Chapitre 3

Automates étendus probabilistes

Nous avons vu au chapitre précédent plusieurs algorithmes de génération de musique. Toutefois, aucun d'entre eux ne s'est avéré totalement satisfaisant. Certains sont trop difficiles à paramétrer, d'autres n'offrent pas assez de flexibilité à l'utilisateur, d'autres encore ne permettent pas de créer assez facilement des variations sur un même thème, et enfin d'autres ne sont pas adaptés à l'utilisation en temps réel. C'est afin de satisfaire tous ces critères que nous introduisons un nouvel algorithme de génération de musique. Dans la première partie de ce chapitre, nous présenterons la description formelle de cet algorithme. Ensuite, nous présenterons une implémentation de l'algorithme adaptée pour le jeu vidéo. Enfin, nous introduirons un exemple d'application que nous reprendrons dans les chapitres de mise en œuvre.

3.1 Retour sur les automates étendus

Au chapitre 2, nous avons souligné un problème existant avec le modèle original d'automates étendus tel que défini par *Cheng et Krishnakumar* [7]. Ce problème provient du fait qu'à moins de supposer que pour toute paire $\langle q, a \rangle \in S \times \Sigma$, toutes les transitions sortant de l'état q après lecture du symbole a portent des fonctions d'activation $f \in C$ mutuellement exclusives, les automates étendus sont non-déterministes. Ce problème a aussi été constaté par *Krishnakumar* [36] dans un article subséquent dans lequel il indique que les automates étendus sont non-déterministes.

Dans ce mémoire toutefois, nous préférerons une solution différente à ce problème, inspirée par *Huang et Lo* [30], qui consiste à déterminiser les automates étendus en introduisant

une valeur de priorité dans chaque fonction d'activation C_i, telles que définies au chapitre 2. Pour ce faire, nous redéfinirons celles-ci par $C_i : V \rightarrow \{0, p\}$ où $p \in \mathbb{N}^*$ est la valeur de priorité de la condition. Ceci signifie que chaque fonction C_i est une fonction booléenne où 0 représente faux et p représente vrai. Ainsi, le fonctionnement d'un automate étendu avec priorités se comprend de cette manière :

Soit q, r et s des états de l'automate, a un symbole appartenant à l'alphabet d'entrée Σ, $f, g \in C$ des fonctions d'activation, et $u, v \in \phi$ des fonctions de mise à jour (voir chapitre 2). Ainsi, si $\langle q, \vec{x} \rangle$ est la configuration courante de l'automate et que $\delta(q, f, a) = \langle r, u \rangle$ et $\delta(q, g, a) = \langle s, v \rangle$ sont deux transitions possibles considérant que le prochain symbole sur le ruban d'entrée est a, que $f(\vec{x}) \neq 0$, et que $g(\vec{x}) \neq 0$, alors l'automate passera dans la configuration $\langle r, u(\vec{x}) \rangle$ si et seulement si $f(\vec{x}) > g(\vec{x})$, sinon il passera dans la configuration $\langle s, v(\vec{x}) \rangle$. On peut définir $f(\vec{x}) = g(\vec{x}) \neq 0$ comme condition d'erreur afin d'éviter de simplement déplacer le problème du non-déterminisme.

Cette nouvelle définition permet de définir plus facilement des automates étendus avec un grand nombre de registres et un grand nombre de transitions sortant du même état.

3.2 Définition

Nous introduisons ici un nouveau type de transducteurs que nous appelons automates étendus probabilistes. Ils combinent la flexibilité des automates étendus et la diversité engendrée par les automates finis probabilistes, tels que vus au chapitre 2. Afin de simplifier la définition d'un automate étendu probabiliste, nous allons supposer que l'état de l'automate est connu à chaque instant t du processus (comme pour les automates étendus). Ainsi on peut les définir de manière analogue aux automates étendus avec priorités :

Soit S un ensemble fini d'états, $S_0 \in S$, F un sous-ensemble de S, Σ un alphabet d'entrée fini, Γ un alphabet de sortie fini, G est une fonction totale de S dans Γ, V un espace n-dimensionnel $V_1 \ldots V_n$, ϕ un ensemble de transformations ϕ_i tel que $\phi_i : V \rightarrow V$, et C un ensemble de fonctions C_i tel que $C_i : V \rightarrow mathbbN$. Soit aussi δ une fonction partielle de $S \times C$ dans $(\mathbb{R} \times \phi)^S$ tel que $(\mathbb{R} \times \phi)^S$ représente un vecteur stochastique augmenté d'une

fonction de mise à jour (appartenant à $\phi \cup Indéfini$) pour chaque état de destination possible. Formellement, un tel vecteur augmenté est défini comme un vecteur $|S|$-dimensionnel de paires ordonnées $\langle Pr(S_j) \in \mathbb{R}, \varphi_j \rangle$ tel que :

- $0 \leq Pr(S_j) \leq 1$,
- $\sum_{j=1}^{|S|} Pr(S_j) = 1$,
- $\varphi_j \in \phi$ si $0 < Pr(S_j) \leq 1$, et
- φ_j est indéfini sinon.

On dit alors que le décuplet $\langle S, S_0, F, \Sigma, \Gamma, V, C, \phi, \delta, G \rangle$ représente un automate étendu probabiliste où S_0 est état initial, F l'ensemble des états finaux, V l'ensemble de registres, C l'ensemble des fonctions d'activations, ϕ l'ensemble des fonctions de mise à jour, δ la fonction de transition, et G la fonction de sortie. On remarque immédiatement que la seule différence avec les automates étendus est que la fonction de transition prend maintenant en compte la probabilité de transition vers chaque état de l'automate. L'exemple suivant illustre le fonctionnement d'un automate étendu probabiliste.

Considérons l'automate étendu probabiliste à trois états (q_0, q_1, q_2) illustré à la figure 3.1. On remarque que l'ensemble C de ses fonctions d'activation est constitué des deux fonctions $f(\vec{x})$ et $g(\vec{x})$ et que l'ensemble ϕ de ses fonctions de mise à jour est constitué des trois fonctions $u(\vec{x}), v(\vec{x})$ et $w(\vec{x})$. De plus, l'état q_0 possède deux transitions : $\delta(q_0, f, a) = \{\langle 0, Indéfini \rangle, \langle 1, u \rangle, \langle 0, Indéfini \rangle\}$ et $\delta(q_0, g, a) = \{\langle 0.25, v \rangle, \langle 0, Indéfini \rangle, \langle 0.75, w \rangle\}$. Supposons maintenant que $\langle q_0, \vec{x} \rangle$ est la configuration initiale et que a est le prochain caractère sur le ruban d'entrée de l'automate. Dans ce cas, la configuration suivante de l'automate est déterminée comme suit :

- Si $f(\vec{x}) = g(\vec{x})$, la configuration suivante n'est pas définie,
- Si $f(\vec{x}) > g(\vec{x})$, l'automate passe dans la configuration $\langle q_1, u(\vec{x}) \rangle$ avec probabilité 100%,
- Sinon, l'automate passe dans la configuration $\langle q_0, v(\vec{x}) \rangle$ avec probabilité 25% ou dans la configuration $\langle q_2, w(\vec{x}) \rangle$ avec probabilité 75%.

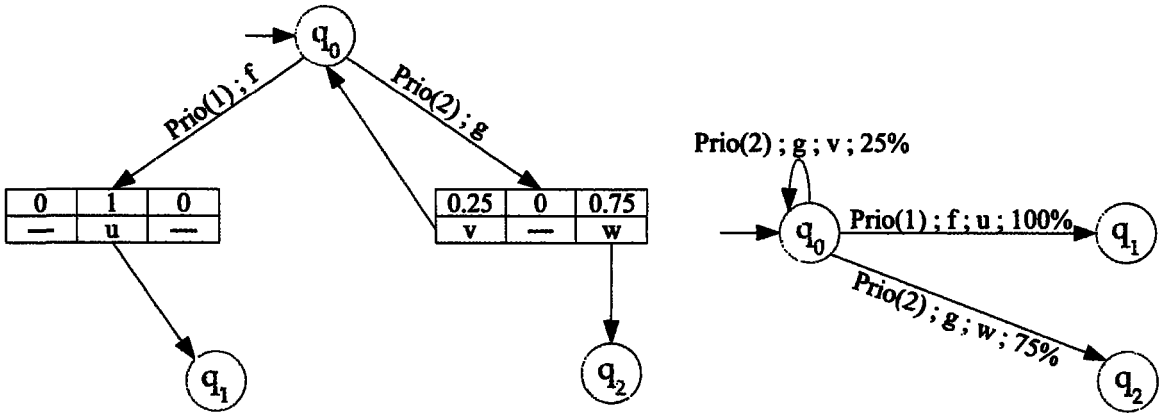

FIG. 3.1 – Deux représentations pour le modèle d'automates étendus probabilistes. Celle de gauche est conforme au modèle mais moins intuitive que celle de droite, qui sera donc utilisée pour toutes les figures subséquentes dans ce mémoire.

Nous pouvons maintenant montrer que les automates étendus probabilistes sont une généralisation appropriée des automates finis probabilistes. Il suffit de poser un automate ayant $f(\vec{x}) = 1$ comme seule fonction d'activation et $\varphi(\vec{x}) = \vec{x}$ comme seule fonction de mise à jour. Les registres sont alors inutilisés et on obtient un transducteur probabiliste standard.

De même, nous pouvons montrer que les automates étendus probabilistes sont une généralisation appropriée des automates étendus avec priorités. Il suffit de poser un automate où l'on impose que les probabilités de transition soient 1 ou 0. Il s'ensuit que pour tout état $q \in S$ et pour toute condition $f \in C$, la fonction de transition $\delta(q, f)$ est déterministe puisqu'elle retourne un vecteur stochastique qui doit donc satisfaire la propriété suivante : $\sum_{j=1}^{|S|} Pr(S_j) = 1$. On obtient la définition d'un automate étendu avec priorités standard en remarquant que la fonction de mise à jour dans le vecteur stochastique augmenté n'est définie que pour l'état ayant une probabilité de 1.

3.2.1 Conditions de bonne formation d'un automate étendu probabiliste

Le modèle d'automates étendus probabilistes comporte un problème inhérent au modèle d'automates étendus. Il s'agit du cas où, sous certaines conditions, aucune transition partant d'un état $q \in S$ ne peut être activée ; ce qui est le cas pour l'état A dans la figure 3.2 lorsque $V_1 \leq 4$. On dit d'un tel état qu'il n'est pas complet puisqu'il existe certaines configura-

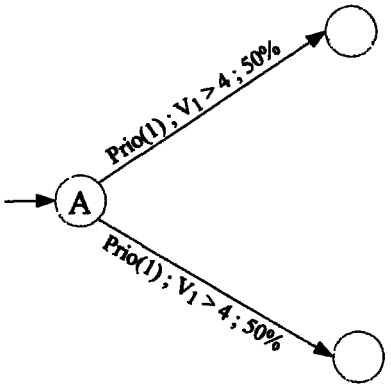

FIG. 3.2 – Aucune transition activable lorsque $V_1 \leq 4$

tions de l'automate où aucune transition sortant de l'état A n'est activable. Malheureusement, cette erreur est difficile à corriger car il n'existe pas de solution unique. De plus, il ne s'agit pas toujours d'une erreur ; certaines configurations de l'automate sont parfois impossibles et c'est pourquoi δ est une fonction partielle.

De plus, sans constituer une malformation à proprement parler, il est aussi possible de créer des transitions inactivables peu importe l'état de l'automate. Par exemple, sur la figure 3.3, on remarque que la transition A porte une contradiction logique comme fonction d'activation ($V_1 < V_1$). Quant aux transitions C et D, elles sont inactivables car elles sortent d'un état marqué comme état final (rappelons-nous que, dans notre implémentation, les états finaux amènent automatiquement l'automate à s'arrêter). Toutes ces transitions peuvent êtres supprimées sans risque de changer le comportement final de l'automate.

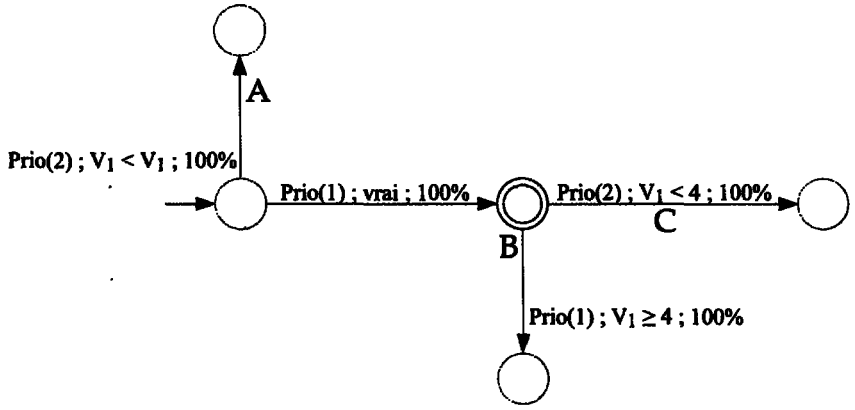

FIG. 3.3 – Transitions inactivables peu importe l'état de l'automate

3.3 Implémentation

Jusqu'ici, par souci de clarté, nous avons présenté le modèle sans faire référence aux jeux vidéo. Dans les lignes qui vont suivre, nous allons adapter celui-ci à une situation d'utilisation en temps réel, ce qui correspond au contexte des jeux vidéo. À cet effet, nous allons ajouter de nouveaux critères à ceux précédemment énumérés au chapitre 2 :

- Communication bi-directionnelle entre l'automate et un processus externe (ex. : jeu, outil de composition, etc.),
- Simplicité.

Définition 3.1 (*Simplicité*)
La *simplicité* est la qualité de ce qui est facile à comprendre ; le caractère de ce qui est facilement utilisable ou réalisable.[1] [6]

À priori, l'introduction de ces nouveaux critères peut sembler redondante puisque l'automate possède déjà un mécanisme lui permettant de communiquer avec des processus externes : ses rubans d'entrée et de sortie. Cette solution peut même sembler idéale par sa simplicité apparente. Toutefois, tel que montré par *Cheng et Krishnakumar* [7], il existe deux manières de traiter l'entrée :

1. En la consommant sur les transitions,
2. En la transférant dans les registres.[2]

Ceci est problématique puisque les deux méthodes n'offrent pas le même niveau de flexibilité. En effet, en transférant l'entrée dans un registre, la deuxième méthode nous permet d'appliquer des fonctions de mise à jour sur celle-ci. De plus, seule cette dernière méthode nous permet de garder la trace de plusieurs paramètres évoluant en parallèle dans l'environnement du jeu vidéo (le processus externe) sans ambiguïté. Remarquons toutefois qu'il nous faut pour cela multiplier les rubans d'entrées, c'est-à-dire qu'il nous faut un ruban par paramètre ou,

[1] `http://www.cnrtl.fr/portail/`

[2] Notons que cette deuxième option ne change en rien le comportement de la tête de lecture par rapport à la première option. L'article de *Cheng et Krishnakumar* [7] montre un exemple qui combine, en fait, les deux méthodes.

dans le cas limite, un ruban par registre de l'automate. Ceci vaut aussi pour les rubans de sortie si l'on veut permettre au jeu de récupérer le contenu des registres de cette manière. On en conclut donc que l'utilisation des rubans d'entrée et de sortie de l'automate pour la communication avec un processus aussi complexe qu'un jeu vidéo n'est pas une si bonne idée qu'il n'y paraît à l'origine. Nous proposerons plus loin une solution fonctionnelle à ce problème.

Mais avant, il nous faut montrer une propriété intéressante qu'acquièrent les automates étendus lorsque l'espace de calcul d'un ou de plusieurs registres est un sur-ensemble de l'alphabet d'entrée. Dans ce cas, il devient possible, en appliquant une transformation similaire à celle présentée à la figure 3.4, de remplacer toutes les transitions par des transitions spontanées. Bien sûr, cela introduit une quantité importante d'états artificiels et complexifie du même coup l'automate mais la solution que nous introduisons au prochain paragraphe sait conserver cette propriété désirable tout en corrigeant ce problème.

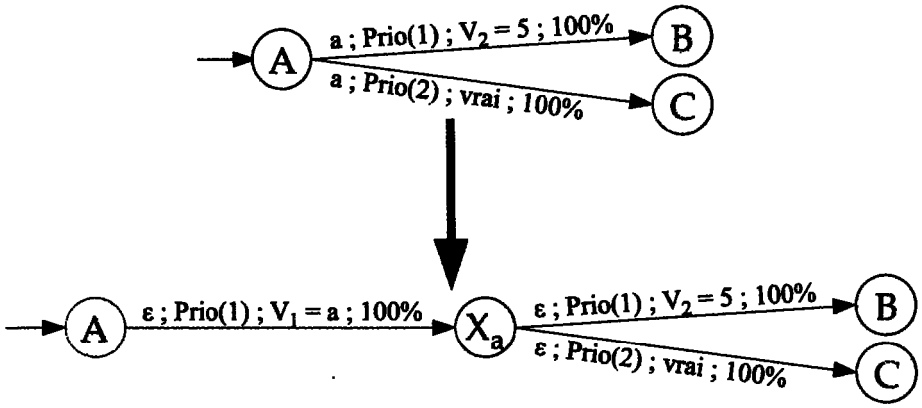

FIG. 3.4 – Suppression des transitions non-spontanées dans un automate étendu (probabiliste)

Notre solution accorde au jeu vidéo un accès direct aux registres de l'automate. Ainsi, elle s'affranchit du ruban d'entrée et des transitions non-spontanées d'une façon qui permet de réduire le nombre d'états artificiels. Enfin, bien que nous aurions aussi pu éliminer les états finaux, puisqu'il n'y a plus d'entrée à reconnaître, nous avons préféré leur accorder une nouvelle fonction : celle de points d'arrêts forcés (états forçant l'arrêt de l'automate).

Afin d'appliquer notre concept, nous avons implémenté un outil de composition et un moteur de musique (l'automate en réalité) entièrement découplés l'un de l'autre. Comme

chacun possède son état propre, il nous faut fréquemment mettre à jour chaque état à partir des informations pertinentes provenant de l'autre. Le fait d'avoir un accès direct aux registres nous a permis d'implémenter facilement ce processus grâce à deux fonctions seulement : `GetRegister` et `SetRegister` (voir chapitre 4).

Enfin, considérant que notre objectif final consiste à produire de la musique pour un jeu vidéo, il nous a fallu encoder la musique pour le ruban de sortie de l'automate. À cet effet, nous avons choisi de définir l'alphabet de sortie de l'automate (Γ) comme un sous-ensemble de $\mathbb{N}$ où chaque nombre correspond à un identificateur pour un segment musical stocké dans une structure externe. Bien entendu, ces segments musicaux devront, eux-aussi, être encodés dans un format utilisable par l'ordinateur. La section suivante compare deux formats d'encodage possible au niveau de la performance et de la flexibilité.

3.3.1 Encodage des segments musicaux

Nous présentons maintenant différentes options par rapport au format d'encodage pour les segments musicaux. Celui-ci affecte directement la performance et la flexibilité du système résultant, deux critères très importants pour l'industrie du jeu vidéo. Pour des fins de commodité, nous avons choisi de ne retenir que les deux formats d'encodage les plus populaires dans l'industrie du jeu aujourd'hui : le format MIDI[3] et le format WAV (voir annexe A pour une description du format MIDI).

L'encodage de la musique sous la forme d'une séquence d'échantillons sonores non compressés est certainement l'option la plus simple car il s'agit de la forme finale que la musique devra adopter avant d'être transmise à la carte de son. Puisque ce format représente l'information sonore à son plus bas niveau, c'est-à-dire comme une séquence d'échantillons, on peut facilement déduire qu'il s'agit du format le moins flexible. Ceci découle du fait que puisque toute l'information musicale de haut niveau est perdue, il devient difficile de modifier le flux musical de façon réaliste, même en appliquant des algorithmes de traitement de signaux complexes. En contrepartie, l'application d'effets spéciaux s'en trouve facilitée. Il existe bien

[3]De nos jours, le format MIDI est surtout populaire pour le développement de jeux pour cellulaires ainsi que comme format de travail pour la composition.

sûr des solutions partielles à ce problème comme l'enregistrement multi-pistes, soit la technique d'enregistrer chaque instrument de musique dans un fichier séparé et de reporter le mixage de ceux-ci au moment de l'exécution du jeu vidéo. De cette façon, le moteur de son du jeu vidéo peut appliquer des effets spéciaux sur chaque instrument indépendamment des autres et on récupère une partie de la flexibilité perdue. Certaines modifications, comme les changements de tempo ou de notes, demeurent difficiles.

Une séquence au format MIDI est formée d'une série de commandes à transmettre à un synthétiseur musical. Ces commandes sont fort simples et espacées dans le temps (ex. : activer note, désactiver note, changer instrument, etc). Ceci fait en sorte que ces séquences sont faciles à modifier en temps réel puisqu'il suffit de modifier le flux de commandes. Toutefois, ceci implique aussi qu'il y a une étape supplémentaire entre la lecture de la séquence MIDI et le mixage de la séquence avec les autres effets sonores utilisés par le jeu, soit la conversion, par un synthétiseur, de la séquence MIDI en séquence d'échantillons WAV. De ce fait, le problème du choix du type de synthétiseur s'impose. Ce problème est important car il affecte directement la performance du logiciel et la qualité du fichier WAV produit. De façon réaliste, deux options s'offrent à nous : utiliser le synthétiseur interne de l'ordinateur ou utiliser un synthétiseur logiciel.

La première option est la plus performante, plus encore que l'utilisation de musique échantillonnée. Ceci est dû au fait que le processeur central de l'ordinateur n'intervient pas dans cette option, même pas au niveau du mixage avec les effets sonores car le synthétiseur interne possède un canal dédié sur la carte de son, c'est-à-dire que la musique produite par celui-ci est jouée simultanément avec les effets sonores sans besoin d'être mélangée. Toutefois, la qualité des synthétiseurs internes peut varier d'un ordinateur à l'autre. De ce fait, il n'existe aucune garantie qu'une même séquence MIDI sera reproduite de la même façon sur différents ordinateurs, ce qui fait que cette solution est plus ou moins appropriée à notre contexte.

L'autre option, l'utilisation d'un synthétiseur logiciel, est plus intéressante pour les jeux car elle garantit une reproduction toujours fidèle de la musique d'un ordinateur à l'autre. De plus, elle offre beaucoup plus de flexibilité que la précédente puisque, dans ce cas, nous

sommes en contrôle de la manière dont le synthétiseur procède pour générer les instruments qui serviront à transformer la séquence MIDI en signal échantillonné. À cette fin, nous avons choisi d'utiliser Fluidsynth[4] [26], un synthétiseur implémentant la technique la plus utilisée de nos jours, soit la synthèse par table d'ondes qui consiste en la modulation et l'interpolation d'échantillons d'instruments réels échantillonnés afin d'obtenir les tonalités désirées [12,26]. Ceci implique que les banques d'instruments d'un tel synthétiseur sont entièrement personnalisables par le compositeur qui est alors libre de fournir ses propres banques d'échantillons. Pour ce faire, nous avons utilisé le format SoundFont (nommé par analogie aux polices de caractères en typographie). Il s'agit d'un format de fichier pouvant contenir jusqu'à plusieurs gigaoctets d'échantillons et pouvant décrire jusqu'à 16384 instruments en superposant ces échantillons et en leur appliquant différents paramètres (ex. : l'enveloppe de volume, l'enveloppe de modulation, la quantité de chorus, la quantité de réverbération, etc). Ces instruments pourront ensuite être utilisés par le synthétiseur pour effectuer le rendu des fichiers MIDI. Remarquons que les banques d'instruments personnalisées d'un compositeur ne sont pas obligées de respecter le standard « General MIDI » qui spécifie un jeu de 128 instruments (ou 256 dans le cas du standard « General MIDI 2 ») que tous les synthétiseurs doivent fournir afin que toutes les séquences MIDI puissent être reproduites de manière prévisible sur tous les synthétiseurs[5]. Remarquons toutefois que, puisque la transformation en temps réel du flux MIDI en signal échantillonné est une opération coûteuse en temps de processeur, il s'agit de l'option la moins performante. Elle n'est donc pas adaptée à l'utilisation dans un jeu vidéo. Toutefois, sa flexibilité la rend idéale pour l'utilisation dans un outil de composition. C'est ce que nous verrons au chapitre 4.

3.4 Exemples

Afin d'illustrer notre modèle, cette section présente quelque cas d'utilisation de musique interactive et/ou algorithmique tirés de jeux réels et explique comment on peut modéliser chacun d'eux à l'aide d'automates étendus probabilistes. Afin d'augmenter la clarté

[4] `http://www.nongnu.org/fluid/`

[5] Dans notre implémentation, la responsabilité incombe à l'utilisateur de fournir une banque d'instruments compatible « General MIDI » si tel est son souhait.

de nos figures, nous nous sommes concentrés sur des éléments spécifiques de la trame sonore des jeux concernés.

Exemple 3.1 On sait déjà que la plupart des jeux font en sorte que l'état du jeu influence la musique. Toutefois, une possibilité intéressante, souvent inexploitée dans les jeux, consiste à faire en sorte que la musique agisse sur l'état du jeu de façon rétroactive. Afin d'illustrer cet effet, prenons l'exemple de *New Super Mario Bros.* [46] dans lequel les ennemis font des pas de danse au rhythme de la musique. Une manière d'implémenter cet effet consiste à séparer chaque pièce de musique en deux segments : *avant pas de danse* et *pendant/après pas de danse* puis d'utiliser un automate étendu probabiliste comme celui représenté à la figure 3.5 dont le but de la fonction de mise à jour est d'indiquer au jeu qu'il est temps pour les ennemis d'effectuer leurs pas de danse. Le jeu remet ensuite le registre à 0.

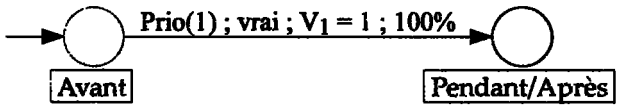

FIG. 3.5 – Rétroaction de la musique vers l'état du jeu

Exemple 3.2 Une technique pratiquement universelle dans tous les médias où la musique joue un rôle accompagnateur consiste à changer de thème musical à des repères bien précis. Dans les jeux, cela se traduit généralement par des changements de thèmes musicaux lorsqu'il se produit un changement majeur dans l'environnement du joueur. Par exemple, dans *Final Fantasy* [60], le joueur peut se déplacer sur un modèle réduit de la carte du monde mais il peut aussi visiter des temples, des villages, et bien d'autres endroits encore. De plus, au cours de ses explorations, il peut rencontrer des ennemis qu'il devra combattre. Pour modéliser ce cas avec un automate étendu probabiliste, il faut d'abord bien cerner le problème. On peut modéliser cela en définissant deux variables : une pour indiquer l'endroit où le joueur se trouve (V_1) et une autre pour indiquer s'il y a combat ou non.

À l'aide de certaines suppositions, généralement tirées du document de conception du jeu, un tableau comme le tableau 3.1 peut être directement traduit en automate par un

	Endroit (V_1)	Combat (V_2)
0	Carte du monde	Non
1	Temple	Oui
2	Village	—

TAB. 3.1 – Extrait des états possibles pour la musique de Final Fantasy

procédé de recouplement des variables. Pour *Final Fantasy*, on suppose qu'il n'est pas possible de se déplacer d'un endroit à l'autre lorsqu'un combat est en cours et qu'il ne peut y avoir de combat dans un village. La figure 3.6 illustre cette situation.

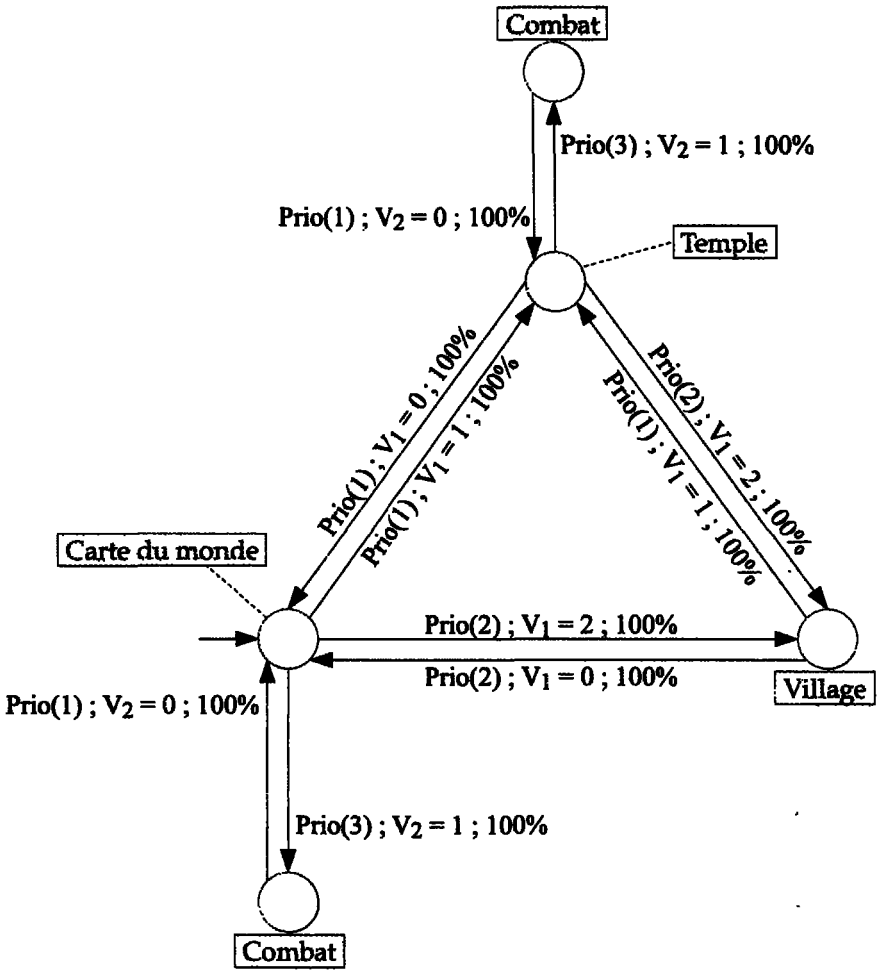

FIG. 3.6 – Changements de thèmes musicaux selon l'environnement du jeu

Exemple 3.3 Afin de combattre la fatigue de l'oreille causée par la répétition constante d'une même musique de fond, certains jeux introduisent des variations algorithmiques dans leur musique de fond. Par exemple, dans *Legend of Zelda : Ocarina of Time* [45], le thème de la « Prairie d'Hyrule » est composé de groupes de 8 mesures choisies aléatoirement parmi 12 possibilités. La figure 3.7 montre un automate étendu probabiliste effectuant la même tâche pour des groupes de 2 mesures choisies parmi 3 possibilités.

Bien entendu, ces trois exemples ne modélisent que des aspects isolés des trames sonores de ces trois jeux qui utilisent souvent une combinaison des techniques décrites et plus encore. Néanmoins, nous croyons que le fait que notre algorithme puisse modéliser une gamme aussi variée de techniques utilisées dans la création de trames sonores interactives et/ou algorithmiques pour les jeux vidéo suffit à prouver sa flexibilité.

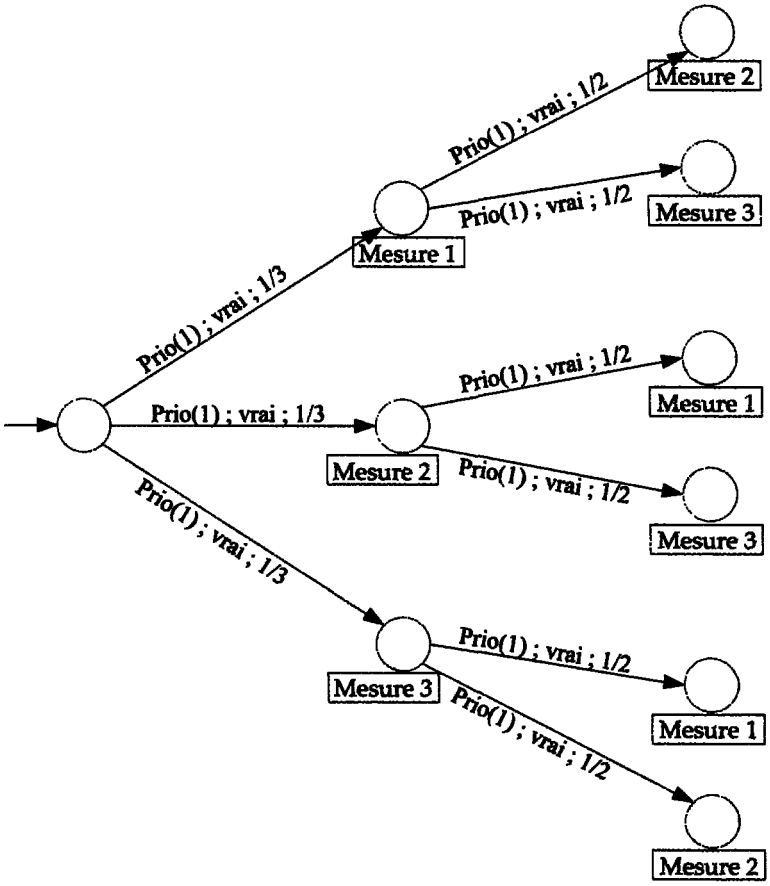

FIG. 3.7 – Permutation de mesures