IRCAM
Paris
Equipe *Repmus*

Encadrants :
Florent JACQUEMARD
Jean BRESSON

Quantification musicale avec apprentissage sur des exemples

Adrien MAIRE

Rapport de stage effectué durant l'été 2013

Résumé : Il est exposé ici une méthode de quantification rythmique avec apprentissage du style du compositeur. Des techniques d'inférence de langage d'arbres de rythme à l'aide d'automate probabilistes sont utilisées à ces fins. L'enjeu est une intégration à l'environnement de composition assisté par ordinateur OpenMusic.

Mots clés : *Informatique musicale, théorie des langages d'arbres, apprentissage d'automates d'arbres probabilistes*

Table des matières

Résumé

La quantification rythmique est un problème qui consiste à générer à partir d'une suite de notes datées en secondes (par exemple au format MIDI), une partition musicale traditionnelle, qui elle correspond à une suite de rationnels. Les solutions commerciales existantes sont calibrées pour des rythmes très simples et ne sont adaptées qu'à un répertoire limité. D'autres approches, plus générales, demandent à l'utilisateur de choisir certains paramètres qui vont guider la quantification.

En fait, le problème de la quantification rythmique n'admet pas de solution dans l'absolu : un même morceau peut avoir plusieurs solutions satisfaisantes en fonction du style voulu par le compositeur, du contexte de génération de la piste d'entrée (il peut par exemple y avoir des imprécisions dues à l'interprétation), ou plus généralement des préférences de l'utilisateur (certains rythmes peuvent parfois avoir plusieurs notations équivalentes).

D'autre part, le rythme étant une notion intrinsèquement arborescente, on peut s'interroger sur la nécessité d'utiliser la notion d'arbre au sein de la quantification.

Au cours de mon stage, je me suis donc efforcé de proposer une solution à ce problème qui utilise la structure arborescente du rythme pour pouvoir le quantifier, tout en apprenant les préférences de l'utilisateur à partir d'exemples que ce dernier fournit en sélectionnant, pour certaines mesures du morceau, les solutions qu'il préfère (ou qu'il rejette) parmi les différentes quantifications proposées par notre quantificateur. La finalité de ce travail serait donc d'implémenter concrètement un tel quantificateur et de l'intégrer à l'environnement OpenMusic, qui est un logiciel de composition musicale assistée par ordinateur développé par l'IRCAM.

Dans un premier temps, je me suis donc documenté sur les divers aspects de ce sujet afin de proposer un cadre et des définitions concrètes concernant notre objectif pour proposer une réponse aux diverses questions que l'on se pose : Que doit réellement apprendre notre algorithme, et par quels moyens ? Comment formaliser mathématiquement les "gouts" et les "préférences" du compositeur ?

Une fois ce travail effectué, j'ai alors élaboré un prototype, qui décrit précisément les différentes étapes que doit suivre notre quantificateur. Ce dernier doit proposer une liste de propositions de quantifications pour chaque mesure du morceau en entrée, et dont le premier élément de la liste est celui qui correspond à la quantification que l'algorithme estime être la plus pertinente, et qui est celle qu'il affiche sous forme de partition par défaut. Puis, l'utilisateur peut sélectionner, parmi les différentes quantifications proposées, celles qu'il juge satisfaisantes, ou insatisfaisantes, voire totalement hors de propos. (Il est également libre d'en ajouter une manuellement, si jamais aucune ne venait à le satisfaire pleinement).

Puis, à l'aide de ces exemples et contrexemples, notre algorithme va générer un

automate d'arbres (les arbres sont ici les quantification candidates pour une mesure donnée) probabiliste A, qui pourra noter les différents arbres en fonction de la probabilité avec laquelle il estime que ceux-ci pourront satisfaire l'utilisateur dans l'absolu (sans pour l'instant prendre en compte la forme de l'entrée). La construction d'un tel automate A se fait en construisant préalablement l'automate qui ne reconnaît que les exemples positifs, puis en étendant le langage par un mécanisme de fusions d'états, sous conditions que les contrexemples ne deviennent pas plus probables que les exemples, et que les contre-exemples absolus (hors de propos), gardent bien une probabilité nulle.

D'autre part, notre algorithme est pourvu d'une fonction d'évaluation B, qui, elle, note les arbre en fonction de leur distance à la mesure d'entrée.

Enfin, il faut alors proposer une solution pour l'énumération de la liste ordonnée des candidats, qui ne soit pas trop gourmande en complexité temporelle. Notre solution actuelle est pour le moins heuristique, néanmoins deux autres pistes sérieuses de recherche, qui se baseraient sur des généralisations d'algorithme classiques de recherche de plus court chemin, sont en cours de réflexion.

Pour conclure, nous avons mis sur pieds un prototype précis de ce que peut être un procédé de quantification rythmique avec apprentissage des goûts/du style de l'utilisateur. Il reste désormais à l'implémenter, à l'intégrer à OpenMusic, puis à valider son bon fonctionnement en poursuivant nos discussions avec les compositeurs de l'IRCAM qui sont les principaux utilisateurs d'OpenMusic.

1 Contexte global

1.1 L'IRCAM et L'équipe Représentation musicale

J'ai effectué ce stage d'été au sein de l'IRCAM, qui, au centre de Paris, est un institut de recherche, de création et de production artistique dans le domaine de la musique et du son. Centre le plus important dans le monde pour la création musicale en relation avec les sciences et les technologies, l'IRCAM a initié plusieurs standards de fait en informatique musicale comme le logiciel Max/MSP, mondialement utilisé, ou le logiciel OpenMusic, développé dans l'équipe Représentations musicales où j'ai fait mon stage, qui est l'environnement de composition assistée par ordinateur le plus largement distribué.

L'équipe Représentations musicales mène des recherches et développements sur la représentation symbolique des structures musicales et les langages et paradigmes informatiques adaptés à la musique. Ces travaux mènent à des applications dans les domaines de la composition assistée par ordinateur (CAO) et de la musicologie computationnelle. La réflexion sur les représentations de haut niveau des concepts et des structures musicales, appuyée sur les langages informatiques originaux développés par l'équipe débouche sur l'implantation de modèles qui peuvent se tourner vers la création comme vers l'analyse musicale.

1.2 La transcription musicale et la quantification rythmique

La transcription musicale est un vieux problème qui consiste à vouloir convertir de façon automatique un enregistrement sonore, en une description musicale détaillée et lisible par un être humain, typiquement une partition musicale au sens traditionnel et occidental du terme. C'est un problème très vaste, qui n'a toujours pas été résolu et qui fait l'objet de nombreuses recherches.

Il se décompose en de nombreux sous-problèmes, parmi lesquels celui de la détection et la différenciation des différentes pistes musicales, celui de la détection de la hauteur des notes jouées par ces dernières (pitch tracking en anglais), ainsi que d'autres problèmes concernés plus spécifiquement par les durées : la recherche des pulsations (beat tracking), celle du tempo (tempo tracking), et enfin le problème de la quantification rythmique, qui est celui sur lequel j'ai travaillé.

En musique, le rythme est la perception d'une structure dans un ensemble d'événements sonores, habituellement, des notes. Il opère une division de la durée, et ces

divisions interviennent à une certaine fréquence qu'on appelle tempo. Les divisions forment, en les rassemblant, des valeurs rythmiques ♩ ♪ ♪ ♩ … Ces trois paramètres constituent ce qui est appelé en musique le rythme [11].

La quantification rythmique est donc un problème classique en recherche musicale. Il peut se définir ainsi : conversion d'un flux de durées exprimées par des valeurs réelles (dans la pratique en format MIDI) en une structure rythmique, hiérarchique et discrète, qui puisse être exprimée de façon significative dans le système habituel de notation musicale. En toute généralité, les étapes préliminaires nécessaires sont la détermination des différentes lignes musicales, lorsque deux notes peuvent être jouées dans un même laps de temps, ainsi que le découpage de notre morceau en mesures, et le découpage de ces mesures en temps (un temps est la durée entre deux pulsations).

Dans le travail que j'ai effectué, nous considérons que ces questions ont déjà été traitées au préalable (automatiquement ou manuellement), par conséquent, le problème auquel nous nous sommes attelés peut se résumer ainsi : construire un procédé automatique qui transforme une suite d'évènements symboliques datés, strictement consécutifs et segmenté en temps, en une partition musicale. Plus précisément, cela est réalisé par une procédure itérative qui s'applique successivement à chaque temps, pour produire le rythme correspondant aux évènements situés à l'intérieur du temps.

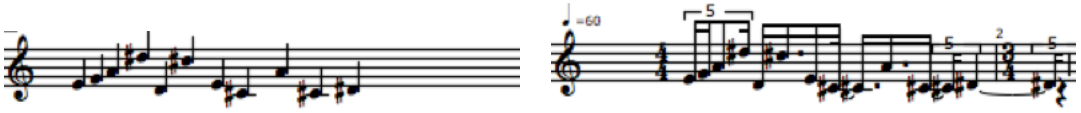

FIGURE 1.1 – Un exemple de quantification : à gauche, une suite de notes dans le temps, et à gauche une partition qui exprime une certaine structure rhytmique

Pour effectuer cette quantification à l'intérieur du temps, on trouve dans la littérature plusieurs approches, qui consistent toujours, peu ou prou, à placer des notes dans une grille. Citons en deux :

Approche naïve. Son principe est de générer une grille de subdivisions régulières, puis de caler les attaques et les durées sur la grille. Cette approche atteint très vite ses limites face à des rythmes complexes, ou même, dès que la subdivision principale choisie ne correspond pas à la subdivision principale du morceau (typiquement si une subdivision binaire est choisie alors que le morceau est ternaire).

Approche Heuristique. Le principe consiste à tester un certain nombre de grilles, qui ne sont pas forcément régulières, et à leur attribuer une note à l'aide d'une mesure d'erreur. Ensuite, il reste à choisir la grille qui a la meilleure note.

C'est une méthode de ce type qui a été utilisé par le quantificateur *Kant* [1], qui est l'ancêtre du quantificateur utilisé dans OpenMusic (dont nous parlerons ci-après). Pour pouvoir attribuer ces notes, il utilisait différentes mesures de distance entre une entrée et un rythme quantifié (voir section 2.6 pour des exemples de telles distances).

1.3 L'environnement OpenMusic

Une motivation importante de mes recherches sur la quantification rythmique est de participer indirectement à l'élaboration, et à l'amélioration du quantificateur d'OpenMusic.

OpenMusic est un environnement de programmation graphique dédié à la composition musicale assistée par ordinateur. Il permet d'écrire tout type de programmes de manière graphique, mais son but premier est de réaliser ces programmes dans une optique compositionnelle.

Basé sur le langage Common Lisp Object System, OpenMusic a été créé par Gérard Assayag, Carlos Agon et Jean Bresson.

Voici la page web du projet : `http://repmus.ircam.fr/openmusic/home`.

Les œuvres (ou les programmes) se construisent dans un patch, dans lequel des boites qui possèdent des entrées et des sorties sont connectées entre elles. (voir figure 1.2)

Arbres de rythme OpenMusic. Le rythme est une structure profondément hiérarchique : un morceau est divisé en mesures, qui comportent par exemple quatre noires, elles-même divisées chacune en deux croches etc. Il est donc tout naturel de représenter un rythme par un arbre. La notation traditionnelle est elle-même intrinsèquement arborescente.

Pour ces raisons, OpenMusic utilise en mémoire des arbres de rythme pour représenter ses données musicales. Plus qu'un outil, ces arbres de rythme constituent la structure de données de base de représentation de la musique dans OpenMusic. Autrement dit, dans OpenMusic, toute piste musicale est un arbre de rythme. Cela est essentiellement dû au fait que les arbres de rythme sont canoniquement convertibles en portée, et que toute donnée musicale doit pouvoir être affichée sur une portée, puisque les utilisateurs d'OpenMusic sont des compositeurs.

La librairie OMquantify. Actuellement, OpenMusic est pourvu d'une libraire nommée OMquantify, qui est un quantificateur rythmique. Il a été développé par Benoît

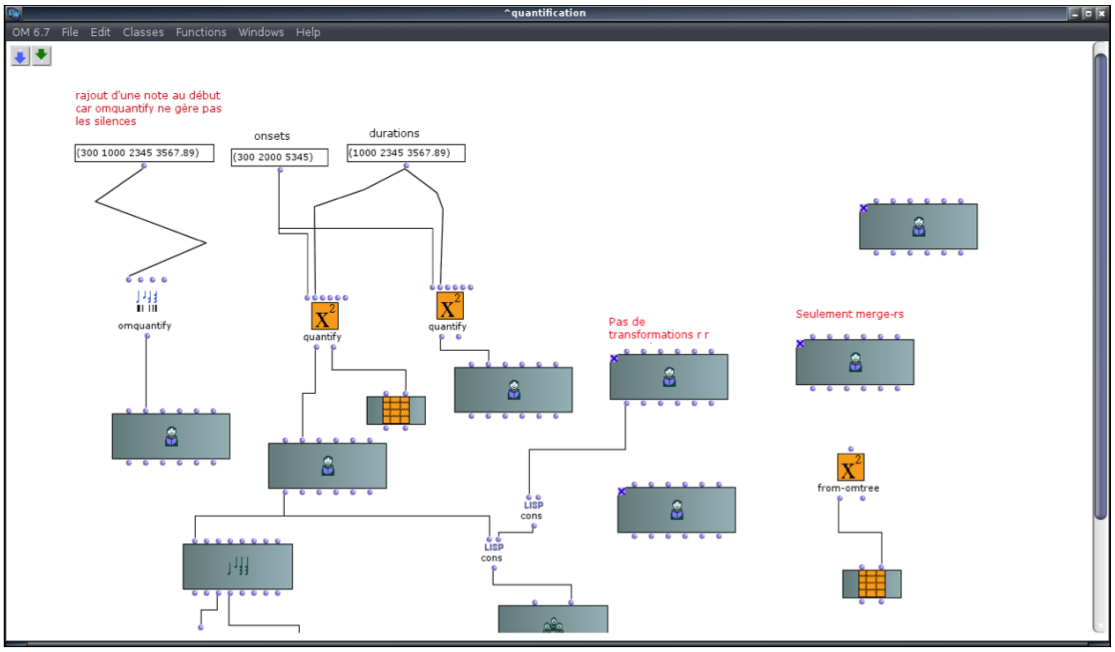

FIGURE 1.2 – Un patch, unité de base d'un programme dans *OpenMusic*.

Meudic [15] et est principalement une amélioration de Kant. Le processus de quantification s'y fait donc sur des grilles, et la solution retenue est alors traduite en dernier lieu en un arbre de rythme.

1.4 L'objectif fixé

Comme nous venons de le voir, les différentes méthodes qui existent pour quantifier un rythme, ne se servent pas de la structure arborescente du rythme. Un des objectif de mon stage est justement de chercher à utiliser la structure d'arbre, qui comme nous venons de le rappeler est fondamentale en théorie musicale, en amont dans le processus de quantification.

Par ailleurs, ceci nous fournira l'occasion d'utiliser des algorithmes sur les arbres, ainsi que des outils de théorie des langages d'arbre, et, comme nous le verrons plus particulièrement, des techniques d'apprentissage d'automates d'arbres pondérés.

D'autre part, le problème de la quantification musicale a ceci de particulier qu'il ne possède pas de solution unique. En effet, pour une même entrée donnée, deux compositeurs différents peuvent très bien attendre des quantifications différentes. Un logiciel de quantification optimal serait donc un logiciel qui s'adapterait à son utilisateur. Il nous faut donc réfléchir à l'idée d'une interaction entre le quantificateur et

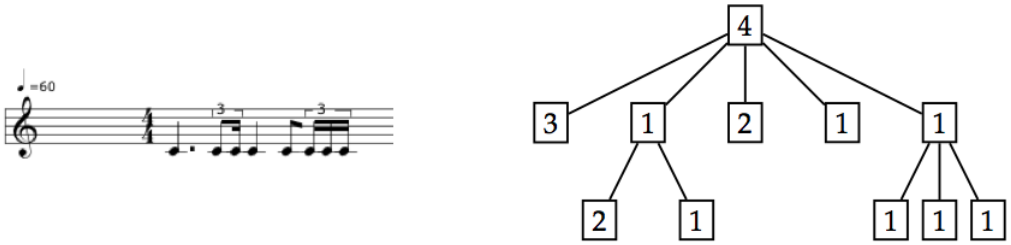

FIGURE 1.3 – À gauche, la représentation traditionnelle d'un rythme, et à droite, la structure d'arbre de rythme qu'utilise OpenMusic pour se le représenter en mémoire. Le chiffre 4, à la racine désigne le nombre de temps dans la mesure, et les autres chiffres correspondent de façon barycentrique (on pourrait par exemple tous les multiplier par 2) à la durée des sous-arbres qu'ils étiquettent.

le compositeur.

Nous venons donc de voir qu'il y a une part de subjectivité dans la quantification rythmique, et que par conséquent, le quantificateur idéal serait donc capable d'apprendre les "goûts" de de son utilisateur en terme d'écriture rythmique et nous pouvons alors essayer de ramener ceci à un problème d'apprentissage de langage d'arbres (le langage modéliserait donc ici les préférences de l'utilisateur). De plus, des travaux de recherche utilisant des techniques d'inférence de langages d'arbres ont déjà été menés dans un cadre musicale, par exemple pour la recherche de similarités musicales dans l'équipe de José-Manuel Iñesta [18, 2].

les "goûts" → le style

Pour résumer, l'objet de nos recherches est donc de proposer une méthode de quantification musicale utilisant la structure arborescente du rythme pour s'adapter aux préférences de l'utilisateur, avec pour ligne de mire, une intégration dans OpenMusic.

2 La méthode suivie

2.1 Définition formelle du problème

La première partie de mon travail a consisté en l'élaboration d'un cadre formel pour définir avec plus de précision le problème que nous souhaitions résoudre. En effet, si l'idée de concevoir un algorithme "apprenne" les "goûts" de l'utilisateur, à l'aide de techniques d'inférence d'automates d'arbres peut sembler intuitivement claire, il a fallu dans un premier temps clarifier notre objectif, et proposer des définitions adéquates pour les notions d'apprentissage et goût.

les "goûts" → le style ?

Commençons par expliciter les entrées, et les sorties de l'algorithme que nous souhaitons élaborer.

Nous rappelons que, d'une part, le découpage en mesures de notre piste d'entrée a été préalablement effectué (de façon automatique, ou bien par l'utilisateur), et que, d'autre part, cette entrée est monophonique : ses notes sont strictement consécutives.

Nous sommes donc naturellement amenés à définir l'entrée comme une suite de séquences de notes :

Définition 1 (Séquence)**.** *Une séquence s de taille n est une liste de n couples de réels $((x_1, y_1), \ldots, (x_n, y_n))$ qui vérifient $x_{i1} < y_{i1} < \ldots < x_{ni} < y_{ni}$ et qui correspondent aux instants d'attaque et de fin des notes.*

Définition 2 (Entrée)**.** *Une entrée $\overline{s}$ (source) à n mesures est une suite de séquences $(s_1, s_2, \ldots, s_n)$. Elle correspond au morceau musical donné en entrée, et préalablement découpé en mesures (chaque séquence correspond à une mesure).*

Notation 1. *Nous noterons S_n l'ensemble des entrées à n mesures, et $S = \bigcup_{n \in \mathbb{N}} S_n$, l'ensemble des entrées possibles.*

Pour ce qui est de la sortie, comme nous l'avons dit précédemment, au lieu de la formaliser comme une suite de rationnels dont la somme sur chaque mesure ferait 1 (ou bien même un autre entier déterminé par l'utilisateur qui correspondrait au nombre de temps dans la mesure), nous préfèrerons la représenter sous forme d'arbre de rythmes.

Il existe plusieurs façon équivalentes de représenter un rythme de façon arborescente, celle que nous retenons ici a été proposée par Pierre Donat-Bouillud, qui effectuait aussi un stage à l'IRCAM sur la transcription rythmique. Elle possède l'avantage d'utiliser un alphabet de seulement quatre lettres (en plus de celles qui déterminent l'arité des noeuds internes), ce qui limitera la complexité de nos algorithmes.

En voici les symboles :

n : note

r : silence (*rest*)

s : liaison avec la note précédente (*slur*)

$=$: le premier symbole différent de $=$ dure "1+ le nombre de $=$ précédents adjacents"

Chacun des fils d'un nœud a une durée égale.

Définition 3 (Arbres de rythme symboliques)**.** *Un arbre de rythme symbolique est un arbre de rang borné*[1] *dont les feuilles sont étiquetées par des symboles de* $\{n, r, s, =\}$*, et dont tous les noeuds internes sont étiquetés par un symbole qui détermine le nombre de ses fils (que l'on omet de noter).*

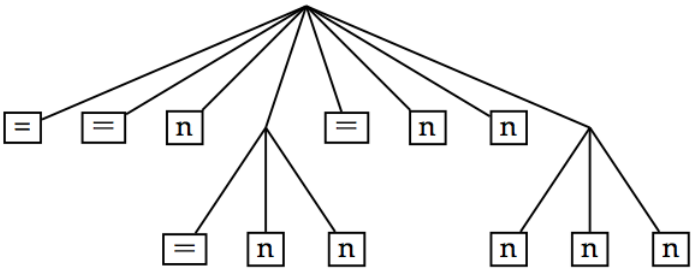

FIGURE 2.1 – Un arbre symbolique de rythme, qui correspond au rythme de la figure 1.3.

Désormais, nous pouvons proposer une formalisation de notre objectif de conception d'algorithme qui apprendrait les préférences de l'utilisateur.

Soit L, un langage d'arbres de rythme, c'est-à-dire un sous-ensemble de l'ensemble des arbres de rythme correspondant à des conditions imposées préalablement par l'utilisateur, voir section 2.3. *L fini ?*

Définition 4 (Sortie)**.** *Une sortie* $\bar{t}$ *(target), associée à une entrée* $\bar{s}$*, dans un langage d'arbres* L*, est une suite* $(t_1, \ldots, t_n)$ *d'arbres de* L *de même longueur que* $\bar{s}$*.*

Notation 2. *Nous noterons* T_n *l'ensemble des sorties à* n *mesures, et* $T = \bigcup_{n \in \mathbb{N}} T_n$ *, l'ensemble des sorties possibles.*

Nous pouvons désormais modéliser les goûts de l'utilisateur par une fonction f, comme ceci : *definition* $f(s_i, t)$

1. voir la définition 10, section 2.2, pour la définition d'arbre.

Définition 5 (Goût)**.** *Le goût de l'utilisateur est décrit par une fonction f définie sur $S \times T$, et à valeur dans $[0,1]$ telle que pour tout entrée $\bar{s}$, $\sum_{\bar{t} \in T} f(\bar{s}, \bar{t}) = 1$, ainsi, $f(\bar{s}, \bar{t})$ désigne la probabilité que la sortie $\bar{t}$, corresponde à la volonté de l'utilisateur.*

Pour pouvoir apprendre, l'utilisateur doit fournir à notre algorithme une liste d'exemples de quantification, en voici donc une définition :

Définition 6 (Exemples)**.** *Une liste d'exemples associée à une entrée $\bar{s} = (s_1, s_2, \ldots, s_n)$, est une famille $E = (E_1, \ldots, E_n)$ d'ensembles finis (éventuellement vides) de sorties telle que pour tout i, $1 \leq i \leq n$, E_i est une partie de L munie d'un ordre total $<_i$ où $t <_i t'$ signifie que $f(s_i, t) < f(s_i, t')$.* [2]

Il s'agit donc d'un tri des préférences de l'utilisateur sur certaines mesures de l'entrée.

Remarque 1. *Le format de proposé pour la suite E n'est pas le seul envisageable, on pourrait également considérer des ensembles d'exemples et/ou de contrexemples de couples arbre/séquence, ou bien un système de notation etc. Si nous avons retenu celui-ci, c'est qu'il nous semblait le plus général possible, en ce sens que les autres alternatives peuvent être ramenées à cette définition. D'ailleurs, dans la solution que nous proposerons par la suite, nous nous contenterons d'exemples et de contrexemples.*

Objectif. Notre problème consiste à chercher un algorithme qui aurait en entrée
- une entrée $\bar{s}$;
- un langage d'arbres de rythme initial L ;
- une liste d'exemples E associée à $\bar{s}$;

De plus, on suppose l'existence d'une fonction de goût f, inconnue mais qu'il ne connaît pas, et dont il va justement chercher à approximer la valeur.

L'algorithme doit retourner en sortie une fonction g approximant au mieux f au vu des informations fournies par E. Cela nous mène à la définition suivante :

Définition 7. *Un algorithme d'apprentissage de quantification rythmique est un algorithme A, qui prend en entrée une entrée $\bar{s}$, un langage d'arbre L, et des exemples E associés à $\bar{s}$ et L, et retourne une fonction $g_{\bar{s},E}$, définie sur T, et à valeur dans $[0,1]$ telle que pour toute entrée $\bar{s}$, $\sum_{\bar{t} \in T} g_{\bar{s},E}(\bar{t}) = 1$, qui vérifie pour toute sortie $\bar{t} \in T$, $\lim_{|E| \to +\infty} g_{\bar{s},E}(\bar{t}) = f(\bar{s}, \bar{t})$.*

2.2 Définitions et outils préliminaires

Pour pouvoir vous présenter la solution que nous avons développée, il est au préalable nécessaire de fournir quelques définitions, et résultats généraux sur les arbres, et automates d'arbres, dont nous nous servirons par la suite.

Notation 3. *On note $\mathbb{N}$ l'ensemble des entiers naturels strictement positifs, et $\mathbb{N}^*$ l'ensemble des chaînes finies d'entiers (le monoïde libre engendré par $\mathbb{N}$), et ϵ la chaîne vide. On note $p.p'$ la concaténation de deux chaînes $p, p \in \mathbb{N}^*$.*

2. Par abus de langage, on ne fait pas de différence entre la séquence s_i et l'entrée de longueur 1 réduite à s_i et de même pour pour les sorties.

Définition 8 (Fermé par préfixe). *On dit qu'un sous-ensemble P de $\mathbb{N}^*$ est fermé par préfixe si pour tout $p, p' \in \mathbb{N}^*$, si $p.p' \in P$, alors $p \in P$.*

Définition 9 (Domaine). *Un ensemble de positions P (ou domaine) est un sous-ensemble fini non vide de $\mathbb{N}^*$ qui est fermé par préfixe, et tel que pour tout $p.i \in P$, on a $p.j \in P$ quand $1 \leq j \leq i$.*

Définition 10 (Arbre). *Un arbre t sur un alphabet Σ (ou Σ-arbre) est une fonction d'un ensemble de positions $Dom(t)$ vers Σ. À chaque position dans $Dom(t)$ correspond un noeud dans l'arbre, étiqueté par $t(p)$. L'arité d'un noeud $p \in Dom(t)$ est le nombre de ses successeurs directs (ou fils), c'est-à-dire le nombre de positions $p.i \in Dom(t)$, où $i \in \mathbb{N}$. Une feuille est un noeud d'arité nulle. La taille $|t|$ d'un arbre t est le cardinal de son domaine, et sa hauteur $h(t)$ est la longueur maximale des chaînes de son domaine.*

Définition 11 (Alphabet gradué). *Un alphabet gradué (ranked alphabet en anglais) est un alphabet Σ que l'on peut partitionner de la sorte $\Sigma = \bigcup_{n=0}^{N} \Sigma_n$.*

Ce qui nous permet d'arriver à la définition d'arbre de rang borné, qui sont ceux que nous utiliserons tout au long de ce travail.

Définition 12 (Arbre de rang borné). *Un arbre de rang borné est un arbre sur un alphabet gradué Σ tel que pour tout $n \leq N$ et tout $f \in \Sigma_n$, l'arité de tout noeud étiqueté par f est n.*

rang → arité

Définition 13 (Automate d'arbres déterministe (vision ascendante)). *Un automate d'arbres déterministe est un quadruplet $A = (Q, \Sigma, \delta, F)$ où*

- *Q est un ensemble fini d'états*
- *Σ est un alphabet gradué*
- *δ est une fonction de $\Sigma \times \bigcup_n Q^n$ dans Q*
- *$F \subseteq Q$ est l'ensemble des états finaux.*

Définition 14 (Calcul et langage d'automate d'arbres deterministe). *Le calcul de l'automate A sur un Σ-arbre t est un Q-arbre r ayant le même domaine que t et tel que pour tout $u \in Dom(t)$ d'arité n, on a $\delta(t(u), r(u.1), ..., r(u.n)) = r(u)$. On dit que le calcul est acceptant si $r(\epsilon) \in F$, et on appelle $L(A)$ le langage des arbres accepté par A.*

Si un langage est reconnu par automate déterministe, alors on peut facilement reconnaître son complémentaire : il suffit de d'inverser les états finaux et non-finaux, ce qui a une complexité linéaire en le nombre d'état.

Si deux langages L_1 et L_2 sont reconnus respectivement par les automates déterministes $A_1 = (Q_1, \Sigma, \delta_1, F_1)$ et $A_2 = (Q_2, \Sigma, \delta_2, F_2)$ alors on peut construire un automate déterministe qui reconnait $L_1 \cup L_2$ en considérant l'automate $A = (Q, \Sigma, \delta, F)$ où :

- $Q = Q_1 \times Q_2$
- $F = F_1 \times Q_2 \cup F_2 \times Q_1$
- $\delta : (f, (q_1, q_1'), ..., (q_n, q_n')) \to (\delta_1(f, q_1, ..., q_n), \delta_2(f, q_1', ..., q_n'))$.

Ce qui nous donne une complexité quadratique en le nombre d'état.

Enfin, on peut aussi reconnaître l'intersection de deux langage en combinant intersection et complémentaire. La propriété suivante résume tout cela :

Théorème 1 (Union, intersection et complémentaire d'automates déterministes)**.** *On peut construire des automates qui reconnaissent l'union, l'intersection et le complémentaire de langages reconnus par automates déterministes, et un temps respectivement quadratique, quadratique et linéaire en le nombre d'états.*

Voici une petite propriété dont nous aurons besoin par la suite :

Théorème 2 (Un langage d'arbres fini est reconnaissable)**.** *Pour tout langage d'arbres fini L, on peut construire un automate d'arbres déterministe reconnaissant exactement L en un temps et d'une taille linéaires en le nombre de noeuds de tous les arbres de L.*

Démonstration. Il suffit de construire un automate dont les états sont l'ensemble des sous-arbres de tous les arbres de L, plus un état poubelle p, et les transitions suivantes $\delta : (f, q_1, ..., q_n) \to f(q_1, ..., q_n)$ si q_1,..., q_n ne sont pas p et que $f(q_1, ..., q_n)$ est bien un sous-arbre de L (et donc un état) ; sinon $\delta : (f, q_1, ..., q_n) \to p$. □

On peut aussi généraliser ces définitions à des automates d'arbres qui ne sont pas nécessairement déterministes, pour lesquels il peut exister plusieurs transitions à partir d'un même membre de gauche :

Définition 15 (Automate d'arbres non déterministe (vision ascendante))**.** *Un automate d'arbres non déterministe est un quadruplet $A = (Q, \Sigma, \delta, F)$ où*
- *Q est un ensemble fini d'états*
- *Σ est un alphabet gradué*
- *$\delta \subseteq \Sigma \times \bigcup_n Q^n \times Q$ est l'ensemble fini des transitions*
- *$F \subseteq Q$ est l'ensemble des états finaux.*

Un calcul d'automate non déterministe se définit de la même manière que dans le cas déterministe, mais, contrairement au cas déterministe, il peut y avoir plusieurs calculs pour un même arbre donné. Un arbre est alors reconnu s'il existe un calcul acceptant :

Définition 16 (Calcul et langage d'automate d'arbres)**.** *Un calcul de l'automate A sur un Σ-arbre t est un Q-arbre r ayant le même domaine que t et tel que pour tout $u \in Dom(t)$ d'arité n, on a $(t(u), r(u.1), ..., r(u.n), r(u)) \in \delta$. On dit que le calcul est acceptant si $r(\epsilon) \in F$, et on appelle $L(A)$ le langage des arbres accepté par A, c'est-à-dire pour lesquels il existe un calcul acceptant.*

Un raisonnement analogue à ce qui a été dit pour les automates déterministes, permettent de calculer l'union et l'intersection de langages reconnus par des automates non-déterministes, en des temps quadratiques. En revanche, cela ne fonctionne plus pour le complémentaire (il existe un algorithme, mais qui prend un temps exponentiel en le nombre d'états). Le théorème suivant résume cela.

Théorème 3 (Union, intersection d'automates d'arbres)**.** *On peut construire des automates qui reconnaissent l'union et l'intersection de langages reconnus par des automates non nécessairement déterministes, et un temps quadratique en le nombre d'états.*

Nous allons maintenant définir des automates pondérés, qui ne se contentent plus seulement de reconnaître un langage, mais qui assigne un poids (dans un semi-anneau, généralement $\mathbb{R}$) à chaque arbre sur lequel ils calculent. Ils définissent alors une fonction (parfois appelée série) de l'ensemble des arbres sur un alphabet dans le semi-anneau. Ils généralisent donc les automates d'arbres, qui eux ne définissent qu'une indicatrice (qui renverrait 1 si on est dans le langage, et 0 sinon).

Définition 17 (Automate d'arbres pondéré)**.** *Soit $\mathbb{K}$ un semi-anneau, un automate d'arbres pondéré sur $\mathbb{K}$ est un quadruplet $A = (Q, \Sigma, \delta, F)$ où*

- *Q est un ensemble fini d'états*
- *Σ est un alphabet gradué*
- *δ est une fonction de $(\bigcup_{n=0}^{N} \Sigma_n \times Q^n) \times Q \to \mathbb{K}$.*
- *$F : Q \to \mathbb{K}$ est une fonction finale.*

Définition 18 (Poids)**.** *Soit t un arbre de rang borné et $A = (Q, \Sigma, \delta, F)$ un automate d'arbres pondéré. Un calcul de A sur t est, de même que dans le cas non pondéré, une fonction r de $Dom(t)$ dans Q.*

- *Le poids $\omega(A, t, r, p)$ d'un calcul r sur un sous-arbre de t à la position p est calculé comme ceci : si l'arité de p dans t est 0, alors $\omega(A, r, t, p) = \delta(t(p), r(p))$, sinon, soit $n > 0$ l'arité de p, alors le poids ω est récursivement défini par*

$$\omega(A, t, r, p) = \delta(t(p), r(p.1), ..., r(p.n), r(p)) \cdot \prod_{i=1}^{n} \omega(A, t, r, p.i).$$

- *Le poids d'un calcul r sur t est $\omega(A, t, r) = F(r(\epsilon)) \times \omega(A, t, r, \epsilon)$, i.e. la fonction finale est appliquée à la racine.*
- *Le poids d'un arbre t est la somme des poids de tous les calculs sur t, i.e. $\omega(A, t) = \sum_r \omega(A, t, r)$.*

Notation 4. *On interprète généralement δ comme un ensemble Δ de transitions pondérées, de la façon suivante :*

$$f(q_1, ..., q_n) \xrightarrow{\omega} q \in \Delta \iff \delta(f, q_1, ..., q_n, q) = \omega \text{ et } \omega \neq 0.$$

On note également Δ_q l'ensemble des transitions menant à l'état q.

Un cas particulier d'automate d'arbres pondéré est l'automate probabiliste, où les poids correspondent à des valeurs de probabilités. Ce sont eux que nous utiliserons principalement dans ce qui va suivre.

Définition 19 (Automate d'arbres probabiliste)**.** *Un automate d'arbres probabiliste est un automate d'arbres pondéré sur $\mathbb{R}$ satisfaisant aux trois conditions suivantes :*

1. *δ et F prennent leurs valeurs dans $[0, 1]$*
2. *$\sum_{q \in Q} F(q) = 1$*
3. *pour tout $q \in Q$, $\sum_{\rho \in \Delta_q} \omega(\rho) = 1$.*

Remarque 2. *Le poids d'un arbre évalué par un automate probabiliste peut-être interprétée comme la probabilité que l'arbre appartienne au langage, même si l'on a pas toujours* $\sum_t w(t) = 1$. *En revanche,* $\sum_t w(t) \leq 1$ *est bien toujours vérifié, voir par exemple [7].*

Remarque 3. *À un automate d'arbres probabiliste A, on peut associer, en un temps linéaire, un automate d'arbres B qui reconnaît le langage* $L = \{t, A(t) \neq 0\}$. *En effet, il suffit de ne considérer que les transitions de poids non nul, et comme états finaux, ceux dont l'image par la fonction finale est non nulle.*

Remarque 4. *Pour un langage fini L, on peut construire un automate probabiliste qui reconnaît avec la même probabilité* $1/|L|$ *tous les éléments de L. Il suffit pour cela d'employer la méthode que dans le cas non-pondéré (voir théorème 2), en donnant un poids 1 à chacune des flèches, et une fonction finale qui donne la valeur* $1/|L|$ *à tous les états qui étaient finaux dans le cas non-pondéré.*

2.3 Spécification du problème

Le problème que nous avons décrit dans la section 2.1 est un cadre un peu trop général pour la solution que nous avons développée. Nous allons désormais énoncer quelques hypothèses restrictives, que nous avons dû poser pour pouvoir répondre au problème. Notre algorithme répond alors à une instanciation du problème général, mais il a vocation à être étendu, ou à servir d'inspiration pour l'élaboration d'une solution plus optimale. Voici donc nos différentes hypothèses :

Limitation du langage La première est que nous nous intéresserons aux langages finis d'arbre de profondeur maximale déterminée et dont l'arité des nœuds est également fixée :

Notation 5. *Soit p un entier,* $Ar = \{a_1, \ldots, a_{|Ar|}\}$ *un ensemble fini d'entiers et s une séquence de taille n. On note indifféremment* $L_{s,Ar,p}$ *ou* $L_{n,Ar,p}$ *l'ensemble des arbres de rythmes à n notes dont la profondeur est au plus p, et dont l'arité des nœuds est dans Ar. Ils correspondent à l'ensemble le plus général qui soit des arbres candidats pour quantifier s.*

Ces données seraient imposées par l'utilisateur.

Localisation de la fonction de goût Une autre restriction que nous ferons, est que nous considérerons que la fonction de goût f n'est non plus définie sur tous les couples (entrée sortie), mais plus localement sur les couples (séquence, arbre de rythme). C'est-à-dire, si l'on note U_n, l'ensemble des séquences à n notes, f est définie sur $\bigcup_{n\in\mathbb{N}} S_n \times L_{n,Ar,p}$. On considère donc que le choix de l'utilisateur pour la quantification d'une mesure est purement local, et ne dépend donc pas de ce qui a été fait sur le reste de la partition. Ce choix a été critiqué par certains musiciens, c'est un des problèmes qu'il faudra résoudre par la suite.

Approche Bayésienne Étant donné le caractère probabiliste que l'on a donné à la fonction f, on peut utiliser, en s'inspirant des travaux d'Ali Taylan Cemgil [4, 3], le théorème de Bayes pour affirmer que $f(e, s) = p(s|e) \sim p(e|s) \times p(s)$, où

— $p(e|s)$ peut être interprété comme inversement proportionnel à la distance entre l'entrée et la sortie. Comme nous le verrons en section 2.6, on peut choisir des distances plus où moins canoniques, ou utiliser des méthodes probabilistes, à l'aide de matrices de covariances, qui ont été développées afin de l'approximer.
— $p(s)$ peut être vue comme une fonction de complexité de la sortie, et ferons le choix de nous restreindre à l'apprentissage de $p(s)$, qui, lui, peut être regardé comme un langage d'arbre probabiliste. C'est sur l'apprentissage de cette valeur que porte l'essentiel de notre travail.

Au final, cette approche, bien qu'assez approximative (car elle n'apprend pas la corrélation voulue entre l'entrée et la sortie, mais seulement le langage attendu pour la sortie) a le mérite de nous ramener à un problème d'apprentissage de langage d'arbres probabiliste. Néanmoins, comme nous allons le voir, nous pourrons peut-être rattraper ce handicap, au moment de l'énumération (voir section 2.7).

2.4 Présentation de l'algorithme

Nous arrivons enfin, au coeur de notre travail. Je vais ici, vous présenter concrètement un prototype complet de ce que serait notre programme de quantification rythmique. Nous avons d'ores et déjà, implémenté sur OpenMusic le modèle graphique de ce celui-ci, et nous allons détailler ci-après les différents algorithmes permettant son fonctionnement.

Les données auxquelles notre programme a accès sont les suivantes :

— D'une entrée e au sens de la définition 2, c'est-à-dire une partition prédécoupée en mesures.
— De trois ensembles d'exemples fournis par l'utilisateur à savoir :

E^+ qui est l'ensemble des exemples positifs, à savoir un ensemble d'arbres que l'utilisateur a sélectionné à un moment donné comme étant une solution convenable pour une certaine mesure. On pourrait également envisager d'avoir initialement une base de donnée optionnelle d'exemples positifs a priori, en fonction de l'utilisateur, du style musical du morceau, etc.

E^- il s'agit de l'ensemble des exemples négatifs. Ce sont des arbres qui n'ont pas satisfait l'utilisateur à un moment donné (il leur en a préféré un autre), mais dont il a néanmoins jugé qu'ils n'avaient pas une forme totalement inacceptable dans le morceau.

E^{--} l'ensemble des contrexemples absolus, c'est-à-dire des arbres que l'utilisateur a déclaré comme étant complètement hors de son langage, donc pour lesquels $p(t) = 0$.

Notre algorithme va :

1. construire un automate d'arbres probabiliste A qui apprendra, à l'aide des ensembles d'exemples, les préférences de l'utilisateur dans l'absolu (sans prendre

en compte l'entrée), c'est-à-dire le langage d'arbres probabiliste de goût de l'utilisateur qui correspond au langage $p(s)$ de la section 2.3. La construction de l'automate est exposée en détail en section 2.5.

2. construire une fonction B qui mesurera la probabilité qu'un arbre soit la bonne sortie en fonction de sa distance l'entrée. Cette fonction, qui, elle, correspond au $P(e|s)$ de la section 2.3 sera explicitée dans la partie 2.6.
3. énumérer les k-meilleurs arbres t candidats, qui maximisent le produit $A(t) \times B(t)$ (où k est un paramètre utilisateur), voir section 2.7.
4. afficher graphiquement le meilleur arbre [3] pour chaque mesure, et permettre, si l'utilisateur le demande, l'affichage, pour une mesure donnée, de la liste des k-meilleurs arbres précédemment calculés (dont il pourra en choisir certains comme exemples, contrexemples ou contre-exemples absolus).

Dans la pratique :

Initialement Les listes d'exemples sont vides (à moins qu'une base de donnée préalable soit sélectionnée par défaut), par conséquent A reconnaît équiprobablement tous les arbres de même profondeur, l'algorithme ne propose pour chaque mesure que les arbres dont la distance est la plus proche de l'entrée.

À un instant t, où l'utilisateur fournit un nouvel exemple L'algorithme doit mettre à jour A, puisque la liste d'exemple a changé, il met donc à jour sa quantification.

2.5 Construction de l'automate

Dans cette session, nous nous intéresserons à l'élaboration de l'automate dont le rôle est d'évaluer les arbres candidats et de leur attribuer une note $p(t)$ correspondant à la probabilité qu'ils ont de satisfaire l'utilisateur dans l'absolu, c'est- à-dire, sans prendre en compte la piste d'entrée.

Pour ce faire, nous nous sommes inspirés de travaux sur l'apprentissage de langage d'arbres par exemples positifs/négatifs, (approches dites *RPNI*, voir références [10, 16]), que nous nous sommes efforcés de généraliser à des langages probabilistes. En effet, utiliser ces travaux tels quels ne nous est pas apparu pertinent pour répondre à notre problématique, car ceux-ci ont été développés pour répondre à des problèmes d'apprentissage sur de grosses bases de données, afin d'apprendre un langage assez précis, or, dans notre cas, le nombre d'exemples est limité et le langage recherché est moins précis : un arbre qui n'est pas un bon exemple à moment donné, peut très bien s'avérer satisfaisant dans un autre contexte (une autre mesure). C'est d'ailleurs cette réflexion qui nous a mené à considérer deux catégories de contrexemples, et à opter pour un langage probabiliste au lieu d'un langage absolu.

Le principe des approches de type RPNI est de commencer par compiler un automate qui reconnaisse uniquement les exemples, puis de faire grossir le langage de

3. C'est un abus de langage, dans la pratique, ce sont des partitions qui sont affichées à l'utilisateur, qui elles sont équivalentes à des arbres.

l'automate en fusionnant tour à tour certains de ses états, avec comme condition limite que les contrexemples restent hors du langage obtenu.

Nous nous sommes également intéressés à d'autres techniques d'apprentissage d'automates d'arbres probabilistes, comme les automates dits *k*-testables ou les techniques d'apprentissage à la limite basée sur la représentation de langages d'arbres stochastiques par des formes linéaires sur un espace vectoriel sur $\mathbb{R}$. Ce dernier type d'approche [7], bien que très élégant d'un point de vue théorique, a l'inconvénient de nécessité un grand nombre d'exemples (plusieurs milliers) pour donner des résultats intéressants. Cette contrainte est donc incompatible avec nos objectifs. Les techniques d'inférence d'automates *k-testables* [17], visent à apprendre directement des langages probabilistes d'arbres à l'aide de statistiques faîtes sur les propriétés locales des arbres (plus exactement sur leur sous-arbres de profondeur $\leq k$). Une telle approche peut-être intéressante pour traiter des représentation arborescentes de données musicales symbolique (voir [2]), néanmoins, cette méthode nous est apparu inadaptée à notre problème de quantification pour que ces statistiques soient porteuses de sens, car sur un nombre trop faible d'exemple, la plupart des arbres seront rejetés (c'est à dire évalués avec une probabilité 0). Pour éviter ce problème, des heuristiques de *lissage* (spécifique au problème de la recherche de similarité) sont appliquées dans [2].

2.5.1 Procédé général

Voici donc le procédé général de construction de notre automate, inspiré, de méthodes de type RPNI, généralisée à des automates probabilistes :

Rappelons que nous disposons :

— Des trois listes d'exemples E^+, E^- et E^{--}

— d'une profondeur maximale p et d'un ensemble Ar d'arités autorisées, qui définit un langage global fini, que nous nommerons G.

Étape 1. La première étape de notre algorithme consiste à générer un automate déterministe qui reconnaît exactement le langage G, on appellera désormais cet automate *Guide*.

Pour ce faire, on peut déjà créer un automate déterministe qui ne reconnait que les arbres d'arités dans Ar : il suffit de faire deux états q_1 et q_2, ou seul q_1 est final, et d'ordonner aux transitions de renvoyer q_2 si et seulement si l'un des fils est déjà dans q_2 ou bien l'arité n'est pas dans Ar.

On peut aussi créer un automates à $p+1$ états qui compte jusqu'à p la profondeur de l'arbre (avec les transitions qui vont bien), et au delà retourne un état non acceptant.

Par intersection, grâce au théorème 1, on peut conclure.

La complexité est donc linéaire en p.

Étape 2. Dans un second temps, nous mettons à jour le *Guide*, en lui retirant les contrexemples absolus. Pour cela, on peut par exemple, créer l'automate qui reconnaît exactement le langage fini E^{--} (voir théorème 2), puis intersecter son complémentaire avec *Guide*. La complexité est donc quadratique en le nombre de noeuds dans la forêt E^{--} et la taille de *Guide*.

Étape 3. On crée l'automate déterministe A qui reconnaît exactement E^+. On le transforme en automate probabiliste (remarque 4), puis on fusionne (grâce à la fonction fusion qui sera détaillée ci-après) tous les couples d'états (q_1, q_2) possibles [4] tant que les deux conditions suivantes sont réunies :

1. $\{t|A(t) \neq 0\}$ est inclus dans le *Guide* ;
2. pour tout t^+ dans E^+, et tout t^- dans E^-, $A(t^+) > A(t^-)$.

La complexité du test 1 est quadratique en le nombre d'états de A et *Guide*. En effet d'après la remarque 3, on peut considérer l'automate qui reconnaît le langage $L = \{t \mid A(t) \neq 0\}$, et le test $L \subseteq Guide$ est équivalent au test $L \cap \overline{Guide} \neq 0$. Or, le test de vacuité d'un langage reconnu par un automate est linéaire en la taille de l'automate [5], et *Guide* étant déterministe, le calcul de $L \cap \overline{Guide}$ est quadratique d'après les propriétés 1 et 3.

La complexité de 2 est de $C_{n,q,a} \times |E^+| \times |E^-|$ où la valeur $C_{n,q,a}$ correspond au temps de l'évaluation d'un arbre par un automate probabiliste qui sera justifiée dans la section 2.5.3.

Le nombre maximal de fusions possibles est quadratique en le nombre d'état initial dans l'automate.

2.5.2 Description de la fonction fusion

La fonction de fusion d'états dans un automate probabiliste est une fonction qui prend en entrée un automate probabiliste et deux états de ce dernier et qui renvoie un automate probabiliste pour lequel ces deux états ne font plus qu'un. En voici le principe :

Supposons que l'on veuille fusionner dans un automate probabiliste $A = (Q, \Sigma, \delta, F)$ deux états p_1 et p_2 de Q en un nouvel état p, il nous reste à déterminer le poids des transitions, ainsi que la fonction finale.

Déterminer le poids des transitions. On cherche à déterminer le poids d'une transition de la forme $f(q_1, ..., q_n) \xrightarrow{\omega} q$

Cas 1 : Aucun des q_i, ni q n'est p. Dans ce cas, cette transition avait déjà un poids dans A, on ne le change pas.

Cas 2 : Certains des q_i sont p, mais $q \neq p$. Dans ce cas, soit $I = \{i \in [1, n] \mid q_i = p\}$, et soit 2^I l'ensemble des fonctions g de I dans l'ensemble $\{p, q\}$ prolongées à $[1, n]$ via $g(i) = q_i$ si $i \notin I$. Le poids $\omega(f(q_1, ...q_n) \to q)$ vaut alors $\sum_{g \in 2^I} \omega(f(g(1), ..., g(n)) \to q)$

Cas 3 : Aucun des q_i n'est p, mais $q = p$. Le poids $\omega(f(q_1, ...q_n) \to p)$ vaut alors $\frac{1}{2} \cdot \big(\omega(f(q_1, ...q_n) \to p_1) + \omega(f(q_1, ...q_n) \to p_2)\big)$.

4. On peut s'intéresser à l'ordre dans lequel on effectue ces fusions, qui modifiera l'aspect de l'automate final, mais cela n'a que peu d'importance dans la mesure où, quel que soit l'ordre retenu, l'automate obtenu, si différent soit-il, répondra à nos attentes. Pour plus de détail dans le cas non pondéré voir les articles [5, 16].

5. voir [6]

Cas 4 : Certains des q_i sont p, et $q = p$. On définit alors I de même que dans le cas 2, et on a alors

$$\omega(f(q_1,...q_n) \to p) = \frac{1}{2} \cdot \Big(\sum_{g \in 2^I} \omega(f(g(1),...,g(n)) \to p_1) + \sum_{g \in 2^I} \omega(f(g(1),...,g(n)) \to p_2) \Big)$$

Déterminer la fonction finale. Pour les états q autres que p, on garde la même valeur pour $F(q)$, et pour p, on pose $F(p) = F(p_1) + F(p_2)$.

Le coût du calcul du poids d'une transition est donc dans le pire des cas de $2^{arite+1}$, or l'arité ne dépasse jamais l'ordre de la dizaine, par conséquent, on peut considérer le calcul du poids d'une transition comme étant constant. D'autre part, dans notre algorithme, on ne calcule pas le poids de toutes les nouvelles transitions à chaque fusion (ce qui ferait exploser la complexité), mais seulement à chaque fois qu'on en a besoin, c'est-à-dire, lorsque qu'elle est appelée au sein d'un calcul sur un arbre. Au final, on peut donc considérer que la fonction de fusion s'exécute en un temps constant.

Théorème 4 (Caractère probabiliste d'un automate après fusion)**.** *Soit $A = (Q, \Sigma, \delta, F)$ un automate d'arbres probabiliste, et p_1 et p_2, deux de ses états. Alors l'automate $A' = (Q', \Sigma', \delta', F')$ obtenu après fusion de ces deux états en un nouvel état p est toujours un automate probabiliste.*

Démonstration. δ' et F' restent clairement à valeur dans $[0,1]$, par construction.
$\sum_{q \in Q'} F(q) = \sum_{q \in Q} F(q) - F(p_1) - F(p_2) + F(p) = \sum_{q \in Q} F(q) = 1$
Pour tout $q \in Q' \setminus \{p\}$,

$$\begin{aligned}
\sum_{r \in \Delta'_q} \omega(r) &= \sum_{n=0}^{max\{Ar\}} \sum_{f \in \Sigma} \sum_{q_1,...,q_n \in Q'} \omega(f(q_1,...q_n) \to q) \\
&= \sum_{n=0}^{max\{Ar\}} \sum_{f \in \Sigma} \sum_{q_1,...,q_n \in Q'} \sum_{I \subset [1,n]} \sum_{g \in 2^I} \omega(f(g(1),...,g(n)) \to q) \\
&= \sum_{n=0}^{max\{Ar\}} \sum_{f \in \Sigma} \sum_{q_1,...,q_n \in Q} \omega(f(q_1,...q_n) \to q) \\
&= \sum_{r \in \Delta_q} \omega(r) \\
&= 1.
\end{aligned}$$

Enfin, pour p :

$$\begin{aligned}
\sum_{r\in\Delta'_p}\omega(r) &= \sum_{n=0}^{max\{Ar\}}\sum_{f\in\Sigma}\sum_{q_1,...,q_n\in Q'}\omega(f(q_1,...q_n)\to p)\\
&= \sum_{n=0}^{max\{Ar\}}\sum_{f\in\Sigma}\sum_{q_1,...,q_n\in Q'}\sum_{I\subset[1,n]}\frac{1}{2}\cdot(\sum_{g\in 2^I}\omega(f(g(1),...,g(n))\to p_1)+\sum_{g\in 2^I}\omega(f(g(1),...,g(n))\to p_2))\\
&= \frac{1}{2}\cdot\sum_{n=0}^{max\{Ar\}}\sum_{f\in\Sigma}\sum_{q_1,...,q_n\in Q}\omega(f(q_1,...q_n)\to p_1)+\frac{1}{2}\cdot\sum_{n=0}^{max\{Ar\}}\sum_{f\in\Sigma}\sum_{q_1,...,q_n\in Q}\omega(f(q_1,...q_n)\to p_2)\\
&= \frac{1}{2}\cdot\sum_{r\in\Delta_{p_1}}\omega(r)+\frac{1}{2}\cdot\sum_{r\in\Delta_{p_2}}\omega(p_2)\\
&= \frac{1}{2}+\frac{1}{2}\\
&= 1.
\end{aligned}$$

□

2.5.3 Méthode d'évaluation d'un automate probabiliste

Comme nous l'avons vu dans la partie 2.1, il faut, en théorie, pour calculer le poids d'un arbre t par un automate pondéré A, sommer chacun des calculs existants sur t : $\omega(A,t)=\sum_r\omega(A,t,r)$. Étant donné que l'on doit faire appel à cette évaluation, pour chaque couple exemple/contrexemple, et avant chaque tentative de fusion, énumérer tous les calculs et les sommer chacun leur tour, serait très couteux en terme de complexité temporelle.

Néanmoins, il est possible d'évaluer le poids d'un arbre de façon plus efficace, en calculant récursivement le poids de chacun de ses sous-arbres. Pour cela, on a besoin d'introduire la notion de quasi-poid, qui est le poids d'un calcul avant l'application de la fonction finale à la racine.

Notation 6. *Soit t un arbre de rang borné et $A=(Q,\Sigma,\delta,F)$ un automate d'arbres pondéré, et r un calcul de A sur t, on note alors $\overline{\omega}(A,t,r)=\omega(A,t,r,\epsilon)=\frac{\omega(A,t,r)}{F(r(\epsilon))}$, qu'on appelle quasi-poids du calcul r de A en t.*

De plus, si pour tout état q on appelle R_q l'ensemble des calculs r menant à l'état q i.e. tels que $r(\epsilon)=q$, alors on note $\overline{\omega_q}(A,t)=\sum_{r\in R_q}\overline{\omega}(A,t,r)$, qu'on appelle quasi-poids de t en q. On a alors $\omega(A,t)=\sum_{q\in Q}F(q)\cdot\overline{\omega_q}(A,t)$.

Démonstration. En effet, on a $\omega(A,t)=\sum_r\omega(A,t,r)=\sum_{q\in Q}\sum_{r\in R_q}\overline{\omega}(A,t,r)\cdot F(r(\epsilon))=\sum_{q\in Q}\overline{\omega_q}(A,t)\cdot F(q)$ □

On peut alors énoncer le théorème qui nous permettra de calculer le poids d'un automate pondéré au vol :

Théorème 5. *Soit $t = f(t_1, ..., t_n)$ un Σ-arbre de rang borné, où $f \in \Sigma$ est l'étiquette de sa racine, et t_1,...,t_n sont ses différents fils, alors on a :*

$$\overline{\omega_q}(A,t) = \sum_{(q_1,...,q_n)\in Q^n} \delta(f(q_1,...,q_n)) \to q) \times \overline{\omega_{q_1}}(A,t_1) \times ... \times \overline{\omega_{q_n}}(A,t_n)$$

Démonstration. En effet, on a :

$$\begin{aligned}
&\sum_{(q_1,...,q_n)\in Q^n} \delta(f(q_1,...,q_n)) \to q) \times \overline{\omega_{q_1}}(A,t_1) \times ... \times \overline{\omega_{q_n}}(A,t_n) \\
&= \sum_{(q_1,...,q_n)\in Q^n} \delta(f(q_1,...,q_n)) \to q) \times \sum_{r_1\in R_{q_1}} \overline{\omega}(A,t_1,r_1) \times ... \times \sum_{r_n\in R_{q_n}} \overline{\omega}(A,t_n,r_n) \\
&= \sum_{(q_1,...,q_n)\in Q^n} \sum_{r\in R_q} \delta(f(q_1,...,q_n)) \to q) \cdot \prod_{i=1}^{n} \omega(A,t,r,i) \text{ (avec la notation de la définition 18)} \\
&= \sum_{r\in R_q} \omega(A,t,r,\epsilon) \\
&= \sum_{r\in R_q} \overline{\omega}(A,t,r) \\
&= \overline{\omega_q}(A,t).
\end{aligned}$$

□

On peut donc évaluer un arbre t en calculant le vecteur $(\overline{\omega_q}(A,t))_{q\in Q}$ grâce au procédé récursif suivant :

— si t est réduit à une feuille f, $\forall q \in Q, \overline{\omega_q}(A,t) = \delta(f,q)$

— sinon, t est de la forme $f(t_1,...,t_n)$, et alors :

1. pour tout t_i, calculer $\overline{\omega_q}(A,t_i)$
2. pour tout $q \in Q$, on a, grâce au théorème précédent :

$$\overline{\omega_q}(A,t) = \sum_{(q_1,...,q_n)\in Q^n} \delta(f(q_1,...,q_n)) \to q) \times \overline{\omega_{q_1}}(A,t_1) \times ... \times \overline{\omega_{q_n}}(A,t_n).$$

Puis, pour obtenir $\omega(A,t)$, il ne nous reste plus qu'à effectuer $\sum_{q\in Q} F(q) \cdot \overline{\omega_q}(A,t)$.

Si l'on note $C_{n,q,a}$ le nombre d'opérations effectuées, dans le pire des cas, pour évaluer le vecteur $(\overline{\omega_q}(A,t))_{q\in Q}$ d'un arbre de profondeur n, de nombre d'états q et d'arité maximale a, avec ce procédé, on obtient les formules suivantes : $C_{0,q,a} = 0$ et $C_{n+1,q,a} = a \times C_{n+1,q,a} + q^{a+1}$.

Un simple calcul nous donne donc que $C_{n,q,a} = \frac{a^n-1}{a-1} \cdot q^{a+1}$.

L'arité et la profondeur étant toujours limitées (de l'ordre de la dizaine au grand maximum), on voit donc que nous avons une complexité polynomiale, de degré assez faible en le nombre d'états.

2.6 Évaluation en fonction de la distance à l'entrée

Comme nous l'avons décrit dans la section 2.5.1, en plus de l'automate A, notre algorithme est pourvu d'une fonction B qui renvoie une estimation de la probabilité que la sortie soit celle qu'on attend en fonction de sa distance à l'entrée et sans prendre en considération les goûts de l'utilisateur.

Pour ce faire, on pourrait être tenté, dans un premier temps, de simplement normaliser (ramener continûment entre 0 et 1) une fonction de la forme $\frac{1}{d(s,t)}$, où d serait une mesure de la distance entre séquence d'entrée s et un arbre de sortie t, qui peut être plus où moins canonique. À titre d'exemple, dans OMKant, voici les diverses distances qui sont utilisées (l'attaque sur la grille correspond à la sortie) :

— somme des rapports cubiques (attaque donnée - attaque sur la grille)
— distance euclidienne entre attaque donnée et attaque sur la grille

Néanmoins, les travaux d'Ali Taylan Cemgil (voir [4]) ont montré que d'expérience, cette valeur ne dépendait pas uniquement de la distance entrée/sortie, mais que la façon dont les notes sont réparties dans la mesure avait aussi son importance. C'est pour cela qu'il suggère d'avoir recours à une matrice de covariance de la forme

$\Sigma(t) = (\rho_{n,m}) \in K^2$ où K est le nombre de notes dans la mesure, et où :

$$\rho_{n,m} = \eta\, e^{-\frac{\lambda^2}{2}\cdot(t_m - t_n)^2}$$

où :
— les t_n sont les instants d'impact des notes dans l'arbre t quantifié,
— η et λ sont des paramètres d'ajustement de la corrélation entre les entrées.

On pose alors $B(s,t) = \frac{1}{(2\pi)^{K/2}\ |\Sigma|^{1/2}} \cdot e^{-\frac{1}{2}\cdot(s-t)^\top\ \Sigma(t)\ (s-t)}$.

2.7 Le problème de l'énumération

La dernière fonction de notre algorithme est celle de l'énumération des k-meilleurs arbres candidats pour la quantification, c'est-à-dire qui maximisent le produit $A(t) \times B(t)$, où k est un paramètre utilisateur.

La méthode que nous retenons pour l'instant est une approche plutôt heuristique, néanmoins, nous disposons de deux autres pistes, sur lesquelles il nous faut encore réfléchir, et dont je vous présenterai les grandes lignes.

Méthode heuristique. C'est une méthode de type force brute, elle consiste à effectuer la mesure $A(t) \times B(t)$, sur toute une liste d'arbres candidats, obtenus, par exemples grâce à d'autres algorithmes de quantifications (par exemples ceux disponibles dans OpenMusic, comme OMKant, en faisant varier les paramètres utilisateurs. Cette méthode est néanmoins coûteuse, car il faut effectuer l'opération $A(t) \times$

$B(t)$ sur chacun des candidats et ensuite trier les k meilleurs candidats, ce qui empêche dans la pratique d'effectuer notre évaluation sur un trop grand nombre de candidats. Par conséquent, notre algorithme se retrouve obligé d'effectuer sa recherche du meilleur candidat dans un ensemble restreint d'arbres, au risque de passer à côté d'un candidat optimal qui n'aurait pas été dans cet ensemble.

Première piste. Une autre méthode d'évaluation, sur laquelle il nous reste à réfléchir, est la suivante :

1. énumérer tous les arbres dont la distance à la mesure d'entrée est inférieure à une valeur x, (fixée ou déterminée par l'utilisateur) ;
2. effectuer une recherche des k meilleurs arbres de cette énumération seulement pour la valeur $A(t)$. Pour cela, on n'est plus obligé d'utiliser la force brute, mais on peut passer par un algorithme efficace comme celui implanté dans la librairie Tiburon [14], qui adapte aux automates d'arbres pondéré l'algorithme de calcul des k-meilleurs chemins de Huang and Chiang [12] mentionné plus bas.

La principale difficulté réside dans l'étape 1 : il nous faut trouver un moyen astucieux d'énumérer tous les arbres dont la distance à la mesure d'entrée est inférieure à x, d'une façon efficace (la méthode naïve ayant une complexité exponentielle en le nombre de notes).

Cette méthode possède donc l'avantage de pouvoir comparer un nombre bien supérieur d'arbres candidats, néanmoins, elle pose la question de déterminer la valeur de x, et n'effectue plus le produit $A(t) \times B(t)$, mais seulement $A(t)$ sur les arbres dont la distance est estimée suffisante.

Seconde piste. Nous suivons également une autre piste, qui, si elle s'avérait fructueuse, s'affranchirait à la fois des problèmes posés par les deux précédentes méthodes, à savoir, qu'elle permettrait, d'une part, de comparer un très grand nombre de candidats, et qu'elle ne singulariserait d'autre part, ni $A(t)$ ni $B(t)$ dans la réalisation de son choix. Qui plus est, elle permettrait également d'étendre l'apprentissage à la distance (et non plus seulement aux arbres dans l'absolu).

Lors d'un pré-traitement, l'automate A et une représentation de la fonction B_s sont traduits dans un hypergraphe pondéré H de sorte que étant donné un arbre t, l'évaluation de t sur H_s retourne $A(t) \times B_s(t)$. Un hypergraphe pondéré est donné par un ensemble de sommets et un ensemble d'arêtes qui envoient un n-uplet de sommets sources vers un sommet destination, chaque arête se voyant associer un poids qui est une fonction de $\mathbb{R}^n \to \mathbb{R}$. Les hypergraphes pondérés généralisent donc les automates d'arbres pondérés, lorsque les sommets représentent les états et les arêtes les transitions.

Après ce pré-traitement, on peut appliquer l'algorithme de recherche des k-meilleurs chemins dans un hypergraphe [12] à H pour obtenir les k meilleurs arbres. Cet algorithme améliore l'extension de Knuth [13] aux hypergraphes de l'algorithme classique de Dijkstra [8] de recherche du meilleur chemin.

3 Conclusions et perspectives

Pour conclure, nous avons donc conçu, sur le papier, un procédé de quantification rythmique avec apprentissage des goûts/du style de l'utilisateur. Il reste désormais à l'implémenter, à l'intégrer à OpenMusic, puis à valider son bon fonctionnement en poursuivant nos discussions avec les compositeurs de l'IRCAM qui sont les principaux utilisateurs d'OpenMusic.

OpenMusic est principalement codé en LISP qui est un langage de programmation fonctionnelle, et je pense que ce langage serait plutôt bien adapté à l'intégration de notre quantificateur, étant donné que les techniques basées sur les arbres et les automates que nous vous avons présenté s'implémenter de façon assez naturelle dans un paradigme fonctionnel. J'ai par ailleurs moi-même testé certaines parties de notre algorithme (plus précisément l'implémentation d'automates d'arbres probabilistes, leur évaluation ainsi que la fusion d'états), en OCaml qui est un autre langage de programmation fonctionnelle, intrinsèquement assez proche du LISP.

Il nous reste encore quelques défis à relever, pour améliorer notre prototype, comme par exemple pour l'énumération (comme nous l'avons expliqué précédemment), ou bien étendre notre solution à des entrées polyphoniques, gérer le problème des petites notes (notes dont la durée n'est pas précisée) qui tiennent à coeur aux compositeurs, ou bien encore améliorer notre gestion des silences.

Un autre aspect sur lequel il peut-être intéressant de se pencher est de chercher à dynamiser l'apprentissage de notre automate. En effet, dans la section 2.5.1, nous avons décrit une méthode pour pouvoir calculer l'automate A de notre problème, à chaque fois que l'utilisateur met à jour une de ses liste d'exemples. Le seul ennui, avec cette méthode, c'est qu'il faut recréer, à chaque fois notre automate à partir de rien, alors que la plupart des exemples sont les mêmes que lors du précédent calcul, on peut donc chercher à s'inspirer de travaux sur l'*apprentissage actif* de langage d'arbres comme ceux les *MAT leaners* [9].

Ce stage a donc été l'occasion pour moi de participer à l'élaboration d'un projet de recherche depuis ses prémisses, c'est-à-dire depuis la recherche même de la définition de nos objectifs, jusqu'à la mise en forme du prototype que nous venons de vous présenter, avec tout ce que cela comporte, à savoir des moments de questionnement, d'interaction avec des compositeurs comme Karim Haddad, Mikhail Malt, ou de rencontre avec des chercheurs spécialisés en théorie des langages d'arbres comme Edouard Gilbert travaillant à L'INRIA à Lille, ou José Luis Iñesta de l'université d'Alicante qui, lui aussi travaille sur des questions d'apprentissage interactif par l'exemple dans des domaines musicaux.

Sur un plan plus technique, stage m'aura permis de m'initier à la théorie des langages, et à des techniques sur les automates d'arbres, et m'aura fait découvrir plus

en détail le monde de la recherche musicale, ses grand projets ainsi que ses grandes problématiques.

Bibliographie

[1] C. Agon, G. Assayag, J. Fineberg, and C. Rueda. Kant : a critique of pure quantification. In *International Computer Music Conference Proceedings (ICMC)*, pages 52–59, 1994.

[2] J. F. Bernabeu, J. Calera-Rubio, J. M. Iñesta, and D. Rizo. Melodic identification using probabilistic tree automata. *Journal of New Music Research*, 40(2) :93–103, june 2011.

[3] A. T. Cemgil. *Bayesian Music Transcription*. PhD thesis, Radboud Universiteit Nijmegen, 2004.

[4] A. T. Cemgil, B. Kappen, and P. Desain. Rhythm quantization for transcription. *Computer Music Journal*, 24(2) :60–76, July 2000.

[5] J. Champavère, R. Gilleron, A. Lemay, and J. Niehren. Schema-guided induction of monadic queries. In *Proceedings of the 9th international colloquium on Grammatical Inference : Algorithms and Applications*, ICGI '08, pages 15–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. `http://tata.gforge.inria.fr`, 2007.

[7] F. Denis, É. Gilbert, A. Habrard, F. Ouardi, and M. Tommasi. Relevant representations for the inference of rational stochastic tree languages. In A. Clark, F. Coste, and L. Miclet, editors, *Proceedings 9th International Colloquium Grammatical Inference : Algorithms and Applications (ICGI)*, volume 5278 of *Lecture Notes in Computer Science*, pages 57–70. Springer, 2008.

[8] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1) :269–271, 1959.

[9] F. Drewes, J. Högberg, and A. Maletti. Mat learners for tree series : an abstract data type and two realizations. *Acta Informatica*, 48(3) :165–189, 2011.

[10] P. García and J. Oncina. Inference of recognizable tree sets. Technical Report DSIC-II/47/93, Universidad de Alicante, 1993.

[11] K. Haddad. Fragments de recherche et d'expérimentation : Eléments de réflexions autour de l'écriture rythmique d'Emmanuel Nunes. Présentation à un séminaire Mamux à l'Ircam, nov 2012.

[12] L. Huang and D. Chiang. Better k-best parsing. In *Proceedings of the Ninth International Workshop on Parsing Technology*, Parsing '05, pages 53–64, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[13] D. Knuth. A generalization of dijkstra's algorithm. *Inform. Process. Lett.*, 6(1), 1977.

[14] J. May and K. Knight. Tiburon : A weighted tree automata toolkit. In O. Ibarra and H.-C. Yen, editors, *Implementation and Application of Automata*, volume 4094 of *Lecture Notes in Computer Science*, pages 102–113. Springer Berlin Heidelberg, 2006.

[15] B. Meudic. *Détermination automatique de la pulsation, de la métrique et des motifs musicaux dans des interprétations à tempo variable d'œuvres polyphoniques*. PhD thesis, Ircam, 2004.

[16] J. Niehren, J. Champavère, R. Gilleron, and A. Lemay. Query Induction with Schema-Guided Pruning Strategies. *Journal of Machine Learning Research*, 14 :927–964, Apr. 2013.

[17] J. R. Rico-Juan, J. Calera-Rubio, and R. C. Carrasco. Probabilistic k-testable tree languages. In *Proceedings of the 5th International Colloquium on Grammatical Inference : Algorithms and Applications*, ICGI '00, pages 221–228, London, UK, UK, 2000. Springer-Verlag.

[18] D. Rizo. *Symbolic music comparison with tree data structures*. PhD thesis, Universidad de Alicante, November 2010.