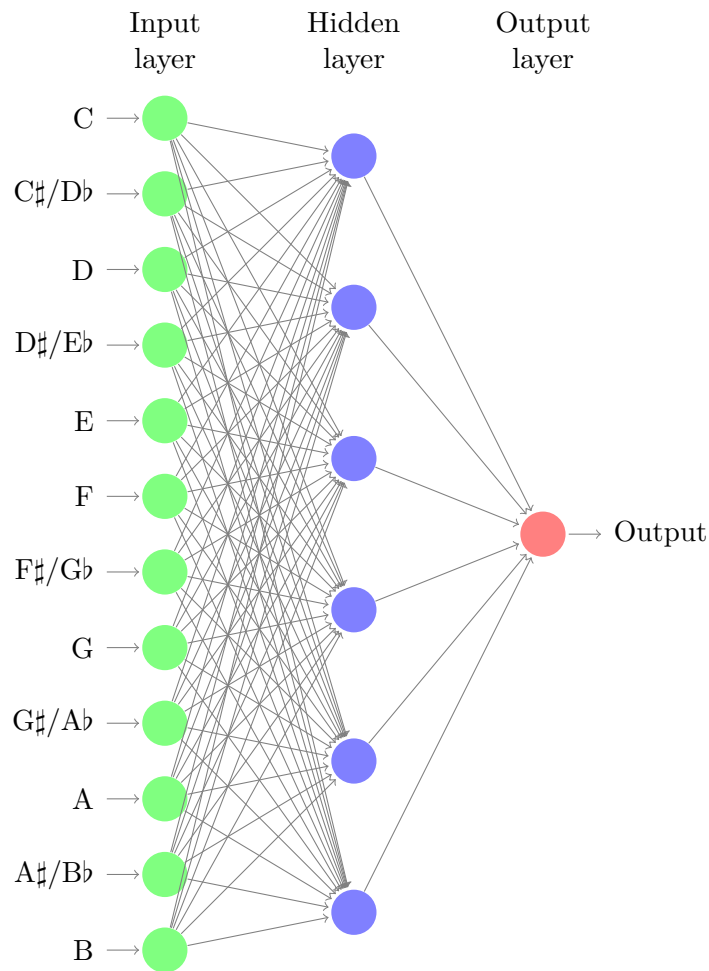
Hoofdstuk 3

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 zo een 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 artificiële neurale 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 het boek ‘Music and Probability’ [10] van David Temperley [2]. Dit model werkt op muziek met een enkele melodielijn en zal ook het model zijn dat als verificateur gebruikt zal worden bij de experimenten in het vervolg van deze masterproef.

3.1 Neuraal Netwerk

In het boek ‘Musimathics’ [5] 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) neurale netwerk[11] op zogenaamde tweeklanken (twee noten die tegelijkertijd gespeeld worden). Het neurale netwerk maakt dan de veralgemening naar hogere orde samenklanken. Het neurale netwerk dat gemaakt werd als verificateur 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 muziktechnisch 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 andere geval



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

krijgt deze een ‘0’ als input. Vervolgens komen we op een hidden layer terecht die volledige 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 dichterbij 1 ligt hoe zekerder het neuraal netwerk is dat het ingegeven akkoord consonant is. Hoe dichterbij 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[1]. 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 ook 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 neurale 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 neurale 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 gehele muziekstuk.

Nadat het neurale netwerk getraind werd, bleek het neurale 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 [14] en Mozart [17] waarop getest werd zelf helemaal niet zo consonant waren als oorspronkelijk gedacht. En dit in die mate dat vaak tot een derde van de akkoorden in zo een stuk als dissonant (tegengestelde van consonant) bestempeld worden. 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 neurale netwerk dat hier gebruikt werd houdt echter geen rekening met deze context. Dit leidt er wel toe dat dit neurale netwerk niet gebruikt kan worden als verificador, aangezien zelfs muziekstukken van Bach en Mozart door de verificador 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 neurale netwerk is beschikbaar in appendix A.1. Er zijn een aantal parameters die ingesteld kunnen worden in dit algoritme. Er is allereerst 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[3] 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[15] waarbij dit niet het geval is. De polyfone(meerstemmige) muziek die beschikbaar is in het juiste formaat bestaat eigenlijk bijna uitsluitend uit klassieke muziek. Verder is er ook nog een redelijk groot corpus beschikbaar van folkmuziek, het zogenaamde Essencorpus[8]. Deze muziek is wel monofoon (eenstemmig). Aangezien de methode met het neurale netwerk uitgelegd 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 delicateser zijn om te behandelen dan stukken folkmuziek die enkel uit een melodieline bestaan. Alle transformaties die zullen besproken worden in de rest van deze masterproef zullen dus ook uit een enkele melodieline bestaan.

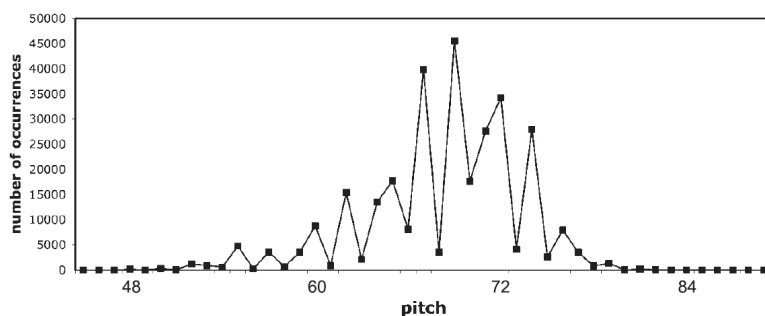
3.3 RPK-Model

Aangezien er nood was aan een framework voor het beoordelen van muziekstukken van slechts een enkele melodieline werd er uiteindelijk uitgekomen bij het zogenaamde RPK-model. Dit model wordt beschreven in het boek ‘Music and Probability’[10]. 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.

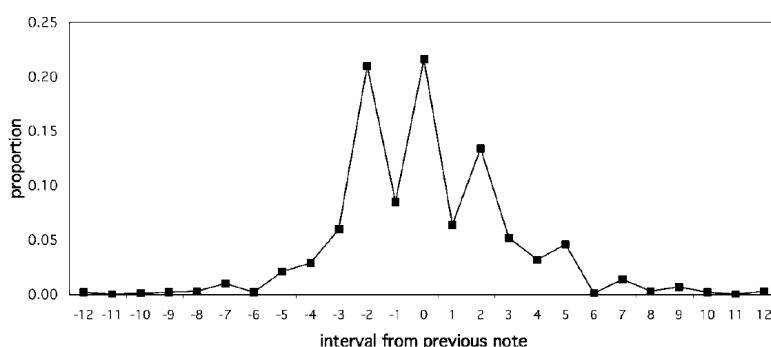
Allereerst is er de zogenaamde *range*, dit is de afwijking tot een centrale toonhoogte. Deze centrale toonhoogte is de noot die in het midden ligt van de distributie van alle noten uit alle muziekstukken van het beschikbare corpus van folkmuziek. Globaal gezien hebben noten die dicht bij die centrum liggen een grotere waarschijnlijkheid tot voorkomen als noten die verder van dit centrum gelegen 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 op te merken is dat in dit model een melodieline 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 of een kwint komt bijvoorbeeld veel vaker voor dan een sprong van een sext. 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.

Ten slotte wordt er ook nog rekening gehouden met de *key*, oftewel de toonaard



FIGUUR 3.2: Distributie van alle noten in het Essencorpus.



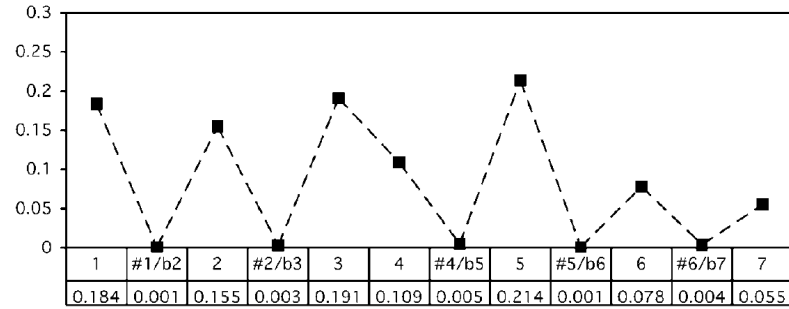
FIGUUR 3.3: Proporties van voorkomen van alle intervallen van opeenvolgende noten in het Essencorpus.

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 normaalverdeling 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.

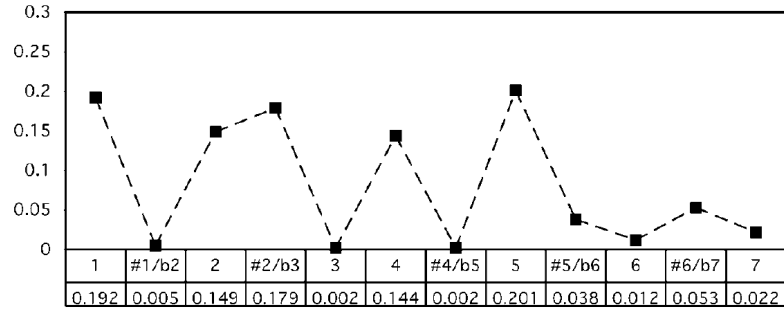
Bepaling score van een volledige melodieline

Nu kan voor elke noot in een melodieline 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

3. OBJECTIEVE BEOORDELING VAN EEN MUZIEKSTUK



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.

score van een volledige melodieline staat dan gelijk aan het gemiddelde score over alle toonhoogtes. De score van een melodieline geeft dus weer wat de gemiddelde waarschijnlijkheid is van een noot in die melodieline volgens het RPK-model. De score van een melodieline 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 melodieline 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.

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 telkens dezelfde noot (zodat de afstand tot de vorige noot telkens 0 is), waarvan deze noot zeer waarschijnlijk is in de toonaard en dicht bij de centrale noot ligt. Het RPK-model geeft dus hoge scores aan stukken die zich dicht rond de centrale noot afspelen en kleine sprongen vertoont 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 de gemiddeld gezien minder afwijking in score teweegbrengt op een muziekstuk, ongeacht in welke richting. 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 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[6] 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 ook in de muziekwereld zo een invloed zou kunnen hebben. Dit bleek niet het geval te zijn. Maar aangezien het ook geen slechtere resultaten gaf 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 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 wordt op de bovenste rij, dan zal de

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.

noot op positie 4 met 5 halve tonen verhoogd worden door de transformatie. Maar ook bijvoorbeeld 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 dat ervoor gekozen is om na uitvoer van de transformaties nog een zogenaamde ‘afronding’ door te voeren. Deze afronding bestaat erin om na de transformatie van een noot in de melodieline, indien deze niet tot de toonaard behoort, te verhogen of verlagen met een halve toon 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 dus 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 ik wel degelijk altijd gebeurt. Vandaar dat het soms ook kan zijn dat het lijkt alsof een transformatie een halve toon te hoog of te laag is uitgevoerd voor een bepaalde noot. Dit ligt dan enkel aan de zonet besproken afronding.

4.2 Afbeelding afhankelijk van positie

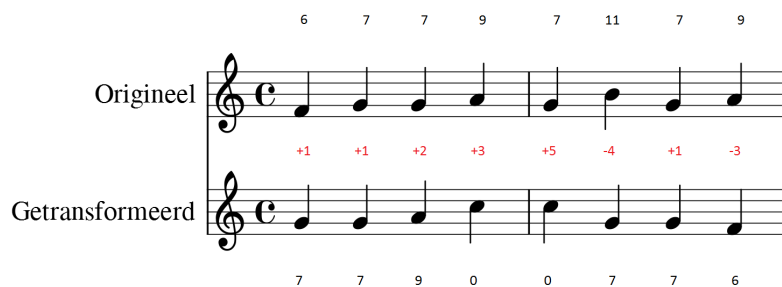
4.2.1 Beschrijving transformatie

Een eerste zeer voor de hand liggende melodische transformatie is er eentje die een noot in een muziekstuk gaat transformeren enkel naargelang zijn positie in de melodieline. 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 melodieline. De bovenste lijn geeft de originele melodie weer, de onderste lijn geeft het getransformeerde resultaat weer. Bij de twee melodielijnen staat bij elke noot telkens ook zijn geheel getal representatie in halve noten (modulo 12). Dit zodat het voor

de lezer makkelijker na te gaan is hoe de transformatie precies verloopt. Tussen de twee notenbalken wordt dan aangegeven welke sprong de transformatie oplegt op de melodieline. 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 het zou moeten zijn. Dit komt door de afronding besproken in onderdeel 4.1.2 aangezien de noot met getalwaarde 8 ($G\sharp/A\flat$) 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 zeer elementair is. Enkel de positie van de noot is van belang met betrekking tot naar welke noot hij 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 totaal 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.

4.3 Afbeelding afhankelijk van 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 in ten opzichte van de vorige noot. Hierbij zal er gekeken worden naar de afstand van de noot op positie 'x' in de originele melodieline tot de noot op positie 'x-1' in de nieuwe melodieline (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 nog 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 staat ten opzichte van de vorige. 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. En 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 melodieline 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 en 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, al zal in dit geval toch de andere richting uit gegaan worden 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 4 is. Een absolute waarde van 4 voor het verschil geeft aanleiding tot een sprong van grootte 1. aangezien het verschil positief is moet de

4.3. Afbeelding afhankelijk van afstand ten opzichte van vorige noot

sprong echter in tegengestelde richting uitgevoerd worden en 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 transformatie afhankelijk van interval ten opzichte van 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.

Hoofdstuk 5

Transformaties Combineren

In dit hoofdstuk worden twee algoritmes behandeld die varianten zijn van elkaar. In beide algoritmes is het de bedoeling om een gegeven muziekstuk te transformeren tot een nieuw muziekstuk op zo een manier dat de totale consonantiescore zo hoog mogelijk is. Dit kan gebeuren door een aantal toegelaten operaties op het oorspronkelijke muziekstuk. Buiten de originele melodieline zullen namelijk 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 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 melodieline heeft het algoritme namelijk de keuze om ofwel de noot te behouden, ofwel deze te transformeren conform een van de transformaties die meegegeven werden aan het algoritme. Het algoritme dat hier voor zorgt wordt beschreven in onderdeel 5.1. In gedeelte 5.2 wordt een algoritme beschreven dat eenzelfde doel heeft maar een extra voorwaarde opgelegd krijgt. Deze voorwaarde stelt dat als met een transformatie wilt toepassen, men telkens een minimum aantal opeenvolgende noten moet transformeren met dezelfde transformatie.

Het is belangrijk op te merken dat het opzet van deze algoritmen niet van die aard is deze daadwerkelijk te gaan gebruiken om muziekstukken te gaan creëren die goed zullen klinken. Er wordt nadrukkelijk geprobeerd om die melodieline te vinden die een zo hoog mogelijke consonantiescore heeft volgens het RPK-model. Er werd echter reeds aangehaald dat muziekstukken met een zeer hoge consonantiescore niet goed klinken (veel dezelfde noten en heel kleine sprongen tussen opeenvolgende noten). Het 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. Een eerste parameter is een originele melodieline. De tweede parameter beschrijft een aantal transformaties (gedefinieerd zoals in onderdeel 4.3), die gebruikt mogen worden door het algoritme. Het doel is nu om gebruik makend van enkel deze verschillende mogelijke transformaties de originele melodieline te transformeren tot de nieuwe melodieline die de hoogste consonantiescore oplevert. 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 melodieline. De tweede mogelijkheid bestaat erin dat de noot ook getransformeerd mag worden, maar denk enkel volgens een van de beschikbare transformaties.

5.1.2 Idee van het algoritme

Een eerste 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). Er is nu echter nog een speciaal geval waarbij een transformatie tot een van de twee extreme gevallen zou leiden 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. 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 dan zichzelf. Dit zijn in totaal 14 verschillende mogelijkheden.

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 best pad (en bijhorende beste score) bijgehouden zal worden voor elk van de 14 mogelijke toonhoogtes die deze noot kan aannemen in de nieuwe melodieline. 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 melodieline 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.

Het algoritme zal tijdens de uitvoering gebruik maken van 3 hulpvariabelen. De eerste twee van deze hulpvariabelen zijn eendimensionale arrays van lengte 14. De eerste array, die de naam *past* meekrijgt, stelt de probabiliteiten voor van de beste paden eindigend bij alle mogelijke noten waarnaar de vorige beschouwde noot kan

getransformeerd worden. De tweede array, die *current* genoemd wordt, bevat de probabiliteiten van de beste paden die eindigen op de huidig beschouwde noot (of een van zijn mogelijke transformaties). De waarden in deze twee arrays, zullen in het algoritme niet de probabiliteiten zelf, maar hun logaritmen zijn, dit om het rekenwerk te vereenvoudigen. De derde hulpvariabele, *matrix*, is een tweedimensionale array en heeft een dimensie van de lengte van de melodieline op 14. Deze variabele gaat voor elke mogelijke waarde die elke noot in het muziekstukje kan aannemen, weergeven vanwaar het beste pad komt dat eindigt bij deze noot.

Om de betekenis van deze variabelen nog iets concreter te maken zal gebruik gemaakt worden van de volgende notatie. De letter ‘C’ voor de toonhoogte van de huidig beschouwde noot in de originele melodieline. De letter ‘P’ voor de toonhoogte van de vorige noot in het originele muziekstukje. Zo zal de waarde C+3 staan voor de noot met een toonhoogte die drie halve tonen hoger ligt dan de huidige noot in het originele stukje. P-4 zal dan bijvoorbeeld staan voor de noot met een toonhoogte die 4 halve tonen lager ligt dan de vorige noot in de originele melodieline. Verder zal, er als er in gesproken wordt over de i-de noot in het muziekstuk, de noot op positie i bedoeld worden. Zo zal de 0-de noot in het muziekstukje de eerste noot zijn, de 1-de noot zal de tweede noot van het muziekstukje zijn enzovoort.

Stel we beschouwen momenteel de n-de noot van het muziekstukje. Dan zullen volgende stellingen gelden. Op positie *past[i]* zal de waarde voor de probabiliteit staan van het beste pad van n noten dat eindigt op de noot ‘P+i-6’. Ook zal tijdens uitvoering van het algoritme de *current* array aangepast worden. Met als doelstelling: op positie *current[i]* zal de waarde voor de probabiliteit berekend worden voor het beste pad van (n+1) noten dat eindigt op de noot ‘C+i-6’. De waarde van het element *matrix[i][j]* verwijst naar vanwaar het beste pad komt van (n+1) noten dat eindigt op de noot ‘C+j-6’. Stel nu dat *matrix[i][j]=k*, dan betekent dit dat het beste pad van (n+1) noten dat eindigt op noot ‘C+j-6’, een verlenging is van het beste pad van n noten dat eindigt op noot ‘P+k-6’.

In figuur 5.1 wordt een voorbeeld van de inhoud van de *matrix*-variabele weergegeven. Voor een zekere noot ‘n’ uit het muziekstuk is nu voor 2 van de mogelijke 14 waarden naarwaar deze kan getransformeerd worden geïllustreerd hoe de beste paden horende bij deze twee toonhoogtes teruggevonden kunnen worden. Een van deze paden wordt in het groen weergegeven het andere in het oranje. De laatste 5 noten van het groene pad hebben als toonhoogte een die respectievelijk 2, 1, 1, -2 en -4 halve tonen hoger ligt dan hun overeenkomstige noot in de originele melodie. De laatste 5 noten van het oranje pad hebben als toonhoogte een die respectievelijk -3, -1, 0, 1 en 4 halve tonen hoger ligt dan hun overeenkomstige noot in de originele melodie.

Initialisatie

De eerste stap in het algoritme zal de noot op positie 1 als huidige noot bekijken en de noot op positie 0 als vorige noot. De hulpvariabelen moeten dus geïnitieerd worden zodat ze daaraan voldoen. Hierdoor zal de array *past* voor al zijn posities een hele lage waarde meekrijgen (-100 als logaritme van de probabiliteit in de broncode weergegeven in appendix A.3) behalve voor de noot op positie 6. Deze noot staat

Index		(n-3)	(n-2)	(n-1)	n	
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 5.1: Illustratie van de *matrix*-variabele. Illustratie van beste pad voor 2 verschillende toonhoogtes.

namelijk voor dezelfde noot als oorspronkelijke noot en aangezien de oorspronkelijke noot op positie 0 nooit getransformeerd wordt, zal $past[6] = 0$. Dit betekent dat dat kans op het behouden van de eerste noot gelijk is aan 1. De *current* array krijgt voor al zijn posities zeer lage waarden mee. De bedoeling hiervan is dat eender welk pad dat als eerste gevonden wordt dat voortgaat op de vorige noot een hogere probabiliteit zal hebben dan deze waarde. Tot slot zal ook elke waarde van *matrix* op een arbitraire startwaarde 0 gezet worden. De waarde die hier in het begin gezet wordt is van geen belang, aangezien deze waarden toch overschreven zullen worden wanneer de beste paden berekend worden.

Een stap in het algoritme

Voor elke noot in het muziekstukje, beginnend vanaf de noot op positie 1, zal nu het volgende uitgevoerd worden. Stel ook dat we momenteel als huidige noot, de noot op positie n beschouwen.

Allereerst gaan we op $current[6]$ de waarde zetten van het beste pad dat de huidige noot niet transformeert. Om dit te doen gaan we kijken naar de probabiliteiten van

diff mod 8	0	1	2	3	4	5	6	7
sprong	5	-4	1	-3	1	1	2	3

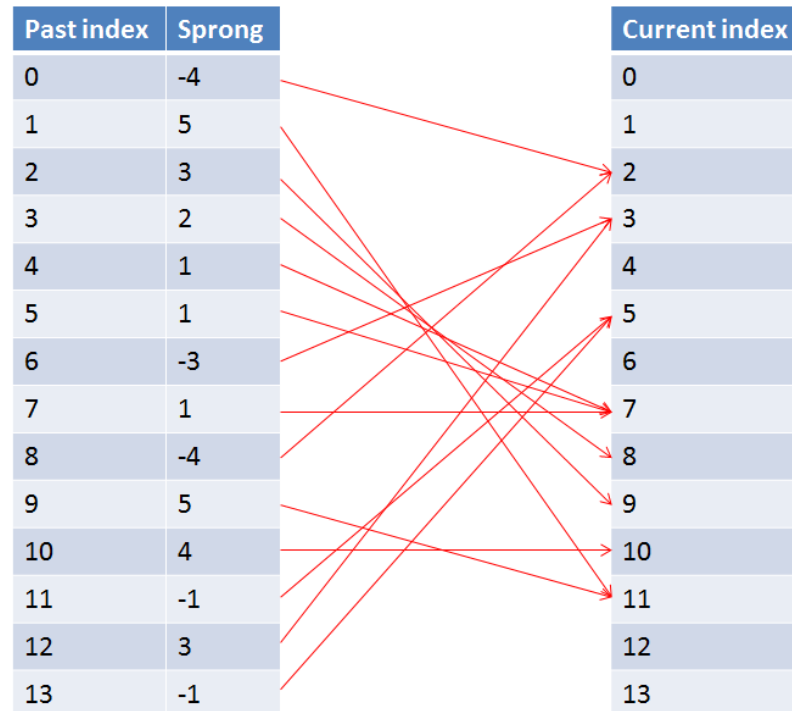
TABEL 5.1: Voorstelling transformatie gebruik in figuur 5.2.

de beste paden die eindigen op alle 14 mogelijke noten die als vorige noot zouden kunnen doorgaan. Deze probabiliteiten staan uiteraard gewoon in *past*. Voor elk element *past[i]* wordt nu zijn waarde bij de afstandsprobabiliteit ten opzichte van de huidig beschouwde noot opgeteld. Als deze waarde hoger is dan de huidige beste waarde van *current[6]*, dan wordt de waarde van *current[6]* overschreven en wordt de waarde *matrix[n][6]* op *i* gezet. Als tweede stap gaan we voor elke mogelijk transformatie kijken naar welke noot de noot op ‘P+i-6’ de noot huidige noot gaat afbeelden. En dit voor alle 14 mogelijke vorige noten. Stel nu dat de noot ‘P-i+6’ de huidig beschouwde noot afbeeldt op noot ‘C+j-6’. Als nu de de som van *past[i]* en de afstandsprobabiliteit tussen deze twee noten groter is dan de huidige waarde *current[j]*, dan zal deze waarde overschreven worden. Alsook zal *matrix[n][j]* gelijk gesteld worden aan *i*. Dit aangezien het nieuwe meest waarschijnlijke pad dat gevonden is dat eindigt op ‘C+j-6’, komt van de noot ‘P+i-6’.

In figuur 5.2 wordt een voorbeeld gegeven van de mapping van probabiliteiten horende bij noten uit de *past*-array naar nieuwe probabiliteiten horende bij noten uit de *current*-array. Bij deze mapping werd gebruik gemaakt van 1 transformatie die gekarakteriseerd wordt door de waarden in tabel 5.1. In dit voorbeeld wordt ook gesteld dat de huidige noot in het originele stuk 3 halve tonen hoger lag dan de vorige noot. Met de informatie van het verschil in toonhoogte tussen de huidige en de vorige noot en de gegeven transformatie kunnen nu de verschillende sprongen bepaald worden die de huidige noot zou ondergaan, afhankelijk van welke van de 14 noten uit de *past*-array als vorige noot beschouwd wordt. De pijlen geven uiteindelijk nog aan naar welke cel in de *current*-array al deze sprongen zouden leiden. Een waarde in de *current*-array wordt enkel aangepast wanneer er een pijl is die aankomt in deze cel, die ook nog leidt tot een hogere score dan de waarde die al in die cel staat. De score waarover gesproken wordt is de som van de waarde in de *past*-array waarvan de deze pijl vertrekt en het logaritme van de afstandswaarschijnlijkheid. Wanneer dit uitgevoerd is, bevat *current* de probabiliteiten van de beste paden van lengte (n+1) die eindigen op de 14 mogelijke toonhoogtes waarop de huidig beschouwde noot kan afgebeeld worden. Het enige wat nu nog rest is om de waarden van *current* over te zetten naar *past*. En om *current* daarna terug te initialiseren op zeer lager waarden. Dit zodat de arrays klaar zijn voor het beschouwen van de volgende noot in het muziekstukje.

Extractie van het beste pad

Na uitvoeren van het vorige gedeelte voor elke noot in het muziekstukje zal *past* de probabiliteiten bevatten van de beste paden voor elk van de 14 mogelijke eindnoten van het geheel van het nieuwe muziekstukje. Als eerste zal er dus gekeken worden welke van deze 14 toonhoogtes het meest waarschijnlijke pad heeft. Dit pad is het



FIGUUR 5.2: Illustratie van de *past*- en *current* variabelen tijdens een stap in het algoritme. De weergave is enkel voor de transformatie weergegeven in tabel 5.1. Dit is een voorbeeld voor het specifieke geval waarin de huidige noot in het originele stuk 3 halve tonen hoger ligt dan de vorige noot.

pad dat we zoeken in dit algoritme. Het enige wat nu nog rest is om vanaf dit punt dat pad te reconstitueren. Dit gebeurt met behulp van de *matrix* array. Beginnen bij de laatste noot en zo opschuivend terug naar voor wordt hiervoor het volgende gedaan. Stel we zitten bij noot n . Noem ‘C+i-6’ de toonhoogte van de huidige noot die in dit beste pad zit. Noem nu $matrix[n][i] = j$. Dan is de vorige noot van het optimale pad de noot met als toonhoogte ‘P+j-6’, waarbij P de toonhoogte is van de overeenkomstige noot in het originele muziekstuk. $matrix[n-1][j]$ geeft dan weer aanleiding tot de noot die daarvoor staat in het optimale pad, enzovoort. Op deze manier kan het volledige optimale pad weer gereconstrueerd worden. Figuur 5.1 kan nogmaals als illustratie dienen voor dit principe.

5.1.4 Performantie en geheugencomplexiteit

De parameters waarvan het algoritme afhankelijk is, zijn de lengte van de melodieline en dus het aantal noten (AN) in de invoer en ook het aantal transformaties (AT) (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 gedefiniëerd 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 f omhoog gaat en de rest constant blijft, dan zal het aantal stappen onveranderd blijven. Het rekenwerk per stap gaat wel met een factor f omhoog (op het constante rekenwerk per stap van het ‘niet transformeren’ na).

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

Geheugen

Het geheugen dat 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 tijdens de uitvoer van het algoritme is onveranderlijk ten opzichte van die lengte. Maar in het total geheugengebruik is de *matrix* array dominant aangezien deze zo veel groter is. Het aantal gebruikte transformaties heeft geen effect op het geheugengebruik van het algoritme.

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

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 melodieline 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 melodieline mag transformeren als hij voor minstens dit opgegeven aantal opeenvolgende noten dezelfde transformatie toepast. Er zijn een aantal overeenkomsten tussen dit algoritme en het algoritme beschreven in sectie 5.1. Bij het bespreken van deze overeenkomsten zal er verwezen worden naar overeenkomstige delen in dat hoofdstuk. Daar waar dit algoritme verschilt van het vorige zal een volledige uitleg gegeven worden.

5.2.2 Idee van het algoritme

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, waarom dit precies noodzakelijk is wordt verder in de tekst nog verduidelijkt. Het idee van dit algoritme bestaat erin om elke noot van het muziekstukje chronologisch te overlopen. En 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 waarin de originele noot kan getransformeerd 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 en de mogelijke transformaties respecteert. De parameter die voor de minimum transformatie lengte staat wordt met ‘ML’ aangeduid. Een eerste mogelijkheid voor zo een optimaal pad is de uitbreiding van eender welk optimaal pad dat eindigt bij de vorige noot met het behouden van de huidige noot. Een tweede mogelijkheid is het uitbreiden van een optimaal pad dat geldig is, eindigt op de vorige noot en eindigt met transformatie f (en dus minstens zijn ML laatste noten met die transformatie is bekomen aangezien we al stelden dat het pad geldig was) uit te breiden met weer dezelfde transformatie f . Tot slot is er ook nog de mogelijkheid om eender welk optimaal en geldig pad van lengte ML korter 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 uitvoering van het algoritme worden er 8 hulpvariabelen (allemaal arrays) gebruikt die hieronder opgelijst staan, met korte uitleg over hun betekenis.

- **prob_keep_past:** Houdt de probabiliteiten bij voor de optimale en geldige paden die eindigen op een van de laatste ML bekeken noten. En dit enkel voor de paden die eindigen zonder transformatie van de laatste noot.
- **prob_keep_current:** Houdt de probabiliteit bij voor het optimaal en geldig pad dat eindigt op de huidig beschouwde noot. En dit enkel voor de paden die eindigen zonder transformatie van hun laatste noot.
- **prob_transform_past:** Houdt voor elke toegelaten transformatie afzonderlijk de probabiliteiten bij voor de optimale en geldige paden die eindigen met deze transformatie. En dit voor al zo een paden die eindigen op een mogelijke transformatie van een van de laatste ML beschouwde noten.
- **prob_transform_current:** Houdt voor elke toegelaten transformatie afzonderlijk de probabiliteiten bij voor de optimale en geldige paden die eindigen met deze transformatie. En dit voor al zo een paden die eindigen op de huidig beschouwde noot.
- **path_end_on_keep_past:** Houdt de optimale en geldige paden bij die horen bij de probabiliteiten weergegeven in *prob_keep_past*.

- **path_end_on_keep_current:** Houdt het optimale pad bij dat hoort bij de probabilmiteit weergegeven in *prob_keep_current*.
- **prob_end_on_transform_past:** Houdt de optimale en geldige paden bij, horende bij de probabilmiteiten van *prob_transform_past*.
- **prob_end_on_transform_current:** Houdt de optimale en geldige paden bij, horende bij de probabilmiteiten van *prob_transform_current*.

Ter verduidelijking worden de vormen en groottes van deze variabelen visueel voorgesteld met een aantal illustraties. In figuur 5.3 worden de datastructuren weergegeven die betrekking hebben tot het ‘niet-transformeren’ van de laatst beschouwde noot. In figuur 5.4 worden de datastructuren weergegeven die betrekking hebben tot het transformeren van de laatst beschouwde noot. Voor elke toegestane transformatie (in het totaal ‘AT’ aantal transformaties) apart wordt er dezelfde soort info bijgehouden als voor het geval van ‘niet-transformatie’. In figuren 5.5 en 5.6 wordt weergegeven hoe de optimale (en geldige) paden bijgehouden worden die horen bij de probabilmiteiten uit de vorige twee figuren. In de twee ‘past’ tabellen uit deze twee figuren valt nog op te merken dat een pad dat rechts van een ander pad in zo een tabel staat, telkens exact een element langer is dan het andere. Dit komt omdat er voor dat optimale pad dat rechts staat al een element extra beschouwd is, dat dan al dan niet getransformeerd kan zijn.

Het valt op dat in dit algoritme optimale paden bijgehouden worden afhankelijk van met welke specifieke transformatie hun laatste noot bekomen is. Dit in tegenstelling tot het vorige algoritme waar er een gemeenschappelijke *matrix* werd opgesteld waaruit het optimale pad later geconstrueerd kon worden. De reden dat zo een matrix wel werke in het vorige algoritme was precies omdat er geen eis van minimum lengte was voor een transformatie. Dit betekent dat voor eender welke noot, er kan besloten worden enkel afgaande op de probabilmiteiten van de mogelijke toonhoogtes die de vorige noot kan aannemen, wat de probabilmiteiten zijn van de toonhoogtes waarnaar de huidige noot getransformeerd kan worden. Stel nu dat we voor het algoritme besproken in dit onderdeel, dezelfde voorstelling zouden willen gebruiken. Dan zijn er twee problemen waar we op botsen. Ten eerste, als we willen transformeren moet dat voor minsten ML opeenvolgende noten zijn, het volstaat dus niet meer om gewoon naar de vorige laag te kijken in de matrix om de nieuwe te bepalen, we gaan ook ML eenheden verder moeten teruggaan om op suboplossingen die voldoende korter zijn ineens ML keer de transformatie toe te passen. Het volgende probleem is dat we zelfs als we dan een nieuw pad vinden, we dit (voorlopig) optimale pad niet zomaar kunnen inschrijven in de *matrix*, aangezien we dan ook andere paden zouden kunnen aanpassen die daardoor mogelijks een deel van hun pad getransformeerd zouden zien over een lengte korter dan ML.

Prob_keep_past			Prob_keep_current	
$P_{(n-ML),0}$...	$P_{(n-1),0}$	$P_{n,0}$	
$P_{(n-ML),1}$...	$P_{(n-1),1}$	$P_{n,1}$	
$P_{(n-ML),2}$...	$P_{(n-1),2}$	$P_{n,2}$	
$P_{(n-ML),3}$...	$P_{(n-1),3}$	$P_{n,3}$	
$P_{(n-ML),4}$...	$P_{(n-1),4}$	$P_{n,4}$	
$P_{(n-ML),5}$...	$P_{(n-1),5}$	$P_{n,5}$	
$P_{(n-ML),6}$...	$P_{(n-1),6}$	$P_{n,6}$	
$P_{(n-ML),7}$...	$P_{(n-1),7}$	$P_{n,7}$	
$P_{(n-ML),8}$...	$P_{(n-1),8}$	$P_{n,8}$	
$P_{(n-ML),9}$...	$P_{(n-1),9}$	$P_{n,9}$	
$P_{(n-ML),10}$...	$P_{(n-1),10}$	$P_{n,10}$	
$P_{(n-ML),11}$...	$P_{(n-1),11}$	$P_{n,11}$	
$P_{(n-ML),12}$...	$P_{(n-1),12}$	$P_{n,12}$	
$P_{(n-ML),13}$...	$P_{(n-1),13}$	$P_{n,13}$	

ML

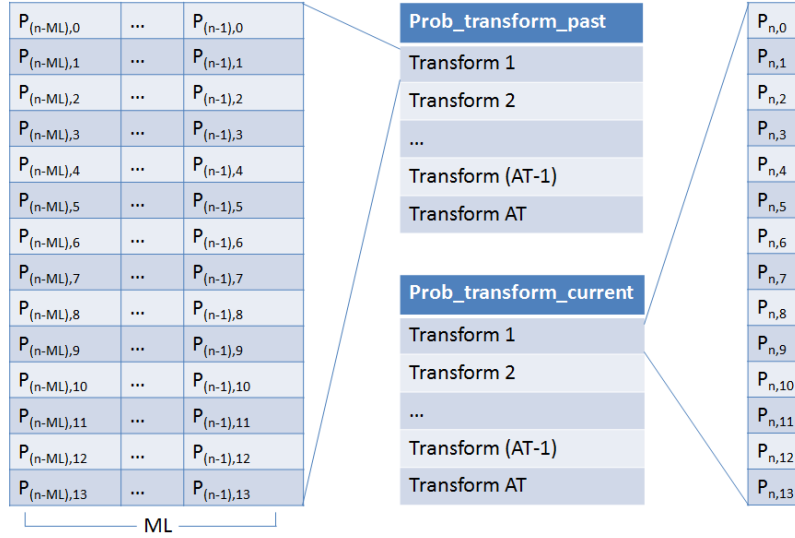
FIGUUR 5.3: Illustratie van de variabelen die probabiliteiten opslaan met betrekking tot het ‘niet-transformeren’ van de laatste noot. Elk element in deze tabel stelt een (logaritme van een) probabieliteit voor.

Initialisatie

De initialisatie verloopt vrij eenvoudig. Alle variabelen die optimale paden bijhouden worden geïnitieerd op een lege array. Aangezien er voor de uitvoer van het algoritme uiteraard nog geen deel van een pad berekend is. De variabelen die betrekking hebben op de probabiliteiten van de optimale paden worden allemaal op een zeer lage waarde geïnitieerd. Dit behalve voor variabele *prob_keep_past*, waarvan de waarde horende bij de probabieliteit op het behouden van de eerste noot op 0 wordt gezet (wat overeenkomt met een kans van 1 aangezien de waarden waarmee gerekend wordt, de logaritmen zijn van de eigenlijke waarden). Dit omdat de eerste noot van een melodieline altijd behouden wordt, want om een transformatie in rekening te brengen moet er een vorige noot zijn die niet bestaat voor de eerste noot.

Een stap in het algoritme

Elke stap in het algoritme komt overeen met het berekenen voor de optimale paden van het algoritme voor de melodieline die een noot meer van het originele stuk bevat dan in de vorige stap. Om deze optimale paden te berekenen wordt gebruik gemaakt van de optimale paden van de vorige stappen in het algoritme. Deze worden dan zo



FIGUUR 5.4: Illustratie van de variabelen die probabiliteiten opslaan met betrekking tot het transformeren van de laatste noot. Elk element in deze tabel stelt een (logaritme van een) probabilliteit voor.

efficiënt mogelijk uitgebreid tot aan de huidig beschouwde noot. In de rest van deze paragraaf wordt er vanuit gegaan dat we momenteel bij noot n in het muziekstuk zitten in het algoritme. En we zoeken nu dus de optimale paden voor het toepassen van het algoritme op de eerste $n+1$ noten (aangezien de eerste noot, noot 0 is).

Een eerste mogelijkheid om een optimaal pad te vinden van $n+1$ noten is om een van de optimale paden van lengte n uit te breiden met een ‘niet-transformatie’ door de noot op positie n in het oorspronkelijk melodie gewoon te behouden. In dit geval maakt het niet uit of zo een optimaal pad van lengte n eindigt op een specifieke transformatie of niet. Indien zo een nieuw pad een hogere waarschijnlijkheid heeft dan het beste pad tot nu toe dan wordt dit pad opgeslagen in *path_end_on_keep_current* en de overeenkomstige probabilliteit wordt bijgehouden in *prob_keep_current*.

Een tweede mogelijkheid is om een nieuw pad van lengte $n+1$ te construeren dat eindigt om een van de toegelaten transformaties. Voor elke transformatie f kan er dan het volgende gedaan worden op mogelijke optimale paden te vinden. Er is een eerste mogelijkheid om een optimaal pad van lengte n dat al eindigde op dezelfde transformatie uit te breiden naar een pad van lengte $n+1$ met deze transformatie. Er is nog een tweede mogelijkheid waarbij een optimaal pad van lengte $n+1-ML$ wordt uitgebreid met ML transformaties volgens transformatie f . Dit om te voldoen aan de eisen van de minimum lengte van een transformatie. Deze paden van lengte $n+1-ML$ waarover gesproken wordt kunnen zowel eindigen op een van de andere transformaties als te eindigen op een ‘niet-transformatie’. Zolang ze daarna maar uitgebreid worden met ML keer de transformatie f . Ook hier zal bij al deze combinaties gekeken worden op de probabilliteit van het nieuwe geconstrueerde pad hoger ligt dan het beste pad tot nu toe. Indien dit het geval is wordt dit nieuwe pad

Path_end_on_keep_past			Path_end_on_keep_current	
$Pa_{(n-ML),0}$...	$P_{(n-1),0}$	$Pa_{n,0}$	
$Pa_{(n-ML),1}$...	$P_{(n-1),1}$	$Pa_{n,1}$	
$Pa_{(n-ML),2}$...	$P_{(n-1),2}$	$Pa_{n,2}$	
$Pa_{(n-ML),3}$...	$P_{(n-1),3}$	$Pa_{n,3}$	
$Pa_{(n-ML),4}$...	$P_{(n-1),4}$	$Pa_{n,4}$	
$Pa_{(n-ML),5}$...	$P_{(n-1),5}$	$Pa_{n,5}$	
$Pa_{(n-ML),6}$...	$P_{(n-1),6}$	$Pa_{n,6}$	
$Pa_{(n-ML),7}$...	$P_{(n-1),7}$	$Pa_{n,7}$	
$Pa_{(n-ML),8}$...	$P_{(n-1),8}$	$Pa_{n,8}$	
$Pa_{(n-ML),9}$...	$P_{(n-1),9}$	$Pa_{n,9}$	
$Pa_{(n-ML),10}$...	$P_{(n-1),10}$	$Pa_{n,10}$	
$Pa_{(n-ML),11}$...	$P_{(n-1),11}$	$Pa_{n,11}$	
$Pa_{(n-ML),12}$...	$P_{(n-1),12}$	$Pa_{n,12}$	
$Pa_{(n-ML),13}$...	$P_{(n-1),13}$	$Pa_{n,13}$	

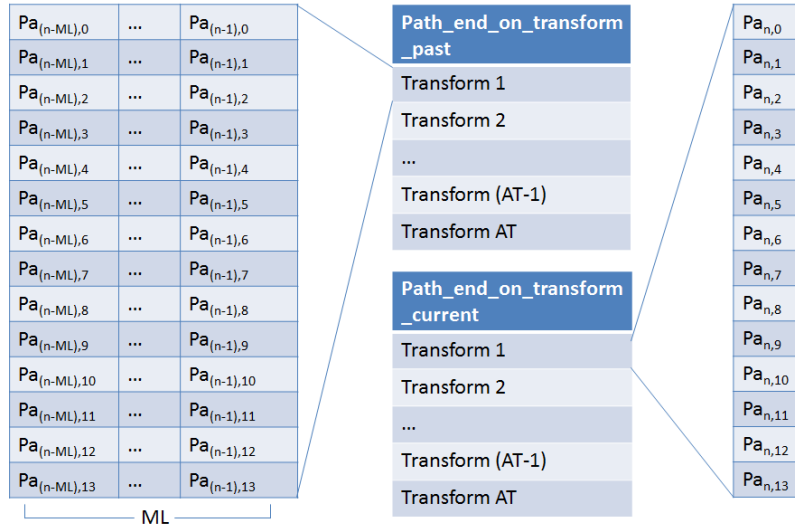
FIGUUR 5.5: Illustratie van de variabelen die optimale (geldige) paden opslaan met betrekking tot het ‘niet-transformeren’ van de laatste noot. Elk element in deze tabel stelt een optimaal en geldig pad voor.

opgeslagen in *path_end_on_transform_current* en de overeenkomstige probabilliteit wordt bijgehouden in *prob_transform_current*.

Als laatste gedeelte van elke stap van het algoritme worden, net zoals in het algoritme beschreven in de vorige sectie, de variabelen weer klaargemaakt voor de uitvoer van de volgende iteratie. Alle probabilliteiten en paden worden een tijdsstap doorgeschoven in hun overeenkomstige variabelen. Paden die meer dan ML korter zijn dan de huidige beschouwde lengte worden niet meer bijgehouden. Dus voor elke iteratie n zullen aan het eind van de iteratiestap alle paden met lengte $n+1-ML$ verwijderd worden omdat ze niet meer noodzakelijk zijn voor het vervolg van het algoritme.

Extractie van het beste pad

In tegenstelling tot het vorige algoritme waar het optimale pad nog moest bepaald worden door terug te lopen door de *matrix*, zal de extractie van het optimale pad na uitvoeren van het algoritme hier een stuk vlotter verlopen. Dit komt doordat de volledige optimale paden van de deelproblemen (en uiteindelijk dus ook van het gehele probleem) telkens opgeslagen worden in hulpvariabelen. Het enige wat moet gebeuren om het beste pad te vinden is dus kijken naar alle mogelijke optimale



FIGUUR 5.6: Illustratie van de variabelen die optimale (geldige) paden opslaan met betrekking tot het transformeren van de laatste noot. Elk element in deze tabel stelt een optimaal en geldig pad voor.

paden voor de volledige melodie. Voor elke mogelijke transformatie en voor elke mogelijke eindnoot die met deze transformatie kan bekomen worden gaan we zo een optimaal pad hebben. En van deze optimale paden gaat het pad gekozen worden met de hoogste waarschijnlijkheid. Dit pad is het gezochte optimale pad dat voldoet aan alle voorwaarden opgelegd aan het algoritme.

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 melodieline (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 $(L-ML)$. 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 invoersequentie 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 toegestane transformatie even veel optimale paden bijgehouden worden die eindigen op die zelfde 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 probabilliteit staat. Deze probabilliteit staat voor de gemiddelde probabilliteit 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 probabilliteit van een bepaald muziekstuk dan betekent dit de gemiddelde probabilliteit van een noot in dit muziekstuk.

6.1 Experimenten met 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 melodielyn 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 melodielyn (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: Resultaten van experiment 6.1.1. Gemiddelde consonantiescores voor de twee soorten transformaties en de originele melodielijnen die getransformeerd 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 melodielij 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 consistentere 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: Resultaten van experiment 6.1.2. Gemiddelde consonantiescores voor de twee Fibonacci transformaties en de originele melodielijnen die getransformeerd 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 melodieline. 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 Experimenten met het RPK-model

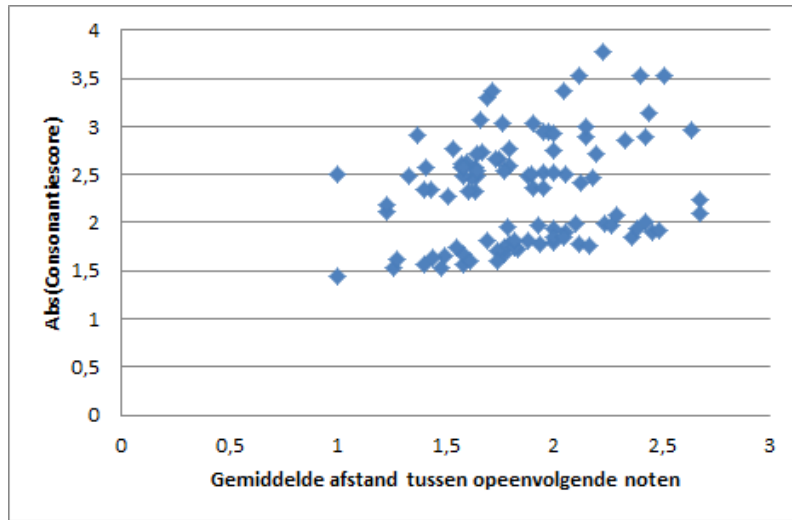
6.2.1 Afhangelijkheid score RPK-Model en gemiddelde afstand tussen opeenvolgende noten

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 resultaten 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 gemiddeld 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

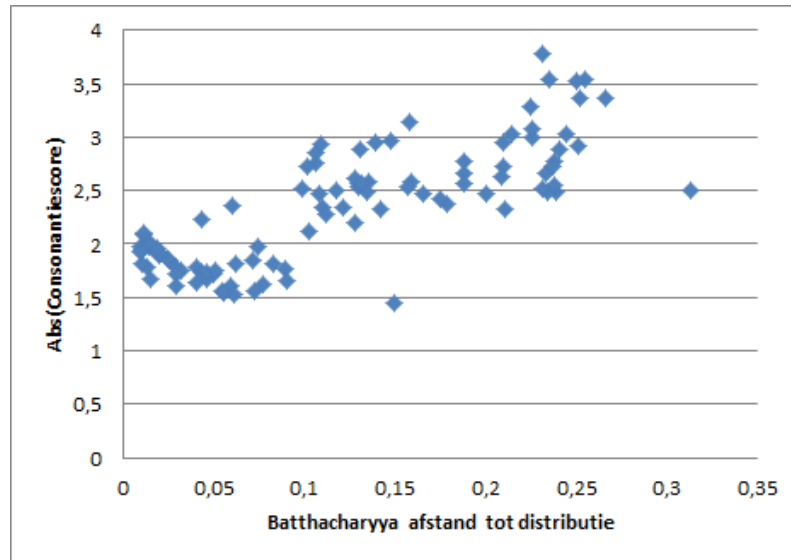
Beschrijving experiment

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 zogenaamde 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 probabiteit 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

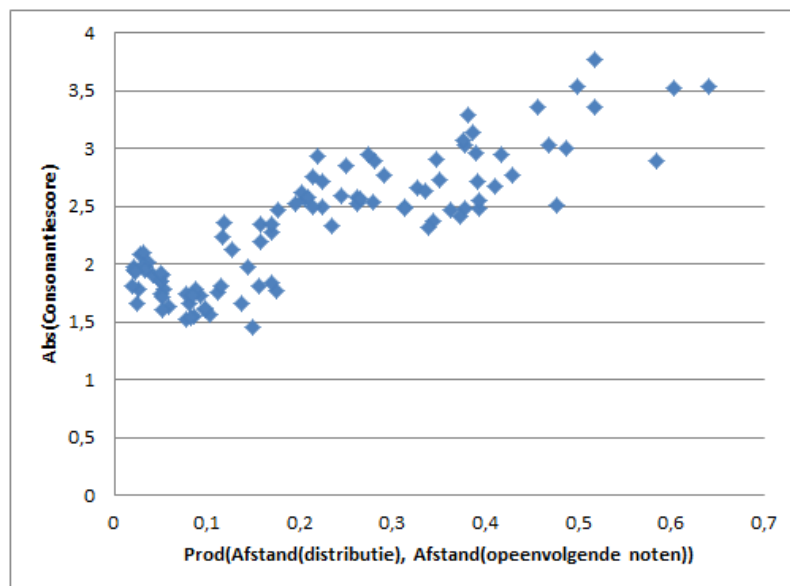
Beschrijving experiment

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 probabiteit 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 Experimenten over de 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.

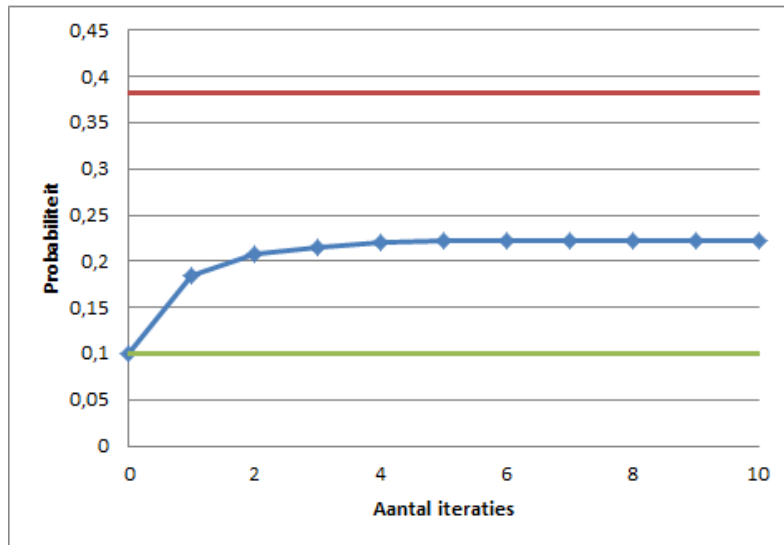
Deze test wordt uitgevoerd op 100 muziekstukken uit het Essencorpus. De gemiddelde probabilliteit van de originele stukken wordt berekend alsook de gemiddelde probabilliteit van het muziekstuk dat optimaal is volgens het RPK-model in de toonaard van de 100 stukken. Nu kan er gekeken worden naar hoe snel 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 resultaten weergegeven van dit experiment. De groene lijn op de figuur geeft de gemiddelde probabilliteit 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 probabilliteit 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 probabilliteit 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 mogelijkheden om een bepaalde noot te transformeren, en kunnen bijgevolg nog steeds de meeste noten niet bereikt worden vanuit eender welke noot.

6.3. Experimenten over de werking van de algoritmes voor het combineren van transformaties



FIGUUR 6.4: Resultaten van het experiment uitgevoerd in deel 6.3.1. De groene lijn staat voor de gemiddelde probabiteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabiteit van de theoretisch beste melodielijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabiteit 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

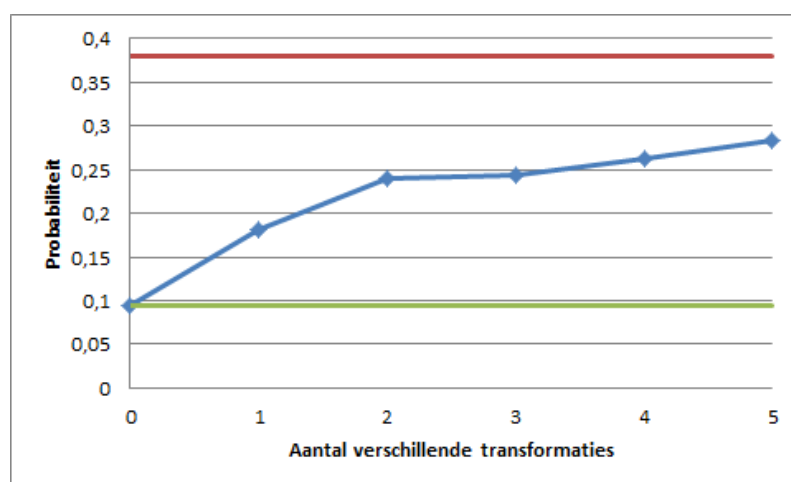
Beschrijving experiment

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 probabiteit van de originele stukken wordt berekend alsook de gemiddelde probabiteit 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 probabilliteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabilliteit van de theoritisch beste melodieliijn in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabilliteit 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 resultaten 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 probabilliteit 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 probabilliteiten kunnen bekomen worden dan in het geval waarbij het aantal iteraties verhoogd wordt in plaats van het aantal mogelijke transformaties.

6.3. Experimenten over de werking van de algoritmes voor het combineren van 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.

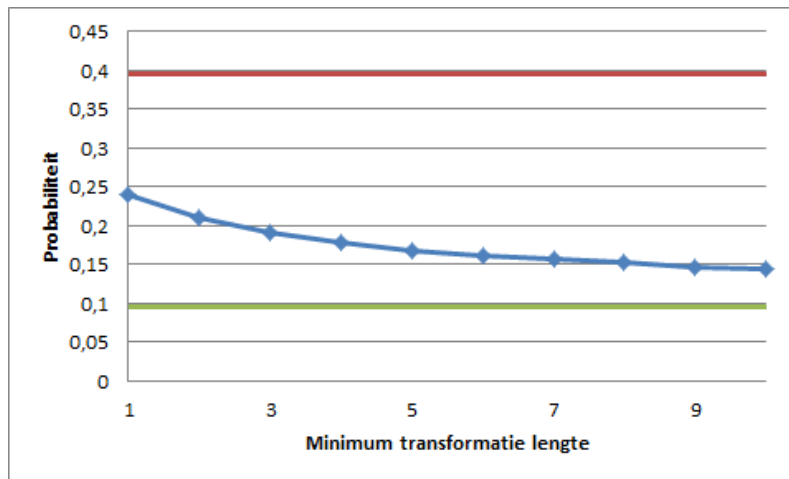
6.3.3 Transformaties combineren: minimum transformatie lengte

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 probabiteit 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 probabiteit van een noot in de originele melodie, de rode lijn voor de gemiddelde probabiteit van de theoretisch beste melodieline in de toonaarden waarop getest werd en de blauwe lijn geeft de gemiddelde probabiteit 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 probabilliteit) 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.

6.4. Experimenten over de tijdscomplexiteit van de algoritmes voor het combineren van transformaties

# 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: Resultaten 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 Experimenten over de tijdscomplexiteit van de algoritmes voor het combineren van transformaties

6.4.1 Tijdsafhankelijkheid algoritme 5.2 van het aantal noten in de melodielyn

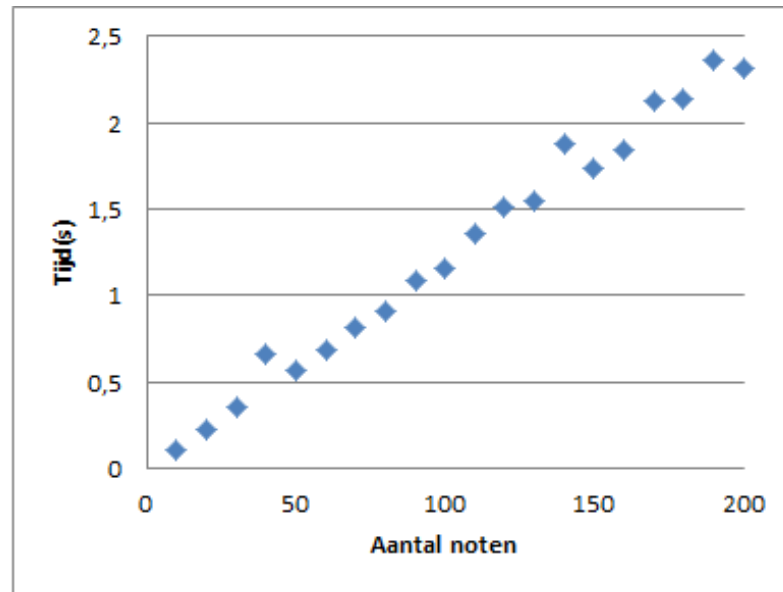
Beschrijving experiment

Van het algoritme beschreven in onderdeel 5.2 werd reeds beargumenteerd dat de uitvoertijd lineair afhankelijk is van het aantal noten van de originele melodielyn. 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 melodielynen (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 melodielyn 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 van 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..

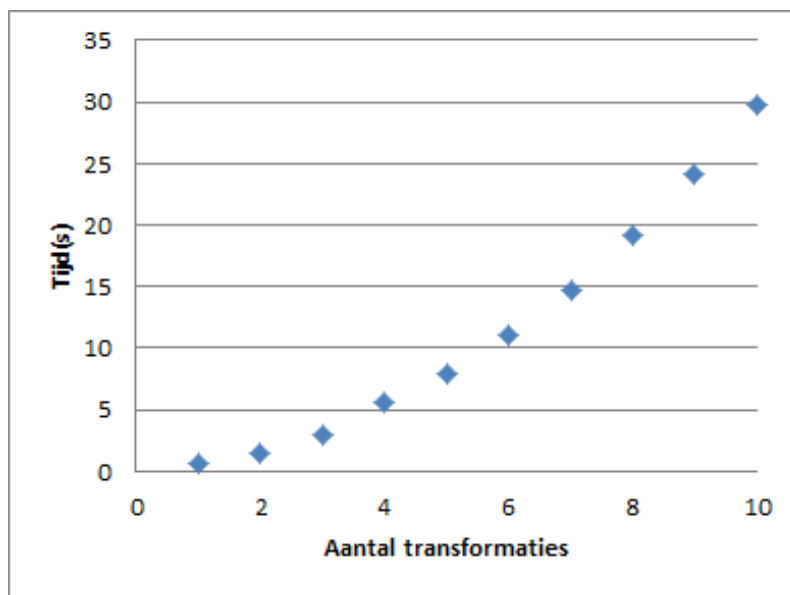
Resultaten

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

6.4. Experimenten over de tijdscomplexiteit van de algoritmes voor het combineren van 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
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.

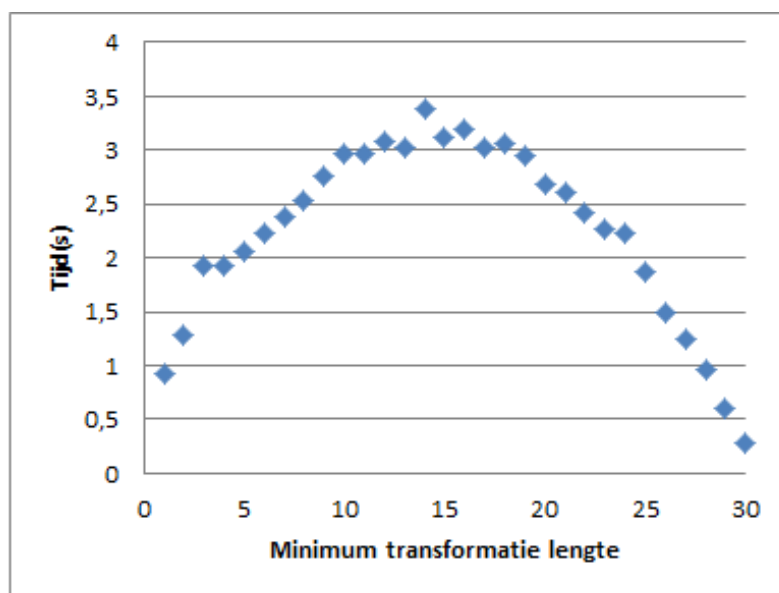
6.4.3 Tijdsafhankelijkheid algoritme 5.2 van minimum transformatie lengte

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.

6.4. Experimenten over de tijdscomplexiteit van de algoritmes voor het combineren van transformaties

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

Een hoofdstuk behandelt een samenhangend geheel dat min of meer op zichzelf staat. Het is dan ook logisch dat het begint met een inleiding, namelijk het gedeelte van de tekst dat je nu aan het lezen bent.

7.1 Eerste onderwerp in dit hoofdstuk

De inleidende informatie van dit onderwerp.

7.1.1 Een item

Een tekst staat nooit alleen. Dit wil zeggen dat er zeker ook referenties nodig zijn. Dit kan zowel naar on-line documenten[16] als naar boeken[9].

7.2 Figuren

Figuren worden gebruikt om illustraties toe te voegen. Dit is dan ook de manier om beeldmateriaal toe te voegen zoals getoond wordt in figuur 7.1.

7.3 Tabellen

Tabellen kunnen gebruikt worden om informatie op een overzichtelijke te groeperen. Een tabel is echter geen rekenblad! Vergelijk maar eens tabel 7.1 en tabel 7.2. Welke tabel vind jij het duidelijkst?



FIGUUR 7.1: Het KU Leuven logo.

gnats	gram	\$13.65
	each	.01
gnu	stuffed	92.50
emu		33.33
armadillo	frozen	8.99

TABEL 7.1: Een tabel zoals het niet moet.

Item		
Animal	Description	Price (\$)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99

TABEL 7.2: Een tabel zoals het beter is.

7.4 Besluit van dit hoofdstuk

Als je in dit hoofdstuk tot belangrijke resultaten of besluiten gekomen bent, dan is het ook logisch om het hoofdstuk af te ronden met een overzicht ervan. Voor hoofdstukken zoals de inleiding en het literatuuroverzicht is dit niet strikt nodig.

Bijlagen

Bijlage A

Broncode

A.1 Neuraal Network 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
transform 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))

```

A. BRONCODE

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

Bijlage B

IEEE_Paper

In de bijlagen vindt men de data terug die nuttig kunnen zijn voor de lezer, maar die niet essentieel zijn om het betoog in de normale tekst te kunnen volgen. Voorbeelden hiervan zijn bronbestanden, configuratie-informatie, langdradige wiskundige afleidingen, enz.

In een bijlage kunnen natuurlijk ook verdere onderverdelingen voorkomen, evenals figuren en referenties[\[4\]](#).

Bijlage C

Poster

Muziek Compositie via Melodische Transformaties

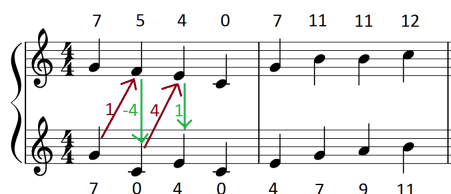
Probleemstelling

- 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 melodieliijn

Gebruik transformatie voor verhoging consonantie melodie

Doel:

- Voor elke noot in de melodieliijn: 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: $O(AN \times AT^2)$** **Geheugen: $O(AN \times ML \times AT)$**
 - 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

Bibliografie

- [1] Backpropagation. URL: <https://en.wikipedia.org/wiki/Backpropagation>.
- [2] David temperley. URL: <http://davidtemperley.com/>.
- [3] Musicxml. URL: <http://www.musicxml.com/>.
- [4] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Del Rey (reprint), 1995. ISBN-13: 978-0345391803.
- [5] G. Loy. *Musimathics*. The MIT Press, Cambridge, Massachusetts, 2006.
- [6] W. MathWorld. Fibonacci number. URL: <http://mathworld.wolfram.com/FibonacciNumber.html>.
- [7] M. Nielsen. *Neural Networks and Deep Learning*. 2016.
- [8] U. of Oxford Text Archive. Essen corpus of german folksong melodies. URL: <http://ota.ox.ac.uk/desc/1038>.
- [9] T. Pratchett and N. Gaiman. *Good Omens: The Nice and Accurate Prophecies of Agnes Nutter, Witch*. HarperTorch (reprint), 2006. ISBN-13: 978-0060853983.
- [10] D. Temperley. *Music and Probability*. The MIT Press, Cambridge, Massachusetts, 2007.
- [11] S. University. Feed-forward networks. URL: <http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Architecture/feedforward.html>.
- [12] Wikipedia. Bhattacharyya distance. URL: https://en.wikipedia.org/wiki/Bhattacharyya_distance.
- [13] Wikipedia. Dynamic programming. URL: https://en.wikipedia.org/wiki/Dynamic_programming.
- [14] Wikipedia. Johann sebastian bach. URL: https://en.wikipedia.org/wiki/Johann_Sebastian_Bach.
- [15] Wikipedia. Musical instrument digital interface. URL: https://nl.wikipedia.org/wiki/Musical_Instrument_Digital_Interface.

- [16] Wikipedia. Scriptie. URL: <http://nl.wikipedia.org/wiki/Masterproef>,
laatst nagekeken op 2010-01-07.
- [17] Wikipedia. Wolfgang amadeus mozart. URL: [https://en.wikipedia.org/
wiki/Wolfgang_Amadeus_Mozart](https://en.wikipedia.org/wiki/Wolfgang_Amadeus_Mozart).

Fiche masterproef

Student: Elias Moons

Titel: Melodische transformatie en evaluatie van muziek

Engelse titel: Melodical transformation and evaluation of music

UDC: 004.9

Korte inhoud:

Deze masterproef beschrijft methodes om melodielijnen van een muziekstuk melodisch te transformeren to nieuwe melodielijnen. Verder wordt er ook aandacht besteed aan een referentiekader waarin deze transformaties geëvalueerd kunnen worden. Tot slot wordt er ook nog 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 is een algoritme beschreven dat gebaseerd is op de principes van *dynamic programming*. Dit algoritme zal, gegeven een aantal mogelijke transformaties en een melodieliijn, de best mogelijke getransformeerde melodieliijn 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

Assessor: Prof. Onbekend

Begeleider: Ir. V. Nys