

# Inhoudsopgave

<b>Lijst van figuren</b>	<b>iv</b>
<b>Listings</b>	<b>v</b>
<b>1 Inleiding</b>	<b>1</b>
1.1 Situering . . . . .	1
1.1.1 Historisch overzicht . . . . .	2
1.1.2 Bestaand werk . . . . .	3
1.2 Doelstellingen . . . . .	3
1.3 Overzicht van de tekst . . . . .	4
<b>2 Muziekrepresentatie</b>	<b>5</b>
2.1 Klassieke muzieknotatie . . . . .	5
2.2 Wat wel en wat niet? . . . . .	6
2.2.1 Melodie . . . . .	7
2.2.2 Samenklank . . . . .	8
2.2.3 Tijd en nadruk . . . . .	9
2.2.4 Details . . . . .	10
2.2.5 Besluit . . . . .	10
2.3 Bestaande representaties . . . . .	11
2.3.1 MIDI . . . . .	11
2.3.2 LilyPond . . . . .	11
2.3.3 Abc notatie . . . . .	13
2.3.4 SoundTracker en afgeleiden . . . . .	13
2.3.5 Humdrum <b>**kern</b> . . . . .	13
2.4 Interwoven Voices List . . . . .	15
2.4.1 Intuïtie . . . . .	16
2.4.2 Definitie . . . . .	16
2.4.3 Voorbeeld . . . . .	17

<b>3</b>	<b>Programmeren in PRISM</b>	<b>18</b>
3.1	PRISM-modellen . . . . .	18
3.1.1	Probabilistische predicaten . . . . .	19
3.1.2	Sample execution subsystem . . . . .	19
3.1.3	Learning subsystem . . . . .	20
3.2	Voor- en nadelen van PRISM . . . . .	22
3.2.1	Voordelen . . . . .	22
3.2.2	Nadelen . . . . .	23
<b>4</b>	<b>Hidden Markov Models</b>	<b>24</b>
4.1	Markov Modellen . . . . .	24
4.2	Verborgen toestanden: HMM's . . . . .	26
4.2.1	Kansparameters . . . . .	27
4.2.2	Topologie . . . . .	27
4.3	Basisalgoritmes voor HMM's . . . . .	28
4.3.1	Forward Pass . . . . .	28
4.3.2	Viterbi . . . . .	29
4.4	Leren van kansparameters . . . . .	29
4.4.1	Basisidee . . . . .	30
4.4.2	Baum-Welch algoritme . . . . .	30
4.5	Implementatie in PRISM . . . . .	32
<b>5</b>	<b>Een PRISM-model voor IVL-muziek</b>	<b>36</b>
5.1	Basismodel . . . . .	36
5.2	Verfijning van het basismodel . . . . .	37
5.2.1	Correcte <code>old/new</code> uitvoer . . . . .	37
5.2.2	Opsplitsen <code>out_note</code> . . . . .	39
5.2.3	Transponeren naar een gemeenschappelijke toonaard . . . . .	41
5.2.4	Harmonische beperkingen . . . . .	41
5.2.5	Voice states . . . . .	42
5.3	Beperkingen op de leer-voorbeelden . . . . .	43
5.3.1	Beperkt aantal stemmen . . . . .	44
5.3.2	Beperkte lengte . . . . .	44
5.4	Besluit . . . . .	45
5.4.1	Samenvatting . . . . .	45
5.4.2	Mogelijke verdere verfijningen . . . . .	46

---

<b>6</b>	<b>Experimenten en resultaten</b>	<b>47</b>
6.1	Automatische classificatie . . . . .	47
6.1.1	Proefopstelling . . . . .	48
6.1.2	Resultaten . . . . .	49
6.1.3	Kritische noot in verband met de duur $L_i$ . . . . .	49
6.2	Automatisch componeren . . . . .	51
6.2.1	Proefopstelling . . . . .	52
6.2.2	Resultaten . . . . .	53
<b>7</b>	<b>Besluit</b>	<b>55</b>
7.1	Samenvatting . . . . .	55
7.1.1	Eerste helft: voorbereiding . . . . .	55
7.1.2	Tweede helft: het echte werk . . . . .	56
7.2	Bereikte resultaten . . . . .	56
7.3	Verder werk . . . . .	57
<b>A</b>	<b>Overzicht gebruikte software</b>	<b>58</b>
<b>B</b>	<b>Omzetting tussen IVL en LilyPond</b>	<b>59</b>
B.1	IVL naar LilyPond . . . . .	59
B.2	LilyPond naar IVL . . . . .	61
B.2.1	LilyPond naar VL . . . . .	61
B.2.2	VL naar IVL . . . . .	63
<b>C</b>	<b>PRISM-code van het muziekmodel</b>	<b>66</b>
	<b>Bibliografie</b>	<b>71</b>

# Lijst van figuren

1.1	Overzicht van de hoofdstukken . . . . .	4
2.1	LilyPond-omzetting van listing 2.1 naar klassieke notatie . . . . .	12
2.2	Hetzelfde voorbeeld als in figuur 2.1, in abc notatie . . . . .	12
2.3	Screenshot van SoundTracker 0.6.7-pre6 . . . . .	14
2.4	Voorbeeld uit figuur 2.1 in SoundTracker-notatie . . . . .	14
2.5	Voorbeeld uit figuur 2.1 in de Humdrum <b>**kern</b> notatie . . . . .	15
3.1	Een mogelijk resultaat na 15 keer opgooien . . . . .	21
3.2	Uitvoer van het leer-algoritme van PRISM . . . . .	21
3.3	Opstartscherm van PRISM . . . . .	23
4.1	Observatie- en overgangskansen voor de springende robot . . . . .	33
4.2	Twee mogelijke observaties van 20 robot-acties . . . . .	33
4.3	Mogelijk resultaat van <code>learn</code> met als invoer <code>robot-voorbeelden.data</code> . . . . .	34
6.1	Te classificeren muziekfragmenten . . . . .	50
6.2	Classificatie van 30 muziekfragmenten . . . . .	50
6.3	Classificatie zonder rekening te houden met duur $L_i$ . . . . .	52
6.4	Automatische compositie gebaseerd op Bach (zonder VS) . . . . .	53
6.5	Automatische compositie gebaseerd op Mozart (zonder VS) . . . . .	53
6.6	Automatische compositie gebaseerd op Bach (3 VS) . . . . .	54
6.7	Automatische compositie gebaseerd op Mozart (3 VS) . . . . .	54

# Listings

2.1	Voorbeeld van de LilyPond-syntax . . . . .	12
3.1	voorbeeld.psm . . . . .	19
4.1	robot.psm . . . . .	32
4.2	robot-voorbeelden.data . . . . .	34
5.1	music-1.psm . . . . .	38
5.2	check_new/4 . . . . .	39
5.3	check_note/3 . . . . .	40
5.4	out_modnote . . . . .	42
B.1	unweave.pl . . . . .	59
B.2	Midi_walker::process() . . . . .	62
B.3	Moment::to_int() . . . . .	62
B.4	weave.pl . . . . .	63
C.1	music.psm . . . . .	66

# Hoofdstuk 1

## Inleiding

*“I don’t know anything about music.  
In my line you don’t have to.”*  
– Elvis Presley (1935-1977)

Componisten van muziek zijn gebonden aan bepaalde – grotendeels ongeschreven – regels, die afhankelijk zijn van de stijl waarin gecomponeerd wordt. Muziekkennis kunnen we beschouwen als het geheel van deze muzikale regels. Deze verhandeling gaat over het modelleren van muziekkennis: het construeren van een model dat op één of andere manier bepaalde muziekkennis beschrijft.

Een bondige situering van de materie van deze verhandeling geven we in een eerste sectie. We geven een historisch overzicht van dit domein (subsectie 1.1.1, gebaseerd op [Con03]) en een schets van recent gelijkaardig werk (subsectie 1.1.2). Vervolgens formuleren we in sectie 1.2 enkele doelstellingen voor deze verhandeling. Tenslotte wordt in sectie 1.3 een overzicht van de verschillende hoofdstukken gegeven.

### 1.1 Situering

Er zijn veel formalismen bekend om strikte, logische regels in uit te drukken. Een voorbeeld daarvan is *logisch programmeren*, zoals de taal Prolog. De meeste stijlen zijn echter niet te vatten in een hoop strikte regels. Inherent aan muziek is namelijk een zekere ‘willekeurigheid’, een soort ‘organische ruis’. Een beetje imperfectie en onnauwkeurigheid zijn in de muziek noodzakelijk om perfectie te bereiken. Dat geldt niet enkel voor de artistieke uitvoering van een muziekstuk (die we hier niet zullen beschouwen) maar ook voor het muziekstuk zelf – de noten op de partituur. Een model met enkel puur logische regels is dus niet zo geschikt om muziekkennis te beschrijven.

In deze verhandeling werken we in een probabilistische context. Muzikale regels zullen we trachten voor te stellen als kansverdelingen van stochastische experimenten. In plaats van handmatig muzikale regels te zoeken en te implementeren zullen we proberen deze regels automatisch af te leiden uit bestaande muziek. Meer bepaald zullen we van een Hidden Markov Model benadering uitgaan. *Probabilistisch logisch programmeren* is in feite logisch programmeren uitgebreid met kansrekening, zodat zowel probabilistische als

relationele kennis op een natuurlijke manier kan uitgedrukt worden. Dit lijkt dan ook een geschikt formalisme om een dergelijk model mee op te stellen.

### 1.1.1 Historisch overzicht

Al in de begindagen van computers hebben psychologen, musicologen, informatici en ingenieurs getracht een automatisch systeem te ontwikkelen om muziek te analyseren, te genereren en beter te begrijpen. Verschillende soorten modellen werden gebruikt om muzikale kennis te beschrijven.

#### Muziekmodellen en modellen voor natuurlijke taal

Het onderzoek naar muziekmodellen heeft een geschiedenis die op veel vlakken gelijkloopt met dat naar modellen voor natuurlijke taal. In de beginjaren (de jaren 1950 en 1960) werden voornamelijk statistische modellen gebruikt, zoals Markov modellen. Al gauw bleek echter dat het construeren van statistische modellen voor muziek-generatie veel moeilijker was dan men in eerste instantie veronderstelde.

Daarom, maar ook omwille van Chomsky's kritiek op het gebruiken van Markov modellen voor taal, verschoof de aandacht in de jaren 1970 en 1980 naar het ontwikkelen van krachtigere grammatica's voor muziekgeneratie. De statistische, empirische component werd daarbij verwaarloosd. 'Ambachtelijk' gemaakte modellen, die slechts een kleine – of zelfs geen – statistische component hadden, werden gebruikt om muziek te genereren. De muzikale ervaring van de ontwerper werd in feite omgezet in een specifieke muzikale grammatica. Een recent voorbeeld van zo'n model wordt uitgewerkt in [Vra03].

In de jaren 1990 kwam er bij de studie van modellen voor natuurlijke taal terug een verschuiving naar statistische modellen, gemotiveerd door de opmerkelijke resultaten daarvan bij spraakherkenning. Die verschuiving was er ook in het onderzoek naar muziekmodellen, gedeeltelijk omwille van de groeiende beschikbaarheid van on-line muziek-gegevensbanken.

#### Synthetische en Analytische modellen

Traditioneel wordt een onderscheid gemaakt tussen *synthetische* en *analytische* modellen voor muziek. Synthetische modellen worden gebruikt om muziek te *genereren* (m.a.w. automatisch te componeren) volgens een bepaalde grammatica van muzikale regels. Analytische modellen zijn ontworpen om o.a. muziekstukken te *classificeren* op genre, door na te gaan of ze aan een bepaalde muzikale grammatica voldoen.

Het onderzoek naar modellen voor natuurlijke taal was van in het begin vooral gefocust op het ontwikkelen van analytische modellen – bedoeld om zinnen te classificeren als grammaticaal correct of niet correct. Slechts een relatief klein deel van het onderzoek gaat naar het generen van natuurlijke taal.

Bij het onderzoek naar modellen voor muziek liggen de verhoudingen omgekeerd. In de beginjaren werd vooral onderzoek gedaan naar synthetische modellen – waarschijnlijk omdat de praktische toepassing van analytische muziekmodellen tot voor kort minder duidelijk was dan die van analytische taalmodellen. Het genereren van muziek, met resul-

taten die onmiddellijk kunnen beluisterd worden, is een boeiend en aantrekkelijk onderzoeksproject. Pas recentelijk schenken onderzoekers aandacht aan analytische modellen, bijvoorbeeld voor muziek-classificatie.

Muziekgeneratie en -analyse worden tegenwoordig als toepassingen van hetzelfde algemeen model gezien [Con03]. Het traditionele onderscheid tussen analytische en synthetische modellen valt daardoor weg. Eens we een analytisch statistisch model hebben, dat hoge kansen toekent aan stukken in een bepaald genre, kunnen we muziekgeneratie gelijkstellen aan bemonstering (sampling) van dat model. Dit is ook de benadering die we zullen volgen in deze verhandeling.

### 1.1.2 Bestaand werk

Zoals Conklin terecht opmerkt is een statistisch model voor classificatie in principe ook bruikbaar voor het genereren van muziek. Een overzicht van verschillende dergelijke statistische modellen voor classificatie van muziek en mogelijke manieren om ze te *samplen* wordt gegeven in [Con03].

Het concept van Markov-ketens is al vaak toegepast voor automatische compositie. Vaak wordt er vertrokken van een model waarvan de parameters handmatig gekozen worden om een bepaalde muziekstijl te verkrijgen. In deze verhandeling zullen we echter vertrekken van bestaande muziek en de model-parameters schatten op basis daarvan – zoals Marom deed met zijn model voor Jazz-improvisaties [Mar97].

Pollastri en Simoncelli [PS01] pasten succesvol Hidden Markov Models toe om melodieën van vijf verschillende componisten te classificeren. Ze vergeleken de resultaten van de HMM-classificatie met die van muziek-experten en met die van amateurs en kwamen tot de conclusie dat HMM's het bijna even goed doen als experts.

## 1.2 Doelstellingen

In het meeste bestaand werk worden modellen gemaakt voor éénstemmige melodieën. We zouden graag een algemeen model hebben dat ook met meerstemmige muziekstukken overweg kan. Verder wensen we een model waarbij probabilistische regels geleerd kunnen worden op basis van *bestaande muziek*, in tegenstelling tot bijvoorbeeld modellen zoals [Vra03] die enkel op strikte regels – die *door de ontwerper* beschreven werden – gebaseerd zijn. We laten ons daarbij inspireren door HMM's.

Als we een dergelijk algemeen muziekmodel geconstrueerd hebben, voeren we er enkele experimenten mee uit. We proberen daarbij aan te tonen dat ons model zowel geschikt is voor analyse-doeleinden (bijvoorbeeld het classificatie-probleem) als voor synthese (automatisch componeren). Dit zou de these van Conklin bevestigen.

We willen met deze verhandeling tenslotte ook nog het nut en de kracht van logisch-probabilistische talen zoals PRISM illustreren.

## 1.3 Overzicht van de tekst

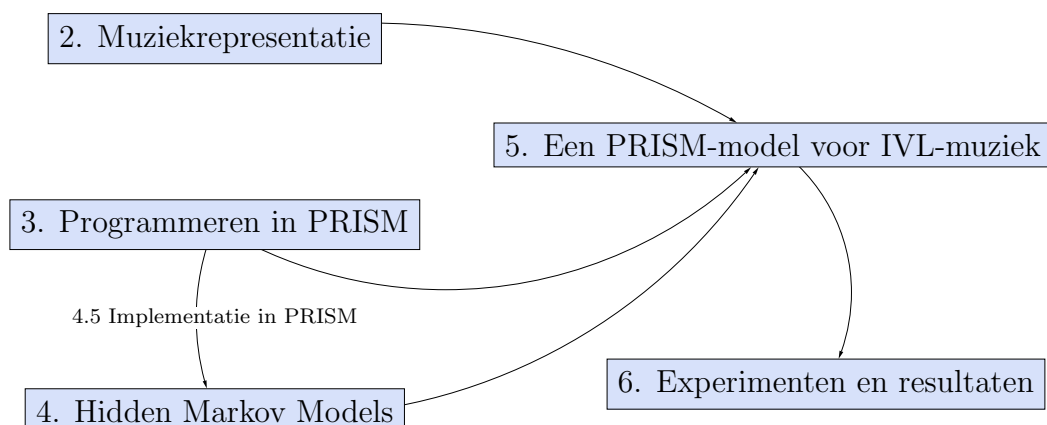
We beginnen met een hoofdstuk over representatie van muziek (hoofdstuk 2), waarin de keuze van de aspecten van muziek die we zullen modelleren wordt gemotiveerd. Tevens wordt een bondige bespreking van bestaande notaties en bestandsformaten gegeven. Dit hoofdstuk eindigt met de definitie van een nieuwe notatie : *Interwoven Voices List* (IVL). Voor wie al enigszins vertrouwd is met muzieknotatie zullen subsectie 2.2.5 en sectie 2.4 voldoende informatie bevatten.

In hoofdstuk 3 wordt *PRISM* besproken, een probabilistisch-logische programmeertaal waarin we zullen werken. Vervolgens gaan we in hoofdstuk 4 dieper in op *Hidden Markov Models* (HMM's). Op het einde van dit hoofdstuk wordt bij wijze van illustratie een HMM geïmplementeerd in PRISM. De vluchtige lezer kan zich desnoods beperken tot secties 3.1.1, 3.1.2, 3.1.3 (enkel de eerste paragraaf en het voorbeeld), 4.1, 4.2 en 4.5.

Geïnspireerd door HMM's construeren we in hoofdstuk 5 een PRISM-model voor muziek in IVL-notatie. Dit model gebruiken we om een tweetal experimenten op te zetten, die beschreven worden in hoofdstuk 6. Ten slotte geven we in hoofdstuk 7 een korte samenvatting, bespreken we in welke mate onze doelstellingen bereikt zijn en wat de mogelijkheden zijn voor eventueel toekomstig werk gebaseerd op deze verhandeling.

Bijlage A geeft een overzicht van de verschillende software-pakketten die gebruikt werden om deze verhandeling te maken. In bijlage B wordt beschreven hoe we op een automatische manier de omzetting van en naar de IVL-notatie kunnen implementeren. De volledige listing van het PRISM-model dat we in hoofdstuk 5 construeerden staat in bijlage C.

In figuur 1.1 wordt een overzicht van de hoofdstukken gegeven. Een pijl van hoofdstuk  $X$  naar hoofdstuk  $Y$  geeft aan dat begrippen uit hoofdstuk  $X$  gebruikt worden in hoofdstuk  $Y$  en hoofdstuk  $X$  dus best eerst gelezen wordt.



Figuur 1.1: Overzicht van de hoofdstukken

## Hoofdstuk 2

# Muziekrepresentatie

*“Da ist nichts Bemerkenswertes daran. Alles was man tun muss, ist, die richtige Note zur richtigen Zeit zu treffen, – und das Instrument spielt von allein.”*  
– Johann Sebastian Bach (1685-1750)

In dit hoofdstuk gaan we onderzoeken hoe we muziek vereenvoudigd kunnen voorstellen, zonder de nodige expressiviteit te verliezen. We beginnen deze zoektocht bij de gebruikelijke, klassieke notatie. Vervolgens maken we in sectie 2.2 een selectie van de muzikale elementen die we willen kunnen voorstellen. Daarna, in sectie 2.3 bespreken we kort enkele bestaande notaties. Tenslotte definiëren we in sectie 2.4 een nieuwe notatie: *Interwoven Voices List*.

### 2.1 Klassieke muzieknotatie

Deze grafische notatie is wereldwijd gekend en gebruikt. Per stem wordt een vijf-lijnige *notenbalk* getekend. Een *sleutel* (vb. “solsleutel”) ikt de notenbalk door de positie van een gekende noot aan te duiden. *Noten* kunnen op of tussen de lijnen van de notenbalk getekend worden. De verticale positie op de notenbalk bepaalt de *toonhoogte* van een noot. In het onderstaande fragment geeft de sol sleutel aan dat de noot op de tweede lijn (van beneden) een sol is. De noten gaan in dit voorbeeld van de *centrale do* (“c4”) naar de do twee octaven erboven (“c6”).



De manier waarop de noot getekend wordt, bepaalt hoe lang die noot duurt – als de zogenaamde *mathematische spatiëring* gebruikt wordt, komt de horizontale positie op de notenbalk ook overeen met die lengte. Bij de zogenaamde *proportionele spatiëring* probeert men de horizontale plaatsing van de noten zoveel mogelijk in overeenstemming te brengen met de nootlengtes, maar voor korte noten wordt voor de leesbaarheid meer plaats gelaten. Een *hele* noot duurt het langst, een *halve* duurt half zo lang, enzovoort. Muziek

wordt onderverdeeld in *maten*, die gescheiden worden door verticale *maatstrepen*. De *maatsoort* bepaalt hoe lang elke maat duurt. In het voorbeeld hieronder duurt één maat even lang als twee kwartnoten, m.a.w. de lengte van één halve noot. We zien hieronder achtereenvolgens een halve, een vierde, twee achtsten, vier zestienden, een achtste gepunt (de punt geeft aan dat de noot met de helft verlengd wordt, dus in dit geval even lang duurt als een achtste en een zestiende), een 32<sup>ste</sup> en twee 64<sup>ste</sup> noten:



Er zijn verschillende conventies om noten te benoemen; de namen do, re, mi, fa, sol, la en si of de letters c, d, e, f, g, a, b worden vaak gebruikt. Op deze manier kunnen we alle noten van de witte toetsen van een piano noteren. Tussen twee opeenvolgende witte toetsen van een piano ligt een zwarte toets (behalve tussen mi en fa en tussen si en do). De noten van zwarte toetsen kunnen op twee manieren geschreven worden. Beschouw bijvoorbeeld de zwarte toets tussen do en re. De noot die daarbij hoort kan ofwel geschreven worden als een verhoogde do (do #, do kruis, “cis”), ofwel als een verlaagde re (re b, re mol, “des”). Het gaat hier over dezelfde toonhoogte, ondanks de verschillende notatie:



Tot zover deze (zeer beknopte) inleiding over de klassieke muzieknotatie. Een meer diepgaande inleiding op muzieknotatie en -theorie wordt gegeven in hoofdstuk 2 van de licentiaatsthesis van Bart Vrancken [Vra03]. Neil Hawes [Haw] maakte een website over muzieknotatie in erg toegankelijke taal. In wat volgt zullen we nieuwe muzikale begrippen en notaties uitleggen wanneer ze voor het eerst gebruikt worden.

Hét grote voordeel van deze notatie is dat ze wereldwijd bekend is; elke (klassiek geschoolde) muzikant kan zonder problemen noten lezen die op deze manier genoteerd staan. We wensen echter dat een *computerprogramma* zonder al te veel problemen kan werken met de notatie die we gebruiken. Daarvoor is de klassieke notenbalk-notatie niet geschikt. We zullen in het vervolg van deze verhandeling wel nog voorbeelden geven in de klassieke muzieknotatie, juist omwille van de goede leesbaarheid voor mensen.

## 2.2 Wat wel en wat niet?

Het zal niet haalbaar zijn om elk aspect van muziek te modelleren. Sterker nog: het is wenselijk om enkel de echt essentiële aspecten te behouden. We moeten dus beslissen welke elementen we als ‘noodzakelijk’ en welke we als ‘verwaarloosbaar’ beschouwen. Uiteindelijk zullen wel dan enkel de ‘noodzakelijke’ aspecten opnemen in ons model. We groeperen de muziek-aspecten die we zullen behandelen in vier categorieën: *melodie*, *samenklank*, *tijd en nadruk* en *details*.

## 2.2.1 Melodie

Een *melodie* is te omschrijven als een opeenvolging van toonhoogtes. Het zal er dus op aankomen om een goede voorstelling van toonhoogtes te vinden.

### Toonhoogte

De toonhoogte (*pitch*) van een noot is in de meeste (westerse) muziek beperkt tot de noten die op een piano kunnen gespeeld worden, m.a.w. de kleinste afstand tussen twee verschillende toonhoogtes is de *halve toon* (de afstand tussen bijvoorbeeld mi en fa of tussen do en do kruis). We zullen dan ook als kleinste interval de halve toon nemen.

Deze beperking laat niet toe om alle melodieën voor te stellen; in sommige exotische muzieksoorten, zoals Arabische of oosterse muziek, komen bijvoorbeeld ook *kwarttonen* voor. Zo'n intervallen klinken echter nogal ongewoon en exotisch omdat ze niet passen in de diatonische melodie-structuur waaraan onze Westerse oren gewend zijn. Het is niet nodig dergelijke intervallen te kunnen voorstellen aangezien ze in Westerse muziek toch niet voorkomen.

Bij sommige instrumenten (bijvoorbeeld viool) kan er een intonatieverschil zijn tussen do kruis en re mol (en analoog tussen re kruis en mi mol, enzovoort). Deze noten worden dan weer als dezelfde beschouwd bij gelijkzwevend gestemde instrumenten (zoals piano). Voor de eenvoud zullen we hier geen rekening houden met een mogelijk intonatieverschil. Het is wenselijk om slechts één voorstelling toe te laten voor dezelfde noot. We kunnen eisen dat alle mollen vervangen worden door kruisen (of omgekeerd) om dit te bereiken.

### Sleutels

De sleutel (solsleutel, fasleutel, ...) is slechts een hulpmiddel in de klassieke notatie om de notenbalk te ijken. Dankzij de sleutel weet degene die de partituur leest de hoogte op de notenbalk van elke toonhoogte. In onze voorstelling zullen we toonhoogtes waarschijnlijk toch niet op een notenbalk aanduiden. Het heeft dan ook weinig zin om nog voor te stellen welke sleutel gebruikt wordt.

### Toonaard en voortekening

De meeste (klassieke) muziek is geschreven in een bepaalde *toonaard*, waarbij vooral de noten gebruikt worden van de *toonladder* die bij die toonaard hoort. Dit houdt in dat bepaalde noten systematisch worden verhoogd of verlaagd. Zo bestaat de toonladder van de toonaard “re groot” (of D majeur) uit de noten re, mi, fa kruis, sol, la, si en do kruis, en zullen dus de noten fa en do systematisch verhoogd worden. Om aan te geven welke noten verhoogd of verlaagd worden, worden in de klassieke notatie een aantal kruisen of mollen aan de sleutel geschreven, die dan op de rest van de notenbalk niet meer herhaald moeten worden. Dit noemen we de *voortekening*.

Er zijn dus twee soorten kruisen en mollen: ‘toevallige’ en die van de voortekening. De ‘toevallige’ tekens worden plaatselijk gebruikt – niet globaal, zoals bij de voortekening – om uitzonderingen op de gebruikte toonladder te noteren. Ze blijven gelden bij

elke herhaling van dezelfde noot, tot het einde van de maat waarin ze voorkomen. Het herstellingsteken ( $\natural$ ) geeft aan dat een noot die normaal gezien verhoogd of verlaagd zou moeten worden (wegens de voortekening of een ‘toevallige’ verhoging van dezelfde noot eerder in de maat) toch niet gewijzigd wordt.

De voortekening kan ook veranderen in het midden van een stuk. Om die verandering duidelijk aan te geven wordt de vorige voortekening eerst ‘hersteld’, waarna de nieuwe voortekening genoteerd wordt. Hieronder wordt drie keer de sequentie “do kruis, fa kruis, do kruis, do” genoteerd, telkens onder een andere voortekening:



We kunnen de voortekening weglaten en overal de verhogingen of verlagingen expliciet uitschrijven. Zo vermijden we dat er verschillende manieren zijn om dezelfde notenreeks te noteren. Bovendien wordt de voorstelling eenvoudiger door het concept van toonaarden weg te laten.

### 2.2.2 Samenklank

Muziek bestaat uit verschillende verweven melodiën. Gewoonlijk wordt elke melodielijn op een aparte notenbalk genoteerd en door een bepaald instrument gespeeld of door een bepaalde zangstem gezongen. Soms worden verschillende stemmen op één notenbalk genoteerd. We kunnen dit echter steeds ‘uitschrijven’ in verschillende notenbalken.

#### Meerdere stemmen

Het gelijktijdig samenklinken van verschillende melodieën is een onverwaarloosbaar aspect van zowat alle muziek. We zullen dus meerdere stemmen toelaten in onze voorstelling. Het is minder belangrijk om toe te laten dat in één stem verschillende noten gelijktijdig kunnen klinken. Er zijn natuurlijk instrumenten waarbij dit mogelijk is (piano, gitaar, viool, ...), maar we kunnen indien nodig meerdere stemmen gebruiken om zo’n instrument voor te stellen. Eigenlijk zijn de twee fenomenen – samenklank binnen één stem en samenklank tussen de stemmen – verschillende voorstellingen van hetzelfde principe (gelijktijdig klinkende noten). Het lijkt dan ook beter om in onze representatie de tweede voorstelling te ‘forceren’.

Niet alleen is het nodig dat meerdere stemmen kunnen voorgesteld worden, het is ook van belang dat we verbanden tussen de stemmen kunnen leggen. Het zal dus niet volstaan een notatie te zoeken voor éénstemmige muziek om dan elke stem apart te noteren in die notatie, want zo is het niet evident welke noten samen klinken.

#### Instrumentatie

Uiteraard is het voor de klankkleur van een muziekstuk niet onbelangrijk welke instrumenten gebruikt worden. Toch zullen we dit aspect verwaarlozen, aangezien de ‘klank’

van een instrument een eerder vaag en subjectief begrip is, waar niet eenvoudig mee te rekenen valt.

### 2.2.3 Tijd en nadruk

Natuurlijk is het ook van belang op welk moment de noten gespeeld worden – hoe snel ze elkaar opvolgen, welke noten langer duren en welke korter. Tevens worden sommige noten benadrukt, beklemtoond.

#### Tempo en ritme

*Tempo* is de algemene snelheid van een muziekstuk, en wordt vaak uitgedrukt in aantal vierde noten per minuut. Het *ritme* van een fragment wordt bepaald door het aantal en vooral de duur van noten en rusten. Beide begrippen hebben te maken met de tijd; tempo is echter een min of meer globale eigenschap van een muziekstuk, terwijl ritme een lokaal begrip is.

Alhoewel het tempo zeker een belangrijke rol speelt in de perceptie van de luisteraar, is het zeker ondergeschikt aan het ritme. We kunnen het tempo beschouwen als iets dat wel de algemene sfeer van een muziekstuk kan bepalen, maar in de muzikale analyse redelijk onbelangrijk is. Ritme is echter een cruciaal aspect van muziek. We zullen dus zeker de duur van noten en rusten moeten voorstellen.

#### Dynamiek

Onder *dynamiek* verstaan we aanduidingen als *forte*, *piano*, *crescendo*, . . . die aangeven op welke manier (vooral: hoe luid) een bepaalde passage gespeeld of gezongen moet worden. Voor de doelstellingen van deze verhandeling willen we onze voorstelling zo eenvoudig mogelijk houden. Het is een min of meer arbitraire keuze om dynamiek als niet-essentieel te beschouwen. Toch wordt de dynamiek van een muziekstuk vaak als ‘afwerking’, en dus eerder als detail beschouwd. We zullen dit aspect niet in onze representatie opnemen.

Het zou een interessante uitbreiding kunnen zijn om dynamiek wél voor te stellen, bijvoorbeeld door met elke noot, naast de toonhoogte en de duur, een volume-waarde te associëren.

#### Maatsoort

De *maatsoort* bepaalt waar de impliciete klemtonen vallen, namelijk op de eerste tel van elke maat. Verder worden maten eigenlijk enkel gebruikt om partituren overzichtelijk te houden door ze in te delen in kleine eenheden.

Eigenlijk is er in de praktijk weinig verschil tussen klemtonen en dynamiek. De beklemtoonde noten worden luider gespeeld, de onbeklemtoonde noten zachter. We zullen klemtonen bijgevolg beschouwen als niet-essentieel. We kunnen de concepten van maten en maatsoort dan ook weglaten.

### 2.2.4 Details

De klassieke muzieknotatie is organisch gegroeid in de voorbije eeuwen. Daardoor zijn er vaak verschillende manieren om hetzelfde uit te drukken, en bestaan er zeer veel symbolen en (vaak Italiaanse) formuleringen om bepaalde nuances uit te drukken.

#### Versieringsnoten

Versieringsnoten hebben meestal een zuiver ornamentele functie, ze kunnen dus in veel gevallen weggelaten worden zonder de melodie fundamenteel te veranderen. In de klassieke notatie worden ze gewoonlijk kleiner getekend en wordt hun duur niet meegeteld. Soms worden ze niet uitgeschreven maar afgekort met bepaalde symbolen boven de te versieren noot. Voorbeelden zijn trillers, mordent, acciaccatura, appoggiatura, glissando en arpeggio. Het is in principe altijd mogelijk versieringsnoten expliciet uit te schrijven, bijvoorbeeld als korte 32<sup>ste</sup> noten. Op de plaatsen waar versieringsnoten belangrijk zijn in de melodie kunnen we ze uitschrijven, op de andere plaatsen laten we ze gewoon weg.

#### Articulatie

In de klassieke notatie bestaan er verschillende symbolen die bij noten geplaatst kunnen worden als aanduiding over hoe die noten gespeeld dienen te worden. Enkele voorbeelden: accent, marcato, staccato, portato, tenuto, legato, . . . Deze articulatie-tekens kunnen we als details beschouwen die we niet opnemen in onze voorstelling.

#### Herhalingen

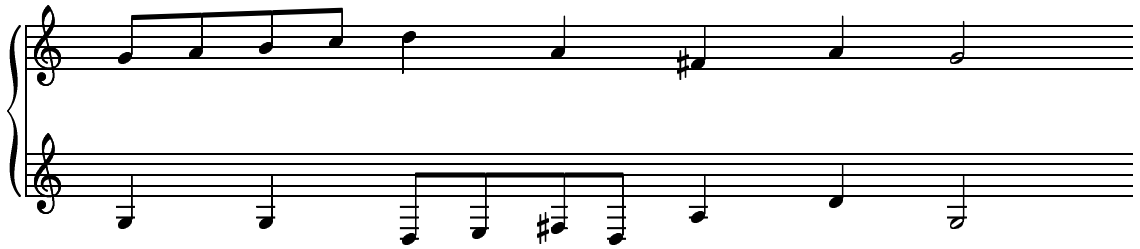
Er bestaan verschillende herhalingstekens die dienen om repetitieve stukken korter op te schrijven. In onze representatie zullen we herhalingen telkens volledig uitschrijven, zodat we het concept van herhaling kunnen laten vallen. Hiermee verliezen we geen expressiviteit en wordt onze voorstelling eenvoudiger, maar wel minder compact.

### 2.2.5 Besluit

De (voor de doelstellingen van deze verhandeling) essentiële muzikale aspecten zijn samenklank, toonhoogte (van noten) en duur (van noten en rusten). Er gaat natuurlijk heel wat muzikale informatie verloren als we alle andere aspecten weglaten. We kunnen dit verlies aan informatie met een eenvoudig voorbeeld illustreren:



Als we van bovenstaand voorbeeld enkel de ‘essentie’ behouden dan krijgen we (voorlopig natuurlijk nog wel in de klassieke muzieknotatie):



## 2.3 Bestaande representaties

Er bestaan uiteraard al verschillende muzieknotaties die meer geschikt zijn om door een computer verwerkt te worden. In deze sectie bespreken we bondig de voor- en nadelen van enkele van die systemen.

### 2.3.1 MIDI

*MIDI* (Musical Instrument Digital Interface) is dé standaard in de muziekindustrie om muziek voor te stellen. Eigenlijk is MIDI een protocol om communicatie tussen muziekinstrumenten en computers te realiseren; gebeurtenissen (events) zoals “instrument X begint noot Y te spelen” worden in bepaalde binaire codes doorgestuurd. Het Standaard MIDI bestandsformaat bestaat uit dergelijke gebeurtenissen met daarbij de tijdstippen (timestamps) waarop ze gebeuren.

De MIDI standaard is ondertussen zo’n twintig jaar oud. Er is dan ook al een behoorlijke hoeveelheid software geschreven om MIDI-bestanden af te spelen, op te nemen of te verwerken. MIDI heeft echter – met onze doelstellingen in het achterhoofd – meer nadelen dan voordelen. Zo zullen er bijvoorbeeld voor elke noot in een MIDI bestand minstens twee events voorkomen: één om de noot te beginnen en één om de noot te stoppen. Bovendien kunnen binnen één stem (*channel* bij MIDI) meerdere noten tegelijk klinken, wat we net wilden vermijden. Het binaire karakter van MIDI bestanden maakt dat het geschikt is voor gespecialiseerde muziekhardware en zorgt voor efficiënt gebruik van het communicatiekanaal. Voor verwerking in een hoog-niveau (logische) programmeertaal is MIDI echter minder handig.

### 2.3.2 LilyPond

LilyPond [Lil, NN03] is een programma dat ‘mooie’ partituren produceert – volgens een proportionele spatiëring – op basis van een plain text muziekbeschrijving. Omzetting naar MIDI is ook voorzien. In figuur 2.1 wordt het resultaat van de verwerking van het voorbeeld uit listing 2.1 door LilyPond gegeven.

De LilyPond-syntax is erg uitgebreid en bevat constructies voor zowat alle mogelijke symbolen uit de klassieke muzieknotatie. Vanuit ons standpunt is dat echter eerder een

Listing 2.1: Voorbeeld van de LilyPond-syntax

```

voiceone = \notes \relative c' {
  \time 2/4
  g'8[ c b c]
  d4 r
}
voicetwo = \notes \relative c {
  \clef "bass"
  \time 2/4
  r16 c[ d e] f[ d e c]
  g'8[ g,] r4
}

\score {
  \notes \context GrandStaff <<
    \context Staff = one <<
      \voiceone
    >>
    \context Staff = two <<
      \voicetwo
    >>
  >>
  \paper{ }
  \midi { \tempo 4 = 80 }
}

```



Figuur 2.1: LilyPond-omzetting van listing 2.1 naar klassieke notatie

---

```

M:2/4
K:C
V:1
L:1/8
gc'bc' | d'2 z2
V:2
L:1/16
z CDE FDEC | G2 G,2 z4

```

---

Figuur 2.2: Hetzelfde voorbeeld als in figuur 2.1, in abc notatie

nadeel. Bovendien worden de verschillende stemmen los van elkaar beschreven, wat een analyse van hoe de stemmen op elkaar inwerken moeilijk maakt.

### 2.3.3 Abc notatie

De *abc notatie* is een (voor mensen althans) eenvoudige ASCII-text muzieknotatie, uitgevonden door Chris Walshaw. Het is de bedoeling dat muziek genoteerd in abc gemakkelijk op zicht te lezen is voor mensen, en is vooral geschikt om korte éénstemmige liedjes te noteren.

Het is ook mogelijk om meerstemmige muziek te noteren. De stemmen worden dan wel apart genoteerd, zoals bij LilyPond. Bovendien bevat de abc syntax nogal wat ‘verkorte notaties’, zoals bijvoorbeeld het symbool “>” (“a>b” is een afkorting voor “a3/2b/2”). Met onze doelstellingen in gedachte is de abc notatie dus niet geschikt. We hebben een representatie nodig waarbij gelijktijdig klinkende noten gemakkelijker te zien zijn.

### 2.3.4 SoundTracker en afgeleiden

SoundTracker is een muziekprogramma dat in 1987 gemaakt is door Karsten Obarski, voor de Commodore Amiga-computer. Ondertussen bestaat er een ruime keuze aan zogenaamde *trackers* die op SoundTracker gebaseerd zijn, zoals bijvoorbeeld Noise Tracker, ProTracker (.MOD), ScreamTracker (.STM en .S3M), Impulse Tracker (.IT) en FastTracker (.XM). Een screenshot van een recente tracker met de weinig originele naam SoundTracker – niet te verwarren met het programma uit 1987 – staat in figuur 2.3.

De onderliggende muziekrepresentatie is in essentie een tabel waarbij de stemmen in de kolommen (*tracks*) naast elkaar gezet worden en gelijktijdig klinkende noten op dezelfde rij staan. Op elke positie in de tabel staat ofwel een nootbeschrijving (instrument, pitch, volume), ofwel niets (de vorige noot blijft klinken), ofwel het “note off” teken (noot stopt met klinken). De muziek wordt van boven naar onder gelezen, waarbij elke rij één tijdseenheid later gespeeld wordt dan de vorige. Met andere woorden, het aantal lege plaatsen na een noot bepaalt hoe lang die noot blijft klinken.

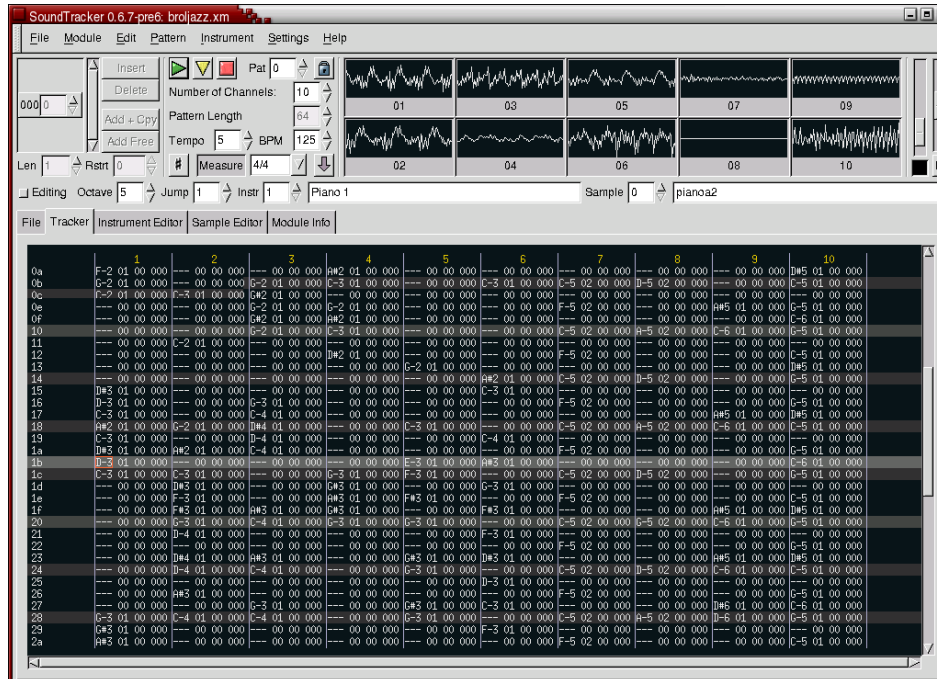
Zoals bij Standaard MIDI bestanden worden de gegevens op een compacte manier door binaire codes voorgesteld. Daardoor zijn deze bestandsformaten voor ons niet echt geschikt om (rechtstreeks) mee te werken. Een ander belangrijk nadeel van dit systeem is dat een vaste ‘resolutie’ gebruikt wordt: als in een muziekstuk zowel heel snelle als trage noten voorkomen, komen er gemakkelijk veel lege plaatsen in de tabel.

Toch heeft de voorstelling die gebruikt wordt in trackers een belangrijk voordeel: samenklank is duidelijk zichtbaar.

### 2.3.5 Humdrum **\*\*kern**

*Humdrum **\*\*kern*** ([www.humdrum.net](http://www.humdrum.net)) is een ASCII voorstelling van muziek, bedoeld voor muzikanalyse, waarbij hetzelfde principe gebruikt wordt als hierboven.

In de SoundTracker-voorstelling duurt elke rij even lang, bij Humdrum **\*\*kern** is dit



Figuur 2.3: Screenshot van SoundTracker 0.6.7-pre6

	1	2
00	G-5 01 00 000	--- 00 00 000
01	--- 00 00 000	--- 00 00 000
02	--- 00 00 000	C-4 01 00 000
03	--- 00 00 000	--- 00 00 000
04	C-6 01 00 000	D-4 01 00 000
05	--- 00 00 000	--- 00 00 000
06	--- 00 00 000	E-4 01 00 000
07	--- 00 00 000	--- 00 00 000
08	H-5 01 00 000	F-4 01 00 000
09	--- 00 00 000	--- 00 00 000
0a	--- 00 00 000	D-4 01 00 000
0b	--- 00 00 000	--- 00 00 000
0c	C-6 01 00 000	E-4 01 00 000
0d	--- 00 00 000	--- 00 00 000
0e	--- 00 00 000	C-4 01 00 000
0f	--- 00 00 000	--- 00 00 000
10	D-6 01 00 000	G-4 01 00 000
11	--- 00 00 000	--- 00 00 000
12	--- 00 00 000	--- 00 00 000
13	--- 00 00 000	--- 00 00 000
14	--- 00 00 000	G-3 01 00 000
15	--- 00 00 000	--- 00 00 000
16	--- 00 00 000	--- 00 00 000
17	--- 00 00 000	--- 00 00 000
18	[-] 00 00 000	[-] 00 00 000
19	--- 00 00 000	--- 00 00 000
1a	--- 00 00 000	--- 00 00 000
1b	--- 00 00 000	--- 00 00 000
1c	--- 00 00 000	--- 00 00 000
1d	--- 00 00 000	--- 00 00 000
1e	--- 00 00 000	--- 00 00 000
1f	--- 00 00 000	--- 00 00 000

Figuur 2.4: Voorbeeld uit figuur 2.1 in SoundTracker-notatie

---

**kern	**kern
*M2/4	*M2/4
*k []	*k []
*C:	*C:
=1	=1
8g	16r
.	16C
8cc	16D
.	16E
8b	16F
.	16D
8cc	16E
.	16C
=2	=2
4dd	8G
.	8GG
4r	4r

---

Figuur 2.5: Voorbeeld uit figuur 2.1 in de Humdrum **\*\*kern** notatie

echter niet zo. Elke noot heeft in Humdrum een expliciet genoteerde duur, en een rij duurt in feite zo lang als de kortste noot op die rij. Op die manier wordt het aantal lege plaatsen erg verminderd.

De syntax van Humdrum **\*\*kern** heeft enkele belangrijke nadelen. Zo wordt de duur van een noot in dezelfde notatie aangegeven als in LilyPond: een achtste noot met “8”, een halve met “16”, een vierde ‘gepunte’ (een vierde plus een achtste) door “4.”, enzovoort. Die notatie is gebaseerd op de klassieke muzieknotatie, maar is onnodig gecompliceerd en enigzins tegen-intuïtief – hoe langer de noot, hoe lager het getal.

Bovendien worden noten met letters aangeduid en blijven er verschillende manieren om dezelfde noot aan te duiden: “C#” is dezelfde noot als “D-” (molteken wordt voorgesteld door “-”). Voortekening, maatsoort en maatstrepen worden expliciet uitgeschreven, terwijl we die als overbodige ballast willen beschouwen. Kortom, het basisprincipe van Humdrum **\*\*kern** is wel bruikbaar, maar de syntax is nog te complex en ‘menselijk’ voor onze doelstellingen.

## 2.4 Interwoven Voices List

De bestaande representaties waren een goede bron van inspiratie. Geen van de besproken systemen zijn echter rechtstreeks bruikbaar. We zullen dus een nieuwe notatie invoeren, een notatie die aansluit bij onze doelstellingen. Aangezien we van plan zijn om PRISM te gebruiken en PRISM gebouwd is op B-Prolog, lijkt het zinvol om een notatie te gebruiken waar Prolog gemakkelijk mee kan werken.

### 2.4.1 Intuïtie

Als we in de SoundTracker of Humdrum <sup>\*\*</sup>kern notatie de lege ruimtes opvullen met een herhaling van de noot die op dat moment klinkt (en rust-tekens op de plaatsen waar geen noot klinkt), kunnen we op elke rij aflezen welke noten er samen klinken. Het probleem is dat we nu geen onderscheid meer kunnen maken tussen één lange noot en veel korte noten. We moeten dus aangeven welke noten ‘nieuw’ zijn en welke ‘al klonken’.

Op deze manier hebben we effectief de verschillende stemmen samengebracht, vandaar de naam *Interwoven Voices List*.

### 2.4.2 Definitie

We stellen een muziekstuk voor als een lijst van de vorm  $[(L_0, V_0, N_0), (L_1, V_1, N_1), \dots, (L_n, V_n, N_n)]$  waarbij  $L_i$  een duur is,  $V_i$  aangeeft in welke stemmen ‘nieuwe’ noten gespeeld worden en  $N_i$  een lijst is van de noten die op moment  $i$  klinken in elke stem. Het drietal  $(L_i, V_i, N_i)$  stelt *fase  $i$*  voor.

#### Duur $L_i$

We stellen de duur voor als een natuurlijk getal. Hoe groter het getal  $L_i$ , hoe langer de fase  $i$  duurt. Meer concreet:  $L_i$  is het aantal 256<sup>ste</sup> noten dat overeenkomt met de duur van fase  $i$ . Zo komt bijvoorbeeld met de duur van een kwartnoot het getal 64 overeen, 32 stelt een achtste noot voor, 96 een gepunte vierde, 512 twee hele noten, enzovoort.

#### Nieuwe noten, oude noten ( $V_i$ )

$V_i$  bevat een lijst van de stemmen waarin een nieuwe noot ingezet wordt in fase  $i$ . Als in een bepaalde stem een nieuwe noot wordt begonnen schrijven we op de overeenkomstige positie ‘new’, als de vorige noot nog blijft klinken schrijven we ‘old’. Een rust die begonnen wordt beschouwen we ook als een nieuwe noot, tenzij ze volgt op een vorige rust, dan beschouwen we ze altijd als ‘old’.

Stel bijvoorbeeld dat  $V_i$  de waarde [new,old,old,new] bevat. Dan beginnen in fase  $i$  de eerste en de vierde stem aan een nieuwe noot, terwijl de vorige noot in de tweede en derde stem nog klinken.

#### Notenlijst $N_i$

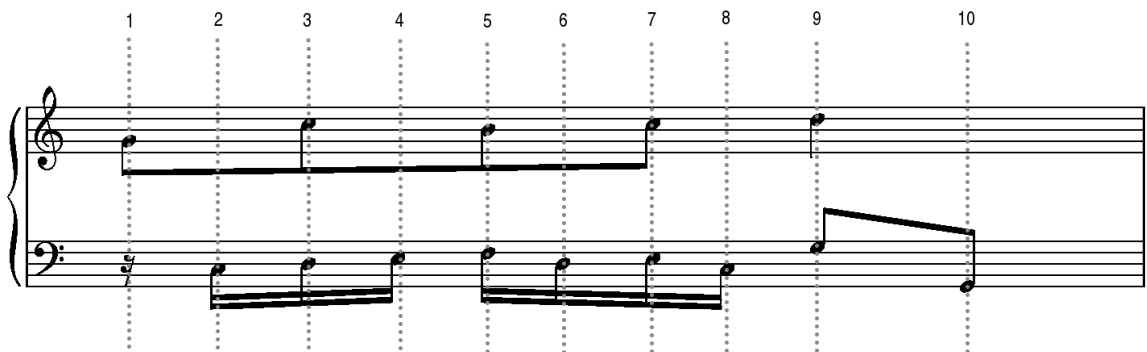
$N_i$  is een lijst van de vorm  $[P_0, P_1, \dots, P_k]$  waarbij  $k$  het aantal stemmen is en  $P_j$  de toonhoogte van de noot die klinkt in de  $j$ -de stem. Meer concreet:  $P_j$  is ofwel een natuurlijk getal ofwel het symbool “r” (rust). De centrale do (c4) wordt voorgesteld door het getal 48, de re door 50, do kruis (en re mol) door 49, enzovoort. De laagste noot die kan worden voorgesteld is dus de do vier octaven onder de centrale do (c0).

### 2.4.3 Voorbeeld

Laat ons bovenstaande definitie illustreren met het voorbeeld uit figuur 2.1 (sectie 2.3.2). Details weggelaten ziet dat voorbeeld er als volgt uit in klassieke notatie:



We bekijken nu de momenten van verandering, die met stippellijn zijn aangeduid op de volgende figuur:



Met elk van die 10 momenten komt een drietal van de vorm  $(L, V, N)$  overeen. De eerste noot in de eerste stem is de sol boven de centrale do en krijgt dus het nummer 55. De tweede stem begint met een rust. De eerste fase duurt even lang als een 16<sup>de</sup> noot, dus 16 tijdseenheden van 256<sup>ste</sup> noten. Het eerste drietal is dus  $(16, [\text{new}, \text{new}], [55, r])$ . In de tweede fase begint de tweede stem met de do een octaaf lager dan de centrale do (48), en klinkt de sol van de eerste stem nog verder. Deze fase duurt weer 16 tijdseenheden. Het tweede drietal is  $(16, [\text{old}, \text{new}], [55, 36])$ . Als we op die manier verdergaan, krijgen we de IVL-versie van het voorbeeld:

```
[ (16, [new,new], [55,r]), (16, [old,new], [55,36]),
  (16, [new,new], [60,38]), (16, [old,new], [60,40]),
  (16, [new,new], [59,41]), (16, [old,new], [59,38]),
  (16, [new,new], [60,40]), (16, [old,new], [60,36]),
  (32, [new,new], [62,43]), (32, [old,new], [62,31]) ]
```

## Hoofdstuk 3

# Programmeren in PRISM

*“The sonatas of Mozart are unique; they are too easy for children, and too difficult for artists.”*

– Arthur Schnabel (1882-1951)

Om probabilistische en logische kennis te modelleren zullen we het PRISM-systeem [Pri] gebruiken. PRISM (***P**rogramming **I**n **S**tatistical **M**odeling*) is een programmeertaal bedoeld voor symbolisch-statistische modellering, ontwikkeld door (o.a.) Taisuke Sato en Yoshitaka Kameya. PRISM is gebouwd op B-Prolog [BPr], een Constraint Logic Programming systeem ontwikkeld door Neng-Fa Zhou. In deze verhandeling gebruiken we PRISM versie 1.6 van 23 maart 2003 – de recentste beschikbare versie toen dit geschreven werd (april 2004).

Eenvoudig gesteld is PRISM te beschouwen als een Prolog-systeem uitgebreid met ingebouwde probabilistische predicaten en een ingebouwd leeralgoritme. Logische kennis kan op een natuurlijke manier in Prolog gemodelleerd worden. Dankzij de ingebouwde probabilistische predicaten kunnen we ook op een relatief intuïtieve manier probabilistische modellen opstellen. Het PRISM-systeem bestaat uit twee subsystemen: één om te leren (*learning subsystem*) en één om ‘uit te voeren’ (*sample execution subsystem*). We zullen nu verder ingaan op de syntax en semantiek van PRISM en de werking van de twee subsystemen schetsen.

### 3.1 PRISM-modellen

In [Sat95] wordt een rigoureuze wiskundige fundering gelegd voor de theoretische semantische basis van PRISM-programma’s, namelijk *distribution semantics*. De PRISM-taal en het PRISM-systeem worden beschreven in [Sat97] aan de hand van enkele voorbeelden. Een gedetailleerd overzicht van de theorie achter PRISM wordt in [SK01] gegeven. In de volgende secties zullen we bondig uitleggen hoe een model (of programma) in PRISM geschreven kan worden. Een basiskennis van Prolog wordt verondersteld.

Alle Prolog-constructies kunnen gebruikt worden in PRISM-programma’s. Er staan bovendien nog een 15-tal nieuwe built-ins ter onzer beschikking in PRISM, waarvan we enkel de belangrijkste zullen beschrijven. In subsectie 3.1.1 worden de PRISM-built-ins

`msw/2`, `values/2` en `target/2` besproken. Daarna, in subsectie 3.1.2, komen `set_sw/2`, `sample/1` en `prob/2` aan bod. Tenslotte worden `learn/1`, `learn/0` en `data/1` in subsectie 3.1.3 behandeld. Voor een gedetailleerder overzicht verwijzen we naar [ZS02]. In [ZS03] worden alle PRISM built-ins opgesomd, voorzien van bondige uitleg.

### 3.1.1 Probabilistische predicaten

Het ingebouwde predicaat `msw(E,R)` (*multi-outcome switch*) stelt de uitvoering van het kans-experiment `E` voor, met als resultaat `R`. Een kans-experiment kan bijvoorbeeld het opgooien van een muntstuk zijn (experiment `muntstuk`), met als mogelijk resultaat `R` ofwel `kop` ofwel `mun`t. Voor elk kans-experiment moet aangegeven worden wat al de mogelijke resultaten zijn. Daarvoor kan het ingebouwde predicaat `values(E,RL)` gebruikt worden, waarbij de mogelijke resultaten voor experiment `E` in de lijst `RL` staan – in het muntstuk-voorbeeld: `values(muntstuk,[kop,munt])`. Elk mogelijk resultaat komt voor met een bepaalde kans. De som van de kansen van alle resultaten voor een bepaald experiment moet uiteraard altijd 1 zijn.

We kunnen het probabilistische predicaat `msw/2` in het algemeen overal waar we willen gebruiken in een PRISM-programma. Zo kunnen we complexe experimenten modelleren die uit meerdere enkelvoudige experimenten (`msw`-predicaten) bestaan. In het voorbeeld in listing 3.1 wordt het experiment “`N` keer een muntstuk opgooien” gedefinieerd door het experiment `muntstuk` `N` keer te herhalen. Het predicaat `target(P,A)` geeft aan dat `P` met ariteit `A` het doel-predicaat is voor de uitvoering en/of het leeralgoritme, in het voorbeeld is dat `gooiNkeer/2`.

Listing 3.1: voorbeeld.psm

```
target ( gooiNkeer , 2 ).
values ( muntstuk , [ kop , munt ] ).

gooiNkeer ( 0 , [] ).
gooiNkeer ( N , [ Worp | Rest ] ) : -
    N > 0 ,
    msw ( muntstuk , Worp ) ,
    N1 is N-1 ,
    gooiNkeer ( N1 , Rest ) .
```

### 3.1.2 Sample execution subsystem

De kansverdeling van een experiment kan ingegeven worden met het directief `set_sw(E,KL)`. Als we in het muntstuk-experiment uit figuur 3.1 bijvoorbeeld 80% kans op kop hebben, dan schrijven we: `set_sw(muntstuk,[0.8,0.2])`.

Eens de kansverdeling van elk enkelvoudig experiment gekend is kunnen we een ‘sample execution’ query op het doel-predicaat uitvoeren. Telkens als bij de uitvoering een `msw(E,R)` wordt tegengekomen zal er willekeurig (volgens de kansverdeling van `E`) een resultaat worden gekozen, dat in `R` terecht komt. Als het resultaat niet kan worden geünificeerd met `R` dan zal die `msw` falen. De built-in `sample(Goal)` voert de query `Goal` uit

in ‘sample execution mode’. In figuur 3.1 wordt een resultaat van `sample(gooiNkeer(15,L))` weergegeven.

De built-in `prob/2` berekent de kans van een gegeven term. Zo zal bijvoorbeeld “`prob(gooiNkeer(2, [kop, munt, munt]), P)`” de kans  $P = 0.032$  geven. Inderdaad, de kans dat eerst kop en dan twee keer munt gegooid wordt is  $P(kop)(P(munt))^2 = 0.032$ .

### 3.1.3 Learning subsystem

In plaats van de kansverdeling zelf te geven met `set_sw` kunnen we de kansverdeling automatisch laten berekenen aan de hand van voorbeelden. De built-in `learn(Example)` zoekt de kansverdeling waarvoor de kans dat voorbeeld `Example` voorkomt zo groot mogelijk is. We kunnen ook een reeks voorbeelden in een bestand zetten en in het PRISM-programma aangeven dat we willen leren aan de hand van dat bestand. Daarvoor gebruiken we het ingebouwde predicaat `data(bestandsnaam)`. In dit geval kunnen we het leer-algoritme starten met de built-in `learn/0`.

Een leer-algoritme voor een bepaalde klasse van programma’s wordt voorgesteld in [Sat95]. Het efficiënter leer-algoritme van PRISM 1.6 wordt in [SK01] behandeld. We zullen ons beperken tot een schets van de werking ervan.

#### Werking

Het leer-algoritme werkt in twee stappen. In de eerste stap worden exhaustief alle mogelijke verklaringen voor de voorbeelden  $V_i$  gezocht. Een *verklaring* is een toekenning van een specifiek resultaat aan elk voorkomen van een experiment. In ons muntstukvoorbeeld is er telkens maar één verklaring voor elke voorbeeld, maar in het algemeen kan het voorkomen dat er verschillende verklaringen mogelijk zijn voor één voorbeeld. Neem bijvoorbeeld het experiment waarbij twee dobbelstenen geworpen worden (waarbij het werpen van een dobbelsteen voorgesteld wordt door een `msw` met 6 mogelijke resultaten) en het resultaat de som van de ogen is. Het voorbeeld “9” heeft dan vier verklaringen: (6,3), (5,4), (4,5) en (3,6). De verschillende verklaringen worden compact bijgehouden als grafen (*explanation graphs* of *support graphs* genaamd) en dankzij *tabling* worden gelijke deelverklaringen maar één keer berekend.

De tweede stap bestaat uit een EM-algoritme dat op een iteratieve hill-climbing manier een lokale Maximum Likelihood (ML) schatting  $\sigma_i$  van de kansverdelingen  $\theta_i$  berekent. Initieel worden de geschatte kansverdelingen  $\sigma_i$  willekeurig gekozen. Vervolgens wisselen de E(xpectation)-stap en de M(aximization)-stap elkaar af. In elke iteratie wordt de schatting van de kansverdelingen verbeterd, in die zin dat de *likelihood* verhoogt. De likelihood  $\ell(\theta_i) = P(V_i|\theta_i = \sigma_i)$  is de waarschijnlijkheid van de voorbeelden, verondersteld dat de schatting correct is. Het algoritme stopt als die verhoging minder dan  $\epsilon$  is, waarbij  $\epsilon$  een klein positief getal is<sup>1</sup>. Er wordt dan verondersteld dat  $P(V_i|\theta_i = \sigma_i)$  naar een (lokaal) maximum is geconvergeerd. We hebben dan (hopelijk) een goede schatting gevonden van de gezochte kansverdelingen.

<sup>1</sup>Default:  $\epsilon = 0.0001$ , kan worden aangepast met `set_epsilon/1`.

---

```

PRISM 1.6, Sato Lab, TITECH, All Rights Reserved. March 2003
B-Prolog Version 6.4, All rights reserved, (C) Afany Software 1994-2003.
| ?- prism(voorbeeld)

table gooiNkeer/2
loading....voorbeeld.psm.out

yes
| ?- set_sw(muntstuk,[0.8,0.2])

Switch muntstuk: unfixed: kop (0.8) munt (0.2)

yes
| ?- sample(gooiNkeer(15,L))

L = [kop,kop,munt,kop,kop,kop,kop,munt,kop,kop,kop,kop,munt,kop,kop]?
yes

```

---

Figuur 3.1: Een mogelijk resultaat na 15 keer opgooien

---

```

| ?- learn(gooiNkeer(5,[kop,kop,munt,munt,kop]))

NODE 0:
NODE 1:      [0] || [0]<>
NODE 2:      [1] || [1]<>
NODE 3:      [2] || [1]<>
NODE 4:      [3] || [0]<>
NODE 5:      [4] || [0]<>
table_statistics([402180,179597820])
ITERATION 1
likelihood0 = -3.365058 likelihood1 = -3.365058
Switch muntstuk: unfixed: kop (0.6) munt (0.4)
Finished learning in 0.01 seconds with 402180 byte table space
(0.01 seconds spent in finding explanations)

```

---

Figuur 3.2: Uitvoer van het leer-algoritme van PRISM

### Voorbeeld

In figuur 3.2 wordt de uitvoer van het PRISM-systeem gegeven bij het leren aan de hand van één voorbeeld. In dit voorbeeld werd vijf keer een muntstuk opgegooid waarvan het resultaat drie keer kop en twee keer munt was. Na één iteratie besluit het leeralgoritme van PRISM dat de kans 60% is dat er kop gegooid wordt en 40% dat munt gegooid wordt.

## 3.2 Voor- en nadelen van PRISM

In deze sectie volgt een korte bespreking van enkele positieve of negatieve eigenschappen van het PRISM-systeem, waarbij de al dan niet positieve evaluatie van een eigenschap bekeken wordt vanuit het de doelstellingen van deze verhandeling.

### 3.2.1 Voordelen

PRISM is uiteraard erg geschikt voor het modelleren van probabilistische en logische (muziek)kennis – de taal is namelijk specifiek daarvoor ontwikkeld. Enkele voordelen van de PRISM-aanpak zijn:

#### Expressiviteit van Prolog

Ingewikkelde verbanden kunnen compact en op hoog niveau uitgedrukt worden in Prolog. Zowat alle voordelen van Prolog blijven behouden in PRISM, aangezien PRISM een uitbreiding van Prolog is. Meer bepaald is het een uitbreiding van B-Prolog, zodat we in principe ook aan CLP<sup>2</sup> (over verschillende domeinen) kunnen doen – al gebruiken we die mogelijkheid niet in deze verhandeling.

#### Probabilistische predicaten

In zuivere Prolog, maar ook in afgeleiden zoals CLP, kunnen enkel strikte logische regels op een natuurlijke manier geformuleerd worden. De evaluatie van een Prolog-query gebeurt op een deterministische manier. Statistische experimenten – zoals het opgooien van een muntstuk – kunnen in die talen echter niet op een eenvoudige manier voorgesteld worden. Eenzelfde query kan verschillende resultaten opleveren afhankelijk van statistische willekeur, wat helemaal niet wenselijk is in een zuiver logische programmeertaal. In PRISM is er dankzij de ingebouwde probabilistische predicaat wél de mogelijkheid om dergelijk statistisch non-determinisme te beschrijven.

#### Ingebouwd algemeen leer-algoritme

Vaak gebeurt er veel ‘trial-and-error’ modelleerwerk vooraleer een geschikt model gevonden wordt. Er kruipt relatief veel tijd en moeite in het schrijven van specifieke leer-algoritmes voor elk uit te proberen model. Eens we een PRISM-model hebben geschreven

---

<sup>2</sup>Constraint Logic Programming

hebben we echter automatisch ook een leer-algoritme dat de onbekende kansparameters op basis van voorbeelden zoekt. We moeten dus niet telkens opnieuw zelf een specifiek leer-algoritme verzinnen/implementeren.

### 3.2.2 Nadelen

Enkele minder wenselijke kenmerken van PRISM zijn:

#### Efficiëntie leer-algoritme

Het ingebouwde leer-algoritme in PRISM is behoorlijk efficiënt voor een *algemeen* leer-algoritme. Het gebruikte EM-algoritme (stap 2 in het leeralgoritme) werkt op de support graphs en is redelijk efficiënt: de complexiteit per iteratie is lineair in de grootte van de grafen. In de praktijk volstaan vaak enkele (tientallen) iteraties om tot convergentie te komen. De ‘bottleneck’ van het leeralgoritme is echter de eerste stap. Exhaustief alle mogelijke verklaringen zoeken (en bijhouden) kan behoorlijk veel rekenkracht (en beschikbaar geheugen) vereisen – zelfs als redundante berekeningen door *tabling* vermeden kunnen worden. Voor modellen waarbij er zeer veel mogelijke verklaringen per voorbeeld zijn is het mogelijk dat het computationeel (quasi-) onmogelijk blijkt om het ingebouwd algoritme te gebruiken. Een leer-algoritme op maat van het specifieke model zal dan toch geschreven moeten worden.

#### Niet platformonafhankelijk

Op dit ogenblik is er enkel een Windows- en een Linuxversie van PRISM beschikbaar. De broncode is niet (gratis) beschikbaar. PRISM mag enkel gebruikt worden voor evaluatie- en educatieve doeleinden. Hetzelfde geldt trouwens voor B-Prolog, waarop PRISM gebouwd is. Voor andere doeleinden is een licentie nodig. Een individuele licentie (enkel binaries) kost 115 dollar, een licentie mét broncode kost 5000 dollar. Bij het opstarten van de evaluatieversie van PRISM wordt elke keer opnieuw expliciet aan de gebruiker gevraagd of hij akkoord gaat met de licentievoorwaarden, zie figuur 3.3.

---

PRISM 1.6, Sato Lab, TITECH, All Rights Reserved. March 2003

```
** I have read the license notice and agree that the evaluation version
** can be used for evaluation and learning purposes only, and a license
** is needed for any other uses.
```

```
Proceed (Type "Yes" to proceed or any others to quit)?
```

---

Figuur 3.3: Opstartscherm van PRISM

## Hoofdstuk 4

# Hidden Markov Models

*“As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.”*  
– Albert Einstein (1879-1955)

Hidden Markov<sup>1</sup> Models (HMM's) zijn statistische modellen voor sequentiële data. Ze zijn succesvol gebleken voor verschillende toepassingen in onder andere artificiële intelligentie, patroonherkenning, spraakherkenning en bio-informatica. We zullen ons baseren op HMM's bij het modelleren van probabilistische en logische muziekkennis. Het is dus nuttig om wat dieper in te gaan op de theorie van HMM's. Dit hoofdstuk is gedeeltelijk gebaseerd op [Ben99]. We beginnen met een algemene beschrijving van HMM's in secties 4.1 en 4.2. Vervolgens worden in sectie 4.3 het *Forward Pass* en het *Viterbi* algoritme besproken. In sectie 4.4 wordt het *Baum-Welch* leer-algoritme beschreven. Tenslotte werken we in sectie 4.5 de implementatie uit van een HMM in PRISM.

### 4.1 Markov Modellen

Met de verkorte notatie  $P(x)$  bedoelen we de kans  $P(X = x)$  dat een kansvariabele  $X$  de waarde  $x$  aanneemt – waarbij het uit de context duidelijk zou moeten zijn welke kansvariabele  $X$  bedoeld wordt.

**Definitie:** *Een Markov model van orde  $k$  is een kansverdeling voor een sequentie  $q_1^T = \{q_1, \dots, q_T\}$  waarvoor de volgende eigenschap over conditionele onafhankelijkheid – ook wel de Markov-veronderstelling (Markov assumption) genoemd – geldt:*

$$\forall t \in \{1, \dots, T\} : P(q_t | q_1^{t-1}) = P(q_t | q_{t-k}^{t-1})$$

---

<sup>1</sup>Andrei Andreyevich **Markov** (1856-1922) was een Russische wiskundige die naast Markov modellen (ook Markov-ketens genoemd) o.a. poëzie en andere teksten bestudeerde als stochastische lettersequenties. Hij was een leerling van Pafnuty Lvovich **Chebyshev** (1821-1894) aan de Universiteit van Sint-Petersburg, waar hij in 1878 afstudeerde en vanaf 1886 les gaf in kansrekenen.

De kans  $P(q_t|q_1^{t-1})$  is de conditionele kans dat het element op de  $t$ -de plaats in de sequentie de waarde  $q_t$  heeft, gegeven al de vorige elementen van de sequentie. Met ander woorden, de  $k$  vorige elementen van de sequentie  $q_{t-k}^{t-1}$  bevatten alle relevante informatie voor het volgende element  $q_t$ . Gewoonlijk wordt  $q_t$  de waarde van de *toestandsvaariabele* genoemd (state variable) op tijdstip  $t$ . Gewoonlijk wordt er gewerkt met *discrete* variabelen, waarbij er dus een eindig aantal mogelijkheden zijn voor  $q_t$ . Omwille van de Markov-veronderstelling kunnen we de kansverdeling van een hele sequentie schrijven als:

$$P(q_1^T) = P(q_1^k) \prod_{t=k+1}^T P(q_t|q_{t-k}^{t-1})$$

Om het concept van een  $k$ -de orde Markov model te illustreren kunnen we het volgende eenvoudig voorbeeld beschouwen. Stel dat een robot rondloopt op een schaakbord, waarbij de robot in elke stap van z'n huidige positie naar een andere (of dezelfde) positie op het bord springt. De kans dat de robot naar een bepaalde positie springt hangt af van de  $k$  vorige posities. Voor  $k = 1$  hangt de kansverdeling voor de volgende positie enkel af van de huidige positie – een plausibele afhankelijkheid zou bijvoorbeeld kunnen zijn dat de kans groot is dat de robot naar een aangrenzende positie springt of blijft staan, en de kans klein is dat de robot naar een afgelegen positie springt. In het geval van  $k = 2$  wordt rekening gehouden met de huidige én de vorige positie – we zouden dan bijvoorbeeld kunnen uitdrukken dat de kans dat de robot in dezelfde richting blijft springen groot is. Het model wordt volledig bepaald door de  $k$  eerste posities van de robot en de conditionele kansen voor de volgende positie gegeven de  $k$  vorige posities. Die kansen kunnen geschat worden door de robot gedurende lange tijd te volgen.

Vaak wordt gewerkt met een Markov model van orde  $k = 1$ . In dat geval is de kansverdeling volledig bepaald door de zogenaamde *begintoestand-kansen*  $P(q_1)$  (initial state probabilities) en de *overgangskansen*  $P(q_t|q_{t-1})$  (transition probabilities):

$$P(q_1^T) = P(q_1) \prod_{t=2}^T P(q_t|q_{t-1})$$

In veel Markoviaanse modellen wordt verondersteld dat de overgangskansen *homogeen* zijn, waarmee bedoeld wordt dat ze gelijk zijn op elk moment in de sequentie. Voor Markov modellen van orde 1 betekent dit dat  $\forall t : P(q_t|q_{t-1}) = P(q_2|q_1)$ . Het aantal parameters wordt sterk gereduceerd door te werken met homogene modellen. Bovendien kunnen we veel gemakkelijker werken met sequenties van verschillende lengte.

Het is zinvol om homogene modellen te gebruiken voor sequentiële data die (min of meer) invariant blijft onder tijd-translaties. In het voorbeeld van de springende robot kunnen we dit aannemen, tenzij bijvoorbeeld de energie-voorraad van de robot zou afnemen naarmate er meer gesprongen wordt, waardoor de robot na een tijd minder ver kan springen en uiteindelijk – als de energie op is – enkel nog kan blijven stilstaan. Om zoiets te modelleren is een niet-homogeen model meer geschikt.

Een homogeen eerste-orde Markov model kunnen we ook beschouwen als een *probabilistische eindige toestand machine* (probabilistic finite state machine) zonder invoer<sup>2</sup>. Bij

<sup>2</sup>'Zonder invoer' is in dit geval equivalent met een constante invoerstring bestaande uit een herhaling

‘gewone’ eindige toestand machines is de volgende toestand een functie van de huidige toestand en het huidige symbool van de invoerstring. In de probabilistische variant wordt de volgende toestand willekeurig bepaald volgens een kansverdeling die afhangt van de huidige toestand en het huidige symbool van de invoerstring. Als er geen invoerstring is kunnen we een dergelijke machine modelleren als een Markov Model. Een niet-homogeen eerste-orde Markov model kunnen we eventueel beschouwen als een probabilistische eindige toestand machine waarbij de invoer op tijdstip  $t$  de waarde  $t$  heeft.

## 4.2 Verborgen toestanden: HMM's

Een belangrijk probleem van  $k$ -de orde Markov modellen is dat ze voor grote  $k$  al snel onhandelbaar worden. Stel dat er  $n$  mogelijke waarden zijn voor de toestandsvariabele  $q_t$ . Het aantal vereiste parameters om de overgangskansen voor te stellen is dan  $O(n^{k+1})$  voor homogene modellen. Voor niet-homogene modellen zijn er zelfs  $O(Tn^{k+1})$  parameters nodig. Daardoor worden we verplicht om kleine waarden voor  $k$  te gebruiken. Spijtig genoeg voldoen veel soorten van sequentiële data die we zouden willen modelleren niet aan de Markov-veronderstelling voor  $k$  klein. Vaak is het zo dat de benodigde informatie over de vorige toestanden echter wel op één of andere manier – al weten we meestal niet precies hoe – kan ‘samengevat’ worden in een andere, nieuwe toestandsvariabele.

We veronderstellen dus dat er een ‘verborgen’ toestandsvariabele  $h_t$  bestaat die aan de Markov-veronderstelling voor lage  $k$  voldoet, maar niet (rechtstreeks) te observeren is. We observeren een sequentie  $o_1^T = \{o_1, \dots, o_T\}$ . De *observatiekansen*  $\epsilon_i(o) = P(o_t = o | h_t = i)$  (observation/emission probabilities) bepalen de kansverdeling van de mogelijke observaties. Deze kansverdeling is enkel afhankelijk van de huidige waarde van de verborgen toestandsvariabele. Dit soort modellen noemen we *Hidden Markov Models* (HMM's).

Laten we ter illustratie even terugkeren naar het voorbeeld van de robot. Stel dat de robot op elke bord-positie bepaalde acties kan doen, met bijhorende kansverdelingen die enkel afhankelijk zijn van zijn huidige positie. Stel nu dat we niet kunnen zien waar de robot zich bevindt en enkel de acties kunnen observeren. De positie van de robot speelt in dit voorbeeld de rol van de verborgen toestand, de reeks acties is de observeerbare sequentie. Later zullen we proberen om uit de observaties van de acties af te leiden wat de overgangskansen (van de ene naar de andere positie) zijn en wat de observatiekansen (die de waarschijnlijkheid van de mogelijke acties op elke positie beschrijven) zijn.

Tenzij anders vermeld zullen we in de rest van deze verhandeling enkel met discrete en homogene HMM's van de eerste orde<sup>3</sup> werken – al zijn sommige resultaten, formules en algoritmes ook geldig voor algemenere HMM's. We veronderstellen dus steeds impliciet dat er een eindig aantal mogelijke waarden is voor de verborgen toestandsvariabele en de observatievariabele. Ook nemen we aan dat de overgangskansen en observatiekansen niet afhankelijk zijn van de plaats in de sequentie, enkel van de huidige verborgen toestand. We zullen steeds met  $n$  het aantal mogelijke waarden voor de verborgen toestand-variabele noteren, met  $T$  de lengte van de sequentie en met  $t$  een tijdstip in de sequentie ( $1 \leq t \leq T$ ).

---

van het enige symbool in het invoer-alfabet.

<sup>3</sup>Voor HMM's wordt bijna altijd de orde  $k = 1$  genomen, aangezien elk HMM van hogere orde kan nagebootst worden door een HMM van orde 1 met voldoende aantal mogelijke waarden voor de verborgen toestandsvariabele.

### 4.2.1 Kansparameters

Volgens de Markov-veronderstelling vormt de toestandsvariabele  $h_t$  een samenvatting van alle relevante vorige waarden van alle vorige geobserveerde en verborgen variabelen:

$$P(o_t|h_1^t, o_1^{t-1}) = P(o_t|h_t) = P(o_1|h_1) \quad (4.1)$$

$$P(h_{t+1}|h_1^t, o_1^t) = P(h_{t+1}|h_t) = P(h_2|h_1) \quad (4.2)$$

De tweede gelijkheid in beide vergelijkingen is uiteraard enkel geldig voor homogene modellen. Hieruit halen we dat de samengevoegde kansverdeling (van een sequentie verborgen toestanden en geobserveerde toestanden) volledig beschreven kan worden in termen van begintoestand-kansen  $P(h_1)$ , overgangskansen  $P(h_t|h_{t-1})$  en observatiekansen  $P(o_t|h_t)$ :

$$P(h_1^T, o_1^T) = P(h_1) \prod_{t=2}^T P(h_t|h_{t-1}) \prod_{t=1}^T P(o_t|h_t) \quad (4.3)$$

Met de (kans-)parameters  $\theta = (\tau, \epsilon, \pi)$  van een HMM bedoelen we die drie reeksen van kansen. In homogene modellen zijn de overgangskansen en observatiekansen niet afhankelijk van het tijdstip  $t$ . We noteren de overgangskans dat verborgen toestand  $i$  gevolgd wordt door verborgen toestand  $j$  dan ook verkort met  $\tau_{ij} := P(h_t = j|h_{t-1} = i)$ . Voor de observatiekans dat de observatie  $o$  gedaan wordt in verborgen toestand  $j$  gebruiken we de verkorte notatie  $\epsilon_j(o) := P(o_t = o|h_t = j)$ . De begintoestand-kansen noteren we ook wel  $\pi_i := P(h_1 = i)$ .

### 4.2.2 Topologie

We kunnen een bepaalde structuur opleggen aan de toestand-overgangen door bepaalde begintoestand- en overgangskansen nul te nemen. Vaak wordt er geëist dat alle sequenties beginnen in dezelfde begintoestand  $x$ . Dat kunnen we doen door de begintoestand-kansen te fixeren op  $\pi_x = P(h_1 = x) = 1$  en  $\forall y \neq x : \pi_y = P(h_1 = y) = 0$ . Analoog kunnen we, door sommige overgangskansen nul te nemen, uitdrukken dat er geen overgang mogelijk is van een bepaalde toestand naar een andere. De beperkingen die op deze manier opgelegd worden aan de mogelijke begintoestanden en overgangen noemen we de *topologie* van het model.

Vooraf gekende informatie over de te modelleren sequenties kunnen we gebruiken om de topologie te bepalen. Hiervoor is het wel nodig dat we (min of meer) weten wat de verborgen toestandsvariabele eigenlijk voorstelt en wat de verschillende mogelijke waarden ervan betekenen. Voor sommige toepassingen is dat helemaal niet zo. In andere toepassingen – zoals spraakherkenning – is het echter heel duidelijk wat de betekenis is van de verborgen toestandsvariabele. Een sterk beperkende topologie waarbij vanuit elke toestand slechts naar één (of enkele) andere toestand(en) kan gegaan worden is in dergelijke toepassingen dan ook mogelijk. Daardoor kan het aantal kansparameters van het model – en dus ook de hoeveelheid benodigde berekeningen – drastisch gereduceerd worden.

## 4.3 Basisalgoritmes voor HMM's

De berekening van  $P(h_1^T, o_1^T)$  is redelijk eenvoudig via vergelijking 4.3 en kan in tijd  $O(T)$  gebeuren. We zijn echter geïnteresseerd in de kansverdeling van  $P(o_1^T)$ , aangezien  $h_1^T$  niet geobserveerd wordt. Een efficiënt algoritme om die kansen te vinden is *Forward Pass*.

### 4.3.1 Forward Pass

We kunnen de kans  $P(o_1^T)$  als volgt berekenen:

$$P(o_1^T) = \sum_{h_1^T} P(h_1^T, o_1^T)$$

Het aantal termen in deze som is onaanvaardbaar groot:  $n^T$ . Gelukkig bestaat er een efficiëntere, recursieve manier om de som uit te rekenen. We kunnen  $P(o_1^t, h_t)$ , de kans dat we de deelsequentie  $o_1^t$  observeren waarbij de laatste verborgen toestand  $h_t$  is, als volgt schrijven in functie van de kansparameters en  $P(o_1^{t-1}, h_{t-1})$ :

$$\begin{aligned} P(o_1^t, h_t) &= P(o_t | o_1^{t-1}, h_t) P(o_1^{t-1}, h_t) \\ &= P(o_t | h_t) \sum_{h_{t-1}} P(o_1^{t-1}, h_t, h_{t-1}) \quad \text{wegens (4.1)} \\ &= P(o_t | h_t) \sum_{h_{t-1}} P(h_t | h_{t-1}, o_1^{t-1}) P(h_{t-1}, o_1^{t-1}) \\ &= P(o_t | h_t) \sum_{h_{t-1}} P(h_t | h_{t-1}) P(o_1^{t-1}, h_{t-1}) \quad \text{wegens (4.2)} \end{aligned}$$

We gebruiken de verkorte notatie  $F_i(t)$  om  $P(o_1^t, h_t = i)$  aan te duiden. Bovenstaande vergelijking kunnen we dus schrijven als:

$$F_i(t) = \epsilon_i(o_t) \sum_{j=1}^n \tau_{ji} F_j(t-1)$$

Op deze manier kunnen we recursief  $F_{h_t}(t)$  berekenen. Het basisgeval in deze recursie vinden we als volgt:

$$F_i(1) = P(o_1, h_1 = i) = P(o_1 | h_1 = i) P(h_1 = i) = \epsilon_i(o_1) \pi_i$$

Deze recursie staat centraal in verschillende algoritmes voor HMM's, en wordt gewoonlijk *forward pass* (of *forward phase*) genoemd. De computationele kost om op deze manier  $F_{h_T}(T) = P(o_1^T, h_T)$  te berekenen is  $O(Tn^2)$ . Hieruit halen we  $P(o_1^T)$  door eenvoudig te sommeren over alle mogelijke eindtoestanden:  $P(o_1^T) = \sum_{h_T} P(o_1^T, h_T) = \sum_{i=1}^n F_i(T)$ .

### 4.3.2 Viterbi

In veel toepassingen van HMM's heeft de verborgen toestandsvariabele een bepaalde betekenis. Bij spraakherkenning komt bijvoorbeeld met elke waarde van de verborgen toestandsvariabele een bepaald phoneem overeen. Het is dan nuttig om uit een sequentie van observaties  $o_1^T$  af te leiden wat de meest waarschijnlijke bijbehorende sequentie van verborgen toestanden  $h_1^T$  was. We vinden dit door volgende maximalisatie:

$$\arg \max_{h_1^T} P(h_1^T | o_1^T) = \arg \max_{h_1^T} \frac{P(h_1^T, o_1^T)}{P(o_1^T)} = \arg \max_{h_1^T} P(h_1^T, o_1^T)$$

Het Viterbi-algoritme vindt het bovenstaande maximum op een vrij efficiënte recursieve manier – vergelijkbaar met forward pass – in  $O(Tn^2)$ . We noemen deze meest waarschijnlijke sequentie van verborgen toestanden  $h_1^T$  de *Viterbi verklaring*<sup>4</sup>. De kans  $P(o_1^T, h_1^T)$  dat de observatiesequentie  $o_1^T$  met zijn Viterbi verklaring  $h_1^T$  voorkomt noemen we de *Viterbi-kans* van  $o_1^T$ .

Forward pass geeft de kans dat een bepaalde sequentie van observaties voorkomt, waarbij in feite alle mogelijke sequenties van verborgen toestanden beschouwd worden. Die kans is uiteraard steeds groter dan de Viterbi-kans van die sequentie.

## 4.4 Leren van kansparameters

De waarden van de kansparameters  $\theta$  van een HMM zijn gewoonlijk onbekend. We hebben enkel een reeks observatie-sequenties ter beschikking – gewoonlijk de *leer-voorbeelden* of *training set* genoemd. Met het *leren van kansparameters* bedoelen we het berekenen of schatten van de kansparameters waarvoor het model een zo goed mogelijke voorstelling vormt voor de leer-voorbeelden. We gebruiken hiervoor het *Maximum Likelihood* (ML) criterium.

We zullen ervan uitgaan dat er maar één sequentie van observaties  $o_1^T$  gegeven is – het is redelijk eenvoudig om dit eventueel uit te breiden naar meerdere sequenties. De likelihood  $\ell(\theta') = P(o_1^T | \theta = \theta')$  van  $\theta'$  is de kans dat de leer-voorbeelden geproduceerd worden door een HMM met kansparameters  $\theta = \theta'$ . Volgens het ML criterium zoeken we een maximum van deze likelihood-functie  $\ell$ . Er is geen algemene manier bekend om analytisch dit maximum te berekenen. Wél kunnen we op een iteratieve manier een benadering voor dit maximum zoeken met een *Expectation-Maximization* (EM) algoritme. In [DLR77] wordt een algemeen EM algoritme voorgesteld. Voor discrete HMM's bestaat er echter al langer een specifiek EM leer-algoritme: het *Baum-Welch* algoritme ([BP66], [BPSW70]). Convergentie is gegarandeerd voor EM-algoritmes, maar het is wel mogelijk dat het resultaat een lokaal maximum is – dus globaal gezien niet noodzakelijk optimaal.

<sup>4</sup>ook wel *Viterbi Alignment* of *Viterbi Path* genoemd

### 4.4.1 Basisidee

Neem het voorbeeld van de springende robot. We observeerden een sequentie van acties. Als we het overeenkomstige traject van de robot zouden kennen, m.a.w. als we zouden weten op welke positie de robot stond op elk moment, dan zou het gemakkelijk zijn om de overgangskansen te schatten. Een goede schatting zouden we bijvoorbeeld bekomen door de relatieve frequentie van sprongen te tellen: hoe vaak de robot van positie  $A$  naar  $B$  sprong gedeeld door hoe vaak hij in positie  $A$  was. De observatiekansen zouden dan ook gemakkelijk te schatten zijn – door de frequentie van de acties per positie te tellen. Zo vinden we een schatting voor de kansparameters via genormaliseerde frequenties.

Helaas kennen we het traject van de robot niet. We zullen dus alle mogelijke trajecten beschouwen. Daarbij geven we elk traject een gewicht gelijk aan de *a posteriori* kans dat dat traject gevolgd werd, gegeven de geobserveerde acties en de huidige schatting van de kansparameters. We zoeken voor elk tijdstip en elke positie de kans dat de robot op dat moment op die positie staat. Zo verkrijgen we een nieuwe schatting van de kansparameters op basis van gewogen frequenties. We beginnen met een ruwe schatting van de kansparameters – in de praktijk meestal een willekeurige initialisatie. Voor de gewichten gebruikten we dus niet de correcte waarden van de kansparameters, waardoor de nieuwe schatting de likelihood niet exact maximaliseert. Het kan echter aangetoond worden [DLR77] dat de likelihood dichter bij een maximum komt. Door iteratief de kansparameters op deze manier te herschatten zullen ze vrij snel convergeren naar een likelihood-maximaliserende waarde.

### 4.4.2 Baum-Welch algoritme

We beginnen met een ruwe schatting van de kansparameters  $\theta_0$ . Desnoods kiezen we de initiële kansparameters  $\theta_0$  willekeurig. In elke iteratie van het Baum-Welch algoritme zullen we een nieuwe, betere schatting van de kansparameters berekenen. Het algoritme stopt als er geen (significante) verbetering meer is – de geschatte kansparameters zijn dan geconvergeerd naar een (lokaal) optimum. We zullen in wat volgt  $P(\dots|\theta = \theta_i)$  vaak afkorten tot  $P(\dots)$ .

#### Backward-kansen

In subsectie 4.3.1 hebben we een efficiënte recursie afgeleid om de forward-kansen  $F_i(t) = P(o_1^t, h_t = i)$  te berekenen. Met  $B_i(t)$  zullen we  $P(o_{t+1}^T | h_t = i)$  aanduiden, de kans van de deelsequentie  $o_{t+1}^T$ , gegeven dat de verborgen toestand op tijdstip  $t$  de waarde  $i$  heeft. We spreken af dat  $B_i(T) = 1$ . Op een gelijkaardige recursieve manier kunnen we de kansen  $B_i(t)$  efficiënt berekenen:

$$B_i(t) = \sum_{j=1}^n B_i(t+1) \tau_{ij} \epsilon_j(o_{t+1}) \quad \text{met } 1 \leq t < T$$

De variabele  $F_i(t)$  noemen we de *forward* variabele,  $B_i(t)$  noemen we de *backward* variabele. Het Baum-Welch algoritme wordt ook wel het *Forward-Backward* algoritme

genoemd. We zullen in functie van deze twee variabelen nog twee andere hulpvariabelen definiëren.

### De *a posteriori* kansen

We definiëren  $\lambda_t(i, j)$  als de *a posteriori* kans, gegeven de observaties, dat de verborgen toestandsvariabele de waarde  $i$  heeft op tijdstip  $t$  en de waarde  $j$  op tijdstip  $t + 1$ :

$$\lambda_t(i, j) := P(h_t = i, h_{t+1} = j | o_1^T) = \frac{P(h_t = i, h_{t+1} = j, o_1^T)}{P(o_1^T)}$$

De noemer van het rechterlid kunnen we schrijven als de teller, gesommeerd over alle mogelijke waarden voor  $h_t$  en  $h_{t+1}$ . De teller kunnen we als volgt uitdrukken in functie van de kansparameters en de forward en backward variabelen:

$$\begin{aligned} P(h_t = i, h_{t+1} = j, o_1^T) &= P(o_1^t, h_t = i)P(h_{t+1} = j | h_t = i)P(o_{t+2}^T | h_{t+1} = j)P(o_{t+1} | h_{t+1} = j) \\ &= F_i(t)\tau_{ij}B_j(t+1)\epsilon_j(o_{t+1}) \end{aligned}$$

We vinden bijgevolg de volgende formule voor  $\lambda_t(i, j)$ :

$$\lambda_t(i, j) = \frac{F_i(t)\tau_{ij}B_j(t+1)\epsilon_j(o_{t+1})}{\sum_{i=1}^n \sum_{j=1}^n F_i(t)\tau_{ij}B_j(t+1)\epsilon_j(o_{t+1})}$$

Verder definiëren we nog  $\gamma_t(i)$  als de *a posteriori* kans dat, gegeven de observaties, de verborgen toestand de waarde  $i$  heeft op tijdstip  $t$ :

$$\gamma_t(i) := P(h_t = i | o_1^T) = \frac{P(h_t = i, o_1^T)}{P(o_1^T)} = \frac{F_i(t)B_i(t)}{\sum_{j=1}^n F_j(t)B_j(t)}$$

We kunnen  $\gamma_t(i)$  ook schrijven in functie van  $\lambda_t(i, j)$ :

$$\gamma_t(i) = \sum_{j=1}^n \lambda_t(i, j)$$

### Herschating van de kansparameters

Na al deze berekeningen op basis van de oude schatting  $\theta_i = (\tau, \epsilon, \pi)$  van de kansparameters kunnen we een nieuwe schatting (re-estimation)  $\theta_{i+1} = (\tau', \epsilon', \pi')$  maken. Deze nieuwe schatting zal  $P(o_1^T | \theta_i)$  zo groot mogelijk maken. Als nieuwe begintoestand-kansen gebruiken we  $\pi'_i = \gamma_1(i)$ . De nieuwe overgangskansen nemen we als volgt:

$$\tau'_{ij} := P(h_{t+1} = j | h_t = i, o_1^T) = \frac{P(h_t = i, h_{t+1} = j | o_1^T)}{P(h_t = i | o_1^T)} = \frac{\sum_{t=1}^{T-1} \lambda_t(i, j)}{\sum_{t=1}^{T-1} \gamma_t(i)}$$

Tenslotte vinden we de nieuwe observatiekansen op de volgende manier, waarbij de selectiefunctie  $S$  gedefinieerd is zodat  $S(x, y) = 0$  als  $x \neq y$  en  $S(x, y) = 1$  als  $x = y$ :

$$\epsilon'_j(o) := P(o_t = o | h_t = j, o_1^T) = \frac{P(o_t = o, h_t = j | o_1^T)}{P(h_t = j | o_1^T)} = \frac{\sum_{t=1}^T S(o_t, o) \gamma_t(j)}{\sum_{t=1}^T \gamma_t(j)}$$

De totale complexiteit van de berekening van de forward- en backward-kansen,  $\lambda$ 's en  $\gamma$ 's en nieuwe kansparameters – en dus de complexiteit per iteratie van het Baum-Welch algoritme – is  $O(Tn^2)$ .

## 4.5 Implementatie in PRISM

We zullen een concrete versie van het voorbeeld van de springende robot implementeren in PRISM. Stel dat de robot drie verschillende acties kan uitvoeren:  $a_1$ ,  $a_2$  en  $a_3$ . Er zijn vier posities op het bord:  $p_1$ ,  $p_2$ ,  $p_3$  en  $p_4$ . De beginpositie is  $p_1$ . In figuur 4.1 wordt een overzicht gegeven van de observatie- en overgangskansen. Een mogelijke implementatie in PRISM van het HMM dat hiermee overeenkomt wordt gegeven in listing 4.1.

Listing 4.1: robot.psm

```
target ( acties , 2 ).
data ( 'robot-voorbeelden.data' ).

values ( begin , [ p1 , p2 , p3 , p4 ] ).
values ( overgang ( - ) , [ p1 , p2 , p3 , p4 ] ).
values ( observatie ( - ) , [ a1 , a2 , a3 ] ).

acties ( Lengte , Lijst ) :-
    msw ( begin , Positie ) ,
    hmm ( 1 , Lengte , Positie , Lijst ).

hmm ( T , Lengte , _ , [ ] ) :- T > Lengte , !.
hmm ( T , Lengte , Positie , [ Observatie | RestLijst ] ) :-
    msw ( observatie ( Positie ) , Observatie ) ,
    msw ( overgang ( Positie ) , VolgendePositie ) ,
    T1 is T+1 ,
    hmm ( T1 , Lengte , VolgendePositie , RestLijst ).

zet_parameters :-
    set_sw ( begin , [ 1 , 0 , 0 , 0 ] ) ,
    set_sw ( overgang ( p1 ) , [ 0.7 , 0.3 , 0 , 0 ] ) ,
    set_sw ( overgang ( p2 ) , [ 0.2 , 0.6 , 0.2 , 0 ] ) ,
    set_sw ( overgang ( p3 ) , [ 0 , 0.2 , 0.5 , 0.3 ] ) ,
    set_sw ( overgang ( p4 ) , [ 0 , 0 , 0.2 , 0.8 ] ) ,
    set_sw ( observatie ( p1 ) , [ 0.9 , 0.1 , 0 ] ) ,
    set_sw ( observatie ( p2 ) , [ 0.5 , 0.5 , 0 ] ) ,
    set_sw ( observatie ( p3 ) , [ 0 , 0.1 , 0.9 ] ) ,
    set_sw ( observatie ( p4 ) , [ 0 , 0.8 , 0.2 ] ) .
```

	$a_1$	$a_2$	$a_3$	$p_1$	$p_2$	$p_3$	$p_4$
$p_1$	0.9	0.1	0.0	0.7	0.3	0.0	0.0
$p_2$	0.5	0.5	0.0	0.2	0.6	0.2	0.0
$p_3$	0.0	0.1	0.9	0.0	0.2	0.5	0.3
$p_4$	0.0	0.8	0.2	0.0	0.0	0.2	0.8

Figuur 4.1: Observatie- en overgangskansen voor de springende robot

We hebben nu een PRISM-model, waarvan we de kansparameters expliciet kennen. Met de built-in `sample` kunnen we een reeks observaties genereren op basis van die kansparameters – zie figuur 4.2. Op die manier kunnen we een bestandje maken met verschillende voorbeelden van sequenties van acties (listing 4.2) dat we vervolgens kunnen gebruiken als invoer voor het leer-algoritme.

In figuur 4.3 wordt een mogelijk resultaat gegeven van het leer-algoritme met als invoer het bestandje uit listing 4.2. De tweede stap in leer-algoritme van PRISM begint met een willekeurige initialisatie van de kansparameters. Vertrekkend vanuit die initialisatie zullen de kansparameters convergeren naar een lokaal optimum. Het is bijgevolg niet zo dat het leer-algoritme voor dezelfde invoer telkens dezelfde resultaten voor de kansparameters geeft.

---

```

PRISM 1.6, Sato Lab, TITECH, All Rights Reserved. March 2003
B-Prolog Version 6.4, All rights reserved, (C) Afany Software 1994-2003.
| ?- prism(robot)

table acties/2
table hmm/4
loading...robot.psm.out

yes
| ?- zet_parameters

Switch begin: unfixed: p1 (1) p2 (0) p3 (0) p4 (0)
Switch overgang(p1): unfixed: p1 (0.7) p2 (0.3) p3 (0) p4 (0)
[...]
Switch observatie(p4): unfixed: a1 (0) a2 (0.8) a3 (0.2)

yes
| ?- sample(acties(20,L))

L = [a1,a1,a1,a1,a1,a1,a1,a1,a2,a1,a1,a1,a3,a2,a1,a1,a1,a1,a1,a1,a1]?
yes
| ?- sample(acties(20,L))

L = [a1,a2,a2,a3,a1,a2,a1,a2,a1,a1,a1,a1,a1,a2,a1,a3,a1,a1,a2,a1]?
yes

```

---

Figuur 4.2: Twee mogelijke observaties van 20 robot-acties

Listing 4.2: robot-voorbeelden.data

```

acties (20, [a1, a1, a1, a1, a1, a1, a1, a2, a1, a1, a1, a3, a2, a1, a1, a1, a1, a1, a1, a1]).
acties (20, [a1, a2, a2, a3, a1, a2, a1, a2, a1, a1, a1, a1, a1, a2, a1, a3, a1, a1, a2, a1]).
acties (20, [a1, a1, a1, a1, a1, a2, a1, a1, a2, a2, a1, a2, a1, a1, a2, a1, a2, a3, a3, a2]).
acties (20, [a1, a1, a1, a2, a2, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a3, a2, a2]).
acties (20, [a1, a1, a1, a3, a3, a1, a1, a1, a1, a2, a1, a3, a2, a3, a3, a2, a2, a3, a1, a1]).
acties (20, [a1, a2, a1, a2, a1, a1, a2, a3, a2, a3, a1, a1, a2, a2, a3, a3, a2, a3, a2, a3]).
acties (20, [a2, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a2, a1, a2, a1, a2, a1, a1, a1]).
acties (20, [a1, a1, a2, a2, a2, a2, a3, a3, a3, a2, a2, a2, a2, a2, a3, a3, a3, a2, a2, a2]).
acties (20, [a1, a1, a2, a1, a3, a2, a1, a3, a2, a3, a2, a2, a2, a3, a2, a3, a3, a2, a3, a3]).
acties (20, [a1, a2, a3, a2, a3, a2, a1, a3, a3, a3, a2, a2, a3, a3, a2, a3, a3, a2, a2, a2]).
acties (20, [a1, a1, a1, a3, a3, a2, a3, a2, a3, a2, a2, a3, a2, a3, a2, a3, a3, a1, a1, a1]).
acties (20, [a1, a2, a1, a1, a1, a2, a1, a3, a2, a2, a3, a2, a3, a2, a2, a3, a3, a2, a1, a2]).
acties (20, [a1, a2, a2, a3, a3, a3, a3, a2, a2, a2, a3, a3, a2, a2, a2, a2, a2, a3, a2, a2]).
acties (20, [a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a1, a2, a2, a3, a2, a2, a2, a3, a2]).
acties (20, [a1, a1, a1, a1, a1, a2, a2, a3, a3, a3, a3, a3, a2, a2, a2, a2, a2, a2, a2, a2]).

```

---

PRISM 1.6, Sato Lab, TITECH, All Rights Reserved. March 2003

B-Prolog Version 6.4, All rights reserved, (C) Afany Software 1994-2003.

| ?- prism(robot)

table acties/2

table hmm/4

loading...robot.psm.out

yes

| ?- learn

[...]

ITERATION 302

likelihood0 = -248.914638 likelihood1 = -248.914541

Switch overgang(p4): unfixed: p1 (0.391240) p2 (0.0) p3 (0.273049) p4 (0.335710)

Switch observatie(p4): unfixed: a1 (0.065633) a2 (0.563788) a3 (0.370577)

Switch overgang(p3): unfixed: p1 (0.001865) p2 (0.0) p3 (0.000002) p4 (0.998131)

Switch observatie(p3): unfixed: a1 (0.0) a2 (0.999988) a3 (0.000011)

Switch overgang(p2): unfixed: p1 (0.047563) p2 (0.884310) p3 (0.068126) p4 (0.0)

Switch observatie(p2): unfixed: a1 (0.825154) a2 (0.167237) a3 (0.007607)

Switch overgang(p1): unfixed: p1 (0.334715) p2 (0.153818) p3 (0.511170) p4 (0.000296)

Switch observatie(p1): unfixed: a1 (0.0) a2 (0.000007) a3 (0.999992)

Switch begin: unfixed: p1 (0.0) p2 (1.0) p3 (0.0) p4 (0.0)

Finished learning in 1.75 seconds with 1234960 byte table space

(0.07 seconds spent in finding explanations)

yes

---

Figuur 4.3: Mogelijk resultaat van learn met als invoer robot-voorbeelden.data

Het resultaat uit figuur 4.3 moeten we niet interpreteren als een benadering van de oorspronkelijke kansparameters waarop de leer-voorbeelden gebaseerd waren. De gevonden kansparameters zijn zó gekozen dat de leer-voorbeelden (lokaal) maximaal waarschijnlijk geproduceerd zijn door het HMM met die kansparameters. Er zijn echter zeer veel verschillende waarden van de kansparameters die essentieel hetzelfde HMM geven – in die zin dat de geobserveerde uitvoer ervan dezelfde kansverdeling heeft. Een permutatie van de verborgen toestanden heeft bijvoorbeeld geen enkele invloed op de observaties. Zo speelt in het resultaat uit figuur 4.3 de toestand  $p_2$  min of meer de rol van  $p_1$  uit de oorspronkelijke kansparameters.

Het heeft bijgevolg weinig zin om verschillende versies van een HMM te vergelijken door de kansparameters naast elkaar te leggen. We zijn enkel geïnteresseerd in de kansverdeling van sequenties van observaties. De kansparameters van een HMM zeggen alleen onrechtstreeks – via het samenspel tussen overgangskansen en observatiekansen – iets over het geobserveerde gedrag. Het is dan ook geen probleem dat andere kansparameters geleerd worden dan die waarop de voorbeelden gebaseerd zijn, zolang we maar kansparameters vinden die een goede verklaring geven voor de observaties.

Het leer-algoritme van PRISM is zoals Baum-Welch een EM-algoritme dat een lokale ML-schatting zoekt. Het zal dan ook gelijkaardige resultaten geven voor de kansparameters van een HMM, op voorwaarde natuurlijk dat we een correcte beschrijving in PRISM hebben van het HMM. Sato en Kameya toonden bovendien in [SK01] aan dat het algemeen leer-algoritme van PRISM dezelfde complexiteit heeft als Baum-Welch – en dus even efficiënt is.

## Hoofdstuk 5

# Een PRISM-model voor IVL-muziek

*“The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.”*  
– John von Neumann (1903-1957)

In dit hoofdstuk construeren we een PRISM-model (zie hoofdstuk 3) voor muziek in de IVL-notatie die we in hoofdstuk 2 definieerden. We baseren ons daarbij op de Hidden Markov Models uit het vorig hoofdstuk. Dit PRISM-model zullen we zowel voor het leren van muziekkennis aan de hand van voorbeelden als voor het classificeren en genereren van muziek gebruiken.

### 5.1 Basismodel

Laat ons veronderstellen dat een componist bij het componeren van een muziekstuk van het begin naar het einde werkt en alle stemmen tegelijkertijd componeert. Anders uitgedrukt veronderstellen we dat het beluisteren en het componeren van een muziekstuk op gelijkaardige manier gebeuren – alle stemmen simultaan en lineair in de tijd. Dit is een redelijk onrealistische veronderstelling. Een componist zal bijvoorbeeld niet noodzakelijk het begin van een muziekstuk eerst te componeren. Ook zal hij vaak eerst een melodie lijn componeren en pas daarna de andere stemmen invullen. Toch is het geen zinloze veronderstelling.

Deze veronderstelling laat namelijk toe om het beluisteren en het componeren van muziek als analoge processen te beschouwen. Het enige verschil is dat bij het beluisteren de muziek geïnterpreteerd wordt en bij het componeren de muziek gecreëerd wordt. Zoals de luisteraar op verschillende momenten tijdens de beluistering verschillende emoties ervaart, zo bevond de componist zich tijdens het componeren in verschillende ‘mentale toestanden’.

We zullen hetzelfde model gebruiken om ‘muziek te beluisteren’ (leren op basis van bestaande muziek en bestaande muziek classificeren) als om ‘muziek te componeren’ (*sam-*

plen van het model). Ons model zal met andere woorden dus zowel een *analytisch* als een *synthetisch* model zijn (zie subsectie 1.1.1 en [Con03]). In zekere zin stellen we daardoor op elk moment in een muziekstuk de ‘mentale toestand’ van de luisteraar gelijk aan de ‘mentale toestand’ waarin de componist zich bevond toen hij de noten op dat moment in het muziekstuk componeerde.

Deze ‘mentale toestand’ van de componist kunnen we natuurlijk niet rechtstreeks observeren – toch zeker niet in het geval van muziek van overleden componisten. Ook het observeren van de ‘mentale toestand’ van een luisteraar is onmogelijk (of toch alleszins niet de bedoeling in deze verhandeling). We kunnen een muziekstuk dan ook beschouwen als de uitvoer van een Hidden Markov Model waarbij de verborgen toestanden overeenkomen met de ‘mentale toestanden’ van de componist. De precieze betekenis van een dergelijke toestand kennen we in feite niet. We nemen aan dat die een soort ‘samenvatting’ bevat van het muziekstuk tot het huidig moment, met de relevante informatie over bijvoorbeeld de harmonische, melodische en ritmische structuren.

Leren op basis van voorbeelden – bestaande muziekstukken – houdt in dat voor elke toestand kansverdelingen worden gezocht voor de mogelijke observaties in die toestand en voor de overgang naar de mogelijke volgende toestanden. In principe is het dus niet nodig om op voorhand vast te leggen wat de toestand betekent: het leer-algoritme zoekt in feite de betekenis die het relevantste is om de bestaande muziekstukken te modelleren. We zullen dan ook geen topologie opleggen aan de overgangskansen – van elke toestand kunnen we naar elke andere toestand overgaan. Wat we wél op voorhand moeten kiezen is het aantal verschillende waarden van de verborgen toestand. Als we dit aantal te klein kiezen dan kan niet alle relevante informatie voorgesteld worden. Als dit aantal echter te groot genomen wordt is er onvoldoende veralgemening. In dat geval zal het leer-algoritme in feite de voorbeelden letterlijk memoriseren in plaats van de onderliggende muzikale mechanismen te leren. Bovendien neemt de reken- en geheugencost toe naarmate het aantal toestanden groeit. In de praktijk zullen we het aantal toestanden gewoonlijk zo groot kiezen als de beschikbare reken- en geheugencapaciteit toelaat.

In listing 5.1 wordt een PRISM programma gegeven dat een dergelijk model vormt voor muziek in IVL-notatie. Het aantal toestanden is hier nogal arbitrair 7 gekozen, maar dat is uiteraard eenvoudig aan te passen. Merk op dat we de lijst in IVL-notatie samen met de lengte en het aantal stemmen ervan ‘verpakken’ in `song/3`. Dit is nodig om te kunnen aangeven hoe lang een te genereren sequentie moet zijn.

## 5.2 Verfijning van het basismodel

In deze sectie zullen we verbeteringen aanbrengen aan het basismodel uit listing 5.1. Deze verbeteringen zijn gebaseerd op intuïtieve muzikale principes of – zoals in het geval van de eerste verfijning – op de wens dat het model enkel zinvolle IVL-uitvoer kan produceren.

### 5.2.1 Correcte old/new uitvoer

Op regel 35 van het basismodel wordt bepaald of een noot “old” of “new” is door dit als een kans-observatie te beschouwen die enkel afhangt van de huidige toestand. Op die

Listing 5.1: music-1.psm

```

1 target(song,3).      % We observe ground atoms song(NV,Length,IVL-List)
2
3 % msw values:
4 values(tr_ss(-),[1,2,3,4,5,6,7]).      % HMM state transitions
5 values(out_L(-),[8,16,32,64]).      % duration symbols
6 values(out_V(-),[old,new]).      % old/new symbols
7 values(out_note(-),[r,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,
8     40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,
9     62,63,64,65,66,67,68,69,70,71,72]). % note symbols
10
11 % model:
12 song(NV,L,IVL):-      % song IVL of length L with NV voices
13     hmm(NV,1,L,1,IVL).      % begin HMM in song state 1
14
15 % NV: number of voices      T: current time (phase)
16 % Length: length of song      SS: current song state
17 % (L,V,N): (duration,old/new-list,notelist)
18 hmm(NV,T,Length,SS,[(L,V,N)|Y]) :-
19     T < Length,
20     observe(NV,SS,L,V,N),      % observation of (L,V,N)
21     msw(tr_ss(SS),SS_Next),      % transition to next song state
22     T1 is T+1,      % increase time
23     hmm(NV,T1,Length,SS_Next,Y). % recursion
24 hmm(NV,T,T,SS,[(L,V,N)]) :-
25     observe(NV,SS,L,V,N).      % done
26
27 observe(NV,SS,L,V,N) :-      % observation of one phase
28     msw(out_L(SS),L),      % duration output is L in this state
29     nlist(NV,SS,N,V).      % check note list
30
31 % NV: number of voices left to check
32 nlist(NV,SS,[Note|RN],[OldNew|RON]) :-
33     NV > 0,
34     msw(out_note(SS),Note),      % check note itself
35     msw(out_V(SS),OldNew),      % check if note is a new one
36     NV1 is NV-1,
37     nlist(NV1,SS,RN,RON).      % next voice
38 nlist(0,-,[],[]).      % done

```

Listing 5.2: check\_new/4

```

% if two notes are the same, the second one can be either 'old' or 'new'
check_new(SS,X,X,OldNew) :-
    number(X),                                     % unless they are both rests
    msw(out_V(SS),OldNew).

check_new(_,r,r,old).                             % second rest is always an 'old' note

check_new(_,r,A,new) :-                           % a rest is always 'new'
    number(A).                                     % when it comes after a note

check_new(_,A,r,new) :-                           % a note after a rest is also always 'new'
    number(A).

check_new(_,A,B,new) :-                           % it must be a 'new' note
    number(A),number(B),                         % when the two non-rest notes
    A \= B.                                       % are different

```

manier kan er ongeldige IVL-uitvoer gegenereerd worden, bijvoorbeeld als “old” gekozen wordt als de huidige noot niet dezelfde toonhoogte heeft als de vorige. We zullen dus aan `nlist/4` ook de lijst van de noten uit de vorige fase meegeven – waardoor `nlist` uiteraard ariteit 5 krijgt – zodat de huidige noten vergeleken kunnen worden met de vorige. We definiëren het hulp-predicaat `check_new/4` zoals in listing 5.2 zodat enkel wanneer zowel “old” als “new” mogelijk zijn `msw/2` wordt gebruikt. Als we nu de `msw/2`-check op regel 35 vervangen door “`check_new(SS,Note,PrNote,OldNew)`” dan is correcte IVL-uitvoer gegarandeerd.

### 5.2.2 Opsplitsen out\_note

Er zijn nogal veel mogelijke waarden voor “out\_note”. Eigenlijk werden drie verschillende aspecten van een noot samengenomen: of het een rust is of een ‘echte noot’, het octaafnummer en het noot-nummer. We kunnen eerst kijken of de noot een rust is of een ‘echte noot’. In het geval van ‘echte noten’ hebben we een toonhoogte die we verder kunnen opsplitsen in een octaafnummer en een noot-nummer tussen 0 (do) en 11 (si).

Deze opsplitsing van toonhoogte in octaaf en noot is muziektheoretisch verdedigbaar: het octaaf waarin een toonhoogte valt, is wat harmonie betreft minder belangrijk dan de noot die die toonhoogte voorstelt. Veel muziek-regels zijn in feite geldig voor alle octaven. Zo is bijvoorbeeld de combinatie van de noten *do* en *fa kruis* altijd dissonant, onafhankelijk van het octaaf waarin deze noten voorkomen. Als we geen opsplitsing zouden maken tussen octaaf en noot dan zouden we deze regel moeten invoeren voor alle octaven. Dit soort redundantie is natuurlijk niet gewenst. Door de opsplitsing te maken vermijden we die redundantie. Bovendien zorgen we er zo voor dat heel wat minder kansparameters moeten geleerd worden. Dat is computationeel van groot belang. Het aantal kansparameters om de observatiekansen van een toonhoogte voor te stellen wordt van een 70-tal gereduceerd tot minder dan 20. Hierbij verliezen we in feite geen (muzikaal relevante) expressiviteit.

Listing 5.3: check\_note/3

```

values(out_rest(_),[rest,note]).           % note or rest?
values(out_octave(_),[-2,-1,0,1,2]).       % relative octave
values(out_modnote(_),[0,1,2,3,4,5,6,7,8,9,10,11]). % pitches

check_note(SS,Note,PrevOct):-
    msw(out_rest(SS),X),                    % is it a rest or a real note?
    check_note(SS,Note,PrevOct,X).

check_note(_ ,Note, _ ,rest):-              % note is a rest
    Note = r.
check_note(SS,Note,PrevOct,note):-         % note is a real note:
    check_real_note(SS,Note,PrevOct).       % check pitch & octave

check_real_note(SS,Note,PrevOct):-
    number(Note),                           % training
    ModNote is Note mod 12,
    Octave is Note // 12,
    OctDiff is Octave - PrevOct,
    msw(out_modnote(SS),ModNote),
    msw(out_octave(SS),OctDiff).

check_real_note(SS,Note,PrevOct):-
    var(Note),                               % sampling
    msw(out_modnote(SS),ModNote),
    msw(out_octave(SS),OctDiff),
    NewOctave is PrevOct + OctDiff,
    sanity_check(NewOctave,SaneNewOctave), % stay within boundaries
    NewNote is ModNote + 12*SaneNewOctave,
    Note = NewNote.

```

Verder zullen we in plaats van absolute octaaf-nummers te gebruiken, werken met relatieve ‘octaaf-verschillen’ ten opzichte van het octaaf-nummer van de vorige noot. De meeste luisteraars horen namelijk niet in welk (absoluut) octaaf een toon ligt, maar ze horen wél of de toon in hetzelfde octaaf, een octaaf hoger of een octaaf lager ligt dan de vorige toon. Als de vorige noot een rust was, wordt er vergeleken met de noot daarvoor (en desnoods met de noot dáárvoor, enzovoort).

In listing 5.3 wordt een definitie van `check_note/3` gegeven waarbij deze opsplitsing van `out_note` in drie componenten `out_rest`, `out_octave` en `out_modnote` wordt gebruikt. Als we dit model gebruiken om IVL-uitvoer te genereren, is het mogelijk dat het absoluut octaaf-nummer op een gegeven moment te hoog of te laag wordt. Daarmee bedoelen we dat noten met dat octaaf-nummer zo laag of zo hoog zijn dat ze buiten het bereik van het menselijk gehoor vallen – of buiten het bereik van elk instrument. Daarom wordt dit met het `sanity_check/2`-predicaat binnen bepaalde redelijke grenzen gehouden. Zie listing C.1 voor de definitie van `sanity_check/2`.

Aan `check_new/4` moesten we de vorige noot doorgeven (zie vorige subsectie). Aan `check_note/3` moeten we het vorige absolute octaaf-nummer doorgeven zodat het verschil kan berekend worden aan de hand van het huidig en het vorig absoluut octaaf-nummer (in het geval we het model trainen) of het huidig absoluut octaaf-nummer kan berekend

worden aan de hand van het vorige absoluut octaaf-nummer en het verschil (in het geval van *sampling*). We kunnen het vorige octaaf-nummer afleiden uit de vorige noot als dat een ‘echte noot’ was. Als de vorige noot een rust was kan dat echter niet. Daarom is het nodig dat we aan `nlist` zowel een lijst van de vorige noten van elke stem (`N_prev`) als een lijst van de vorige absolute octaaf-nummers van elke stem (`O_prev`) doorgeven. De lijst `N_prev` is gewoon gelijk aan de notenlijst `N` uit de vorige IVL-fase. Het predicaat `octaves/4` (zie listing C.1) construeert de lijst `O_prev` op basis van `N_prev` en de vorige versie van `O_prev`.

In de eerste IVL-fase hebben we nog geen ‘vorige noten’. We beginnen daarom eenvoudigheidshalve pas vanaf de tweede fase. Bij het genereren van IVL-uitvoer zullen we dus naast de lengte en het aantal stemmen ook de eerste noot van elke stem en de duur van de eerste fase als gegeven beschouwen. Als in de eerste fase in sommige stemmen rusten voorkomen, dan hebben we voor de eerste ‘echte noot’ in die stemmen geen vorige absoluut octaaf-nummer om mee te werken. In dat geval gebruiken we de default-waarde “4” (centraal octaaf) als zagezegd vorig octaaf-nummer.

### 5.2.3 Transponeren naar een gemeenschappelijke toonaard

Essentieel verandert een muziekstuk niet als het getransponeerd wordt, d.w.z. als elke noot systematisch dezelfde toonafstand verhoogd of verlaagd wordt. Als elke noot van een stuk geschreven in re groot (D) twee halve tonen verlaagd wordt, krijgen we een stuk in do groot (C). Het is dus redelijk om ervoor te zorgen dat hetgeen geleerd wordt op basis van voorbeelden onafhankelijk is van de toonaard van die voorbeelden.

We zullen aan elke voorbeeld-sequentie een getal `BT` (BasisToon) tussen 0 en 11 toekennen dat uitdrukt in welke toonaard het staat: 0 voor C/Amin, 2 voor D/Bmin, etc. Daarbij maken we geen onderscheid tussen grote en kleine toonaarden als die dezelfde toonladder gebruiken. Als we elke toonhoogte verlagen met dat getal, worden de voorbeelden in zekere zin ‘genormaliseerd’ tot de gemeenschappelijke toonaard do groot/la klein. Dit getal zullen we manueel vaststellen per voorbeeld. Er bestaan heuristieken om dit automatisch te doen, maar dat zou ons te ver leiden. Meestal volstaat het om naar de voortekening en de slotnoten te kijken.

### 5.2.4 Harmonische beperkingen

De meeste muziekstukken gebruiken vooral noten uit de toonladder van de toonaard waarin ze geschreven staan, aangevuld met noten uit ‘nabijgelegen’ toonladders. De noten van de toonladder van do groot zijn *do, re, mi, fa, sol, la* en *si*. De ‘nabijgelegen’ toonaarden van do groot zijn *fa* groot (subdominant), *sol* groot (dominant) en do klein, met respectievelijke toonladders (*fa, sol, la, si mol, do, re, mi*), (*sol, la, si, do, re, mi, fa kruis*) en (*do, re, mi mol, fa, sol, la mol, si mol*).

In plaats van elke toonhoogte toe te laten in elke toestand, zullen we voor elke toestand de mogelijke toonhoogtes beperken tot de noten van een bepaalde toonladder. Niet alleen beperken we zo het aantal mogelijke uitvoer-noten, wat het aantal kansparameters verkleint, maar ook ‘forceren’ we de toestandsvariabele om een bepaalde ‘harmonische

Listing 5.4: out\_modnote

```

                                % relative pitches (scaled to C)
                                % possible values depend on SS
                                % 1: only C chord notes
% c e g
values(out_modnote(1),[0,4,7]).
% c d e f g a b
values(out_modnote(2),[0,2,4,5,7,9,11]). % 2: seven notes of C
% c d e fis g a b
values(out_modnote(3),[0,2,4,6,7,9,11]). % 3: seven notes of G
% c d e f g a bes
values(out_modnote(4),[0,2,4,5,7,9,10]). % 4: seven notes of F
% c es e f g as bes
values(out_modnote(5),[0,3,4,5,7,8,10]). % 5: +/- Cmin and Fmin7
% c d es f a bes
values(out_modnote(6),[0,2,3,5,9,10]). % 6: +/- F and Bes
% c d e f g a b
values(out_modnote(7),[0,1,2,3,4,5,6,7,8,9,10,11]). % 7: all pitches

```

toestand' uit te drukken. Op die manier zorgen we ervoor dat de kennis van de verschillende veelvoorkomende toonladders al expliciet in het model aanwezig is zonder dat dit geleerd moet worden uit voorbeelden. Het leren kan zich dan concentreren op melodieën binnen een toonaard en overgangen tussen toonaarden.

Merk op dat deze 'harmonische beperkingen' mogelijk gemaakt worden dankzij de vorige verfijning (transponeren naar een gemeenschappelijke toonaard). We kunnen nu immers veronderstellen dat het stuk geschreven is in do groot. In listing 5.4 wordt een mogelijke verfijning van `out_modnote` gegeven. Een andere harmonische indeling is natuurlijk ook mogelijk.

### 5.2.5 Voice states

Tot hier toe namen we aan dat – onafhankelijk van het aantal stemmen – al de relevante informatie over alle vorige fases kan samengevat worden in één enkele toestandsvariabele, de 'song state' (SS). Zoals we in het volgende hoofdstuk zullen zien is dit voor sommige toepassingen (classificatie) een veronderstelling die een bruikbaar model geeft. Voor andere toepassingen hebben we echter een gedetailleerder model nodig en is deze veronderstelling te sterk.

Het lijkt namelijk erg onwaarschijnlijk dat de 'song state' – die uiteindelijk om computationele redenen slechts een beperkt aantal waarden kan aannemen – voldoende informatie kan bevatten om lokaal gedrag op het niveau van een stem te beschrijven, hoe groot het aantal stemmen ook is. Bovendien is dit model in zekere zin niet onafhankelijk van het aantal stemmen. Als de 'song state' namelijk verondersteld wordt om ook informatie over de ontwikkelingen binnen de stemmen te bevatten, dan zijn de kansparameters die geleerd worden op basis van voorbeelden eigenlijk afhankelijk van het aantal stemmen van die voorbeelden.

## ‘Song HMM’ en ‘Voice HMM’

Om bijvoorbeeld melodische ontwikkelingen in een stem te modelleren zou het interessant zijn als we voor elke stem ook een verborgen toestand zouden bijhouden. We zullen dus naast het ‘song HMM’ ook voor elke stem een ‘voice HMM’ invoeren. De noot-observatie krijgt dan een kansverdeling die afhankelijk is van de huidige ‘song state’ van het globaal ‘song HMM’ en van de huidige ‘voice state’ van het ‘voice HMM’ van de stem. Na elke fase gaat – zoals voorheen – het ‘song HMM’ over naar de volgende toestand met een kansverdeling die afhankelijk is van de huidige ‘song state’. Nu gaat echter ook elk ‘voice HMM’ over naar de volgende toestand met een kansverdeling die afhankelijk is van de huidige ‘song state’ én van de huidige toestand van het ‘voice HMM’. Met andere woorden, de observatie- en overgangskansen van een ‘voice HMM’ hangen af van de huidige toestand ervan én van de huidige ‘song state’.

In feite beschrijft de ‘song state’ de *globale* toestand van alle stemmen tesamen terwijl de ‘voice states’ de *lokale* toestand van elke stem apart beschrijven. Het ‘song HMM’ modelleert aspecten zoals de harmonische onderbouw en het algemeen ritme (via `out_L`). Het ‘voice HMM’ modelleert aspecten zoals melodie en plaatselijk ritme (via `out_V`). We maken daarbij de impliciete veronderstelling dat elke stem zich fundamenteel gelijkaardig gedraagt – de parameters van het ‘voice HMM’ zijn voor elke stem identiek – maar niet noodzakelijk op elk moment dezelfde toestand moet hebben.

### Totale toestand

De *totale toestand* van het model op een gegeven tijdstip (IVL-fase) is nu gegeven door de ‘song state’ en voor elke stem de toestand van het ‘voice HMM’ van die stem (de ‘voice states’). Met  $N$  stellen we het aantal stemmen voor, met  $S$  het aantal mogelijke waarden voor de ‘song state’ en met  $V$  het aantal mogelijke waarden voor de ‘voice state’. Het aantal mogelijke totale toestanden per fase is dan gelijk aan  $SV^N$ . Zonder ‘voice states’ was dit slechts  $S$  en dus niet afhankelijk van het aantal stemmen  $N$ .

Leren op basis van muziekstukken met een groot aantal stemmen (bijvoorbeeld een symfonie met  $N = 10$ ) was dan ook computationeel mogelijk in het model zonder ‘voice states’. Dit zal met de huidige implementatie van PRISM en beschikbare rekenkracht niet meer mogelijk zijn als we ‘voice states’ invoeren. We zullen dus het  $N$  niet te groot mogen nemen. Naast het aantal stemmen  $N$  zullen we ook het aantal mogelijke waarden voor de ‘voice state’  $V$  redelijk klein moeten kiezen. Uiteraard is  $V = 1$  een zinloze en triviale keuze die op hetzelfde neerkomt als een model zonder ‘voice states’. We zullen voor zowel  $N$  als  $V$  gewoonlijk de waarde 2, 3 of 4 nemen.

## 5.3 Beperkingen op de leer-voorbeelden

We hebben nu een model dat kan leren op basis van geldige IVL-voorbeelden en geldige IVL-uitvoer kan produceren. In de praktijk blijkt echter dat het leer-algoritme van PRISM al na één iteratie stopt als we hele muziekstukken als leer-voorbeelden gebruiken. Aangezien de likelihood zó dicht bij nul ligt, zal – vanwege de beperkte rekennauwkeurigheid – de log-likelihood zowel voor als na de iteratie “-inf” zijn. Aangezien de log-likelihood gelijk

blijft, is aan het stop-criterium voor het leer-algoritme voldaan. Het zal dus noodzakelijk zijn om beperkingen op te leggen aan de leer-voorbeelden zodat dit niet kan gebeuren.

De kans dat een IVL-sequentie geproduceerd wordt door ons model neemt in het algemeen sterk af naarmate het aantal stemmen  $N$  en de lengte  $L$  van de sequentie toenemen. In de volgende subsecties zullen we hiervoor een intuïtieve verklaring geven. We zullen dus het aantal stemmen  $N$  en de lengte  $L$  van de leer-voorbeelden moeten beperken. Deze twee beperkingen op de leer-voorbeelden zijn er puur omwille van praktische, computationele redenen. In theorie blijft ons model dus even algemeen als voorheen.

### 5.3.1 Beperkt aantal stemmen

De kans van een IVL-sequentie is gelijk aan de som over alle mogelijke sequenties van totale toestanden  $T_1^L$  van het product van  $L - 1$  totale overgangskansen en  $L - 1$  totale observatiekansen. Met *totale overgangskans* bedoelen we de kans  $P(T_{i+1}|T_i)$  en met *totale observatiekans* bedoelen we de kans dat een bepaald IVL-drietal geobserveerd wordt in een gegeven totale toestand. Beide kansen worden in het algemeen kleiner naarmate het aantal stemmen  $N$  toeneemt. De totale overgangskans is immers het product van een overgangskans van het ‘song HMM’ en  $N$  overgangskansen van de ‘voice HMMs’. Dat verklaart alvast waarom de kans van een IVL-sequentie kleiner wordt als het aantal stemmen toeneemt.

Zoals in de vorige sectie al is aangehaald zullen we voorbeelden gebruiken met een redelijk klein aantal stemmen. Gewoonlijk zullen we ons beperken tot twee- of driestemmige stukken. Deze beperking alleen zal echter niet volstaan.

### 5.3.2 Beperkte lengte

Als de lengte van de sequentie één groter wordt, dan wordt het aantal mogelijke sequenties van totale toestanden vermenigvuldigd met een factor  $SV^N$ . De te berekenen som zal dus  $SV^N$  keer meer termen bevatten. Elke term wordt met één extra totale overgangskans en één extra totale observatiekans vermenigvuldigd, doordat de lengte  $L$  met één verhoogd werd. Totale overgangskansen zijn echter gemiddeld ongeveer  $1/(SV^N)$ . Daardoor wordt gecompenseerd voor het verhoogde aantal termen. Het netto-effect is dus een extra factor van ongeveer één totale observatiekans. Deze totale observatiekansen zijn typisch erg kleine kansen. De hele kans van de IVL-sequentie zal dus (exponentieel) verkleinen naarmate de lengte toeneemt.

Een volledig muziekstuk in IVL-notatie bestaat typisch uit enkele honderden tot duizenden fases. We zullen moeten werken met voorbeeld-sequenties die veel korter zijn dan volledige stukken. Daarom zullen we elk stuk eerst opsplitsen in deelsequenties van een twintigtal fases. Die lengte is natuurlijk nogal arbitrair gekozen.

## 5.4 Besluit

De volledige broncode van het PRISM-model zoals we het construeerden in secties 5.1 en 5.2 wordt gegeven in listing C.1 (in bijlage C). In deze sectie geven we een bondige samenvatting in woorden van de werking van dit uiteindelijke model. We sluiten dit hoofdstuk af met een schets van mogelijke toekomstige verbeteringen aan dit model.

### 5.4.1 Samenvatting

Het ‘song HMM’ start in initiële toestand “1”, elk ‘voice HMM’ start in initiële toestand “a”. Tevens nemen we voor elke stem als ‘vorige absoluut octaaf-nummer’ de waarde 4. We beginnen nu elke fase te overlopen, te beginnen met de eerste.

Stel dat we in fase T in totale toestand (SS,VS) zitten. Eerst observeren (of genereren) we een IVL-drietal (L,V,N), waarbij buiten de totale toestand ook de lijsten `N_prev` (vorige noten) en `O_prev` (vorige octaaf-nummers) van belang zijn. In de eerste fase slaan we deze observatie-stap over. Vervolgens gaan we over naar de volgende totale toestand (`SS_Next`,`VS_Next`) volgens de overgangskansen `tr_ss(SS)` en `tr_vs(SS,VS)`. We berekenen nu met `octaves/4` de absolute octaaf-nummers `N_octave` van de geobserveerde (of gegenereerde) notenlijst N, waarbij we het vorige absoluut octaaf-nummer nemen als de noot in kwestie een rust is. Nu kunnen we naar de volgende fase T+1 gaan, waarin we in totale toestand (`SS_Next`,`VS_Next`) zitten. Als lijst `O_prev` in fase T+1 nemen we de zojuist berekende lijst `N_octave`. De lijst `N_prev` in fase T+1 is gelijk aan de notenlijst N uit fase T.

In de observatie-stap (`observe/9`) controleren (of genereren) we eerst de duur L van de huidige fase (`out_L`). Vervolgens controleren (of genereren) we voor elke stem de huidige klinkende noot in die stem (met `check_note/5`), en of het een ‘nieuwe’ noot is of niet (`out_V`). Om de huidige noot te bepalen (of te controleren) gaan we eerst na of het een rust is of een ‘echte noot’ (`out_rest`). Als het om een ‘echte noot’ gaat bekijken we de toonhoogte, die we opsplitsen in een relatief octaaf-nummer (`out_octave`) en een noot-nummer (`out_modnote`). In het geval dat het absoluut octaaf-nummer niet binnen redelijke grenzen ligt passen we dit aan met `sanity_check`.

Bij het bepalen of een noot ‘nieuw’ is of niet, is het nodig om te vergelijken met de vorige noot. Een noot kan bijvoorbeeld enkel ‘oud’ zijn als ze gelijk is aan de vorige. Daarom geven we de lijst `N_prev` door aan `observe/9`. Verder hebben we het vorige absoluut octaaf-nummer nodig om de toonhoogte te vinden aan de hand van een relatief octaaf-nummer en een noot-nummer. De lijst `O_prev` moet daarom ook doorgegeven worden.

Alle observatiekansen zijn afhankelijk van de huidige totale toestand, behalve `out_L`. Aangezien de duur L niet bij een bepaalde stem hoort maar bij heel de huidige IVL-fase, hangt de kansverdeling ervan enkel af van de huidige ‘song state’.

Als we de laatste fase bereikt hebben doen we nog een laatste observatie-stap. In het geval van *sampling* eindigen we dan; in het geval van leren op basis van voorbeelden zullen nu door *backtracking* alle mogelijke verklaringen (sequenties van totale toestanden) gezocht worden.

## 5.4.2 Mogelijke verdere verfijningen

Het model dat we in dit hoofdstuk construeerden houdt nog geen (of onvoldoende) rekening met een aantal dingen. Er zijn dan ook tal van verdere verfijningen van dit model denkbaar.

In subsectie 2.2.3 hadden we besloten dat ritme belangrijker is dan tempo. Eigenlijk is bijvoorbeeld de opeenvolging (8<sup>ste</sup>, 4<sup>de</sup>, 8<sup>ste</sup>) in zekere zin muzikaal equivalent met de opeenvolging (16<sup>ste</sup>, 8<sup>de</sup>, 16<sup>ste</sup>). Ons model houdt hier geen rekening mee. Een mogelijke oplossing hiervoor is werken met relatieve fase-duur in plaats van absolute.

De kansverdeling van `mod_note` is in ons model enkel afhankelijk van de huidige ‘song state’ en ‘voice state’. Het is echter zo dat langdurige lage noten bijna nooit dissonant zijn, terwijl korte hoge nootjes vaak juist wel dissonante (versierings-)noten zijn. We zouden dus misschien beter `mod_note` ook van bijvoorbeeld de duur van de fase en/of van het huidig (absoluut) octaaf-nummer laten afhangen.

Ons model legt geen topologie op aan de mogelijke overgangen tussen de toestanden. Het is ook niet echt duidelijk welke topologie gepast zou kunnen zijn, aangezien we de betekenis van de toestanden niet precies kennen. We zouden echter verschillende strengere topologieën kunnen uitproberen. Voor het ‘voice HMM’ zouden we bijvoorbeeld een cyclische topologie kunnen gebruiken: met vier toestanden ( $a, b, c, d$ ) betekent dat dat we van  $a$  naar  $a$  en  $b$  kunnen, van  $b$  naar  $b$  en  $c$ , van  $c$  naar  $c$  en  $d$  en van  $d$  naar  $d$  en  $a$ . Met een dergelijke topologie zijn er slechts  $4S$  kansparameters voor 4 toestanden – zonder topologie zouden dat er  $12S$  zijn. Op die manier kunnen we dus een groter aantal toestanden gebruiken zonder het aantal te leren kansparameters te verhogen.

We gaan er van uit dat in elke stem elke toonhoogte in principe kan voorkomen, binnen de grenzen opgelegd door `sanity_check/2`. Dat is niet realistisch, aangezien de meeste instrumenten een bepaald bereik hebben dat maar enkele octaven omvat. Een mogelijke verfijning is om van elke stem vast te leggen wat het bereik is van het instrument dat die stem speelt. In plaats van elke stem dezelfde ruime grenzen op te leggen kunnen we dan nauwkeurige grenzen gebruiken die anders kunnen zijn voor elke stem.

Uitbreidingen van de IVL-notatie – en de daarbij horende uitbreidingen van het PRISM-model – zijn natuurlijk ook een mogelijkheid. We zouden bijvoorbeeld ook de dynamiek kunnen voorstellen door elke noot een (absolute of relatieve) volume-waarde te geven. In feite zouden we bijna elk aspect van muziek kunnen toevoegen aan de IVL-notatie waarvan we in sectie 2.2 besloten dat het niet essentieel is.

Het zou ons te ver leiden om al deze verfijningen en uitbreidingen te implementeren. We zullen het houden bij het model van listing C.1. In het volgende hoofdstuk zullen we zien dat zelfs dit al bij al relatief ruw model interessante resultaten geeft.

## Hoofdstuk 6

# Experimenten en resultaten

*“The first question I ask myself when something doesn’t seem to be beautiful is why do I think it’s not beautiful. And very shortly you discover that there is no reason.”*  
– John Cage (1912-1992)

Met het model dat we in het vorige hoofdstuk construeerden zullen we twee experimenten uitvoeren. In dit hoofdstuk wordt beschreven hoe we daarbij te werk gaan. Verder worden enkele resultaten van de experimenten gegeven en bondig besproken.

Voor het eerste experiment – automatisch classificeren op componist (sectie 6.1) – kan de kwaliteit van de resultaten objectief nagegaan worden. Hoe meer stukken juist geïdentificeerd worden, hoe beter. In het tweede experiment – automatisch componeren van muziek (sectie 6.2) – is de beoordeling van de kwaliteit van de uitvoer echter voor een groot deel een kwestie van persoonlijke smaak.

In bijlage B wordt de automatische omzetting van muziek in Lilypond-formaat naar IVL-notatie en omgekeerd geïmplementeerd. De leer-voorbeelden in dit hoofdstuk zijn op die manier automatisch omgezet naar IVL. Ze komen uit het *Mutopia Project* [Mut], een project dat tot doel heeft een bibliotheek van vrij te downloaden muziek in Lilypond-formaat aan te leggen.

Met de default-waarde  $\epsilon = 0.0001$  voor het leer-algoritme van PRISM (zie subsectie 3.1.3) worden te veel EM-iteraties gedaan waardoor te weinig gegeneraliseerd wordt. We zullen die default-waarde dus verlagen (bijvoorbeeld tot  $\epsilon = 0.05$ ) met behulp van de built-in `set_epsilon/1`. Op die manier zal het EM-algoritme sneller eindigen, waardoor een beter evenwicht tussen generalisatie en memorisatie bereikt wordt.

### 6.1 Automatische classificatie

In een eerste experiment proberen we de componist van een onbekend stuk te raden aan de hand van gekende stukken. Dit is het probleem van *automatische classificatie* op componist. Analog kunnen we in plaats van te classificeren op componist, classificeren op andere criteria – bijvoorbeeld op muziekgenre of misschien zelfs op emotionele inhoud (“Klinkt dit muziekstuk eerder vrolijk of eerder droevig?”). De enige voorwaarde is dat

er voldoende voorbeeld-muziekstukjes beschikbaar zijn van elke klasse. Of die klasse nu een componist, genre of emotie uitdrukt, speelt eigenlijk geen rol.

Een mogelijke toepassing hiervan is de volgende: stel dat we in een gegevensbank van muziekstukken op zoek gaan naar een bepaald muziekstuk. Vaak weten we niet wie de componist van dat muziekstuk is of wat de titel ervan is. Het wordt dan heel moeilijk om dat stuk op te zoeken. Als we queries zoals “het gezochte muziekstuk klinkt zoals  $X$  van componist  $Y$ ” zou kunnen ingeven, dan zou dit het zoekwerk veel eenvoudiger maken. Een andere mogelijke toepassing: een ‘automatische piloot’-knop voor een radiozender waarbij de liedjes automatisch gekozen worden zonder storende stijlbreuken. Het volgende liedje wordt dan zo gekozen dat het qua genre voldoende dicht bij het huidige ligt.

### 6.1.1 Proefopstelling

Om het experiment niet te ingewikkeld te maken, nemen we aan dat bij de classificatie gekozen moet worden tussen twee componisten. De classificatie-methode die we gebruiken is op een eenvoudige manier uit te breiden naar meerdere componisten.

De twee componisten waarvan we muziek zullen proberen te classificeren zijn Bach<sup>1</sup> en Mozart<sup>2</sup>. We kozen deze twee componisten, omdat ze een duidelijk verschillende componeerstijl hebben en omdat er op de *Mutopia*-site [Mut] voldoende werken van beide componisten beschikbaar zijn.

#### Twee modellen

Model  $M_B$  bekomen we door het model uit het vorig hoofdstuk (zie listing C.1) te trainen met leer-voorbeelden uit het werk van Bach. Analoog bekomen we model  $M_M$  uit het werk van Mozart.

Om het rekenwerk doenbaar te houden, werken we met tweestemmige stukken, opgesplitst in sequenties van lengte 20. Ook gebruiken we het model zonder Voice States (anders gezegd: met slechts één Voice State). Uit preliminaire testen bleek dat we vergelijkbare resultaten krijgen als we wél met Voice States werken. Voor classificatie is het dus niet nodig om het ingewikkeldere model (met Voice States) te gebruiken. Zoals we in sectie 6.2 zullen zien is dit niet altijd zo. Aangezien de benodigde reken- en geheugen-capaciteit veel hoger is voor modellen met Voice States zullen we het hier dus houden op het eenvoudigere model zonder Voice States.

Als leer-voorbeelden voor het Bach-model  $M_B$  gebruiken we enkele tweestemmige stukken voor klavecimbel: *Inventies* nummers 1, 2, 4 en 8 (BWV 77{2,3,5,9}). In IVL-notatie

<sup>1</sup>Johann Sebastian **Bach** (1685-1750) was een Duitse componist uit de Barok-periode, die beschouwd wordt als één van de grootste componisten aller tijden. Zijn werken – die getuigen van intellectuele diepgang, technische perfectie en artistieke schoonheid – waren een bron van inspiratie voor zowat elke componist in de Europese traditie, van Mozart tot Schönberg. Zijn bekendste werken zijn de *Brandenburgse Concerten*, het *Wohltemperierte Klavier*, de *Kunst der Fuge*, de *Mis in B-mineur* en de *Mattheüspassie*.

<sup>2</sup>Wolfgang Amadeus **Mozart** (1756-1791), geboren in Salzburg in het hedendaagse Oostenrijk, was een componist uit de Klassieke periode die – zoals Bach – tot de belangrijkste componisten uit de muziekgeschiedenis gerekend wordt. Bekende werken zijn het *Requiem*, het klarinetconcerto, de latere pianoconcerti, de strijkkwartetten, de latere symfonieën en de opera’s *Nozze di Figaro*, *Don Giovanni* en *Die Zauberflöte*.

(in sequenties van lengte 20) zijn dit 72 voorbeelden met een totale lengte van 1428 fases. Voor het Mozart-model  $M_M$  gebruiken we zeven duetten voor hoorn uit opus KV 487: nummers 1 (*Allegro*), 2 (*Menuetto*), 5 (*Larghetto*), 8 (*Allegro*), 9 (*Menuetto*), 10 (*Andante*) en 12 (*Allegro*). Dit komt overeen met 50 voorbeelden, in totaal 948 fases.

### Classificatie - methode

We zullen nu van fragmenten uit andere stukken trachten te achterhalen van welke componist ze zijn. Meer bepaald zullen we de 30 fragmenten uit figuur 6.1 – die zoals de leer-voorbeelden elk van lengte 20 zijn – correct proberen te classificeren als ‘gecomponoerd door Bach’ of ‘gecomponoerd door Mozart’.

De kans  $P_B$  dat een te classificeren fragment de uitvoer is van model  $M_B$ , vinden we met het ingebouwde `prob/2`-predicaat. We vergelijken die kans met de kans  $P_M$  dat het fragment de uitvoer is van model  $M_M$ . Als  $P_B > P_M$  besluiten we dat het stuk door Bach gecomponeerd werd, als  $P_M > P_B$  besluiten we dat het een stuk van Mozart is. In het geval dat  $P_M = P_B$  kunnen we het stuk niet classificeren – dit zal echter niet vaak voorkomen.

### 6.1.2 Resultaten

De resultaten van de automatische classificatie staan in figuur 6.2. Alle fragmenten worden correct geclassificeerd. Dit is een opmerkelijk en goed resultaat, zeker als we rekening houden met de nogal beperkte IVL-representatie, het redelijk ruw model en het klein aantal leer-voorbeelden. Die leervoorbeelden waren bovendien tweestemmig en toch werden ook de driestemmige stukken juist geclassificeerd.

Merk op dat er met logaritmische kansen gewerkt wordt. Sommige kansen zijn blijkbaar nul, of op logaritmische schaal  $-\infty$  (“-inf”). Dit is mogelijk te wijten aan de beperkte rekenprecisie. Er is echter ook een andere mogelijke verklaring, die problematischer is voor de significantie van deze resultaten...

### 6.1.3 Kritische noot in verband met de duur $L_i$

Als we de leer-voorbeelden en de te classificeren fragmenten bestuderen merken we op dat de stukken van Bach systematisch ritmisch sneller zijn dan die van Mozart. Bij muziek van Bach is de duur  $L_i$  van de IVL-fases overwegend 16 (16<sup>de</sup> noot als ‘ritmische eenheid’), terwijl dit bij Mozart vaker 32 is (8<sup>ste</sup> noot als ‘ritmische eenheid’). Dit maakt de classificatie gemakkelijker: model  $M_B$  krijgt kleinere kansen voor `out_L(-, -) = 32` dan model  $M_M$  waardoor  $P_B$  voor stukken van Mozart steeds kleiner is dan  $P_M$ . Hetzelfde geldt omgekeerd voor stukken van Bach.

Nog erger is het als een waarde voor  $L_i$  niet voorkomt in leer-voorbeelden maar wel in het te classificeren stuk. Zo heeft bij de leer-voorbeelden voor het Bach-model  $M_B$  geen enkele  $L_i$  de waarde 96. Het leer-algoritme zal daardoor de kans dat `out_L(-, -) = 96` in  $M_B$  gelijkstellen aan nul. De te classificeren fragmenten `mozart_b1` en `mozart_b2` bevatten echter wél fases met duur 96. Die fragmenten kunnen dus niet verklaard worden

fragment	uit het muziekstuk	# stemmen	componist
bach_a1 ⋮ bach_a5	<i>Inventie n° 13</i> in A-mineur (BWV 784)	2 (klavecimbel)	Bach
mozart_a1 ⋮ mozart_a5	<i>Duet n° 7 (Adagio)</i> in F (KV 487)	2 (hoorns)	Mozart
bach_b1 ⋮ bach_b10	<i>Sinfonia n° 12</i> in A (BWV 798)	3 (klavecimbel)	Bach
mozart_b1 ⋮ mozart_b10	<i>Divertimento II (Allegro)</i> in Bes (KV 229)	3 (hobo, klarinet, fagot)	Mozart

Figuur 6.1: Te classificeren muziekfragmenten

fragment	$P_B$		$P_M$	classificatie	
bach_a1	-128.9242	>	-654.9777	Bach	OK
bach_a2	-119.2298	>	-585.3842	Bach	OK
bach_a3	-119.0030	>	-inf	Bach	OK
bach_a4	-116.1041	>	-inf	Bach	OK
bach_a5	-120.6330	>	-inf	Bach	OK
mozart_a1	-297.2974	<	-135.6940	Mozart	OK
mozart_a2	-123.8396	<	-113.6923	Mozart	OK
mozart_a3	-302.4484	<	-120.1393	Mozart	OK
mozart_a4	-202.9020	<	-155.1590	Mozart	OK
mozart_a5	-151.7031	<	-115.7385	Mozart	OK
bach_b1	-143.8003	>	-235.7590	Bach	OK
bach_b2	-189.3584	>	-422.4009	Bach	OK
bach_b3	-177.6644	>	-215.6288	Bach	OK
bach_b4	-157.7476	>	-317.2182	Bach	OK
bach_b5	-156.0695	>	-607.5714	Bach	OK
bach_b6	-166.0751	>	-234.4263	Bach	OK
bach_b7	-174.2113	>	-491.5094	Bach	OK
bach_b8	-161.3694	>	-165.9104	Bach	OK
bach_b9	-152.8914	>	-463.6443	Bach	OK
bach_b10	-171.3188	>	-176.2599	Bach	OK
mozart_b1	-inf	<	-552.7813	Mozart	OK
mozart_b2	-inf	<	-230.1960	Mozart	OK
mozart_b3	-431.4452	<	-234.0330	Mozart	OK
mozart_b4	-283.3101	<	-165.0584	Mozart	OK
mozart_b5	-423.6916	<	-188.6801	Mozart	OK
mozart_b6	-323.5870	<	-119.2280	Mozart	OK
mozart_b7	-428.5111	<	-175.3572	Mozart	OK
mozart_b8	-397.1515	<	-168.5853	Mozart	OK
mozart_b9	-330.0027	<	-179.0431	Mozart	OK
mozart_b10	-381.2333	<	-175.4383	Mozart	OK

Figuur 6.2: Classificatie van 30 muziekfragmenten

door het model  $M_B$  waardoor  $P_B$  nul wordt - logaritmisches uitgedrukt “-inf”.

De ‘ritmische eenheid’ van Bach is blijkbaar (althans voor de stukken die we beschouwden) sneller dan die van Mozart. Het is dus in principe een geschikt criterium om fragmenten te classificeren. Toch voelt het een beetje aan als ‘valsspelen’. Eigenlijk gebruiken we geen eigenschappen van de *muziek* zelf, maar eerder van de *muzieknotatie*. Als we eerst alle stukjes naar dezelfde ‘ritmische eenheid’ zouden omzetten, dan zouden we er eigenlijk muzikaal niets aan veranderen. We vermoeden echter dat de classificatie-tabel uit figuur 6.2 er misschien heel anders had kunnen uitzien als we dat zouden doen.

We stellen ons dan ook de vraag of de classificatie nog steeds correct zou gebeuren als we niet zo’n systematische duur-verschillen zouden hebben. Om op die vraag een antwoord te krijgen zouden we alle leer-voorbeelden en te classificeren fragmenten eerst ritmisch kunnen herschalen zodat ze min of meer dezelfde ‘ritmische eenheid’ hebben. Op die manier gaat weinig informatie verloren.

Een minder omslachtige manier van werken is gewoon geen rekening houden met de duur  $L_i$  van de IVL-fases. We verwijderen daarom regel 93 – “msw(out\_L(SS),L)” – uit het PRISM-model (listing C.1). Nu herhalen we het experiment van hierboven.

## Resultaten

De resultaten van dit aangepast experiment worden gegeven in figuur 6.3. Nu worden 4 van de 30 fragmenten incorrect geassocieerd. Dit is nog steeds een sterk resultaat: een (menselijke) muziekexpert zou het waarschijnlijk niet veel beter doen.

De verschillen tussen  $P_B$  en  $P_M$ , die in zekere zin uitdrukken hoe zeker we zijn van de classificatie, zijn in dit experiment – zoals te verwachten – kleiner dan in het vorige. Geen enkele kans heeft nu nog de waarde “-inf”. Het valt op dat de kansen groter zijn dan in het vorig experiment. De reden daarvoor is dat de factor  $\prod_{i=1}^{20} P(L_i)$  wegvalt.

Merk op dat de vier fout geassocieerde fragmenten telkens stukken van Mozart zijn die onterecht geassocieerd werden als ‘gecomponeerd door Bach’. Een mogelijke – maar erg speculatieve – verklaring hiervoor is dat de componist Mozart beïnvloed is door Bach maar niet omgekeerd, waardoor het ‘Bach-model’ sommige fragmenten van Mozart kan verklaren.

Tot hier toe hebben we enkel (korte) fragmenten uit muziekstukken geassocieerd. Een mogelijke methode om volledige muziekstukken te classificeren is de volgende: we splitsen het stuk op in kleinere fragmenten, vervolgens classificeren we elk van die fragmenten en de meest voorkomende classificatie geven we als resultaat. Hoe vaak de meest voorkomende classificatie voorkomt ten opzichte van het totaal drukt uit hoe zeker we zijn van de classificatie van het volledig stuk. Als we de vier muziekstukken van figuur 6.1 op die manier zouden classificeren, dan zouden ze alle vier correct geassocieerd worden.

## 6.2 Automatisch componeren

De doelstelling van het tweede experiment is heel wat ambitieuzer. We zullen trachten op een automatische manier een muziekstuk te componeren, gebaseerd op voorbeelden.

fragment	$P_B$		$P_M$	classificatie	
bach_a1	-131.7985	>	-139.9345	Bach	OK
bach_a2	-117.5514	>	-125.2330	Bach	OK
bach_a3	-120.9500	>	-142.2960	Bach	OK
bach_a4	-121.6009	>	-140.7881	Bach	OK
bach_a5	-120.5054	>	-143.6630	Bach	OK
mozart_a1	-109.6897	<	-102.5448	Mozart	OK
mozart_a2	-100.4641	>	-102.4500	Bach	FOUT
mozart_a3	-120.4684	<	-110.4892	Mozart	OK
mozart_a4	-102.3158	<	-99.9453	Mozart	OK
mozart_a5	-98.5936	>	-101.4389	Bach	FOUT
bach_b1	-143.6279	>	-162.1477	Bach	OK
bach_b2	-152.1257	>	-158.1435	Bach	OK
bach_b3	-161.1178	>	-171.0204	Bach	OK
bach_b4	-162.6107	>	-167.5636	Bach	OK
bach_b5	-155.0678	>	-155.7069	Bach	OK
bach_b6	-164.9219	>	-168.5871	Bach	OK
bach_b7	-167.8643	>	-172.0629	Bach	OK
bach_b8	-155.6751	>	-167.1436	Bach	OK
bach_b9	-156.1293	>	-165.5039	Bach	OK
bach_b10	-147.0801	>	-153.6163	Bach	OK
mozart_b1	-246.3015	<	-177.6869	Mozart	OK
mozart_b2	-177.0579	<	-148.8940	Mozart	OK
mozart_b3	-236.5225	<	-166.7288	Mozart	OK
mozart_b4	-153.6892	<	-153.6512	Mozart	OK
mozart_b5	-178.7821	<	-181.9799	Bach	FOUT
mozart_b6	-162.3160	<	-108.5185	Mozart	OK
mozart_b7	-173.5196	<	-167.9018	Mozart	OK
mozart_b8	-186.9003	<	-154.7144	Mozart	OK
mozart_b9	-156.2027	>	-169.4681	Bach	FOUT
mozart_b10	-184.6396	<	-167.0463	Mozart	OK

Figuur 6.3: Classificatie zonder rekening te houden met duur  $L_i$ 

Zoals in het vorige experiment bepalen we eerst de kansparameters van het model door het te trainen met voorbeelden. In het vorig experiment gebruikten we de PRISM-built-in `prob/2` om de waarschijnlijkheid van een gegeven stuk te bepalen. Nu zullen we `sample/1` gebruiken om een willekeurig stuk te genereren. Daarbij wordt natuurlijk rekening gehouden met de kansverdelingen van het model.

Er zijn veel mogelijke toepassingen van een programma dat automatisch muziek componeert. Automatische composities kunnen een nieuwe bron van inspiratie zijn voor (menselijke) componisten. In situaties waar niet aandachtig naar muziek geluisterd wordt – denk aan achtergrondmuziek in een station, winkelcentrum of wachtzaal – kan automatisch gegenereerde muziek worden gebruikt in plaats van bestaande muziek die vaak duur is – want auteursrechtelijk beschermd.

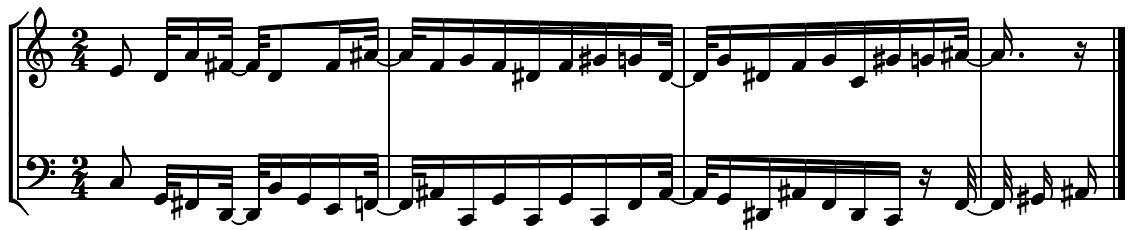
### 6.2.1 Proefopstelling

We trainen het model uit het vorig hoofdstuk (eerst zonder Voice States, dan met drie Voice States) met de leervoorbeelden uit het vorig experiment: enkele Inventies van Bach en enkele Hoorn-duetten van Mozart. Door middel van de built-in `sample/1` genere-

ren we muziek op basis van het getraind model. Meer bepaald gebruiken we de query “sample(song(2,L,0,[(32,[new,new],[36,52])|IVL]))” zodat we een tweestemmig IVL-stukje van lengte L krijgen. Vervolgens converteren we het gegenereerde IVL-stukje naar een LilyPond-bestand, zoals in bijlage B.1 beschreven wordt. Uiteindelijk converteren we dit LilyPond-bestand naar een MIDI-bestand, dat we kunnen beluisteren.

## 6.2.2 Resultaten

Laat ons eerst de uitvoer van het model zonder Voice States bestuderen. Typische voorbeelden daarvan worden gegeven in figuur 6.4 (getraind met Inventies van Bach) en figuur 6.5 (getraind met duetten van Mozart). Ze zijn voor de leesbaarheid in klassieke notatie weergegeven, met een gepaste sleutel.



Figuur 6.4: Automatische compositie gebaseerd op Bach (zonder VS)

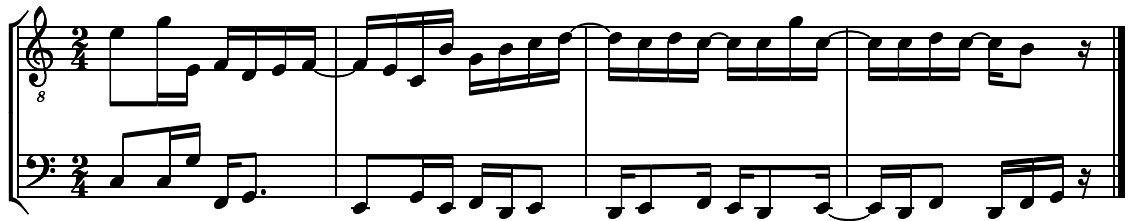


Figuur 6.5: Automatische compositie gebaseerd op Mozart (zonder VS)

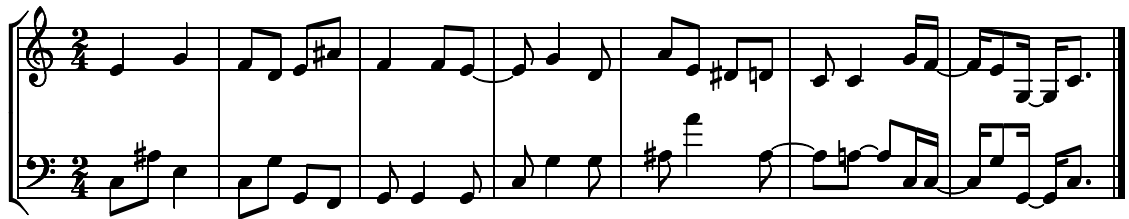
Deze automatisch gegenereerde composities klinken erg artificieel en atonaal. Eigenlijk zijn ze maar een beetje interessanter dan een willekeurige opeenvolging van noten. Behalve op het gebied van het – nogal triviale – gemiddeld ritme (cfr. subsectie 6.1.3) is er in feite geen muziek-kennis die duidelijk merkbaar geleerd is door het model.

Het model mét (drie) Voice States produceert betere uitvoer. In figuren 6.6 en 6.7 worden twee typische voorbeelden gegeven van automatisch gecomponeerde muziekstukjes op basis van dit model.

Deze automatische composities zijn muzikaal gezien redelijk interessant en zijn mogelijk een goede bron van inspiratie voor een menselijke componist. Ze klinken echter niet zoals de stukken waarop ze gebaseerd zijn (waarmee het model getraind is). Vooral op gebied van ritme zijn ze erg verschillend. Dit is waarschijnlijk te wijten aan het erg ruw ritme-model. Er wordt geen rekening gehouden met de maatsoort, wat tot uiting komt in



Figuur 6.6: Automatische compositie gebaseerd op Bach (3 VS)



Figuur 6.7: Automatische compositie gebaseerd op Mozart (3 VS)

de vele syncopes<sup>3</sup>. Verder valt het op dat er vrij veel dissonanten voorkomen, meer dan in de leer-voorbeelden. Toch is het duidelijk, bijvoorbeeld uit de volledige authentieke cadens<sup>4</sup> op het einde van de compositie uit figuur 6.7, dat bepaalde muziek-kennis geleerd wordt uit de voorbeelden.

Al klinken de gegenereerde stukjes niet echt zoals de muziek waarop ze gebaseerd zijn, toch kunnen we met een beetje goede wil wel enige gelijkenissen opmerken. Zoals we al uitlegden in de vorige sectie is de ‘ritmische eenheid’ de 16<sup>de</sup> noot bij de stukken van Bach en de 8<sup>ste</sup> noot bij de stukken van Mozart. Dat verschil komt duidelijk tot uiting in de respectievelijke automatisch componeerde stukjes. Een ander verschil tussen de stukken van Bach en Mozart is dat bij Mozart vaker dezelfde noot herhaald wordt terwijl dit bij Bach eerder zelden gebeurt. In de automatische compositie gebaseerd op Mozart (figuur 6.7) zien we daar een voorbeeld van in de derde maat.

Uiteraard moeten we niet noodzakelijk tweestemmige stukjes genereren als we trainen met tweestemmige stukken. Er is geen verband tussen het aantal stemmen in de invoer (de leer-voorbeelden) en het aantal stemmen in de uitvoer (de automatisch gecomponeerde stukjes). We kunnen dus in principe ook éénstemmige, driestemmige of zelfs tienstemmige muziek genereren.

<sup>3</sup>*syncope* (Van Dale): verschuiving van het accent door verbinding van het volgende zware aan het voorafgaande zwakke of lichte maatdeel.

<sup>4</sup>*volledige authentieke cadens*: opeenvolging van de drie harmonische functies *tonica*, *subdominant* en *dominant*, eindigend op dominant-tonica, waardoor de toonsoort bevestigd wordt.

# Hoofdstuk 7

## Besluit

*“La musique exprime ce qui ne peut être dit et sur quoi il est impossible de rester silencieux.”*

– Victor Hugo (1802-1885)

In dit laatste hoofdstuk geven we een bondige samenvatting van de vorige hoofdstukken. Daarna bespreken we het geleverd werk en toetsen we het aan onze oorspronkelijke doelstellingen. Tenslotte vermelden we enkele mogelijkheden voor verder werk.

### 7.1 Samenvatting

Eigenlijk bestaat deze verhandeling uit twee helften. In de eerste helft van deze verhandeling, meer bepaald hoofdstukken 2, 3 en 4, beschreven we drie ingrediënten : IVL, PRISM en HMM's. In de tweede helft van deze verhandeling, hoofdstukken 5 en 6, brachten we deze drie ingrediënten samen. Het grootste deel van het eigen werk is dan ook geconcentreerd in de tweede helft van deze verhandeling.

#### 7.1.1 Eerste helft: voorbereiding

We begonnen in hoofdstuk 2 met de zoektocht naar een muziekrepresentatie die gemakkelijk te gebruiken is in PRISM en waarin de volgens ons essentiële aspecten van muziek – samenklank, toonhoogte en duur – uitgedrukt kunnen worden. Geen enkele bestaande notatie bleek te volstaan, we introduceerden dus een nieuwe notatie: IVL. In deze notatie wordt een muziekstuk voorgesteld als een Prolog-lijst van drietallen  $(L, V, N)$  die elk een fase voorstellen. Daarbij stelt  $L$  de duur voor,  $N$  een lijst van de noten die op dat moment klinken. Welke noten ‘nieuw’ zijn en welke al begonnen in de vorige fase staat in  $V$ .

In hoofdstuk 3 beschreven we PRISM, een taal voor logisch-probabilistisch programmeren/modelleren die we kunnen beschouwen als Prolog uitgebreid met ingebouwde probabilistische predicaten. Aan de hand van een eenvoudig voorbeeld (een aantal keer opgooien van een muntstuk) beschreven we de twee subsystemen van PRISM: het zogenaamde *sampling execution subsystem* en het ingebouwd leer-algoritme (*learning subsystem*).

Daarna kwamen Hidden Markov Models aan bod, in hoofdstuk 4. Dit zijn Markovketens waarvan de toestanden zelf niet geobserveerd kunnen worden, maar enkel ‘observaties’ zichtbaar zijn, waarvan de kansverdeling afhangt van de ‘verborgen toestand’. Op het einde van dat hoofdstuk (sectie 4.5) implementeerden we ter illustratie het voorbeeld-HMM van de ‘springende robot’ in PRISM.

### 7.1.2 Tweede helft: het echte werk

In hoofdstuk 5 construeerden we een PRISM-model voor muziek in IVL-notatie, gebaseerd op HMM’s. We begonnen met een redelijk eenvoudig basismodel, waar we vervolgens verschillende verfijningen aan aanbrachten. Zo kwamen we tot ons uiteindelijk model, dat in subsectie 5.4.1 wordt samengevat en waarvan de broncode in listing C.1 wordt gegeven.

Dit model is een soort ‘hiërarchisch HMM’ waarbij één ‘song HMM’ de ‘globale toestand’ beschrijft en verschillende ‘voice HMMs’ (één per stem) de ‘lokale toestand’ in een bepaalde stem beschrijven. De ‘voice HMMs’ (of beter gezegd de overgangs- en observatiekansen van elk ‘voice HMM’) zijn afhankelijk van de toestand van het ‘song HMM’ (vandaar ‘hiërarchisch’). De uitvoer van de verschillende HMM’s nemen we op een bepaalde manier samen om zo een muziekstuk in IVL-notatie te bekomen.

Op basis van dit model voerden we twee experimenten uit die in hoofdstuk 6 beschreven worden. In een eerste experiment probeerden we een 30-tal twee- en driestemmige muziekfragmenten automatisch te classificeren op componist. Tenslotte probeerden we in het tweede experiment om op een automatische manier ‘typische’ muziekstukjes te componeren.

## 7.2 Bereikte resultaten

Onze oorspronkelijke doelstellingen (zie sectie 1.2) zijn bereikt. We hebben namelijk een algemeen model gemaakt voor meerstemmige muziek. De parameters van dit model kunnen automatisch geleerd worden op basis van voorbeelden, dankzij het ingebouwd leer-algoritme van PRISM.

Dit model is zowel bruikbaar voor classificatie als voor automatisch componeren, zoals de resultaten van de experimenten in hoofdstuk 6 aantonen. Hiermee geven we een empirische bevestiging van de these van Conklin [Con03]. Het is daarbij gebleken dat automatisch componeren ‘moeilijker’ is dan classificatie, in die zin dat de resultaten van classificatie overtuigender zijn, ondanks het feit dat een eenvoudiger model daarvoor volstond. Alle muziekfragmenten werden namelijk correct geclassificeerd door ons model. Zelfs toen we het ritme-aspect negeerden werden nog 26 van de 30 fragmenten correct geclassificeerd.

De automatische composities van het uiteindelijk model zijn zeker muzikaler dan willekeurige opeenvolgingen van noten. Het is ook duidelijk dat er bepaalde muziekkennis geleerd wordt uit de voorbeelden. Toch slaagden we er niet in om stukjes te produceren die (voor een menselijke luisteraar) klinken zoals de leer-voorbeelden waarop ze gebaseerd

zijn. Het zou echter te ambitieus zijn – en een belediging voor echte componisten – om zoiets te verwachten van ons model.

Tenslotte kunnen we stellen dat deze verhandeling een goede illustratie vormt van de mogelijkheden van een logisch-probabilistische taal zoals PRISM. In sectie 3.2 gaven we enkele voor- en nadelen van PRISM. Bij de voorbereidingen voor deze verhandeling hebben we veel verschillende soorten modellen uitgetoetst. Het was dan erg belangrijk dat we het ingebouwd leer-algoritme van PRISM konden gebruiken, zodat we niet zelf een leer-algoritme voor elk model moesten implementeren. Dat bespaarde veel overbodig werk, aangezien de meeste van die modellen uit de *trial-and-error*-fase van ons onderzoek uiteindelijk toch werden weggegooid. Anderzijds werden we door dit ingebouwd algemeen leer-algoritme genoodzaakt om enkele redelijk sterke (computationele) beperkingen op te leggen aan de leer-voorbeelden (sectie 5.3).

## 7.3 Verder werk

In subsectie 5.4.2 gaven we een overzicht van mogelijke bijkomende verfijningen aan ons model. Een voor de hand liggend vervolg op deze verhandeling is dan ook om deze bijkomende verfijningen te implementeren en na te gaan wat de gevolgen daarvan zijn voor de classificatie- en compositie-resultaten.

In zijn thesis ontwikkelde Bart Vrancken [Vra03] een programma voor het componeren van polyfone (koor-)muziek. Het gebruikt de (menselijke) invoer van één stem om de andere drie stemmen automatisch te genereren. De door ons model gegenereerde éénstemmige stukken zouden we kunnen gebruiken als invoer voor dat programma. We hebben dan een programma dat zonder invoer muziek produceert die voldoet aan strikte harmonische constraints. Misschien is (een deel van) het programma van Bart Vrancken zelfs te integreren in ons model.

In [CW95] en [CA01] worden zogenaamde *viewpoints* voorgesteld: verschillende manieren om naar een melodie te kijken. Aan de hand daarvan worden patronen gezocht in koorwerken van Bach. Dit idee wordt in [Con02] uitgebreid naar meerstemmige muziek. Het is mogelijk een interessante denkpiste om in ons model ook op één of andere manier dergelijke *viewpoints* te gebruiken. Hoe we dat precies kunnen doen, is echter zeker geen triviale kwestie.

# Bijlage A

## Overzicht gebruikte software

**LilyPond 2.1.0** is een software-pakket dat erg goed is in het automatisch produceren van esthetisch verantwoorde partituren. Omzetting naar MIDI-formaat is ingebouwd en verschillende importermogelijkheden vanuit andere formaten bestaan. LilyPond werd gebruikt om de muziekvoorbeelden in deze verhandeling mooi weer te geven. De bestaande muziekstukken die we gebruikten om ons model te trainen, waren oorspronkelijk in LilyPond-formaat. Ook hebben we de automatisch gecomponeerde stukjes omgezet naar LilyPond-formaat (en naar MIDI) om ze gemakkelijker te kunnen lezen (en beluisteren). Zie subsectie 2.3.2, bijlage B en [NN03].

Homepage: <http://www.lilypond.org>; Licentie: GNU GPL

**Perl 5.8.1** (*Practical Extraction and Report Language*) is een programmeertaal die heel geschikt is om informatie te halen uit een bestand en in een andere vorm terug weer te geven. Perl combineert eigenschappen van C, sed, awk en sh. We gebruiken Perl in bijlage B

Homepage: <http://www.perl.com/>; Licentie: Artistic License/GNU GPL

**PRISM 1.6** (*Programming In Statistical Modeling*) wordt besproken in hoofdstuk 3.

Homepage: <http://mi.cs.titech.ac.jp/prism/>; Licentie: zie subsectie 3.2.2

**teTeX 2.0.2** is een volledige distributie van L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> voor UNIX-compatibele systemen. Deze verhandeling is gemaakt met L<sup>A</sup>T<sub>E</sub>X, een macro-pakket, geschreven door Leslie Lamport, dat gebruik maakt van T<sub>E</sub>X, een computerprogramma van Donald E. Knuth dat speciaal ontworpen is voor het zetten en drukken van wiskundige teksten en formules.

Homepage: <http://www.tug.org/teTeX/>; Licentie: GNU GPL

**TiMidity++ 2.13.0-rc1** is een MIDI-player die *patches* van hoge kwaliteit gebruikt om de klanken van de instrumenten na te bootsen. De meeste andere MIDI-players gebruiken de ingebouwde synthesizer van de geluidskaart, die vaak een erg artificiële klank geeft.

Homepage: <http://www.timidity.jp>; Licentie: GNU GPL

## Bijlage B

# Omzetting tussen IVL en LilyPond

In deze appendix gaan we in op de concrete implementatie van de omzetting van IVL naar LilyPond en omgekeerd.

## B.1 IVL naar LilyPond

De omzetting van Interwoven Voices List naar LilyPond-formaat is relatief gemakkelijk, aangezien de IVL-notatie erg eenvoudig is. Het perl-scriptje `unweave.pl` (cfr. listing B.1) implementeert een converterprogramma dat IVL via `stdin` inleest en een geldig LilyPond bestand uitprint op `stdout`.

Listing B.1: `unweave.pl`

```
#!/usr/bin/perl
#####
# convert interwoven voices list (IVL) format to lilypond #
#####

$IVL = "";
while (<>) {
    chomp();
    s/\s//g; # remove spaces
    $IVL .= $_;
}

($header,$IVL) = split /\x5B\x28/, $IVL ; # header[(IVL-list
($n,$l) = split(",",$header); # $l: song length
$n =~ s/song\x28//; # number of voices
$IVL =~ s/\x29\x5D\x29.// ; # remove ")])." (at the end)

@notes = split /\x29,\x28/, $IVL; # split between "),(
$voices = (scalar(split(",",$notes[0])) - 1)/2; # number of voices

$n == $voices or die "Inconsistent number of voices";
$l == scalar(@notes) or die "Inconsistent song length";

for ($i = 0; $i < $voices; $i++) {
    $voice[$i] = "voice" . chr(65+$i) . " = \ notes \{n";
}
}
```

```

@notenames = ("c", "cis", "d", "dis", "e", "f", "fis", "g", "gis", "a", "ais", "b");

foreach $note (@notes) {
  ($info, $notelist) = split /\x5D,\x5B/, $note; # split between "],[",["
  chop $notelist; # contains comma-separated list of pitches
  ($duration, $changed_channels) = split /\x5B/, $info;
  @diffs = split(", ", $changed_channels);
  @cnotes = split(", ", $notelist);
  for ($i = 0; $i < $voices; $i++) {
    # if a new note is started
    if ($diffs[$i] eq "new") {
      # add duration of previous note
      &addDuration;
      # add new notename
      &addNoteName;
      $notelength[$i] = $duration;
    } else {
      # previous note is still sounding
      $notelength[$i] += $duration;
    }
  }
}
for ($i = 0; $i < $voices; $i++) {
  &addDuration; # add duration of last note
  $voice[$i] .= "\n}\n";
}

# output
for ($i = 0; $i < $voices; $i++) {
  print $voice[$i];
}
print "\\score_{\n____\\ notes_{\n context_StaffGroup_<<";
for ($i = 0; $i < $voices; $i++) {
  print "\n_____\n context_Staff_=" . chr(65+$i) . "<<\n";
  print "_____\n voice" . chr(65+$i) . "\n____>>";
}
print "\n____>>\n____\n paper_{\n____\n midi_{\n tempo_4_=_40_\n}";

# subroutines
sub addDuration {
  if ($notelength[$i] > 0) {
    $lilyduration = "";
    $lengthleft = $notelength[$i];
    while ($lengthleft > 0) {
      for ($k = 256; $k > 0; $k /= 2) {
        if ($lengthleft >= $k) {
          $lengthleft -= $k;
          $lilyduration .= 256/$k;
          last;
        }
      }
    }
    last if ($lengthleft == 0);
    if ($lastnote[$i] eq "r") {
      $lilyduration .= "_r";
    } else {

```

```

        $lilyduration .= "˘˘".$lastnote[$i];
    }
}
$voice[$i] .= $lilyduration."˘";
}
}
sub addNoteName {
    $notenumber = $notes[$i];
    if ($notenumber eq "r") {
        $notename = "r"; $octave = "";
    } else {
        $notename = $notenames[$notenumber % 12];
        $octave = int($notenumber/12);
        if ($octave >= 4) {
            $octave = "" x ($octave - 3);
        } elsif ($octave == 3) {
            $octave = "";
        } else {
            $octave = "," x (3 - $octave);
        }
    }
    $voice[$i] .= $notename.$octave;
    $lastnote[$i] = $notename.$octave;
}
}

```

## B.2 LilyPond naar IVL

Het is iets minder eenvoudig om LilyPond-bestanden te converteren naar IVL. De LilyPond-syntax is namelijk redelijk ingewikkeld. We zullen deze conversie in twee stappen doen. In een eerste stap zetten we LilyPond om naar een tussenformaat dat we “Voice List” (VL) zullen noemen. Deze VL-notatie lijkt sterk op de IVL-notatie waar we naartoe willen, maar de stemmen worden nog wel gescheiden genoteerd. In de volgende stap ‘weven’ we dan de stemmen door elkaar tot IVL.

### B.2.1 LilyPond naar VL

LilyPond heeft een ingebouwde omzetting naar MIDI. Een voor de hand liggende maar ietwat onelegante manier om hiervan gebruik te maken is in de LilyPond-broncode die deze omzetting naar MIDI realiseert enkele toevoegingen te doen. Telkens als LilyPond een `start_note`-event wegschrijft naar de MIDI-file zullen we het nootnummer en de lengte printen naar `stdout`. Concreet kunnen we dat implementeren op deze manier:

Aan de functie `Midi_walker::process()` uit `lily/midi-walker.cc` voegen we enkele regels toe (15-18 en 22-24 in listing B.2). De functie `Moment::to_int()` wordt in die regels gebruikt en moeten we dus nog definiëren en aan `lily/moment.cc` toevoegen (listing B.3). Deze functie zet een duur van interne LilyPond-representatie om naar onze notatie. Om de stemmen te scheiden in de output, zullen we een nieuwe regel beginnen nadat een channel is weggeschreven. Dat doen we door in `lily/performance.cc` in de functie `Performance::output()` op het einde van de `for`-lus de regel `"printf("\n");"` toe te voegen.

Listing B.2: Midi\_walker::process()

```

1 void
2 Midi_walker::process ()
3 {
4   Audio_item* audio = (*items_)[index_];
5   int rust;
6
7   do_stop_notes (audio->audio_column->at_mom ());
8
9   if (Midi_item* midi = Midi_item::get_midi (audio))
10    {
11     midi->channel_ = track->channel_;
12     if (Midi_note* note = dynamic_cast<Midi_note*> (midi))
13     {
14      if (note->get_length ().to_bool ()) {
15       // jump in timestamp indicates rest
16       rust = audio->audio_column->at_mom ().to_int ()
17             - last_mom_.to_int ();
18       if (rust > 0) printf ( "%i,r:" , rust);
19
20       do_start_note (note);
21
22       // new note started
23       printf ( "%i,%i:" , note->get_length ().to_int (),
24             note->get_pitch ()+48);
25     }
26   }
27   else
28   {
29     output_event (audio->audio_column->at_mom (), midi);
30   }
31 }
32 }

```

Listing B.3: Moment::to\_int()

```

1 int
2 Moment::to_int () const
3 {
4   int i = (num() * 256) / den();
5   return i;
6 }

```

We hebben nu een versie van LilyPond die VL-output naar `stdout` stuurt. Als we bij wijze van illustratie deze LilyPond loslaten op het voorbeeld uit 2.3.2 dan krijgen we volgende output:

```
32,55:32,60:32,59:32,60:64,62:
16,r:16,36:16,38:16,40:16,41:16,38:16,40:16,36:32,43:32,31:
```

## B.2.2 VL naar IVL

De tweede stap – van VL naar IVL – is vrij eenvoudig te implementeren, bijvoorbeeld met het perl-scriptje `weave.pl` (cfr listing B.4). Om een LilyPond-bestand `input.ly` te converteren naar `output.ivl` in IVL-formaat kan een commando als “`lilypond input.ly | weave.pl > output.ivl`” gebruikt worden.

Listing B.4: `weave.pl`

```
#!/usr/bin/perl
#####
# convert VL to IVL                                     #
#####

# input: n lists of notes
#
# 32,r:96,60:128,48:48,50:16,52:48,53:16,52:
# 64,36:32,r:32,36:64,33:32,r:32,33:64,29:
#
# output: one list in IVL format
#
# e.g.
# song(2,7,[ (32,3,[r,36]),(32,1,[60,36]),(32,2,[60,r]) ,...]).

$n = 0;          # number of voices
while (<>) {
    chomp();
    $n++;
    $channel[$n] = $_;
    $position[$n] = 0;
    $finished[$n] = 0;
    $channel_length[$n] = scalar(split(":",$channel[$n]));
    $changed[$n] = 1; # initially, all notes are new
}

print "song(".$n.", ";
$output = "";
$song_length = 0;

# get first notes
for ($i = 1; $i <= $n; $i++) {
    @notes = split(":",$channel[$i]);
    $next_note[$i] = $notes[$position[$i]];
    ($next_duration[$i], $next_pitch[$i]) = split(",",$next_note[$i]);
}
}
```

```

while (1) {
  # find shortest note
  $shortest=0;
  for ($i = 1; $i <= $n; $i++) {
    if ($finished[$i] == 0) {
      if ($shortest == 0 || $shortest > $next_duration[$i]) {
        $shortest=$next_duration[$i];
      }
    }
  }
}

# compute $changednotes
$changednotes = "[";
for ($i = 1; $i <= $n; $i++) {
  if ($changed[$i] == 1) {
    $changednotes .= "new";
  } else {
    $changednotes .= "old";
  }
  if ($i < $n) { $changednotes .= ","; }
}
$changednotes .= "];";

# print an IVL triplet
$output .= "(".$shortest." ,".$changednotes." ,[";
for ($i = 1; $i <= $n; $i++) {
  $output .= $next_pitch[$i];
  if ($i < $n) { $output .= ","; }
}
$output .= "])";
$song_length++;

print STDERR ".";

# decrease remaining duration of current notes
for ($i = 1; $i <= $n; $i++) {
  if ($finished[$i] == 0) {
    $next_duration[$i] -= $shortest;
    if ($next_duration[$i] == 0) {
      $prev_pitch = $next_pitch[$i];
      # note ended, get next note
      if ($position[$i] < $channel_length[$i] - 1 ) {
        $position[$i]++;
        @notes = split (":", $channel[$i]);
        $next_note[$i] = $notes[$position[$i]];
        ($next_duration[$i], $next_pitch[$i]) = split (",", $next_note[$i]);
      } else {
        $finished[$i] = 1;
        print STDERR "[END". $i ."]";
        # this voice has ended, padding with rests
        $next_pitch[$i]="r";
        $changed[$i] = 0;
      }
    }
    if ( ($prev_pitch eq "r") && ($next_pitch[$i] eq "r") ) {
      $changed[$i] = 0; # repeating rests are never 'new' notes
    } else {

```

```
        $changed[$i] = 1;
    }
  } else {
    # note still sounding
    $changed[$i] = 0;
  }
}
}

# stop if all voices have ended
$finish = 0;
for ($i = 1; $i <= $n; $i++) { $finish += $finished[$i]; }
if ($finish == $n) { last; }

# if not, iterate
$output .= ",_";
}

print $song_length." ,[_". $output." _]".\n";
```

## Bijlage C

# PRISM-code van het muziekmodel

Listing C.1: music.psm

```

1 target(song,4). % We observe ground atoms song(NV,Length,BaseTone,IVL)
2 data(bach). % training data
3
4 %-----
5 % msw values :
6
7 % song HMM: 7 states
8 values(tr_ss(-),[1,2,3,4,5,6,7]). % song state transitions
9
10 % voice HMM's: 3 states
11 values(tr_vs(-,-),[a,b,c]). % voice state transitions
12 % SS,PrevVS
13
14 % 32 16 8 8. 4 4. 2
15 values(out_L(-),[8,16,32,48,64,96,128]). % duration symbols
16
17 values(out_V(-,-),[old,new]). % old/new symbols
18
19 values(out_rest(-,-),[rest,note]). % note or rest?
20
21 values(out_octave(-,-),[-2,-1,0,1,2]). % relative octave
22
23 % relative pitches (scaled to C)
24 % c e g % possible values depend on SS
25 values(out_modnote(1,-),[0,4,7]). % 1: only C chord notes
26 % c d e f g a b
27 values(out_modnote(2,-),[0,2,4,5,7,9,11]). % 2: seven notes of C
28 % c d e fis g a b
29 values(out_modnote(3,-),[0,2,4,6,7,9,11]). % 3: seven notes of G
30 % c d e f g a bes
31 values(out_modnote(4,-),[0,2,4,5,7,9,10]). % 4: seven notes of F
32 % c es e f g as bes
33 values(out_modnote(5,-),[0,3,4,5,7,8,10]). % 5: +/- Cmin and Fmin7
34 % c d es f a bes
35 values(out_modnote(6,-),[0,2,3,5,9,10]). % 6: +/- F and Bes
36 % c d e f g a b
37 values(out_modnote(7,-),[0,1,2,3,4,5,6,7,8,9,10,11]). % 7: all pitches
38 % SS,VS

```

```

39
40 %-----
41 % Model:   (base)
42
43 song(NV,L,BT,IVL):-      % song IVL of length L with NV voices (base tone BT)
44   format("processing_song(~p,~p,~p,...)~1n",[NV,L,BT]),
45   initlist(NV,4,O_prev), % initialize previous octaves to "4"
46   initlist(NV,a,VS),     % initialize voice states to "a"
47   hmm(NV,1,L,1,VS,BT,BT,IVL,noprevnotes,O_prev). % begin in song state 1
48
49
50 % NV: number of voices
51 % T: current time (phase)           Length: length of song
52 % SS: current song state           VS: current voice states (list)
53 % OrigBT: song base tone (has to be determined manually)
54 % BT: current base tone (= OrigBT, unless we implement modulations)
55 % (L,V,N): (duration, old/new-list, notelist)
56 % N_prev: previous notelist
57 % O_prev: previous octave list
58 hmm(NV,T,Length,SS,VS,OrigBT,BT,[(L,V,N)|Y],N_prev,O_prev):-
59   T < Length,
60   T > 1,
61 % observations
62   observe(NV,SS,VS,BT,L,V,N,N_prev,O_prev),
63
64 % state transitions
65   tr_voicestates(NV,SS,VS,VS_Next), % next voice states
66   msw(tr_ss(SS),SS_Next),          % next song state
67
68 % recursion to next phase
69   octaves(BT,N,O_prev,N_octave), % compute new 'previous octave list'
70   T1 is T+1,                     % increase time
71   hmm(NV,T1,Length,SS_Next,VS_Next,OrigBT,BT,Y,N,N_octave).
72
73
74 % first phase: (assuming first tuple is given)
75 % The observations of the 1st phase are not checked. We start in phase 2.
76 % Doing so we always have meaningful N_prev and O_prev lists.
77 hmm(NV,1,Length,SS,VS,OrigBT,BT,[_L,-AllNew,N]|Y,-,O_prev):-
78   octaves(BT,N,O_prev,N_octave),
79   tr_voicestates(NV,SS,VS,VS_Next),
80   msw(tr_ss(SS),SS_Next),
81   hmm(NV,2,Length,SS_Next,VS_Next,OrigBT,BT,Y,N,N_octave).
82
83 % last phase:
84 hmm(NV,T,T,SS,VS,-,BT,[(L,V,N)],N_prev,O_prev):-
85   observe(NV,SS,VS,BT,L,V,N,N_prev,O_prev). % final observations
86
87
88 %-----
89 % Model:   (details)
90
91 % observation of one (L,V,N)-tuple
92 observe(NV,SS,VS,BT,L,V,N,N_prev,O_prev):-
93   msw(out_L(SS),L), % duration is L in this state
94   nlist(NV,SS,VS,BT,N,N_prev,V,O_prev). % check note list

```

```

95
96 % NV: number of voices left to check
97 % SS: current song state
98 % VS: voice state for current voice
99 % BT: base tone
100 nlist (NV,SS,[VS|RVS],BT,[Note|RN],[PrNote|RPN],[OldNew|RON],[PrOct|RPO]) :-
101     NV > 0,
102     check_note(SS,VS,BT,Note,PrOct),           % check note itself
103     check_new(SS,VS,Note,PrNote,OldNew),       % check if note is a new one
104     NV1 is NV-1,
105     nlist(NV1,SS,RVS,BT,RN,RPN,RON,RPO).      % next voice
106 nlist(0,-,[],-,[],[],[],[]).                % done
107
108
109 check_note(SS,VS,BT,Note,PrevOct) :-
110     msw(out_rest(SS,VS),X),                    % is it a rest or a real note?
111     check_note(SS,VS,BT,Note,PrevOct,X).
112
113 check_note(_,-,-,Note,-,rest) :-              % note is a rest
114     Note = r.
115 check_note(SS,VS,BT,Note,PrevOct,note) :-    % note is a real note:
116     check_real_note(SS,VS,BT,Note,PrevOct).  % check pitch & octave
117
118 check_real_note(SS,VS,BT,Note,PrevOct) :-
119     number(Note),                              % training
120     ModNote is (Note-BT) mod 12,
121     Octave is (Note-BT) // 12,
122     OctDiff is Octave - PrevOct,
123     msw(out_modnote(SS,VS),ModNote),
124     msw(out_octave(SS,VS),OctDiff).
125
126 check_real_note(SS,VS,BT,Note,PrevOct) :-
127     var(Note),                                  % sampling
128     msw(out_modnote(SS,VS),ModNote),
129     msw(out_octave(SS,VS),OctDiff),
130     NewOctave is PrevOct + OctDiff,
131     sanity_check(NewOctave,SaneNewOctave),    % stay within boundaries
132     NewNote is BT+ ModNote + 12*SaneNewOctave,
133     Note = NewNote.
134
135
136 sanity_check(NewOctave,NewOctave) :-
137     NewOctave > 1, NewOctave < 7.             % acceptable
138
139 sanity_check(NewOctave,SaneNewOctave) :-
140     NewOctave > 6,                             % too high
141     SaneNewOctave = 6.
142
143 sanity_check(NewOctave,SaneNewOctave) :-
144     NewOctave < 2,                             % too low
145     SaneNewOctave = 2.
146
147 % if two notes are the same, the second one can be either 'old' or 'new'
148 check_new(SS,VS,X,X,OldNew) :-
149     number(X),                                  % unless they are both rests
150     msw(out_V(SS,VS),OldNew).

```

```

151 check_new( _ , _ , r , r , old ) .           % second rest is always an 'old' note
152
153 check_new( _ , _ , r , A , new ) :-         % a rest is always 'new'
154     number(A) .                             % when it comes after a note
155
156 check_new( _ , _ , A , r , new ) :-         % a note after a rest is also always 'new'
157     number(A) .
158
159 check_new( _ , _ , A , B , new ) :-         % it must be a 'new' note
160     number(A) , number(B) ,                 % when the two non-rest notes
161     A =\= B .                               % are different
162
163
164
165 % octaves(BT,NL,POL,OL): copy octave values of note-list NL to OL, using
166 %                               previous octave value when the note is a rest
167 octaves(BT, [r | RestN] , [PrevOct | RestPO] , [PrevOct | RestO]) :-
168     octaves(BT, RestN , RestPO , RestO) .
169 octaves(BT, [Note | RestN] , [_ | RestPO] , [Octave | RestO]) :-
170     number(Note) ,
171     NoteOctave is (Note-BT) // 12 ,
172     Octave = NoteOctave ,
173     octaves(BT, RestN , RestPO , RestO) .
174 octaves( _ , [] , [] , [] ) .
175
176
177 % initialize list to NV times the value N
178 initlist(0, _ , []).
179 initlist(NV,N,[N|R]) :-
180     NV > 0 ,
181     NV1 is NV-1 ,
182     initlist(NV1,N,R) .
183
184 % choose the next states for the voice-scope HMM's
185 tr_voicestates(0, _ , [] , []).
186 tr_voicestates(NV,SS,[VS|RVS],[VS_Next|RVS_Next]) :-
187     NV > 0 ,
188     msw(tr_vs(SS,VS),VS_Next) ,             % choose voice state
189     NV1 is NV-1 ,
190     tr_voicestates(NV1,SS,RVS,RVS_Next) . % next voice

```

# Bibliografie

- [Ben99] Yoshua Bengio. Markovian models for sequential data. *Neural Computing Surveys*, 2:129–162, 1999.
- [BP66] Leonard E. Baum en Ted Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, 37:1554–1563, 1966.
- [BPr] B-prolog homepage. <http://www.probp.com>.
- [BPSW70] Leonard E. Baum, Ted Petrie, George Soules, en Norman Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *Annals of Mathematical Statistics*, 41:164–171, 1970.
- [CA01] Darrell Conklin en Christina Anagnostopoulou. Representation and discovery of multiple viewpoint patterns. In *Proceedings of the International Computer Music Conference*, pages 479–485, Havana, Cuba, 2001.
- [Con02] Darrell Conklin. Representation and discovery of vertical patterns in music. In C. Anagnostopoulou, M. Ferrand, en A. Smaill, editors, *Music and Artificial Intelligence: Proc. ICMAI 2002*, number 2445 in Lecture Notes in Artificial Intelligence, pages 32–42. Springer-Verlag, 2002.
- [Con03] Darrell Conklin. Music generation from statistical models. In *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*, pages 30–35, Aberystwyth, Wales, 2003.
- [CW95] Darrell Conklin en Ian H. Witten. Multiple viewpoint systems for music prediction. *International Journal of New Music Research*, 24(1):51–73, 1995.
- [DLR77] A. P. Dempster, N. M. Laird, en D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, Series B (Methodological)*, 39(1):1–38, 1977.
- [Haw] Neil V. Hawes. Basic music theory on the world wide web. [http://ourworld.compuserve.com/homepages/Neil\\_Hawes/theory.htm](http://ourworld.compuserve.com/homepages/Neil_Hawes/theory.htm).
- [Lil] Lilypond homepage. <http://www.lilypond.org>.
- [Mar97] Yuval Marom. *Improvising Jazz using Markov chains*. Honours Thesis, University of Western Australia, 1997.

- [Mut] Mutopia project. <http://www.mutopiaproject.org>.
- [NN03] Han-Wen Nienhuys en Jan Nieuwenhuizen. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics*, Firenze, Italy, may 2003.
- [Pri] Prism homepage. <http://mi.cs.titech.ac.jp/prism/>.
- [PS01] Emanuele Pollastri en Giuliano Simoncelli. Classification of melodies by composer with Hidden Markov Models. In *Proceedings of the First International Conference on WEB Delivering of Music*, pages 88–95, Firenze, Italy, 2001.
- [Sat95] Taisuke Sato. A statistiscal learning method for logic programs with distribution semantics. In *Proceedings of the 12th International Conference on Logic Programming*, pages 715–729, Tokyo, Japan, 1995.
- [Sat97] Taisuke Sato. PRISM: A symbolic-statistical modeling language. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1330–1335, Nagoya, Japan, 1997.
- [SK01] Taisuke Sato en Yoshitaka Kameya. Parameter learning of logic programs for symbolic-statistical modeling. *Journal of Artificial Intelligence Research*, 15:391–454, 2001.
- [Vra03] Bart Vrancken. *Componeren van polyphone muziek*. Licentiaatsthesis, K.U. Leuven, 2003.
- [ZS02] Neng-Fa Zhou en Taisuke Sato. *Toward a High-performance System for Symbolic and Statistical Modeling*, 2002.
- [ZS03] Neng-Fa Zhou en Taisuke Sato. *A Reference Guide to PRISM*, Feb 2003.