Programmation et apprentissage bayésien de comportements pour personnages synthétiques – application aux personnages de jeux vidéos

Ronan Le Hy

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
| | | | | | | | | | |

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : *Sciences Cognitives*

préparée au laboratoire LIG et l'INRIA Rhône-Alpes, dans le cadre de l'École doctorale
Ingénierie pour la Santé, la Cognition et l'Environnement

présentée et soutenue publiquement par

Ronan LE HY

le 6 avril 2007

Titre :

Programmation et apprentissage bayésien de comportements pour des personnages synthétiques
Application aux personnages de jeux vidéos

Directeur de thèse :

Pierre BESSIÈRE

Composition du jury :

M.	Augustin LUX	Président
M.	Rachid ALAMI	Rapporteur
M.	Stéphane DONIKIAN	Rapporteur
M.	Denis DUFOUR	Examinateur
M.	Pierre BESSIÈRE	Directeur de thèse

Remerciements

J'ai commencé cette thèse grâce à Olivier Lebeltel (*Odge*). Je l'ai terminée grâce à Pierre Bessière, qui m'a soutenu scientifiquement, moralement, et financièrement.

J'ai eu la chance de collaborer en pratique avec (dans le désordre alphabétique) Juan-Manuel Ahuactzin, Anthony Arrigoni, Kevin Barbier, David Bellot, Vincent Carpy, Francis Colas, Sébastien Laborie, Kamel Mekhnacha, Diego Pisa. Quant aux amis qui m'ont aidé à finir, j'ai mieux pour eux que cette page de remerciements. Il y a Fabrice, Alain, David B., Francis, Lucile, Anne, Claire, Jean, et tous les autres...

L'équipe Laplace/SHARP/Cybermove/e-Motion est formidable. Véronique Roux, Anne Pasteur, Olivier Malrait, Carole Bienvenu, Helen Pouchot ont souvent fait plus que leur travail pour m'aider. Je n'aurais peut-être pas fini sans l'aide de Razika Hammache.

Résumé

Nous nous intéressons à l'acquisition de comportements par des personnages autonomes (*bots*) évoluant dans des mondes virtuels, en prenant comme exemple les jeux vidéos. Deux objectifs essentiels sont poursuivis :

- réduire le temps et la difficulté de programmation pour le développeur, qui doit peupler un monde virtuel de nombreux personnages autonomes ;
- offrir au joueur une nouvelle possibilité : apprendre à des bots comment jouer.

Alors que les environnements virtuels sont complexes, et que les comportements des bots doivent être riches, le défi est d'offrir des méthodes simples de programmation et d'apprentissage. Celles-ci doivent de plus se plier à des contraintes importantes sur la mémoire et le temps de calcul disponibles.

Nous commençons par présenter les méthodes actuelles de programmation de tels personnages par une étude de cas avec *Unreal Tournament*, un jeu de combat à la première personne. Dans ce jeu, les comportements s'appuient sur un langage de programmation spécialisé pour la description de machines d'états finis. Cette méthodologie est caractérisée par une grande flexibilité, une faible formalisation et une grande complexité. Elle se prête difficilement à l'apprentissage.

Nous proposons une méthode alternative de construction de comportements basée sur la programmation bayésienne, un formalisme de description de modèles probabilistes. D'une part, cette méthode permet de maîtriser la complexité de programmation de comportements composés. D'autre, part elle sépare clairement le travail de programmation de celui d'ajustement d'un comportement : ce dernier peut être fait par un non-informaticien. Techniquement cette méthode repose essentiellement sur deux innovations :

- Une technique générique de définition de tâches élémentaires, appelée *fusion par cohérence améliorée*. Elle permet de fusionner un nombre important de consignes exprimées comme des tableaux de valeurs définissant des distributions de probabilités. Chacune de ces consignes peut être soit prescriptive (que faire) soit proscriptive (que ne pas faire).
- Une technique de mise en séquence de ces tâches élémentaires, qui permet de construire le comportement complet du personnage à partir des tâches élémentaires précédentes, appelée *programmation inverse*. Elle repose sur un modèle de Markov caché spécialisé, qui peut lui aussi être vu comme une machine d'états finis mais dont la spécification est plus condensée qu'avec un langage de programmation classique.

Contrairement à l'approche classique, cette méthode de construction de comportement permet facilement l'apprentissage par démonstration. Un bot apprend son comportement en observant un humain qui joue. Les tâches élémentaires, comme les séquences, peuvent ainsi être apprises. Pour les tâches élémentaires, l'identification des paramètres se fait directement. Pour les séquences, il est nécessaire reconnaître les « intentions » du joueur (les tâches élémentaires) à partir des actions de son avatar. Cela est rendu possible en utilisant soit une méthode de reconnaissance à base d'heuristiques spécifiques, soit une méthode de reconnaissance bayésienne basée sur l'algorithme de Baum-Welch incrémental.

Mots-clés : programmation bayésienne, jeux vidéos, fusion par cohérence améliorée, programmation inverse, apprentissage par démonstration.

Summary

We treat the problem of behaviours for autonomous characters (*bots*) in virtual worlds, with the example of video games. Our two essential objectives are :

– to reduce time and difficulty of behaviour development ;

– to give to the player a new possibility : teaching bots how to play.

We propose a method to build behaviours based on Bayesian programming (a formalism to describe probabilist models). It lays on two innovations :

– a generic technique for definition of elementary tasks, called **enhanced fusion by coherence** ;

– a technique for sequencing these elementary tasks, called **inverse programming**.

In contrast with classical approaches, this method allows to efficiently learn behaviours by demonstration.

Keywords : bayesian programming, video games, enhanced fusion by coherence, inverse programming, learning by demonstration.

Table des matières

Chapitre 1

Introduction

1.1 Les mondes virtuels

Un monde virtuel est la réunion d'un environnement simulé et de personnages qui le peuplent.

1.1.1 La motivation : un lieu d'interaction complexes

World of Warcraft (figure 1.1) est un exemple commun de monde virtuel. Il se présente comme un monde virtuel imaginaire, peuplé de créatures. Bien que virtuel, ce monde est semblable en certains points à ce que nous percevons du monde réel. Les lois élémentaires de la physique y semblent respectées ; les créatures et les lieux, dans leur apparence, leur comportement et leurs interactions, ressemblent à ceux que nous connaissons dans le monde réel. L'idée au centre du jeu est que chaque humain prend le contrôle d'une des créatures. Il existe aussi des créatures complètement autonomes, contrôlées par la machine.

World of Warcraft est principalement le lieu d'interactions entre humains. La richesse du monde qu'il propose est en grande partie celle des relations entre joueurs humains. Les personnages autonomes, dits *personnages non joueurs* (PNJ), participent principalement à la crédibilité du monde virtuel, mais n'accomplissent pas les mêmes tâches que les joueurs humains.

Un second exemple commun de monde virtuel dans les jeux vidéos est *Unreal Tournament* (figure 1.2). Des personnages humanoïdes s'y combattent, chacun pour soi ou par équipes, dans une arène de combat en trois dimensions (3D). Ces personnages peuvent être contrôlés par des humains, ou par la machine. Ce type de jeux de combats met donc la machine et l'humain sur le même pied, dans des relations de compétition ou de collaboration de pair à pair.

Un jeu vidéo présentant un monde virtuel met donc en jeu trois sortes d'interactions. En premier lieu, les interactions **monde-monde**, sont celles liées à la simulation du monde virtuel. Deuxièmement, les interactions **monde-personnage**, concernent la problématique d'interface des personnages dans le monde, et de dynamique de celui-ci. Enfin, les interactions **personnage-personnage** consistent en tous les aspects de communication.

FIG. 1.1 – World of Warcraft : un monde virtuel peuplé de personnages contrôlés par des humains, et de créatures autonomes (image Francis Colas).

Notre propos est la construction de personnages dont les interactions avec le monde et les autres personnages soient pertinentes pour les objectifs du monde virtuel.

FIG. 1.2 – Unreal Tournament : des personnages synthétiques et contrôlés par des humains s'affrontent dans une arène (image unrealtournament.com).

1.1.2 Les personnages

Les facettes possibles d'un personnage autonome

Un personnage autonome dans un monde virtuel complexe peut posséder chacune des caractéristiques que Ferber attribue [Ferber 95] aux *agents*.

Il **agit dans l'environnement** : se déplace, utilise une outil ou une arme, ramasse des objets... Il **communique avec d'autres personnages** : il prend sa place dans une équipe, fait des gestes pour indiquer son humeur ou ses intentions... Il est **conduit par des motivations et des buts** : tuer le plus grand nombre d'ennemis, capturer le drapeau de l'équipe adverse, accumuler des points d'expérience... Il possède des **ressources propres** : des armes, un niveau de santé, une armure, des sorts à jeter... Il **perçoit une partie de son environnement** : il voit une partie de l'environnement proche, dans la limite de son champ de vision et des objets opaques qui lui cachent la vue, entend les sons émis à sa proximité... Il possède une **représentation partielle de son environnement**, ou **pas de représentation** : il se construit une carte de l'environnement visité jusqu'ici, ou éventuellement ne mémorise rien. Il possède des **compétences** : défendre une zone, attaquer, aller chercher le drapeau ; et peut **offrir des services** : se plier à des ordres lors d'un jeu par équipes. Il peut se **reproduire** : la reproduction elle-même semble rare dans les jeux vidéos modernes, mais il est fréquent que les personnages puissent renaître quand ils meurent.

Enfin, son comportement tend à satisfaire ses objectifs, en prenant en compte ses ressources et compétences, selon ses perceptions, ses représentations et des communications qu'il met en jeu.

Les défis de la programmation d'agents autonomes

Le premier défi de la programmation de comportements pour personnages autonomes est celui de la construction de la complexité. Il s'agit de créer des comportements intéressants en vue de l'interaction avec des joueurs humain. Cela demande l'illusion de l'intelligence, ou de l'*humanité*. L'idéal de cette illusion serait un personnage joueur autonome pour *World of Warcraft* qui puisse fournir la même qualité d'interaction que les joueurs humains. Les mondes virtuels fournissent ainsi le cadre d'une forme de *test de Turing* [Turing 50] où les joueurs humains sont en permanence en position d'évaluer l'humanité des personnages autour d'eux.

Le second défi est celui de la gestion de la complexité engendrée par cette quête de crédibilité : la puissance des comportements et leur maîtrise sont des directions antagonistes. De plus, l'existence d'un grand nombre de personnages complexes (qu'ils soient contrôlés par des humains ou par la machine) dans un monde complexe fournit un cadre impossible à appréhender dans son ensemble pour un personnage donné. La programmation du comportement d'un personnage autonome doit donc faire face à une incomplétude inévitable des représentations et raisonnements sur le monde – comme un robot réel en environnement non contrôlé.

1.1.3 Les questions pratiques du développement de personnages autonomes

Dans la perspective de construire le comportement d'un personnage autonome, nous nous plaçons du point de vue du développeur, puis de celui du joueur.

Le point de vue du développeur

La première tâche du développeur est la programmation initiale du comportement. Confronté à une méthodologie particulière de programmation, il se pose les questions suivantes.

- Est-ce facile ? Un non-spécialiste de programmation (scénariste, concepteur de niveau...) peut-il décrire lui-même le comportement ? Quelle quantité d'information faut-il spécifier, et sous quelle forme ? Cette forme est-elle proche de l'idée intuitive que je me fais du fonctionnement du comportement ? Cette série de questions met en jeu le niveau d'abstraction de la méthodologie de programmation.
- Puis-je faire tout ce que j'imagine ? Puis-je atteindre une complexité importante en gardant le contrôle sur mon programme ? Mon programme risque-t-il de devenir difficile à comprendre et à modifier ? Ces questions couvrent la notion (floue) d'expressivité du système de programmation de comportements.
- Puis-je programmer des comportements complets dans les limites des ressources processeur et mémoire qui me sont allouées ? Dans un cadre industriel, un système de gestion de comportements doit en effet se contenter d'une minorité des ressources processeur et mémoire.
- Puis-je fixer les paramètres de mon comportement par simple apprentissage par pilotage ? Puis-je alléger la charge de programmation en ajustant certains paramètres au moyen de techniques d'apprentissage automatique ? Une simple démonstration peut devenir une technique puissante de programmation quand la méthodologie s'y prête. Par exemple, le jeu de course *Colin McRae Rally 2* contient des conducteurs autonomes contrôlés par des réseaux de neurones artificiels, dont l'ajustement des paramètres peut facilement être fait par apprentissage.

Une implémentation initiale faite, il devient nécessaire de modifier le comportement pour l'ajuster selon des besoins particuliers, le faire évoluer ou en régler des problèmes *a posteriori*.

- Puis-je identifier les paramètres pertinents de mon modèle pour ajuster une partie donnée du comportement ?
- Puis-je donner un sens à ces paramètres ? Puis-je fixer leur valeur avec mon intuition ? Puis-je comprendre pourquoi mon comportement ne fait pas ce que je veux ?
- Puis-je analyser un comportement que je n'ai pas écrit, pour le comprendre et le modifier ? Puis-je me reposer sur un formalisme, une description systématique et organisée ?

Le point de vue du joueur

Le joueur humain qui interagit avec notre personnage autonome est intéressé en premier lieu par la qualité de cette interaction. S'il est difficile de définir positivement cette qualité au-delà

de la notion d'*humanité*, les défauts possibles d'un comportement peuvent être énumérés.

- Prédictabilité, répétitivité : un comportement est opérant mais prédictible, mécanique. Par exemple, les personnages autonomes d'*Unreal Tournament* se déplacent dans leur environnement 3D le long de chemins stereotypés, ce qui les identifie clairement comme non-humains.
- Rigidité : un comportement est inopérant dans certaines situations particulières de blocage qui n'ont pas été prévues lors de la conception. Cela survient particulièrement quand le joueur peut remarquer les motifs répétitifs du comportement du personnage autonome, et en tirer partie systématiquement.
- Habileté mal ajustée : un comportement est trop faible ou trop fort quant aux tâches spécifiques du jeu, pour un joueur humain donné. L'intérêt d'un jeu vidéo repose sur le fait que le joueur puisse atteindre les objectifs du jeu, moyennant des efforts raisonnables.

Au-delà de ces défauts qui marquent par leur présence ou leur absence la qualité d'un comportement, certaines caractéristiques sont recherchées pour la valeur ajoutée qu'elles apportent à l'intérêt d'un jeu.

- Le joueur peut-il montrer à un personnage comment se comporter ? Il s'agit de donner au joueur la possibilité d'entraîner par l'exemple les personnages qui jouent avec lui – par exemple dans un combat par équipes dans *Unreal Tournament*. Dans les mondes virtuels permanents comme *World of Warcraft*, il serait aussi intéressant qu'un personnage autonome puisse remplacer temporairement un joueur humain, en basant son propre comportement sur celui observé du joueur. Son personnage devient alors tantôt avatar[1], tantôt robot autonome – une seule créature cohérente et persistante du monde.
- Le joueur peut-il sauver, charger, échanger des comportements pour ses personnages – comportements qu'ils aurait éventuellement programmé lui-même par démonstration ?

1.2 La problématique de la construction de personnages synthétiques autonomes

Nous dressons ici une catégorisation des questions qui font notre problématique de construction de comportements. Nous prendrons ce jeu de questions comme fil conducteur de notre étude.

(1) *En quoi consiste la description d'un comportement ?* Ce groupe de questions porte sur le formalisme de description de comportements. Il s'agit de caractériser la structuration de l'information sur un comportement. Dans cette information, nous comprenons tout ce qui fait qu'un comportement donné est distinct d'un autre comportement. Cela peut regrouper des éléments tels que la structure d'un modèle graphique, des paramètres numériques, l'expression d'un algorithme dans un langage de programmation... Dans la suite, nous désignons cette information par le terme de *paramètres* d'un comportement.

(Q1a : structure des paramètres) *Les paramètres d'un comportement sont-ils localisés, identifiés et structurés ?* Nous avons vu que les paramètres d'un comportement doivent pouvoir

[1] Un avatar est la représentation d'un humain dans le monde virtuel.

être compris et manipulés. Cela nécessite qu'ils soient localisés (réduits à un ensemble déterminé de valeurs ou d'expressions), identifiés (qu'un rôle puisse être affecté à chacun), et structurés (qu'ils soient organisés, groupés, modularisés de manière intelligible). Un comportement décrit comme un programme arbitraire dans un langage de programmation généraliste donné est un exemple d'information mal localisée et difficile à identifier ; l'identité d'un comportement est alors constituée d'un programme complet. A l'autre extrême, un comportement formé d'un régulateur Proportionnel-Dérivée (PD) est un exemple d'information très bien localisée et identifiée : les paramètres du comportement sont deux constantes dont le sens est bien compris. Un critère d'évaluation de la localisation et de l'identification optimale des paramètres d'un comportement est la facilité avec laquelle il est possible de les manipuler et les modifier de manière automatique (comme cela peut être fait lors d'un processus d'apprentissage). Pour notre premier exemple, il est ainsi très difficile de manipuler un comportement dont la description est un programme arbitraire.

(Q1b : paramètres et intuition) *Les paramètres d'un comportement sont-ils accessibles à l'intuition ?* Le développeur doit pouvoir passer de son intuition d'un comportement à son implémentation, et vice-versa. Il s'agit de combler le vide entre la représentation humaine d'un comportement (« ce personnage doit suivre le couloir, retrouver son chef, et le suivre ») et sa traduction en termes de valeurs pour les paramètres du comportement (la structure d'un réseau de neurones artificiel, un ensemble de règles logiques...). Combler ce vide est nécessaire pour pouvoir passer de l'intuition d'un comportement à son implémentation d'une part (problématique de programmation), et de l'implémentation d'un comportement à l'intuition de ses mécanismes de fonctionnement (problématique de maintenance) d'autre part. Ce problème est celui de l'*ancrage des symboles* [Harnad 90]. Au moins deux voies – non exclusives – sont possibles pour l'aborder. La première est de s'assurer que les paramètres d'un comportement ont un sens intuitif pour le programmeur, comme c'est le cas par exemple pour un arbre de décision formé de règles logiques portant sur des variables de haut niveau. La seconde est de fournir au programmeur les moyens de manipuler les paramètres du comportement de manière intuitive et partiellement automatisée. Cela peut en particulier être fait par apprentissage par démonstration. Un critère pratique d'évaluation de l'accessibilité des paramètres pour l'intuition est la possibilité pour un non-spécialiste, comme un scénariste ou un joueur même, de manipuler les paramètres du comportement.

(Q1c : apprentissage) *Est-il possible d'apprendre les paramètres du comportement par démonstration ?* L'apprentissage par démonstration (ou un expert humain montre au personnage virtuel comment se comporter) est en premier lieu un élément de l'intérêt de certains jeux. Il est aussi un moyen pour le développeur de comportements de manipuler de manière intuitive les paramètres d'un comportement. Ces deux aspects ne peuvent cependant que rester marginaux en l'absence d'un système de description de comportements qui permet l'apprentissage par démonstration. Nous cherchons une méthode qui permettrait un apprentissage par démonstration à la volée, en temps réel – ce qui est nécessaire pour pouvoir s'intégrer aux concepts de jeu d'une application ludique.

(2) *Peut-on exécuter un comportement complexe dans des limites fixées de ressources maté-*

rielles ? Dans le cadre d'une application complète comme un jeu vidéo, le système de gestion de comportements doit s'exécuter dans des limites fixées de ressources matérielles. Ces ressources sont principalement la mémoire centrale, et le temps processeur. Il s'agit de déterminer d'une part si un comportement complexe typique peut être exécuté dans les limites fixées, et d'autre part comment évolue l'occupation des ressources quand la complexité d'un comportement est augmentée. Pour la mémoire comme pour le temps processeur, il s'agit de pouvoir évaluer dans la pratique les besoins du comportement (ceux-ci pouvant évoluer dans le temps), puis d'agir si possible sur les paramètres du comportement pour en réduire les besoins.

(Q2a : complexité en mémoire) *Un comportement complexe peut-il être exécuté dans les limites typiques de mémoire disponible ?* L'occupation mémoire est liée à la nécessité de stocker des données liées à une représentation du monde virtuel et de l'appareil sensori-moteur du personnage, et à celle de raisonner sur la base de ces données.

(Q2b : complexité en temps de calcul) *Un comportement complexe peut-il être exécuté dans les limites typiques de temps processeur disponible ?* Le temps processeur alloué au système de gestion des comportements dans un jeu vidéo est généralement limité à 10% à 20% par la place prépondérante que prennent les systèmes de simulation et de rendu graphique du monde virtuel. Nous nous intéressons à la complexité théorique des méthodes de décision, mais aussi à un critère pratique : sur un processeur moderne, combien de décisions pouvons-nous faire par seconde ?

1.2.1 Récapitulatif des questions de problématique

Voici la liste des questions que nous avons développées, et qui nous serviront de fil conducteur dans la suite de notre étude.

- **(1)** *En quoi consiste la description d'un comportement ?*
  - **(Q1a : structure des paramètres)** *Les paramètres d'un comportement sont-ils localisés, identifiés et structurés ?*
  - **(Q1b : paramètres et intuition)** *Les paramètres d'un comportement sont-ils accessibles à l'intuition ?*
  - **(Q1c : apprentissage)** *Est-il possible d'apprendre les paramètres du comportement par démonstration ?*
- **(2)** *Peut-on exécuter un comportement complexe dans des limites fixées de ressources matérielles ?*
  - **(Q2a : complexité en mémoire)** *Un comportement complexe peut-il être exécuté dans les limites typiques de mémoire disponible ?*
  - **(Q2b : complexité en temps de calcul)** *Un comportement complexe peut-il être exécuté dans les limites typiques de temps processeur disponible ?*

1.3 Contribution

Nous proposons une méthode de programmation de comportements basée sur la Programmation Bayésienne des Robots [Lebeltel 99]. Il s'agit d'une technique de modélisation probabiliste permettant d'approcher à plusieurs égards les machines d'états finis.

Cette méthode permet la construction incrémentale de modèles structurés, fortement modularisés **(Q1a)**. Comme certaines techniques classiques, elle est basée sur le séquencement et la hiérarchisation de tâches élémentaires.

La technique dite de *programmation inverse* permet de casser la complexité de la description des séquencements de tâches, tant en représentation mémoire **(Q2a)** qu'en temps de calcul **(Q2b)**.

Au cœur de nos comportements, se trouvent des tables de paramètres numériques, codant des distributions de probabilités. Nous montrerons que ces paramètres sont particulièrement accessibles à l'intuition **(Q1b)**, et qu'ils se prêtent aux manipulations automatiques **(Q1a)**.

En particulier, nous montrerons que grâce à cette représentation il est possible d'apprendre par démonstration nos modèles de comportement **(Q1c)** en temps réel **(Q2b)**.

1.4 Plan de lecture

Nous présenterons dans un premier temps les techniques de l'état de l'art industriel de la programmation de comportements, sur l'exemple d'*Unreal Tournament*. Ce jeu, représentatif des jeu de combat à la première personne, nous permettra d'illustrer la simplicité et la puissance des méthodes couramment employées aujourd'hui pour programmer des comportements. Nous extrairons les points forts et les faiblesses de ces méthodes, pour asseoir la discussion des techniques que nous proposons.

Nous décrirons ensuite, du simple au complexe, la construction d'un comportement pour un personnage autonome fonctionnel. Nous traiterons de la construction de comportements simples, qui peuvent être conçus d'un bloc. Nous décrirons en guise d'illustration certains des comportements de base d'un personnage autonome jouant à *Unreal Tournament*.

Nous montrerons comment combiner ces comportements simples dans le temps, à l'aide d'une technique de mise en séquence probabiliste. Cette technique permet de construire un personnage autonome complet jouant à *Unreal Tournament*, combinant tâches élémentaires.

Nous décrirons enfin comment à chaque étape les programmes construits se prêtent à l'apprentissage : les comportements simples, comme ceux mis en séquence peuvent être appris. Il sera donc possible pour un joueur de montrer à notre personnage comment se battre.

Nous conclurons en résumant les aspects critiques de notre méthode de construction de comportements, et comment elle se compare aux méthodes classiques telles que celles vues dans le jeu *Unreal Tournament* original.

Chapitre 2

Etude de cas : *Unreal Tournament*, un système industriel de gestion de comportements

Unreal Tournament est un jeu vidéo de combat à la première personne[1]. Nous en présentons les caractéristiques qui en font une plateforme de référence pertinente pour la programmation de comportements. Nous nous consacrons ensuite à décrire la méthodologie de construction de comportements en son sein.

2.1 *Unreal Tournament* : fonctionnement général

2.1.1 Le monde virtuel d'*Unreal Tournament*

Le jeu se présente comme un environnement 3D (voir figure 2.1), représentant des bâtiments et des paysages futuristes, généralement labyrinthiques. Des personnages humanoïdes armés évoluent dans cet environnement. Ils peuvent se déplacer (marcher, courir, sauter, se téléporter), faire usage de leurs armes, ramasser des objets (armes, bonus de santé...), actionner des commandes (ascenseurs, chausse-trappes...). Ces personnages ont des buts. Le plus commun est le match à mort, où chaque personnage se bat pour tuer le plus grand nombre de fois possible ses adversaires. Des variantes existent où ce mode de jeu se pratique par équipes ; dans d'autres, le but principal est de capturer le drapeau de l'équipe adverse ou de garder le contrôle de certaines zones de l'environnement. Chacun des personnages est contrôlé soit par un joueur humain dont

[1] un jeu à la première personne est un jeu où le joueur se perçoit à l'intérieur de son avatar dans le monde virtuel. Il voit sur l'écran ses mains et son fusil.

FIG. 2.1 – *Unreal Tournament* : un personnage autonome se bat dans l'arène.

il est l'avatar, soit par un programme informatique.

Ces caractéristiques font d'*Unreal Tournament* un objet d'étude intéressant en regard de nos objectifs. En premier lieu, il présente un monde virtuel complexe. La complexité se présente à l'oeil de chaque personnage (autonome ou contrôlé par un joueur humain), en la taille de l'environnement 3D et la relative richesse des interactions qu'il offre. Mais elle se trouve surtout dans l'absence de maîtrise du comportement des autres personnages – alors même que les buts du jeu reposent sur l'interaction avec ceux-ci.

En second lieu, *Unreal Tournament* nous fournit une plateforme de simulation où les personnages sont contraints par le temps. Pour être crédible aux yeux des joueurs humains, un personnage autonome doit décider et agir rapidement et régulièrement.

Enfin, le système de contrôle des personnages autonomes est dans *Unreal Tournament* une brique essentielle de l'intérêt du jeu (comme le sont de plein droit la qualité du rendu graphique, et la plausibilité de la simulation du monde virtuel sans ses personnages). Dans l'idéal, les personnages autonomes présenteraient l'illusion d'être commandés par des joueurs humains, d'habileté proche de celle du vrai joueur humain.

2.1.2 Grands traits de l'implémentation d'*Unreal Tournament*

Dans la perspective de décrire le système de gestion de comportements mis en œuvre dans *Unreal Tournament*, nous en présentons en premier lieu l'organisation générale.

Nécessité de la séparation en couches

Un moteur de simulation est généralement séparé en couches indépendantes. Cela répond à des exigences d'ordre industriel : développement de différentes parties en parallèle, réutilisation de composants dans différents produits, facilitation de la maintenance et des extensions.

Les couches logicielles

Le système repose sur un moteur de **rendu graphique**. Son but est la représentation à l'écran du monde virtuel tridimensionnel. Il regroupe des routines de plus ou moins haut niveau de rendu et d'animation de triangles, de murs, d'effets atmosphériques (brouillard, neige...). Il est d'ordinaire le plus gros consommateur de ressources du système global. Une grande partie des efforts de développement du jeu complet y sont consacrés.

Sur ce moteur de rendu, sont ajoutés des systèmes d'**animation**. L'animation se place à plusieurs niveaux, de la simple lecture d'animations stockées sur disque, basées sur l'interpolation entre positions-clés (par exemple, animation du cycle de marche d'un personnage), à la composition d'animations pour la création de mouvements complexes (animation d'un personnage courant à travers l'environnement, tout en rechargeant son pistolet et en bougeant la tête).

Ce premier bloc de rendu et d'animation peut généralement (c'est le cas d'*Unreal Tournament*) être clairement séparé du système de simulation du monde, d'une part, et de contrôle des personnages, d'autre part.

La **simulation du monde** est ce qui concerne la gestion des objets animés (ascenseurs, portes, personnages en tant qu'objets physique virtuels...), et de leurs interactions. C'est la partie qui se consacre à construire le monde virtuel.

Enfin, en utilisant ces systèmes de base, il est possible de construire un système de **contrôle** des personnages. Cela consiste en la spécification du comportement de chaque élément autonome dans le monde, du plus simple (ascenseur), au complexe (personnage humanoïde). Cette spécification est au mieux décrite comme l'écriture séparée, pour chaque créature, d'un comportement, sur la base des capteurs et moteurs virtuels fournis par le système de simulation du monde.

Le contrôle de certains personnages peut être délégué à un humain. Ces personnages sont nommés *avatars*. L'interface de contrôle de ces personnages (les systèmes moteurs et de retour perceptuel) peuvent faire l'objet d'un module séparé. Les plateformes auxquelles nous nous intéressons ici utilisent généralement le clavier, la souris et un rendu graphique 3D sur écran.

Objet de notre étude

Dans notre étude, nous laisserons de côté le moteur de rendu, le système d'animation, et le module de simulation du monde. Nous nous consacrerons au problème de la construction de comportements. Nous nous appuierons sur ceux implémentés effectivement dans Unreal Tournament, dont les sources nous sont accessibles.

2.2 Développement de comportements pour *Unreal Tournament*

Nous nous intéressons ici à la façon dont sont programmés et exécutés les comportements des personnages autonomes d'*Unreal Tournament*.

2.2.1 Technologie de description des comportements dans *Unreal Tournament*

UnrealScript : un langage de description de machines d'états finis

La description des comportements dans *Unreal Tournament* repose sur un langage propre de description de machines d'états finis : *UnrealScript*.

Ce langage possède les caractéristiques d'un langage de scripts moderne. Il est dynamique (certains éléments, comme les tables de symboles, sont accessibles et modifiables à l'exécution), sûr (il garantit l'intégrité de la mémoire), objet (les programmes sont organisés autour des données), et est compilé vers un code-objet pour une machine virtuelle imbriquée dans le moteur du jeu. Il se prête à des développements rapides et de haut niveau.

Chaque élément dynamique (personnage, ascenseur, porte...) est contrôlé par une machine d'états finis propre décrite grâce à *UnrealScript*. À l'exécution, tous les automates des nombreux agents du jeu sont simulés en parallèle.

Pour un personnage, décrire un comportement revient donc à décrire :

– des **états**, auxquels sont associées des actions ;
– des **gestionnaires d'évènements**, qui sont déclenchés quand certains évènements surviennent, dans le contexte d'états particuliers ; par exemple quand le personnage bute contre un mur alors qu'il est en train de sauter.
– des **transitions** d'état à état, qui sont déclenchées soit dans les actions, soit dans les gestionnaires d'évènements.

Stratégies de développement

Nous nous attachons dans la suite de ce chapitre à décrire la construction ascendante d'un comportement complet pour un personnage d'*Unreal Tournament*. Cette démarche correspond à un processus de développement incrémental, et permet d'ajuster des comportements fonctionnels de plus en plus complexes par essai-erreur, et tests progressifs.

La vue générale d'un comportement complet d'un personnage autonome d'*Unreal Tournament* est représentée sur la figure 2.2. Cette figure correspond au comportement de combat en match à mort, tel qu'il est implémenté dans le jeu original. Sont figurés les états (bulles étiquetées) et les transitions (flèches d'état à état) d'une machine d'états finis de taille importante. Nous allons exposer la structure de cette celle-ci, des éléments de base au comportement complet ; nous détaillerons donc les niveaux suivants :

– la fonction ;
– la tâche ;
– la méta-tâche ;
– le comportement complet.

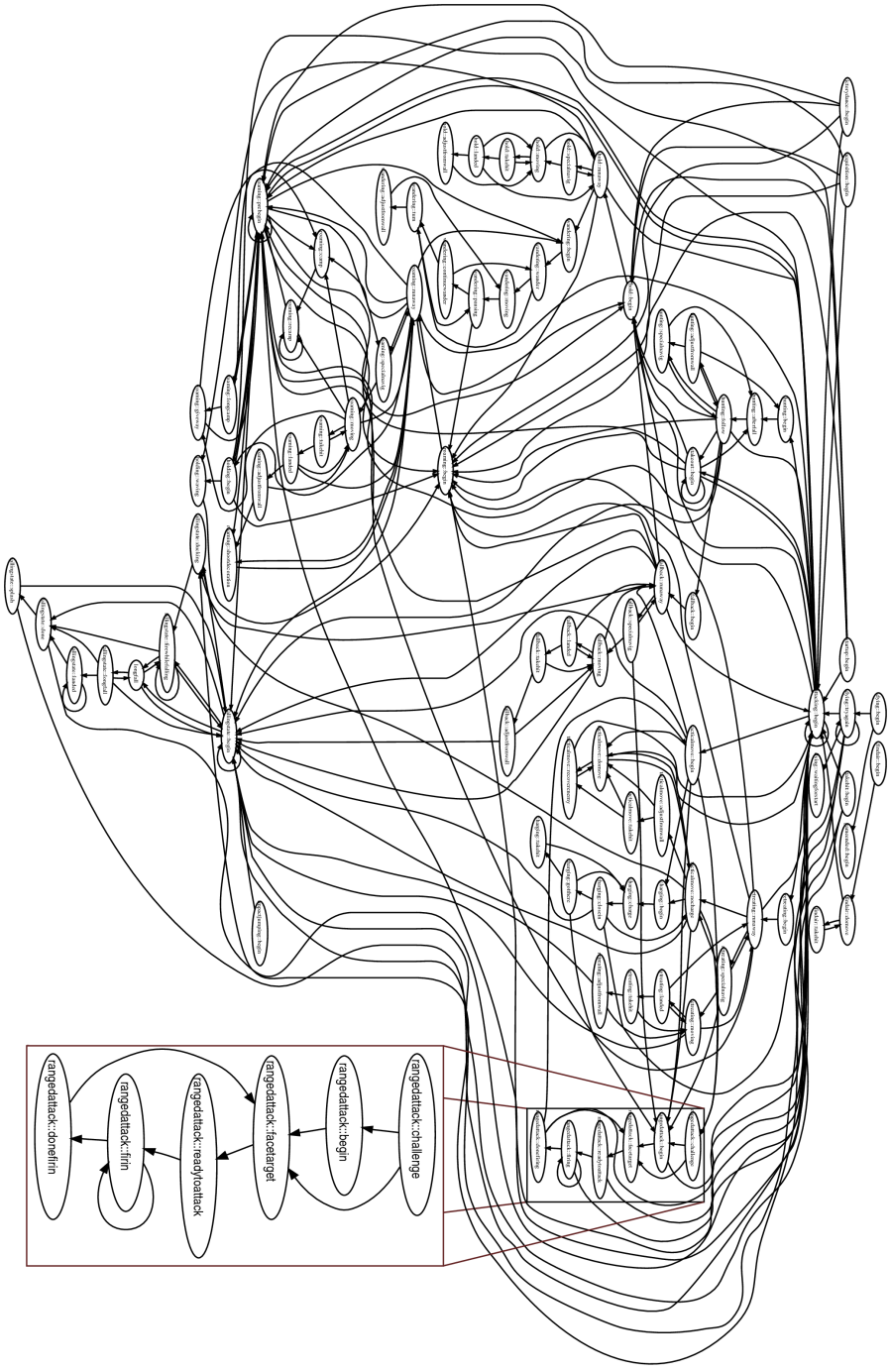

FIG. 2.2 – Extrait du graphe de la machine d'états finis du comportement complet d'un personnage Unreal Tournament. Chaque bulle correspond à une sous-tâche. Les annotations des transitions sont omises : celles-ci sont déclenchées suivant l'arrivée d'évènements et les valeurs des variables du programme correspondant au graphe. La tâche grossie, `RangedAttack`, correspond à une tâche d'attaque à distance.

2.2.2 Une fonction : `PlayCombatMove()` (jouer un mouvement de combat)

La figure 2.3 montre un des éléments de base d'un comportement : une fonction écrite en *UnrealScript*.

```
function PlayCombatMove() {
  if ( (Physics == PHYS_Falling) && (Velocity.Z < -300) )
     FastInAir();
  else
     PlayRunning();
  if ( Enemy == None )
     return;
  if ( !bNovice && (Skill > 0) )
     bReadyToAttack = true;
  if ( Weapon == None ) {
     bAltFire = 0;
     bFire = 0;
     return;
  }
  if ( bReadyToAttack && bCanFire ) {
     if ( NeedToTurn(Enemy.Location) ) {
        if ( Weapon.RefireRate < 0.99 )
           bFire = 0;
        if ( Weapon.AltRefireRate < 0.99 )
           bAltFire = 0;
     } else
        FireWeapon();
  } else {
     // keep firing if rapid fire weapon unless can't see enemy
     if ( Weapon.RefireRate < 0.99 )
        bFire = 0;
        if ( Weapon.AltRefireRate < 0.99 )
           bAltFire = 0;

        if ( (bFire + bAltFire > 0) &&
             ((Level.TimeSeconds - LastSeenTime > 1) ||
              NeedToTurn(Enemy.Location)) ) {
           bFire = 0;
           bAltFire = 0;
        }
  }
}
```

FIG. 2.3 – Fonction `PlayCombatMove()` : jouer un mouvement de combat. Cette fonction déclenche le tir quand cela est approprié. Il s'agit d'un programme procédural classique, se basant sur la manipulation de variables et l'appel d'autres fonctions (en vert).

Cette fonction est appelée pour faire jouer au personnage une action élémentaire de combat : se déplacer et faire feu, alors que la cible est connue. Elle se base sur la manipulation (lecture et écriture) de diverses variables propres à l'état du personnage. Par exemple, la variable `bFire` indique que le personnage doit faire feu. Il est fait appel à d'autres fonctions (en vert), comme `FireWeapon()` qui joue l'animation du tir et signale au moteur de simulation l'envoi d'un

projectile.

2.2.3 Une tâche : `RangedAttack` (attaque à distance)

Il est possible de construire une tâche en exécutant des fonctions les unes après les autres, c'est-à-dire par un processus de **séquencement**.

Ainsi, la fonction `PlayCombatMove` est une des fonctions utilisées pour la construction de la tâche `RangedAttack`. Cette tâche consiste à attaquer un ennemi en restant à distance ; elle est utilisée quand l'arme du personnage a un rayon d'action important, comme c'est le cas pour le lance-roquettes. Un extrait en est présenté sur la figure 2.4.

Pour des raisons de clarté, la fonction `PlayCombatMove` n'apparaît pas sur cette figure. On peut cependant noter l'appel de fonctions qui réalisent une animation (`FinishAnim()`), un déplacement (`TweenToFighter()`, `TurnToward()`), ou une combinaison d'animations et d'actions complexes (`PlayRangedAttack()`).

```
state RangedAttack {
// [...]
Challenge:
    Acceleration = vect(0,0,0); // stop
    DesiredRotation = Rotator(Enemy.Location - Location);
    PlayChallenge();
    FinishAnim();
    TweenToFighter(0.1);
    Goto('FaceTarget');
Begin:
    Acceleration = vect(0,0,0); // stop
    DesiredRotation = Rotator(Target.Location - Location);
    TweenToFighter(0.16 - 0.2 * Skill);
    // Goto('FaceTarget') implicite
FaceTarget:
    if ( NeedToTurn(Target.Location) )
    {
     PlayTurning();
     TurnToward(Target);
     TweenToFighter(0.1);
    }
    FinishAnim();
    // Goto('ReadyToAttack') implicite
ReadyToAttack:
    DesiredRotation = Rotator(Target.Location - Location);
    PlayRangedAttack();
    // Goto('Firing') implicite
Firing:
    TurnToward(Target);
    // Goto('DoneFiring') possible, déclenché par certains évènements
    Goto('Firing');
DoneFiring:
    KeepAttacking();
    Goto('FaceTarget');
```

FIG. 2.4 – État `RangedAttack` (attaque à distance). Les étiquettes internes de l'état sont marquées en vert, et les sauts vers ces étiquettes, en bleu.

Cette tâche est décrite comme une machine d'états finis, dont les états sont les étiquettes (en vert) : `Challenge`, `Begin`, `FaceTarget`... Les transitions sont déclenchées par l'appel à `Goto()`

(en bleu). Certains appels à `Goto()`, comme celui vers l'étiquette `DoneFiring`, sont déclenchés indirectement par l'arrivée de certains évènements, comme la fin du jeu d'une animation de tir. En l'absence d'un `Goto()` explicite, le flux de contrôle passe d'un état au suivant. L'état initial de cette machine d'états finis est l'état nommé `Begin`. La figure 2.5 reprend la structure interne de l'état `RangedAttack`, sous forme graphique.

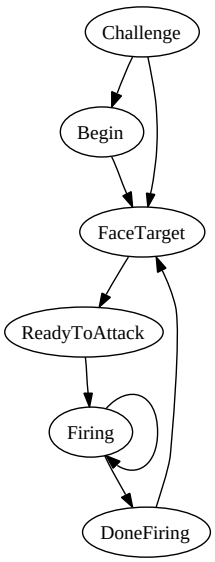

FIG. 2.5 – Tâche élémentaire `RangedAttack` (attaque à distance) : représentation graphique de la structure interne.

2.2.4 Une méta-tâche : attaque

La tâche `RangedAttack` décrit un mode particulier d'attaque : l'attaque à distance. En décrivant les autres tâches élémentaires de l'attaque (charger, se déplacer pour éviter les tirs...), il est possible de construire une méta-tâche d'attaque. Cela est fait par un nouveau processus de **séquencement**, qui ébauche une **hiérarchie** : la méta-tâche d'attaque est une machine d'états finis dont chaque état (comme `RangedAttack`) englobe une machine d'états finis.

Les transitions d'une tâche à l'autre sont décrites localement, dans la tâche source. Elles peuvent être déclenchées soit dans le cours normal de l'exécution des instructions et fonctions de la tâche, soit à la réception d'un évènement alors que le personnage est en train d'exécuter la tâche source.

Sur la figure 2.6, nous reprenons la partie de la tâche `RangedAttack` consacrée aux transitions sortantes, c'est-à-dire au basculement vers d'autres tâches se situant au même niveau que `RangedAttack`.

Les transitions sortantes sont déclenchées par l'appel à `GotoState()`. Ainsi l'appel `GotoState('TacticalMove')` déclenche le basculement vers une tâche de déplacements tactiques. Ce type d'appel se fait soit dans le cours normal des états de la

```
state RangedAttack {
  function StopFiring() {
     Super.StopFiring();
     GotoState('Attacking');
  }
  function EnemyNotVisible() {
     //let attack animation complete
     if ( bComboPaused || bFiringPaused ) return;
     if ( (Weapon == None) || Weapon.bMeleeWeapon|| (FRand() < 0.13) ) {
          bReadyToAttack = true;
          GotoState('Attacking');
          return;
     }
  }
  function KeepAttacking() {
     if ( bComboPaused || bFiringPaused ) {
          if ( TimerRate <= 0.0 ) {
               TweenToRunning(0.12);
               GotoState(NextState, NextLabel);
          }
          if ( bComboPaused ) return;
     }
     if ( (Enemy == None) || (Enemy.Health <= 0) || !LineOfSightTo(Enemy) ) {
          bFire = 0;
          bAltFire = 0;
          GotoState('Attacking');
          return;
     }
     if ( (Weapon != None) && Weapon.bMeleeWeapon ) {
          bReadyToAttack = true;
          GotoState('TacticalMove');
          return;
     }
  }
  function AnimEnd() {
     if ( (Weapon == None) || Weapon.bMeleeWeapon
          || ((bFire == 0) && (bAltFire == 0)) ) {
          GotoState('Attacking');
          return;
     }
     decision = FRand() - 0.2 * skill;
     if ( !bNovice ) decision -= 0.5;
     if ( decision < 0 ) GotoState('RangedAttack', 'DoneFiring');
     else {
          PlayWaiting();
          FireWeapon();
     }
  }
  function SpecialFire() {
     if ( Enemy == None )
          return;
     NextState = 'Attacking';
     NextLabel = 'Begin';
  }

Challenge:
  // ...
Begin:
  if ( Target == None ) {
     Target = Enemy;
     if ( Target == None ) GotoState('Attacking');
  }
  // ...
FaceTarget:
  // ...
ReadyToAttack:
  DesiredRotation = Rotator(Target.Location - Location);
  PlayRangedAttack();
  if ( Weapon.bMeleeWeapon ) GotoState('Attacking');
  Enable('AnimEnd');
Firing:
  if ( Target == None ) GotoState('Attacking');
  TurnToward(Target);
  Goto('Firing');
DoneFiring:
  Disable('AnimEnd');
  KeepAttacking();
  Goto('FaceTarget');
}
```

FIG. 2.6 – Tâche élémentaire `RangedAttack` (attaque à distance) : transitions vers des états extérieurs. Les transitions vers une autre tâche sont marquées en rouge. Les fonctions appelées par le déclenchement d'un évènement sont marquées en vert.

tâche (`Challenge`, `Begin`, `FaceTarget`) tels que nous les avons vus au paragraphe précédent, soit en réaction à certains évènements. Par exemple, la fonction `EnnemyNotVisible()` est appelée quand le personnage ne voit plus son ennemi courant, et peut déclencher un saut vers la tâche `Attacking`. Dans les deux cas, les conditions de transition sont exprimées au moyen de tests logiques portant sur les variables qui décrivent l'état du personnage.

La figure 2.7 reprend sous forme graphique la décomposition de la méta-tâche d'attaque. En dehors de la tâche `RangedAttack` (attaque à distance), les états de cette machine d'états finis sont les tâches `TacticalMove` (déplacement tactique), `Charging` (charge), `StakeOut` (surveiller une zone), `Roaming` (vagabonder), `Retreating` (se replier), `Hunting` (chasser, poursuivre) et `Hold` (tenir une position). Les transitions sortant de la tâche `RangedAttack` sont reprises en rouge sur cette figure.

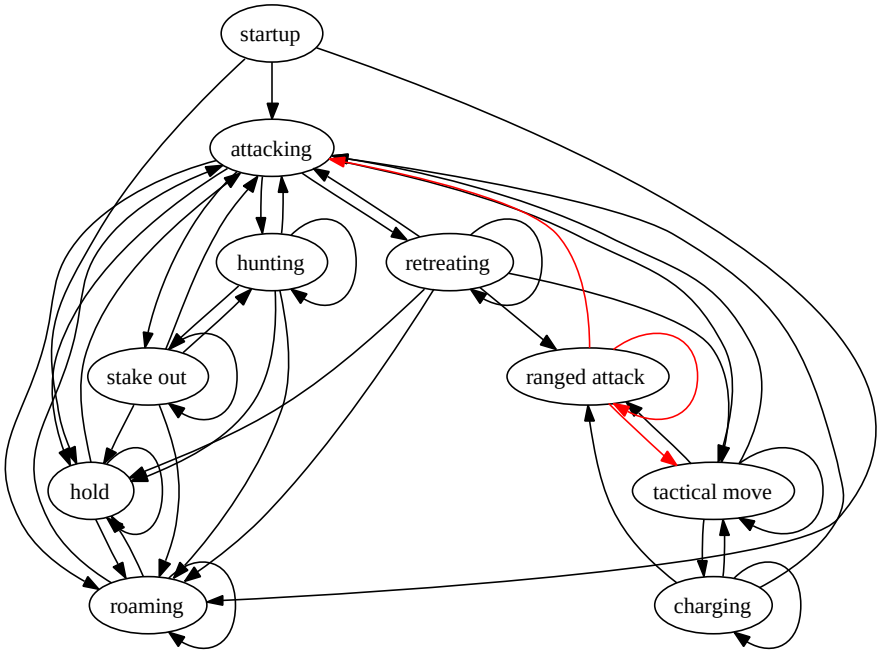

FIG. 2.7 – Décomposition de la méta-tâche d'attaque. Les transitions sortant de la tâche `RangedAttack` sont marquées en rouge.

L'idée de regrouper plusieurs tâches à l'intérieur de la méta-tâche d'attaque est de réduire le nombre de points d'entrée pour des transitions, et donc le nombre de transitions possibles. De plus, il devient possible de tester la méta-tâche d'attaque séparément, avant d'avoir construit le comportement complet (qui intègre la recherche d'armes, de munitions, l'escorte d'autres personnages...).

2.2.5 Le comportement complet

La combinaison de toutes les méta-tâches, par un nouveau séquencement, permet d'aboutir au comportement complet. La figure 2.2 (présentée plus haut) reprend ce modèle complet, où les méta-tâches et les tâches sont éclatées. La tâche `RangedAttack` est encadrée et grossie. Il est pertinent de considérer cette forme éclatée plutôt qu'une forme regroupant tâches et méta-tâches, dans la mesure où c'est la forme réelle que prend la machine d'états finis dans le programme *UnrealScript*. En effet, au-delà de l'organisation théorique que nous avons présentée, le programme ne reflète pas formellement l'organisation des méta-tâches, et permet à une tâche de sauter vers n'importe quel état intermédiaire d'une autre tâche (pas seulement l'état `Begin`).

2.2.6 Discussion

Pourquoi des machines d'états finis ?

Les machines d'états finis sont un formalisme ancien et bien connu pour la modélisation.

Leur concept fondamental pour la synthèse de comportements est la **mise en séquence de tâches** élémentaires connues, pour créer une tâche de plus haut niveau. Quand une décision doit être prise (régulièrement, ou quand les conditions sensorielles l'imposent), la tâche suivante à exécuter est sélectionnée selon la tâche en cours et les conditions sensorielles.

Par exemple, attaquer un adversaire peut se décomposer comme le séquencement de tâches élémentaires comme s'approcher de l'adversaire, s'en éloigner, choisir une arme, et se déplacer sur le côté. Ce processus, décrit dans un langage de programmation générique, peut être modélisé trivialement par une machine d'états finis.

La mise en séquence de tâches dans ce formalisme peut se faire à plusieurs niveaux, ce qui permet de créer des **hiérarchies de comportements**. Il suffit de mettre en séquence des tâches qui sont elles-mêmes des mises en séquence d'autres tâches de plus bas niveau. Cela conduit à gérer des machines d'états finis imbriquées.

Par exemple, le comportement d'un personnage autonome complet dans *Unreal Tournament* ferait la mise en séquence de plusieurs tâches de haut niveau, comme fuir et attaquer, qui peuvent elles-mêmes être décrites comme des tâches complexes issues d'une mise en séquence.

En abordant la description de comportements comme la construction de machines d'états finis imbriquées, plutôt que comme un programme écrit directement dans un langage de programmation générique, cette méthode apporte un cadre formel pour mieux maîtriser la construction du comportement. En particulier, il est possible de représenter un comportement sous forme d'un graphe annoté (comme sur la figure 2.2).

Cependant, il reste très facile d'intégrer diverses techniques à ce formalisme. Ainsi, il est possible de faire appel à des fonctions arbitraires (par exemple, une routine de planification) dans les tâches élémentaires. Il est aussi possible de mettre en œuvre des méthodes de décision variées (réseaux de neurones...) dans la logique de séquencement de tâches.

Par exemple, notre personnage autonome dans *Unreal Tournament* connaîtra une tâche de recherche de munitions. À cet effet, il est possible d'appeler directement un algorithme basé sur

A* permettant de calculer un chemin planifié pour retrouver les boîtes de munitions.

Réponses d'*UnrealScript* aux questions

– **(Q1a : structure des paramètres)**

Une tâche simple est décrite par un script. Cette forme de description est puissante mais aucun paramètre du comportement n'est isolable. La structure du programme et les valeurs d'un certain nombre de variables globales tiennent lieu de paramètres. S'ils sont bien identifiés, ils ne sont pas localisés. Ils sont également peu structurés.

La méthode de mise en séquence de tâches élémentaires repose sur le déclenchement scripté de transitions. Les déclenchements se font sur la base de décisions programmées de manière classique. Il faut noter que l'utilisation de fonctions de réactions à certains évènements tend à fractionner la charge de décision, et à la rendre plus facilement gérable. Cependant, quand le nombre de transitions augmente à partir d'un état, la charge de maintenance de la logique de décision basée sur des si-alors-sinon devient très grande. En effet, elle dérive soit vers une imbrication profonde de si-alors-sinon, soit vers une longue séquence de tests logiques. Les paramètres du séquencement sont donc assez bien identifiés, mais leur maîtrise devient problématique quand le nombre d'états séquencés augmente.

Quant au modèle de hiérarchisation, son avantage est de briser la complexité des comportements en la répartissant verticalement (alors que le simple séquencement la repartit horizontalement). Il permet un développement réellement ascendant, en construisant des comportements du simple au complexe, de bas en haut de la hiérarchie. Les paramètres du comportement, bien que dilués ultimement dans un grand script, sont structurés, et regroupés par blocs cohérents.

Finalement, les paramètres des comportements décrits par *UnrealScript* sont dilués dans le script, et ne sont pas isolables. Cela empêche de considérer directement un comportement comme des données : le sauver sur disque, le charger, le transmettre sur le réseau, l'apprendre...

– **(Q1b : paramètres et intuition)**

L'utilisation de scripts pour la description de l'état parle à l'intuition du programmeur. Mais dans la mesure où ajuster le comportement consiste à modifier un programme, il n'est pas envisageable de laisser cette tâche à un non-spécialiste.

De même, les paramètres du séquencement sont des conditions logiques portant sur les variables d'un script. Leur valeur intuitive est la même que celle du script lui-même, et il n'est pas envisageable de les confier à un non spécialiste.

La hiérarchisation correspond à un découpage intuitif des tâches en sous-tâches, récursivement. Ce découpage peut se faire sur le papier par des concepteurs de comportements ne programmant pas. Par contre, il est difficile pour un non spécialiste de modifier un comportement existant : la structure hiérarchique se trouve masquée par sa propre implémentation, et son abstraction est perdue.

La méthodologie de construction de machines d'états finis hiérarchiques est bien connue. Elle apporte de la facilité à l'activité de construction de comportements vue comme la programmation d'un système complexe. Cependant, c'est là aussi sa limite : un comportement dans *Unreal Tournament* ne peut être plus intuitif qu'un programme de plusieurs milliers de lignes.

– **(Q1c : apprentissage)**

Il n'est pas possible d'établir de méthodologie générale d'apprentissage d'un comportement élémentaire décrit sous cette forme.

Les transitions d'état à état sont elles aussi difficilement apprenables. Il est cependant envisageable d'utiliser un formalisme adapté supplémentaire, comme les arbres de décision, pour apprendre ces transitions.

L'apprentissage de la hiérarchie peut se faire sous les mêmes conditions que celui du séquencement : l'adoption d'un formalisme restrictif de gestion des décisions. En général, il n'est pas possible.

La description des machines d'états finis en *UnrealScript* ne peut permettre telle quelle l'apprentissage des modèles de comportements. Cet apprentissage serait nécessaire tant au niveau des tâches élémentaires, que des transitions entre tâches.

– **(Q2a : complexité en mémoire, Q2b : complexité en temps de calcul)**

Les besoins en mémoire et temps de calcul de ce genre de comportements simples ne peuvent être déterminés en général, et doivent être envisagés au cas par cas. Cependant, leur évaluation pour un programme donné est possible aux moyen de techniques de calcul et de mesure classiques.

L'utilisation d'un langage de script pour décrire le séquencement des états permet une représentation efficace de la table de transitions de l'automate : cette représentation est creuse. L'utilisation de conditions logiques garantit de plus des performances en temps de calcul prédictibles, et suffisantes dans la plupart des cas.

L'idée même de hiérarchie conduit à briser l'augmentation du nombre de transitions avec celle du nombre d'états. Cela se traduit par des décisions mieux maîtrisées, et efficaces pour les mêmes raisons que pour le mécanisme de séquencement.

L'utilisation de la logique classique et de simples machines d'états finis permet de réaliser des programmes légers autant en mémoire qu'en temps de calcul. Ils n'utilisent que peu de données, dans la mesure où la description de la structure des machines d'états finis est faite par du code. De plus, les méthodes de décision (séquencement) qu'ils proposent sont simples et efficaces.

Finalement, il faut constater l'efficacité pratique d'*UnrealScript* pour construire des comportements : les personnages autonomes d'Unreal Tournament sont un succès. Nous devons conclure que les qualités de cette méthode de description de comportement sont essentielles et ne peuvent être sacrifiées gratuitement.

Au-delà d'*UnrealScript*

Dans la suite de notre discussion, nous présentons un modèle de machines d'états finis qui tend à répondre aux manquements du modèle *UnrealScript*, sans se départir de ses principaux avantages de simplicité et d'efficacité.

Nous présenterons d'abord la programmation avec ce modèle :

– quels sont ses éléments de base ;
– comment il est possible d'assembler des tâches à partir de ces éléments ;
– comment ces tâches peuvent être séquencées pour produire un comportement complet.

Nous nous pencherons ensuite sur l'avantage primordial de notre modèle par rapport au modèle classique utilisé par *Unreal Tournament* : sa capacité d'apprentissage.

Dans ces deux parties, nous construirons un comportement pour un personnage autonome capable de jouer contre des personnages natifs d'*Unreal Tournament*, et contre d'autres personnages contrôlés par des humains. Nous aurons le souci de traiter au fur et à mesure nos questions de problématique.

Chapitre 3

Programmer un comportement

3.1 Vocabulaire : comportement, tâche simple, séquencement

Nous appelons **comportement** un modèle permettant pour un personnage de répondre à la question : que faire, sachant ce que je perçois du monde et de mon état ?

Une **tâche simple** est un comportement programmé d'un bloc, comme une entité logique unique. Elle peut être exécutée dans des circonstances spécifiques, pour tendre à certains buts particuliers. Elle est réactive, et ne se base que sur des variables observables (pas de variables internes au modèle).

La **mise en séquence** de tâches consiste à exécuter plusieurs tâches les unes après les autres, en en choisissant l'ordre selon l'état du monde et du personnage, en vue de tendre à certains buts. Chacune des tâches peut être une tâche simple, ou une tâche issue d'une mise en séquence de plus bas niveau.

3.2 Principes de la programmation bayésienne

3.2.1 Le formalisme de la programmation bayésienne

La programmation bayésienne est un cadre formel de modélisation s'appuyant sur le calcul des probabilités. Il prend en compte des variables sur des domaines discrets finis.

Par exemple, nous pouvons considérer la variable A, prenant des valeurs dans l'intervalle entier $[\![0; 2]\!]$, et la variable B prenant des valeurs dans $[\![-1; 1]\!]$.

Nous pouvons nous intéresser à la distribution de probabilité sur la variable A :

$$P(A)$$

Nous pouvons nous intéresser à la probabilité de cette distribution pour une valeur précise

de A :

$$P([A = 1]) \;=\; 0.2$$

Nous pouvons considérer la distribution de probabilité sur la conjonction des variables A et B :

$$P(A\,B)$$

Nous pouvons aussi considérer la distribution de probabilité conditionnelle de la variable A sachant B :

$$P(A \mid B)$$

Cette distribution conditionnelle correspond à une famille de distributions : pour chaque valeur de B, elle donne une distribution sur A. Elle est définie par la règle du produit :

$$P(A\,B) \;=\; P(B)\,P(A \mid B) \;=\; P(A)\,P(B \mid A)$$

Enfin, la règle de marginalisation nous permet de lier $P(A)$ à $P(A\,B)$:

$$P(A) \;=\; \sum_{B} P(A\,B)$$

Sur cette base théorique très simple, la programmation bayésienne permet de construire des modèles liant plusieurs variables au moyen de distributions de probabilités conditionnelles.

Décrire un programme bayésien [Lebeltel 99] [Lebeltel 00b] [Lebeltel 00a] [Lebeltel 04a] [Lebeltel 04b] consiste à :

- choisir les **variables pertinentes** du problème ;
- **décomposer la distribution conjointe**, c'est-à-dire la distribution de probabilité de la conjonction de toutes les variables ;
- **assigner une forme** à chacun des facteurs de cette décomposition (par exemple : une table de probabilité, une gaussienne dont les paramètres sont à déterminer, une sigmoïde...) ;
- **instancier les paramètres** éventuels de chacune de ces formes.

Le programme écrit, nous pouvons l'utiliser en lui posant des **questions**. La résolution d'une question s'appelle **inférence**. Elle s'appuie sur la règle du produit et celle de marginalisation (énoncées plus haut) : il est possible de calculer à partir de la décomposition une distribution conditionnelle quelconque portant sur une sous-partie des variables du programme. La conduite de ce calcul (en particulier les simplifications et approximations qui permettent de traiter des modèles de grande taille) est un sujet important de recherches, qui ne nous occupe pas ici. Nous nous focalisons sur la construction de modèles pour lesquels l'inférence peut être faite de manière exacte et rapide.

Nous traiterons d'exemple très simples de programmes bayésiens, avant de montrer comment construire des comportements fonctionnels pour un personnage jouant à *Unreal Tournament*.

3.2.2 Le formalisme de la programmation bayésienne : comportement de détection du danger

Nous pouvons, sur les principes esquissés ci-dessus, programmer (figure 3.1) un comportement très simple de détection du danger pour un personnage autonome qui joue à *Unreal Tournament*. Ce comportement va consister à tourner sur soi-même très vite, pour essayer de percevoir des ennemis éventuels.

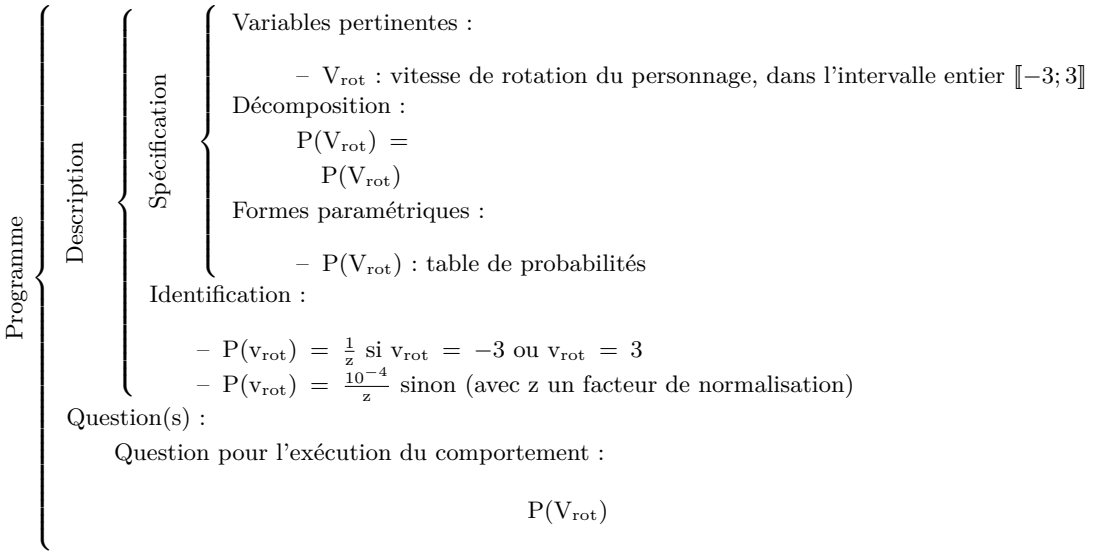

FIG. 3.1 – Comportement de détection du danger

Ce comportement ne porte que sur une variable : la vitesse de rotation du personnage. $v_{rot} = -3$ signifie tourner vite vers la droite, et $v_{rot} = 3$ tourner vite vers la gauche. La vitesse de translation du personnage est maintenue constante : il court tout le temps.

Puisque nous n'avons qu'une seule variable, la décomposition de la distribution conjointe se réduit trivialement au seul facteur $P(V_{rot})$, que nous décrivons comme une table de probabilités.

Nous posons que les vitesses de rotation extrêmes sont chacune 10^4 fois plus probables que n'importe laquelle des autres vitesses de rotation. Le facteur de normalisation z est calculé en tenant compte de la normalisation de la table, qui comporte deux valeurs extrêmes et cinq valeurs intérieures :

$$2 \times \frac{1}{z} + 5 \times \frac{10^{-4}}{z} = 1$$

La seule question que nous pouvons poser à notre modèle est : quelle vitesse de rotation choisir ?

$$P(V_{rot}) \, ?$$

3.2.3 Comportement de suivi de cible

Nous pouvons, sur le même modèle, construire un comportement de suivi de cible (figure 3.3). Notre personnage va pouvoir être attiré par un élément statique (un point de navigation, un objet...) ou mobile (un autre personnage) de l'environnement. Il perçoit l'orientation de cet attracteur, relativement à sa propre direction de déplacement (figure 3.2).

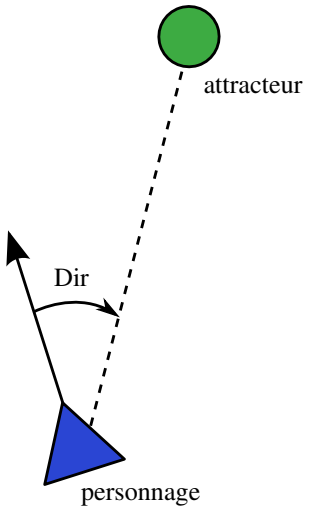

FIG. 3.2 – Un personnage (triangle bleu) perçoit l'orientation relative Dir d'un attracteur (disque vert).

Par rapport au comportement de détection de danger, nous avons ajouté une variable sensorielle : la direction de la cible Dir. Le personnage perçoit les cibles dont la direction est dans l'intervalle réel $[-\pi;\ \pi]$. Nous discrétisons cet intervalle en 9 valeurs entières, en assignant à la variable Dir l'intervalle entier $[\![-4; 4]\!]$.

Nous décomposons la distribution conjointe de façon à pouvoir exprimer $P(V_{rot} \mid Dir)$: la vitesse de rotation à adopter en fonction de la direction de la cible.

Pour une direction d'attraction donnée, la probabilité de la vitesse de rotation $P(V_{rot} \mid Dir)$ doit être forte pour la valeur de V_{rot} qui ramène le personnage instantanément face à sa cible. Elle est faible pour les autres valeurs. Cette idée correspond à la densité de probabilité continue représentée sur la partie gauche de la figure 3.4, pour $Dir = -\frac{\pi}{3}$. Pour construire une table de probabilités à partir de cette densité, il suffit d'intégrer cette densité sur chacun des intervalles centrés sur les valeurs entières de V_{rot}. Pour $Dir = -\frac{\pi}{3}$, cela donne la distribution représentée sur la partie droite de la figure 3.4.

Nous avons l'ébauche d'un comportement générique : nous pouvons l'instancier pour différentes cibles, et combiner les consignes obtenues. Nous allons maintenant dresser un panorama général des techniques de construction de comportements en programmation bayésienne.

Programme

Description

Spécification

Variables pertinentes :

- V_{rot} : vitesse de rotation du personnage, dans $[\![-3;3]\!]$
- Dir : direction relative de la cible, dans $[\![-4;4]\!]$

Décomposition :

$$P(V_{rot}\ Dir) = P(Dir)\ P(V_{rot} \mid Dir)$$

Formes paramétriques :

- P(Dir) : inconnu
- $P(V_{rot} \mid Dir)$: table de probabilités

Identification :

$P(V_{rot} \mid Dir)$: table de probabilité établie à partir d'une densité de probabilités sur le modèle de la figure 3.4.

Question(s) :

Question pour l'exécution du comportement :

$$P(V_{rot} \mid Dir)$$

FIG. 3.3 – Comportement de suivi de cible

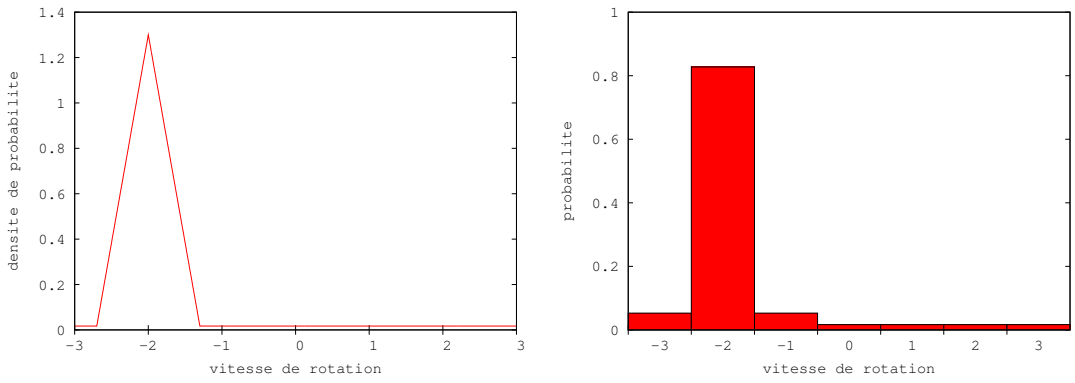

FIG. 3.4 – Densité de probabilité continue pour une consigne en vitesse de rotation pour un attracteur en Dir $= -\frac{\pi}{3}$, et table de probabilité associée.

3.2.4 Un panorama partiel des comportements réactifs en programmation bayésienne

Nous avons esquissé le formalisme et la méthode de base que nous utilisons pour les comportements de notre personnage Unreal Tournament. La panoplie de méthodes élémentaires disponibles dans le cadre de la programmation bayésienne est très étendue. Nous en énumérons brièvement quelques unes ici.

Exprimer des consignes prescriptives et proscriptives

Nous avons montré comment exprimer une consigne d'attraction, c'est-à-dire une consigne prescriptive. Il est possible d'exprimer directement une consigne proscriptive, qui interdit certaines valeurs motrices au lieu de les conseiller. Cela peut être utile en particulier pour l'écriture d'un modèle d'évitement d'obstacle [Koike 03] (problème tout aussi important en robotique mobile que pour la programmation de personnages synthétiques).

L'idée essentielle est de fournir une consigne d'évitement d'obstacle qui interdit les déplacements conduisant à une collision, mais autorise de manière uniforme tous les autres. Cette consigne peut ensuite être combinée avec une autre consigne, qui indique la direction dans laquelle le robot veut se déplacer sans tenir compte des obstacles.

Fusion bayésienne naïve

Le schéma de fusion bayésienne naïve est un dispositif très commun, permettant de combiner une série d'indications S_0 ... S_n en une seule variable V les résumant (voir figure 3.5). L'hypothèse essentielle en est que sachant la variable pivot V, chaque S_i est indépendant des autres S_j.

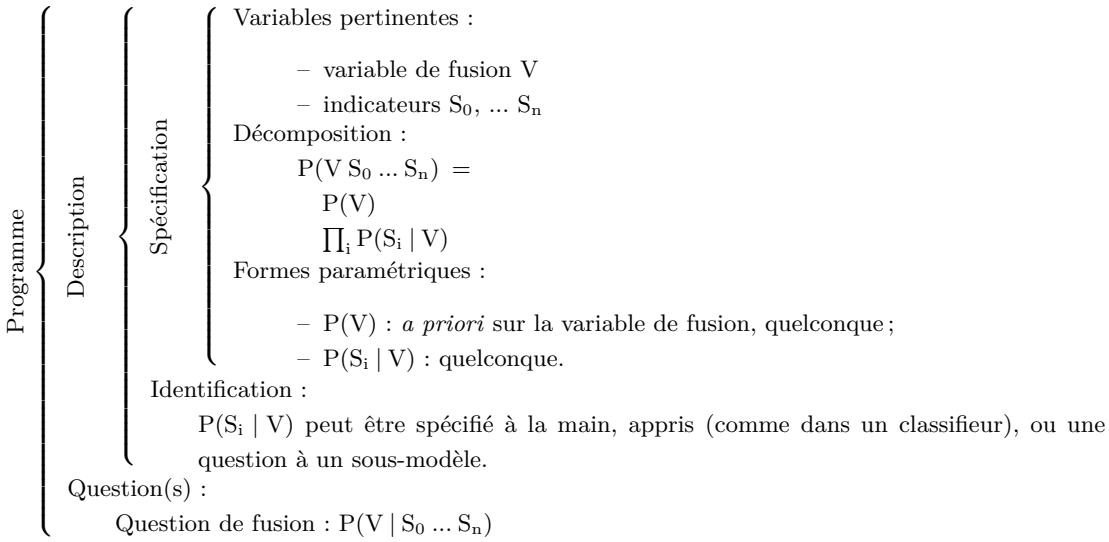

FIG. 3.5 – Fusion bayésienne naïve

Fusion sensorielle naïve Dans un schéma de fusion sensorielle naïve, la variable pivot V est un capteur virtuel (comme la direction d'un objet Dir), synthétisant les informations de plusieurs capteurs réels S_0 ... S_n. Les facteurs $P(S_i \mid V)$ peuvent alors être spécifiés directement à la main, ou appris.

Par exemple, nous pouvons générer un capteur virtuel pour notre personnage *Unreal Tournament*, qui résume la dangerosité d'un ennemi (figure 3.6).

La table 3.1 correspond à P(Arme|Danger) telle que nous pourrions la spécifier pour construire notre capteur virtuel. La colonne bleue correspond à $P(\text{Arme} \mid [\text{Danger} = \text{faux}])$: distribution de l'arme de l'ennemi sachant qu'il n'y a pas de danger. La colonne grise correspond à $P(\text{Arme} \mid [\text{Danger} = \text{vrai}])$: distribution de l'arme de l'ennemi sachant qu'il y a du danger. x et x' sont des probabilités que nous ne spécifions pas explicitement, mais qui sont contraintes par le fait que chaque colonne doit se sommer à un. Par exemple le x de la colonne bleue vaut $x = 1 - 0.1 - 0.01 = 0.89$.

Dans cette table, nous avons transcrit les hypothèses suivantes :

– quand il n'y a pas de danger (colonne bleue), nous savons que l'arme de l'ennemi n'est pas forte, et sans doute pas moyenne ;
– quand il y a du danger (colonne grise), nous ne savons rien sur l'arme de l'ennemi (le danger peut être dû à son arme, ou à sa proximité, ou à son niveau de santé).

Programme

Description

Spécification

Variables pertinentes :

- Danger (booléen) : dangerosité de l'ennemi ;
- Arme (faible, moyenne ou forte), Distance (dans $[0; 5]$), Santé (faible, moyenne ou forte) : caractéristiques observables de l'ennemi.

Décomposition :

P(Danger Arme Distance Santé) =
P(Danger)
P(Arme | Danger)
P(Distance | Danger)
P(Santé | Danger)

Formes paramétriques :

- P(Danger) : uniforme ;
- P(Arme | Danger) : table de probabilités, donnée en exemple sur la table 3.1 ;
- P(Distance | Danger) : table de probabilités ;
- P(Santé | Danger) : table de probabilités.

Identification :

Nous pouvons spécifier chacune des tables de probabilités à la main.

Question(s) :

Question de fusion : P(Danger | Arme Distance Santé)

FIG. 3.6 – Fusion sensorielle : dangerosité d'un ennemi

	Faux	Vrai
faible	x	x'
moyenne	0.1	x'
forte	0.01	x'

TAB. 3.1 – P(Arme | Danger)

Fusion motrice naïve Dans un schéma de fusion motrice, la variable pivot V est une variable motrice (comme une vitesse de rotation V_{rot}), et les variables S_i sont des variables sensorielles. Chaque facteur $P(S_i \mid V)$ est une question posée à un sous-modèle capable de relier une partie des capteurs à la commande motrice. Le schéma permet de générer une commande motrice unique à partir des consignes données par les sous-modèles.

Utilisation d'un état : filtres bayésiens

Les filtres bayésiens sont les modèles qui au cadre purement réactif ajoutent une variable d'état. Cet état résume les informations qui sont propagées dans le modèle, d'un pas de temps à l'autre.

Utiliser un état permet d'introduire une logique temporelle dans le programme. Cela permet également une forme de mémoire locale au comportement. Avec le temps, et la mémoire, il devient

possible de décrire des comportements sortant du cadre réactif. Une illustration élaborée peut en être trouvée dans [Koike 05].

Les modèles de séquencement que nous développerons dans la suite utilisent cette technique d'état. Nous reviendrons alors sur une description plus formelle du filtrage.

Modéliser l'interaction, inférer le comportement : les cartes bayésiennes

L'idée des cartes bayésiennes [Diard 03b] [Diard 03c] [Diard 03d] [Diard 04] [Diard 05] pour le contrôle d'un personnage est de modéliser la dynamique du système (personnage, monde), et d'envoyer les commandes motrices qui correspondent le mieux à l'état du monde recherché. De cette façon, le personnage est piloté en spécifiant les circonstances sensorielles (l'état du monde) qui lui sont favorables. C'est le processus d'inférence bayésienne qui détermine les commandes motrices les meilleures pour conduire à ces circonstances.

Écrire un programme dans ce paradigme ne revient plus à décrire des heuristiques directes pour le contrôle du personnage (*comment agir, sachant les observations ?*), mais à modéliser une partie du monde. Cette modélisation, et l'apprentissage des facteurs de sa décomposition, permet au prix d'une complexité de programmation accrue, une flexibilité importante.

Par exemple, la spécification d'un comportement peut consister à spécifier que le personnage doit rechercher les situations où son niveau de santé est fort. La prise en compte d'un modèle de l'évolution du monde (quelles actions engendrent quel état, et en particulier quel niveau de santé) et de l'état courant du monde (le niveau de santé et les autres variables pertinentes de l'état du monde, comme la position dans l'environnement) permet, par inférence, de remonter aux commandes motrices les mieux à même de conduire à un niveau de santé fort.

[Simonin 05] présente un exemple de l'utilisation de cartes bayésiennes pour les comportements d'un petit robot mobile. Les modèles de perception et de déplacement qui y sont développés sont proches de ceux que nous utilisons pour notre personnage autonome.

3.2.5 Un exemple de modèle complexe : sélection de l'action avec focalisation de l'attention

La figure 3.7 montre la décomposition de la distribution conjointe utilisée dans [Koike 05]. Il s'agit d'un modèle de comportement pour un robot autonome. Il est composé de plusieurs tâches modélisées séparément qui sont combinées pour former le comportement global. Il a pour vocation de prendre des décisions motrices en fonction de l'ensemble des observations passées. Cela est réalisé au moyen de plusieurs filtres bayésiens imbriqués, et d'une technique de focalisation de l'attention qui permet de réduire les informations sensorielles à prendre en compte. Cela permet de briser la complexité de représentation de l'état du monde et du robot, la complexité de modélisation de son évolution, et la complexité de prise en compte des informations sensorielles.

Ce modèle répond au défi de la construction de comportements complexes et réalistes ; mais sa complexité le rend difficile à comprendre. En particulier, il n'est pas possible ici d'en détailler tous les mécanismes, qui sont nombreux et imbriqués. Le lecteur intéressé est invité à se réfèrer à [Koike 05].

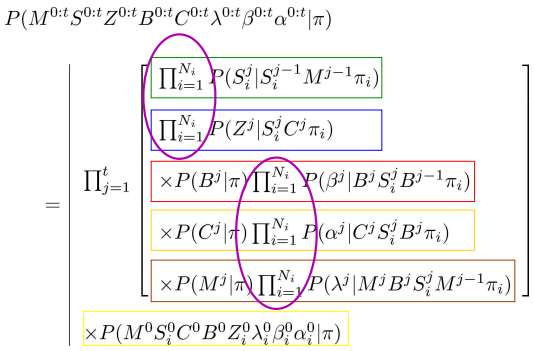

FIG. 3.7 – Décomposition de la distribution conjointe pour le modèle de comportement de [Koike 05]. Ce modèle est basé sur la combinaison de plusieurs filtres bayésiens. Il a l'ambition de prendre une décision motrice basée sur l'ensemble des états sensori-moteurs passés du robot.

Nous sommes en quête d'une méthodologie simple qui permettra d'approcher les mêmes effets de manière plus rapide et plus facile. Nous développons donc dans la suite deux techniques basées sur la programmation bayésienne : la fusion par cohérence améliorée, et le séquencement par programmation inverse.

3.3 Programmer une tâche simple : fusion par cohérence améliorée

La fusion par cohérence (aussi appelée fusion avec diagnostic) [Pradalier 03b] [Pradalier 03a] [Lorieux 03] [Pradalier 04] est une technique de combinaison de consignes. La figure 3.8 décrit son mécanisme général, sur le cas de la fusion de consignes de vitesses de rotation.

Ce modèle repose sur l'idée que des sous-modèles fournissent une liste de consignes en vitesse de rotation, sous la forme de distributions de probabilités :

$$\mathrm{P(V^0_{rot})} \ldots \mathrm{P(V^n_{rot})}$$

Nous voulons déterminer une commande $\mathrm{P(V_{rot})}$ qui combine ces consignes. Cela est réalisé en décrivant la relation de chaque consigne à la commande, à l'aide des variables de cohérence :

$$\mathrm{M^0} \ldots \mathrm{M^n}$$

La variable $\mathrm{M^i}$ est une variable binaire. $\mathrm{M^i} = 1$ signifie que la commande $\mathrm{V_{rot}}$ est cohérente avec la consigne $\mathrm{V^i_{rot}}$. A l'inverse, $\mathrm{M^i} = 0$ signifie que la commande $\mathrm{V_{rot}}$ est en désaccord avec la consigne $\mathrm{V^i_{rot}}$.

La question posée à ce modèle est celle de la commande $\mathrm{V_{rot}}$ sachant la cohérence avec chaque consigne. La consigne $\mathrm{V^i_{rot}}$ peut être prescriptive (la commande doit être en accord avec cette consigne), et nous posons la question avec $[\mathrm{M^i} = 1]$. Elle peut être proscriptive (la commande ne doit pas être proche de la consigne), et nous posons la question avec $[\mathrm{M^i} = 0]$.

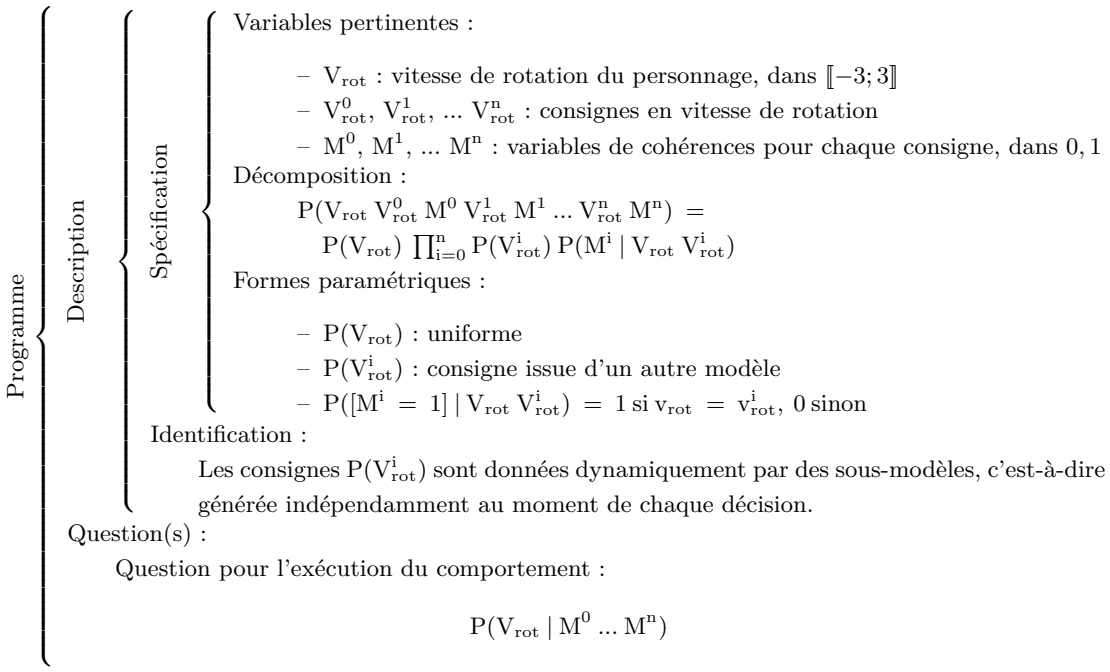

FIG. 3.8 – Fusion par cohérence : exemple de fusion de consignes en vitesse de rotation

Nous utilisons ces deux manières de combiner des consignes dans les comportements de notre personnage *Unreal Tournament* ; elles contribuent à faire de la fusion par cohérence un outil générique de construction de comportements simples.

3.3.1 Fusion de consignes prescriptives : comportement d'exploration

Nous construisons un comportement d'exploration en combinant plusieurs consignes de suivi de cible (figure 3.3) à l'aide d'un schéma de fusion par cohérence (figure 3.8).

Notre personnage est muni de 4 capteurs, indiquant la direction de 4 points proches de l'environnement, pertinents pour le comportement d'exploration : Dir^0, Dir^1, Dir^2, Dir^3. Pour un personnage *Unreal Tournament*, ces points sont les nœuds d'un graphe de navigation décrivant l'environnement courant. La figure 3.9 montre un tel graphe. Deux nœuds y sont liés s'il est possible de se déplacer directement de l'un à l'autre sans buter sur un obstacle (trou, mur). Nos quatres capteurs correspondront donc aux quatres point visibles et atteignables les plus proches du personnage.

Pour chacun de ces capteurs, nous instancions un comportement de suivi de cible, qui répond à la question :

$$P(V_{rot} \mid [Dir = dir^i])$$

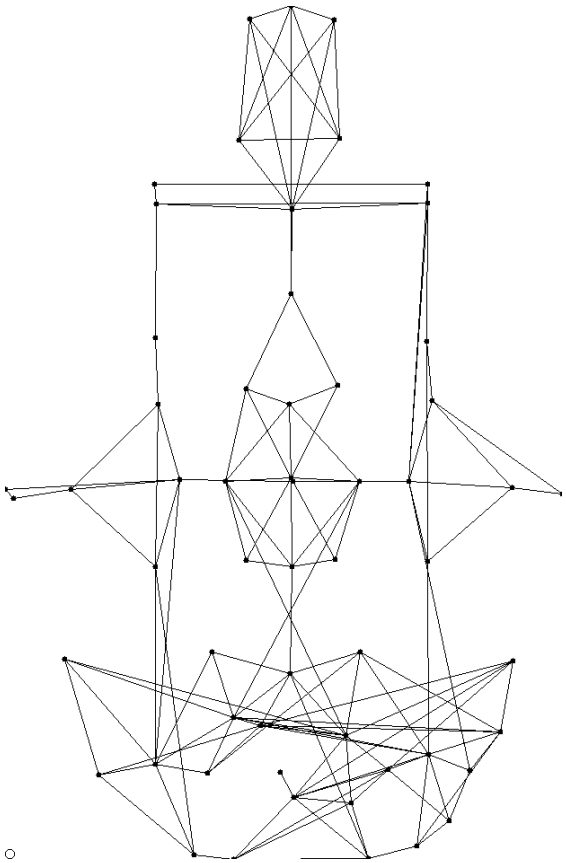

FIG. 3.9 – Graphe de navigation pour le niveau *DM-Stalwart* d'*Unreal Tournament*.

Cette question est la consigne de vitesse de rotation, d'après le comportement de suivi de la cible i. L'abstraction de ces 4 consignes nous donne quatre consignes distributions :

$$\mathrm{P}(\mathrm{V}^0_{\mathrm{rot}}),\ \mathrm{P}(\mathrm{V}^1_{\mathrm{rot}}),\ \mathrm{P}(\mathrm{V}^2_{\mathrm{rot}}),\ \mathrm{P}(\mathrm{V}^3_{\mathrm{rot}})$$

Nous réalisons la fusion de ces consignes prescriptives selon le schéma de fusion par cohérence, en cherchant à ce que notre commande soit proche de chacune des consignes. Nos variables de cohérence M^0, M^1, M^2, M^3 sont donc toutes mises à la valeur 1.

Notre schéma de fusion nous donne la réponse à la question :

$$\begin{aligned}
&\mathrm{P}(\mathrm{V}_{\mathrm{rot}} \mid [\mathrm{M}^0 = 1]\,[\mathrm{M}^1 = 1]\,[\mathrm{M}^2 = 1]\,[\mathrm{M}^3 = 1]) \\
&= \frac{1}{\mathrm{Z}} \sum_{\mathrm{V}^{0:3}_{\mathrm{rot}}} \mathrm{P}(\mathrm{V}_{\mathrm{rot}}) \prod_{\mathrm{i}=0}^{3} \mathrm{P}(\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}})\, \mathrm{P}([\mathrm{M}^{\mathrm{i}} = 1] \mid \mathrm{V}_{\mathrm{rot}}\, \mathrm{V}^{\mathrm{i}}_{\mathrm{rot}}) \\
&= \frac{1}{\mathrm{Z}}\, \mathrm{P}(\mathrm{V}_{\mathrm{rot}}) \prod_{\mathrm{i}=0}^{3} \sum_{\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}}} \mathrm{P}(\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}})\, \mathrm{P}([\mathrm{M}^{\mathrm{i}} = 1] \mid \mathrm{V}_{\mathrm{rot}}\, \mathrm{V}^{\mathrm{i}}_{\mathrm{rot}}) \\
&= \frac{1}{\mathrm{Z}} \underbrace{\mathrm{P}(\mathrm{V}_{\mathrm{rot}})}_{=\mathrm{U}} \prod_{\mathrm{i}=0}^{3} \Bigg[\mathrm{P}([\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}} = \mathrm{v}_{\mathrm{rot}}])\, \underbrace{\mathrm{P}([\mathrm{M}^{\mathrm{i}} = 1] \mid \mathrm{V}_{\mathrm{rot}}\, [\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}} = \mathrm{v}_{\mathrm{rot}}])}_{=1} \\
&\qquad\qquad + \sum_{\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}} \neq \mathrm{v}_{\mathrm{rot}}} \mathrm{P}(\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}})\, \underbrace{\mathrm{P}([\mathrm{M}^{\mathrm{i}} = 1] \mid \mathrm{V}_{\mathrm{rot}}\, \mathrm{V}^{\mathrm{i}}_{\mathrm{rot}})}_{=0} \Bigg] \\
&= \frac{1}{\mathrm{Z}'} \prod_{\mathrm{i}=0}^{3} \mathrm{P}(\mathrm{V}^{\mathrm{i}}_{\mathrm{rot}} = \mathrm{v}_{\mathrm{rot}})
\end{aligned} \tag{3.1}$$

Dans ce calcul, Z et $\mathrm{Z}' = \frac{1}{\mathrm{Z}\,|\mathrm{V}_{\mathrm{rot}}|}$ correspondent à des facteurs de normalisation, qui assurent que la distribution obtenue est normalisée sur la variable $\mathrm{V}_{\mathrm{rot}}$.

Notre question obtient donc la réponse suivante : la commande est directement le produit des consignes.

Les figures suivantes illustrent ce processus de fusion de consignes prescriptives par cohérence. Pour la clarté de celles-ci, nous ne considérons que trois consignes, au lieu des quatre que notre personnage utilise en réalité pour son comportement d'exploration. Les consignes, toutes deux générées par le programme d'attraction présenté plus haut, sont : $\mathrm{P}(\mathrm{V}^0_{\mathrm{rot}})$, $\mathrm{P}(\mathrm{V}^1_{\mathrm{rot}})$ et $\mathrm{P}(\mathrm{V}^2_{\mathrm{rot}})$ (figure 3.10). Le résultat de la fusion est la commande $\mathrm{P}(\mathrm{V}_{\mathrm{rot}})$ (figure 3.11).

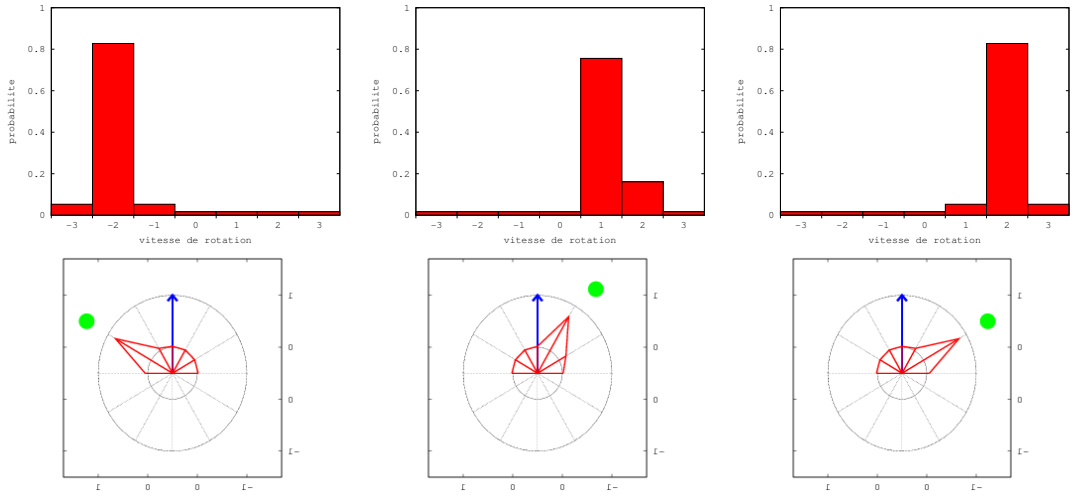

FIG. 3.10 – Représentations des histogrammes des consignes d'attractions en coordonnées cartésiennes (haut) et polaires (bas). À **gauche** : $P(V^0_{rot})$ – attracteur en Dir $= -\frac{\pi}{3}$; au **centre** : $P(V^1_{rot})$ – attracteur en Dir $= \frac{\pi}{5}$; à **droite** : $P(V^2_{rot})$ – attracteur en Dir $= \frac{\pi}{3}$. Les directions des attracteurs sont indiqués par des disques verts. Sur les figures en coordonnées polaire, le petit cercle indique la probabilité zéro, et le grand cercle la probabilité un ; le personnage regarde vers le haut (flèche bleue).

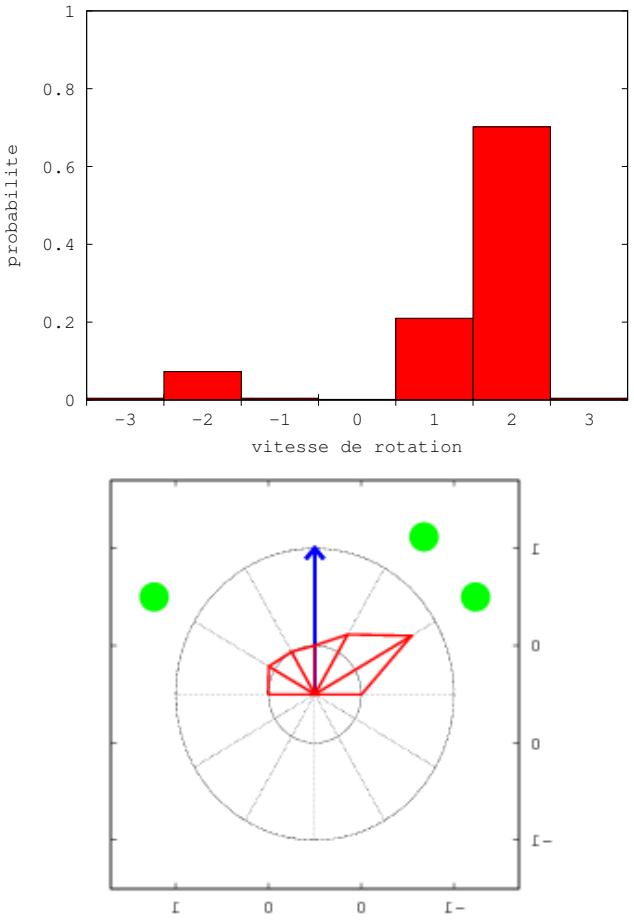

FIG. 3.11 – Commande P(V_{rot}), résultat de la fusion des consignes prescriptives.

3.3.2 Fusion de consignes proscriptives : comportement de fuite

Le schéma de fusion par cohérence peut aussi être utilisé pour spécifier des consignes proscriptives, c'est-à-dire des zones indésirables de l'espace de commande. Il suffit pour cela de considérer que nous avons pour notre comportement d'exploration posé la question :

$$\mathrm{P}(\mathrm{V_{rot}} \mid [\mathrm{M}^0 = 1]\,[\mathrm{M}^1 = 1]\,[\mathrm{M}^2 = 1]\,[\mathrm{M}^3 = 1]) \tag{3.2}$$

Cette question est posée en sachant que la commande doit être cohérente avec chaque consigne. Nous pouvons, à l'inverse, spécifier que la commande doit être en désaccord avec les consignes :

$$\mathrm{P}(\mathrm{V_{rot}} \mid [\mathrm{M}^0 = 0]\,[\mathrm{M}^1 = 0]\,[\mathrm{M}^2 = 0]\,[\mathrm{M}^3 = 0]) \tag{3.3}$$

Nous programmons un comportement de fuite par fusion proscriptive de quatre consignes qui attirent le personnage vers les quatre ennemis les plus proches. La commande produite est celle qui tend à éviter chacune des directions qui nous approchent d'un ennemi.

La question de la commande se résout de la façon suivante :

$$\begin{aligned}
&\mathrm{P}(\mathrm{V_{rot}} \mid [\mathrm{M}^0 = 0]\,[\mathrm{M}^1 = 0]\,[\mathrm{M}^2 = 0]\,[\mathrm{M}^3 = 0]) \\
&= \frac{1}{\mathrm{Z}} \sum_{\mathrm{V_{rot}^{0:3}}} \mathrm{P}(\mathrm{V_{rot}}) \prod_{\mathrm{i}=0}^{3} \mathrm{P}(\mathrm{V_{rot}^i})\,\mathrm{P}([\mathrm{M}^\mathrm{i} = 0] \mid \mathrm{V_{rot}}\,\mathrm{V_{rot}^i}) \\
&= \frac{1}{\mathrm{Z}}\,\mathrm{P}(\mathrm{V_{rot}}) \prod_{\mathrm{i}=0}^{3} \sum_{\mathrm{V_{rot}^i}} \mathrm{P}(\mathrm{V_{rot}^i})\,(1 - \mathrm{P}([\mathrm{M}^\mathrm{i} = 1] \mid \mathrm{V_{rot}}\,\mathrm{V_{rot}^i})) \\
&= \frac{1}{\mathrm{Z}}\,\mathrm{P}(\mathrm{V_{rot}}) \prod_{\mathrm{i}=0}^{3} \left[\sum_{\mathrm{V_{rot}^i}} \mathrm{P}(\mathrm{V_{rot}^i}) - \sum_{\mathrm{V_{rot}^i}} \mathrm{P}(\mathrm{V_{rot}^i})\, \underbrace{\mathrm{P}([\mathrm{M}^\mathrm{i} = 1] \mid \mathrm{V_{rot}}\,\mathrm{V_{rot}^i})}_{=1 \text{ si } \mathrm{v_{rot}^i} = \mathrm{v_{rot}},\, 0 \text{ sinon}} \right] \\
&= \frac{1}{\mathrm{Z}}\,\mathrm{P}(\mathrm{V_{rot}}) \prod_{\mathrm{i}=0}^{3} \left[1 - \mathrm{P}([\mathrm{V_{rot}^i} = \mathrm{v_{rot}}])\right]
\end{aligned} \tag{3.4}$$

Notre commande est donc le produit des compléments à 1 des consignes.

Nous pouvons reprendre notre illustration, à partir de trois consignes proscriptives (au lieu des quatre que notre personnage utilise en réalité). La commande 3.13 est le résultat de la fusion proscriptive des consignes de la figure 3.12. Cette commande ressemble à une distribution uniforme avec des "trous" au niveau des consignes.

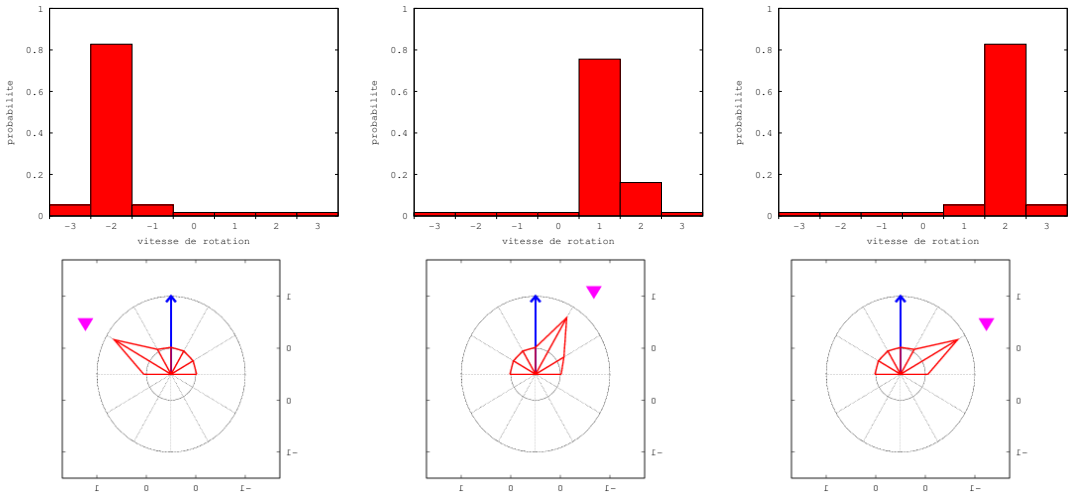

FIG. 3.12 – À **gauche** : $P(V^0_{rot})$ – attracteur en Dir $= -\frac{\pi}{3}$, au **centre** : $P(V^1_{rot})$ – attracteur en Dir $= \frac{\pi}{5}$, à **droite** : $P(V^2_{rot})$ – attracteur en Dir $= \frac{\pi}{3}$. Les triangles inversés violets figurent les attracteurs.

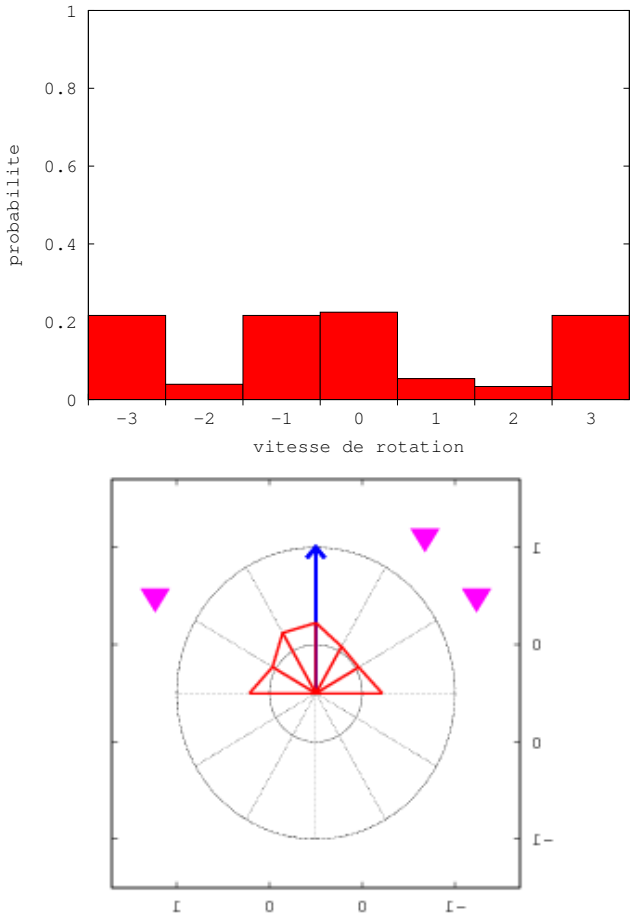

FIG. 3.13 – Commande $P(V_{rot})$, résultat de la fusion des attractions comme des consignes proscriptives.

3.3.3 Fusion de consignes prescriptives et proscriptives : comportement de fuite amélioré

Nous avons montré comment fusionner des consignes qui sont toutes prescriptives (les quatre attracteurs du comportement d'exploration), et qui sont toutes proscriptives (les quatre répulseurs du comportement de fuite). Nous avons, dans les deux cas, utilisé un nombre fixe de consignes.

Nous pouvons en pratique utiliser le schéma de fusion par cohérence comme un mécanisme plus générique :

- chaque consigne peut être fusionnée de manière prescriptive ($M^i = 1$) ou proscriptive ($M^i = 0$), indépendamment des autres consignes ;
- nous pouvons fusionner facilement un nombre variable (borné) de consignes.

Nous utilisons ce mécanisme pour améliorer notre comportement de fuite. Avec notre comportement de fuite initial, notre personnage fuit ses ennemis en choisissant une direction de sorte à ne faire face à aucun d'eux. Ce faisant, il ne tient pas compte de la géométrie de l'environnement, et peut se diriger vers un mur, ou un fossé. Pour prendre ces éléments en compte, nous pouvons ajouter les points de navigation comme attracteurs. Parmi ces points de navigation, nous utilisons ceux qu'il est possible d'atteindre en ligne droite sans toucher d'obstacle immobile.

Nous reprenons donc sur la figure 3.14 les mêmes trois consignes proscriptives, qui correspondent aux directions des ennemis les plus proches. Nous y ajoutons les consignes attractives de la figure 3.15, qui correspondent aux points de navigation de l'environnement qui sont accessibles directement. La figure 3.16 montre le résultat de la fusion de ces consignes. La distribution obtenue présente un pic important autour de 0, qui correspond à la consigne prescriptive $P(V^3_{rot})$ mais n'est pas couvert par les consignes proscriptives.

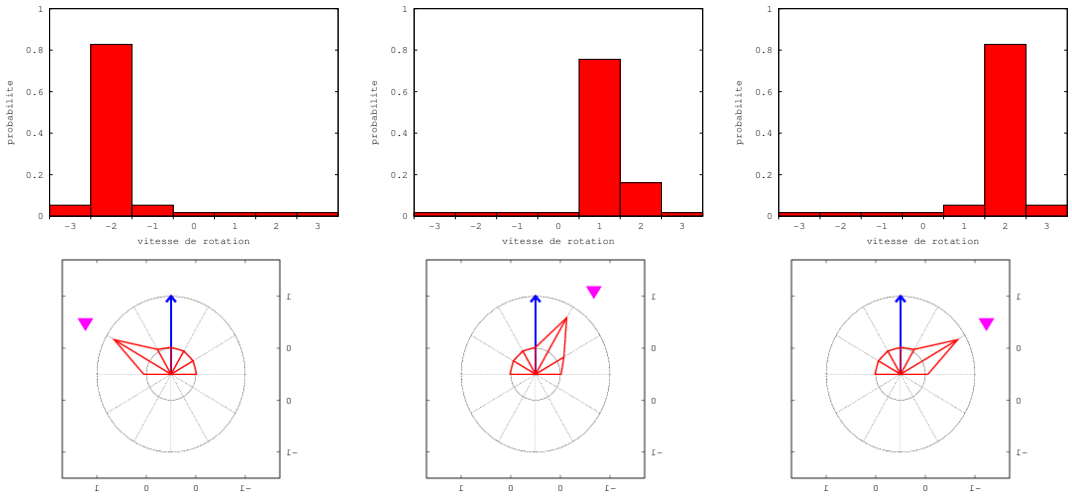

FIG. 3.14 – À **gauche** : consigne $P(V^0_{rot})$ – attracteur en Dir $= -\frac{\pi}{3}$, au **centre** : $P(V^1_{rot})$ – attracteur en Dir $= \frac{\pi}{5}$, à **droite** : $P(V^2_{rot})$ – attracteur en Dir $= \frac{\pi}{3}$.

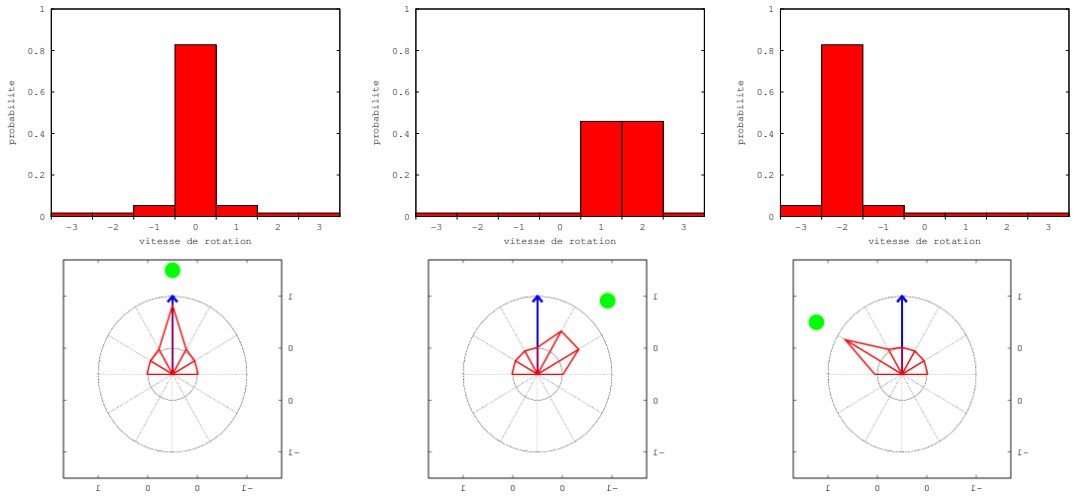

FIG. 3.15 – À **gauche** : consigne $P(V^3_{rot})$ – attracteur en Dir $= 0$, au **centre** : $P(V^4_{rot})$ – attracteur en Dir $= \frac{\pi}{4}$, à **droite** : $P(V^5_{rot})$ – attracteur en Dir $= -\frac{\pi}{3}$.

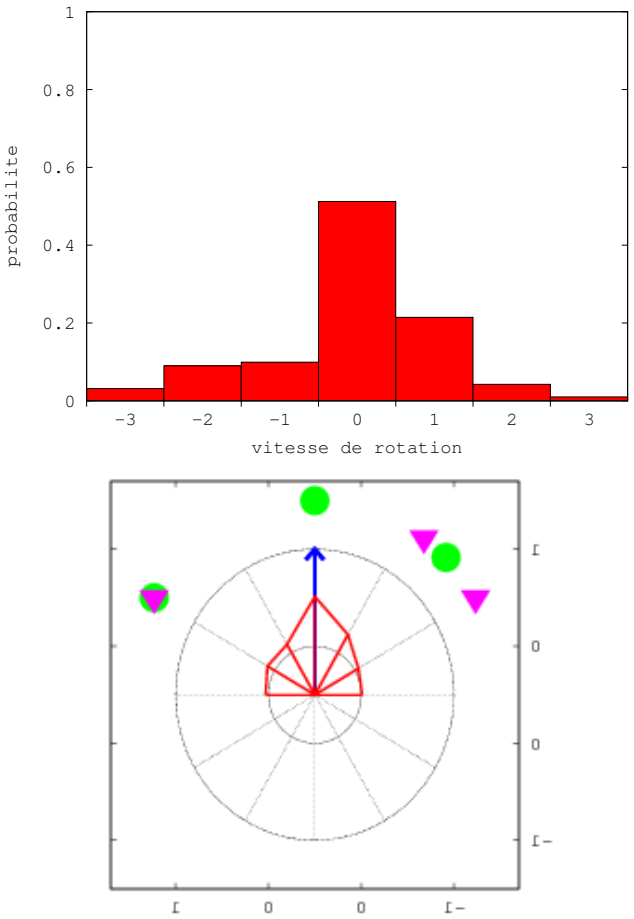

FIG. 3.16 – Commande $P(V_{rot})$, résultat de la fusion de trois consignes proscriptives ($P(V_{rot}^{0,1,2})$) et trois consignes prescriptives ($P(V_{rot}^{3,4,5})$).

3.3.4 La fusion par cohérence : un mécanisme générique

Nous avons vu qu'il est possible de fusionner un nombre fixe de consignes, en les considérant chacune comme prescriptive ou proscriptive.

La fusion d'un nombre variable de consignes est faite en considérant que chacune des quatre consignes est active ou inactive. Pour le comportement de fuite, si deux ennemis seulement sont perçus,

- les deux premières consignes sont activées : $P(V^0_{rot})$ et $P(V^1_{rot})$ sont spécifiées ;
- les deux autres sont désactivées : $P(V^2_{rot})$ et $P(V^3_{rot})$ sont réduites à la distribution uniforme[1] :

$$P(V^2_{rot}) \;=\; P(V^3_{rot}) \;=\; U \;=\; \frac{1}{Card(V_{rot})}$$

De cette façon, la commande ne dépend que des consignes actives $P(V^0_{rot})$ et $P(V^1_{rot})$:

$$\begin{aligned}
&P(V_{rot} \mid M^0\, M^1\, M^2\, M^3)\\
&= \frac{1}{Z} \sum_{V^{0:3}_{rot}} P(V_{rot}) \prod_{i=0}^{3} P(V^i_{rot})\, P(M^i \mid V_{rot}\, V^i_{rot})\\
&= \frac{1}{Z}\, P(V_{rot}) \prod_{i=0}^{1} \sum_{V^i_{rot}} P(V^i_{rot})\, P(M^i \mid V_{rot}\, V^i_{rot})\\
&\times \prod_{i=2}^{3} \sum_{V^i_{rot}} \underbrace{P(V^i_{rot})}_{=U}\, P(M^i \mid V_{rot}\, V^i_{rot})\\
&= \frac{1}{Z}\, P(V_{rot}) \prod_{i=0}^{1} \sum_{V^i_{rot}} P(V^i_{rot})\, P(M^i \mid V_{rot}\, V^i_{rot})\\
&\times \prod_{i=2}^{3} \sum_{V^i_{rot}} \frac{1}{Card(V_{rot})} \underbrace{P(M^i \mid V_{rot}\, V^i_{rot})}_{=1 \text{ si } v_{rot} = v^i_{rot},\, 0 \text{ sinon}}\\
&= \frac{1}{Z'}\, P(V_{rot}) \prod_{i=0}^{1} \sum_{V^i_{rot}} P(V^i_{rot})\, P(M^i \mid V_{rot}\, V^i_{rot})\\
&\times \prod_{i=2}^{3} 1\\
&= \frac{1}{Z'}\, P(V_{rot}) \prod_{i=0}^{1} \sum_{V^i_{rot}} P(V^i_{rot})\, P(M^i \mid V_{rot}\, V^i_{rot}) \qquad (3.5)
\end{aligned}$$

Enfin, chaque consigne est une distribution portant sur la variable de fusion V_{rot}. Cette distribution peut être obtenue par divers moyens. Par exemple, pour notre comportement d'exploration elle est obtenue en interrogeant un sous-modèle, et correspond à

$$P(V^i_{rot} \mid Dir^i_{attracteur})$$

[1] Il est aussi possible de désactiver la consigne i en posant simplement $P(M^i \mid V_{rot}\, V^i_{rot}) \;=\; U$.

Dans le modèle de fusion, nous ne prenons pas en compte la variable $\mathrm{Dir}^{i}_{\text{attracteur}}$: c'est un détail d'implémentation du sous-modèle, qui est masqué lors de la fusion. La seule contrainte imposée au sous-modèle est qu'il fournisse une distribution sur V_{rot}.

Enfin, sa construction comme un programme bayésien est intuitive, et conduit à un résultat tout aussi intuitif : la commande est simplement le produit des distributions consignes.

3.3.5 Fusion par cohérence améliorée contre fusion motrice naïve

Nous avons vu plus haut le schéma de fusion motrice naïve. Il est intéressant de comparer celui-ci à la fusion par cohérence améliorée.

Considérons un ensemble de consignes exprimées directement sur la variable de fusion. Chacune d'elles peut s'écrire $c^i = P(V \mid S_i)$, où V est la variable de fusion et S_i est un ensemble de variables pertinentes pour le sous-modèle fournissant la consigne i. Ce type de consigne directe est très courant : il consiste à spécifier une variable motrice sachant des capteurs.

Nous avons vu que la fusion par cohérence améliorée consiste à faire le produit de ces consignes directes :

$$P(V) = \frac{1}{Z} \prod_i c^i$$

Au contraire, dans la fusion motrice naïve, la commande générée ne correspond pas à ce produit. Pour écrire la fusion de ces consignes directes, nous devons les inverser, pour obtenir les consignes inversées $c^{i\prime} = P(S_i \mid V)$:

$$c^{i\prime} = P(S_i \mid V) = \frac{P(S_i)\, P(V \mid S_i)}{P(V)} = \frac{1}{Z} P(S_i)\, c^i$$

où Z est un facteur de normalisation :

$$Z = P(V) = \sum_{S_i} P(S_i)\, P(V \mid S_i)$$

Et nous pouvons ensuite les fusionner :

$$P(V) = \frac{1}{Z} \prod_i c^{i\prime} = \frac{1}{Z} \prod_i P(S_i)\, c^i$$

Le calcul de la consigne inversée $c^{i\prime} = P(S_i \mid V)$ à partir de la consigne directe c^i fait intervenir l'*a priori* $P(S_i)$. Dans le schéma de fusion motrice naïve, cet *a priori* doit donc être spécifié, et influe sur la commande issue de la fusion. Or il est fréquent que nous n'ayons pas d'idée pertinente sur cette distribution sur les variables sensorielles. Le résultat de la fusion motrice naïve dépend alors dangereusement de facteurs qui ne sont pas pertinents, et peut être imprévisible ou difficile à interpréter.

Dans le cas où il est naturel de spécifier nos consignes directement ($P(V \mid S_i)$: variable motrice sachant les variables sensorielles), il est donc plus intéressant de réaliser une fusion par cohérence améliorée.

3.3.6 Tâches bayésiennes simples : conclusion

Nous avons décrit les briques simples de nos comportements : des modèles bayésiens réactifs liant capteurs et moteurs. Leur construction repose sur une identification des variables pertinentes, de la décomposition de la distribution conjointe sur ces variables, et du choix des formes précises des facteurs de cette décomposition. Nous verrons dans la suite comment les paramètres de ces facteurs peuvent être appris, au lieu d'être entièrement spécifiés à la main.

Les paramètres des tâches bayésiennes simples sont bien identifiés : ce sont les paramètres des distributions de probabilité formant les facteurs de la décomposition. Ils ont un sens intuitif : ils peuvent être fixés manuellement, et ils peuvent être compris et interprétés.

Les comportements simples produits peuvent être combinés entre eux selon des techniques génériques (fusion motrice, fusion capteurs) pour produire d'autres briques simples plus puissantes.

Cependant, la taille des briques, c'est-à-dire leur nombre de variables et de paramètres est limitée par ce qui peut être géré par le programmeur. Il devient nécessaire de pouvoir les combiner de manière modulaire, pour briser la croissance de la complexité de programmation. La mise en séquence apporte une réponse à ce besoin : elle permet de combiner plusieurs tâches en un comportement de plus haut niveau.

3.4 Programmer un comportement comme une séquence : la programmation inverse

3.4.1 Mise en séquence, sélection de l'action

Reprenons la définition d'un agent donnée par Ferber, pour en extraire deux caractéristiques :

- un agent possède des compétences : il sait accomplir un certain nombre de tâches, comme celles simples décrites auparavant ;
- il a des motivations et des buts.

Le problème de la sélection de l'action consiste à appliquer les compétences du personnage les unes après les autres, en tenant compte de la situation du personnage dans l'environnement, pour tendre à ses buts.

Informellement, il s'agit de répondre à la question "que dois-je faire pour tendre à mes buts ?". Le problème de programmation est de définir une méthode de choix de tâche.

Plus formellement, nous voulons définir une fonction π nommée **politique** telle que Tache $=$ π(Capteurs, EtatInterne) soit une réponse à la question de sélection de l'action. Dans une situation ou le personnage perçoit des Capteurs et se trouve dans un certain EtatInterne, Tache est alors une tâche appropriée pour tendre à ses buts.

En termes de modélisation probabiliste, la politique π est un programme bayésien capable de répondre à la question P(Tache $|$ Capteurs EtatInterne π).

3.4.2 La problématique de la mise en séquence

Dans ce contexte, deux questions se posent :

- Sur quoi baser le choix d'une tâche à accomplir ? C'est-à-dire, comment choisir les variables Capteurs et EtatInterne ?
- Comment mettre en œuvre le choix de la tâche ? En termes probabilistes, comment décrire la décomposition $P(\text{Tache Capteurs EtatInterne} \mid \pi)$?

Sur quoi baser le choix d'une tâche à accomplir ?

Capteurs Nous pouvons en guise de capteurs utiliser différentes grandeurs facilement calculables dans l'environnement simulé. Parmi celles-ci, le nombre de personnages proches, leur distance au personnage autonome, la vitesse du personnage autonome...

Il nous est également possible de construire des capteurs virtuels de plus haut niveau, tirant parti des possibilités d'un environnement complètement simulé. Par exemple, il est possible de calculer la position du centre de gravité d'une foule ou d'un troupeau, ou les coordonnées relatives d'un point d'intérêt déterminé par un algorithme de planification.

Choisir nos capteurs est crucial – cela correspond au choix de certaines des variables du modèle probabiliste. Ce choix délimite la portée des comportements que nous pourrons construire. Par exemple, la gestion d'un comportement d'attaque dans *Unreal Tournament* nécessite de prendre en compte des informations sur les ennemis proches. De plus, l'utilisation de capteurs élaborés, comme ceux issus d'un calcul de chemins planifiés peut être une source d'intelligence pour le comportement global.

Faire ce choix demande une expertise sur l'environnement et les comportements à produire : il s'agit de faire la liste des capteurs disponibles (directement, ou selon des précalculs élaborés), et d'en extraire un jeu pertinent. Il est envisageable dans certains cas d'automatiser cette sélection des capteurs pertinents [Dangauthier 05b]. Nous faisons ici le choix d'en rester à l'expertise humaine.

État interne L'état interne de notre personnage peut être réduit à rien. La politique de choix de tâche est alors dite purement réactive : elle se base uniquement sur les informations instantanées disponibles. À un instant t, il peut aussi contenir une mémoire d'actions ou perceptions précédentes, comme Tache_{t-1} ou Tache_{t-1} Tache_{t-2} ... Tache_{t-n}. Enfin, il peut contenir une représentation à jour de l'environnement du personnage, comme la position de ses ennemis (y compris ceux qu'il a vu récemment mais ne voit plus). Une partie de l'apparence d'intelligence de notre personnage se trouve dans cette capacité de mémoire et de représentation.

Comment mettre en œuvre le choix de la tâche ?

Techniques délibératives : approche cognitive L'approche cognitive de construction de comportements repose sur l'idée que le personnage maintient une représentation de son environnement. Il prend des décisions en intégrant l'effet de ses actions dans son environnement.

La représentation de l'environnement peut être locale ou globale ; symbolique ou basée sur les interactions [Simonin 05] sensori-motrices. Cette représentation peut être exploitée à plusieurs échelles temporelles, par exemple pour définir des plans à court termes mis à jour en continu [Petti 05].

A* est une technique de ce type omniprésente dans les jeux vidéos modernes. Il s'agit d'un algorithme de planification le plus souvent utilisé pour calculer des chemins d'un point à un autre dans l'environnement. Il nécessite un modèle complet de l'environnement. Des variations sur A* et d'autres algorithmes [Mazer 98] de planification permettent d'effectuer ces calculs de planifications de manière incrémentale, en temps limité, et en explorant partiellement l'environnement, qui peut être dynamique.

Cette vue descendante du comportement, basée sur sa représentation, a aussi donné lieu à des algorithmes de calcul de politiques optimales. Les techniques d'apprentissage par renforcement [Sutton 98] permettent la construction de stratégies. Elles reposent sur un formalisme mathématique solide (processus de décision de Markov (MDP) et processus de décision de Markov partiellement observables (POMDP)). Elles s'appuient sur la construction simultanée d'un modèle probabiliste de l'interaction du personnage avec l'environnement, et de la politique maximisant une fonction de coût. Ainsi, les POMDP [Howard 60] prennent en compte l'état du monde, l'observation qui en est faite, l'action du personnage, et une récompense ou une pénalité associée à chaque état ; il est possible de calculer une politique qui permet de choisir les actions qui maximisent les récompenses obtenues globalement [Cassandra 94].

Le point commun de ces techniques délibératives est de reposer sur une représentation de l'environnement, et de l'interaction du personnage avec celui-ci. Cette représentation peut être connue globalement ou localement, et parfois apprise par exploration (comme en apprentissage par renforcement). Les mieux maîtrisées de ces méthodes trouvent leur place en pratique quand elles peuvent s'appuyer sur la représentation existante de l'environnement dans le moteur de rendu et de simulation.

Elles sont utilisées pour des problèmes spécifiques : faire naviguer un personnage dans un labyrinthe, contrôler les déplacements d'une bras articulé... Elles peuvent en théorie s'appliquer pour réaliser des plans dans des espaces autres que l'espace des positions spatiales (comme l'espace des actions élémentaires du personnage), mais cela nécessite des modèles compliqués de l'environnement et des interactions, et rend leur complexité peu compatible avec les contraintes d'un monde simulé en temps réel. Dans l'état présent, ces méthodes se prêtent mal à l'apprentissage par imitation.

Mise en compétition de tâches Une revue des mécanismes de sélection d'action tels qu'ils ont été modélisés en éthologie, et appliqués en robotique, peut être trouvée dans [Tyrrell 93].

La forme la plus basique de sélection de tâche consiste à les mettre en compétition, chacune fournissant une grandeur nommée activation donnant son applicabilité. Celle qui obtient l'activation la plus importante est alors exécutée. Le calcul de l'activation est fait pour chaque tâche sur la base des conditions sensorielles. Il est possible de prendre également en compte des inhibitions [Tinbergen 51] d'une tâche sur l'autre. Par exemple, une tâche Manger va calculer son activation

comme proportionnelle à la faim, et inversement proportionnelle à la distance de la nourriture, mais peut être inhibée par l'activation d'une tâche de fuite des prédateurs.

[Maes 91] décrit un mécanisme détaillé sur cette base. Il s'agit d'un modèle de sélection d'action appliqué à la robotique, prenant en compte les préconditions logiques nécessaires à l'application de chaque tâche.

Le modèle de comportement décrit par [Koike 05] réalise la sélection de l'action sans décider explicitement d'une tâche à chaque instant. Le choix de l'action est couplé à un mécanisme de focalisation de l'attention qui réduit les informations sensorielles prises en compte. Les commandes motrices sont déterminées par la fusion des commandes données par chaque tâche, en proportion de leur probabilité.

Notre approche : mise en séquence de tâches Nous proposons une technique de mise en séquence de tâches. Celle-ci permet de choisir à chaque instant une tâche à exécuter sachant un ensemble de variables sensorielles, et la tâche précédente. Elle est essentiellement réactive : sa seule mémoire est celle de la tâche exécutée au pas de temps précédent. Nous montrerons qu'elle permet de rivaliser avec les modèles de choix de tâches mis en œuvre dans *Unreal Tournament*, en simplicité comme en expressivité.

3.4.3 Un modèle bayésien inspiré des automates pour mettre des tâches en séquence

Programme bayésien de séquencement de tâches

Nous considérons un personnage autonome jouant à *Unreal Tournament*. Pour son contrôle, nous prenons en compte les variables suivantes :

- T^{t-1} : la tâche du personnage au temps $t-1$, à valeurs dans {Attaque, RechArme, RechSanté, Exploration, Fuite, DétDanger}.
- T^t : la tâche au temps t.
- Santé : le niveau de santé du personnage à t, dans {Bas, moyen, Haut}.
- Arme : l'arme du personnage à t, dans {Aucune, Moyenne, Forte}.
- ArmeE : l'arme de l'ennemi courant à t, dans {Aucune, Moyenne, Forte}.
- Bruit : indique si un bruit a été entendu récemment, à t, booléen.
- NbE : le nombre d'ennemis proches à t, dans {Aucun, Un, DeuxOuPlus}.
- ArmeProche : indique si une arme est proche à t, booléen.
- SantéProche : indique si une recharge de santé est proche à t, booléen.

Les commandes motrices élémentaires du personnage sont les valeurs des variables T^t et T^{t-1}. Elles incluent :

- une tâche d'attaque, dans laquelle le personnage tire sur un ennemi en maintenant une distance constante, et en se déplaçant en crabe ;
- des tâches pour aller ramasser une arme ou une recharge de santé que le personnage a remarqué dans son environnement ;
- une tâche pour naviguer dans l'environnement et en découvrir les parties encore inexplorées ;

– une tâche de fuite, qui consiste à essayer de s'échapper d'un ennemi ; et
– une tâche de détection du danger, qui consiste à se retourner pour essayer de détecter la présence d'un ennemi.

La décision se fait au temps t : T^{t-1} et les variables sensorielles à t sont connus, T^t doit être déterminé. Cette décision est faite au moyen du programme bayésien décrit dans la figure 3.17.

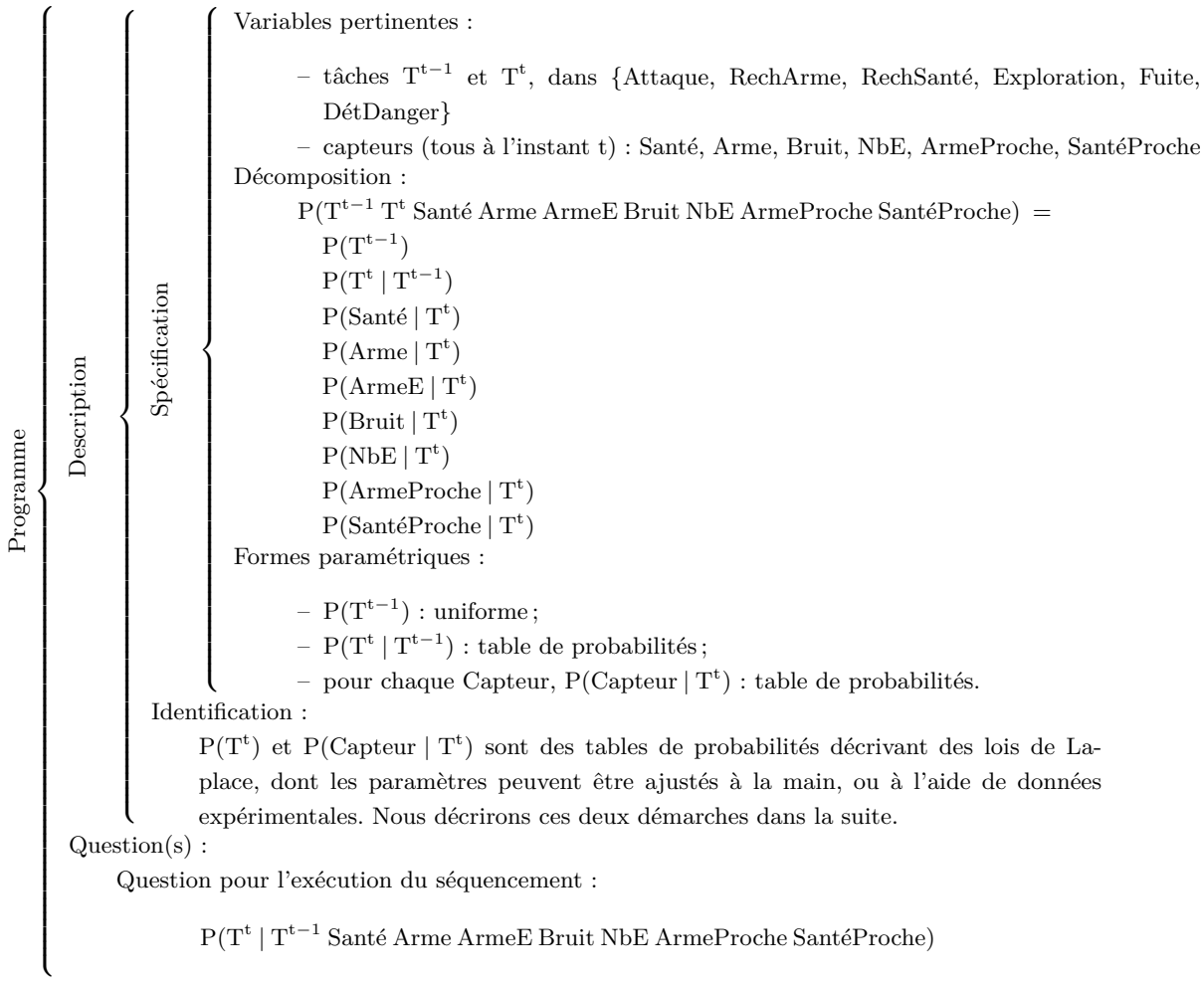

FIG. 3.17 – Modèle de mise en séquence de tâches par programmation inverse

La décomposition de ce programme est faite avec l'hypothèse que sachant la tâche suivante T^t, les variables sensorielles sont indépendantes deux à deux. Cela fait de notre modèle un modèle de Markov caché (en Anglais : *Hidden Markov Model*, HMM [Rabiner 89]), où les observations sont indépendantes sachant l'état. Bien que cette hypothèse réduise l'expressivité de notre modèle, elle permet de le spécifier d'une manière très condensée. Nous reviendrons sur l'idée de modèle de Markov caché quand nous nous préoccuperons d'apprendre notre modèle.

Un modèle de Markov caché est souvent utilisé pour construire un capteur virtuel : la variable d'état. On est alors en présence d'un modèle de fusion sensorielle (naïve ou non), qui intègre une série d'observations pour estimer l'état. Nous utilisons notre HMM comme un modèle de fusion motrice : notre état (la tâche) est la variable motrice, qui permet de faire agir le personnage. Cette fusion est naïve au sens où les variables sensorielles sont indépendantes deux à deux sachant la tâche.

Sachant la tâche courante et les valeurs des capteurs, nous voulons connaître la nouvelle tâche que le personnage doit poursuivre. Chaque fois que notre personnage doit prendre une décision (dix fois par secondes dans nos expérimentations), nous posons donc la question suivante :

$$\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1}\ \text{Santé Arme ArmeE Bruit NbE ArmeProche SantéProche}) \tag{3.6}$$

Le processus d'inférence bayésienne consiste à résoudre cette question en se ramenant à la décomposition de la distribution conjointe :

$$\begin{aligned}
&\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1}\ \text{Santé Arme ArmeE Bruit NbE ArmeProche SantéProche}) \\
&\quad = \frac{\mathrm{P}(\mathrm{T}^{\mathrm{t}}\ \mathrm{T}^{\mathrm{t}-1}\ \text{Santé Arme ArmeE Bruit NbE ArmeProche SantéProche})}{\mathrm{P}(\mathrm{T}^{\mathrm{t}-1}\ \text{Santé Arme ArmeE Bruit NbE ArmeProche SantéProche})} \\
&\quad = \frac{\mathrm{P}(\mathrm{T}^{\mathrm{t}-1})\,\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1})\,\Pi_{\mathrm{i}}\,\mathrm{P}(\mathrm{Capteur}_{\mathrm{i}} \mid \mathrm{T}^{\mathrm{t}})}{\sum_{\mathrm{T}^{\mathrm{t}}}(\mathrm{P}(\mathrm{T}^{\mathrm{t}-1})\,\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1})\,\Pi_{\mathrm{i}}\,\mathrm{P}(\mathrm{Capteur}_{\mathrm{i}} \mid \mathrm{T}^{\mathrm{t}}))} \\
&\quad = \frac{\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1})\,\Pi_{\mathrm{i}}\,\mathrm{P}(\mathrm{Capteur}_{\mathrm{i}} \mid \mathrm{T}^{\mathrm{t}})}{\sum_{\mathrm{T}^{\mathrm{t}}}(\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1})\,\Pi_{\mathrm{i}}\,\mathrm{P}(\mathrm{Capteur}_{\mathrm{i}} \mid \mathrm{T}^{\mathrm{t}}))}
\end{aligned} \tag{3.7}$$

Nous tirons une valeur sur la distribution obtenue pour décider de la nouvelle tâche. Cette tâche correspond à un comportement élémentaire du personnage, et peut être exécutée directement.

Si nous notons NCapteurs le nombre de capteurs (dans notre cas, 7), et NTaches le nombre de tâches (dans notre cas, 6), nous tabulons une distribution à NTaches valeurs. Nous calculons le numérateur de la formule 3.7 pour chacune des valeurs possibles de T^{t} ; nous normalisons ensuite le tableau obtenu de façon à ce que la somme de ses éléments soit 1, sans calculer explicitement le dénominateur de la formule 3.7.

Le calcul de la question nécessite donc :

- $(\text{NCapteurs} + 1) \times \text{NTaches}$ accès à un tableau et $\text{NCapteurs} \times \text{NTaches}$ multiplications pour le calcul du numérateur pour toutes les tâches T^{t} ;
- NTaches additions, NTaches accès à un tableau, et NTaches divisions pour normaliser la distribution.

En termes de mémoire, notre modèle nécessite de stocker :

- $\text{NTaches} \times \text{NTaches}$ flottants pour la table de transition $\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1})$;
- $\text{NTaches} \times \sum_{\mathrm{i}} |\mathrm{Capteur}_{\mathrm{i}}|$ flottants pour toutes les tables $\mathrm{P}(\mathrm{Capteur}_{\mathrm{i}} \mid \mathrm{T}^{\mathrm{t}})$.

La complexité de calcul de la question est donc linéaire dans le nombre de tâches, et dans le nombre de capteurs. La complexité en mémoire est quadratique dans le nombre de tâches, et linéaire dans le nombre de capteurs.

Programmation inverse

Ce modèle de séquencement, par sa structure et son utilisation, peut être comparé à une machine d'états finis (MEF). Le problème est celui que traite une MEF : sachant un état (notre tâche) courant et des valeurs sensorielles, il s'agit de déterminer un état suivant.

Considérons le cas où chacune de nos n variables sensorielles a m_i ($1 \leq i \leq n$) valeurs possibles.

Dans une MEF décrivant ce comportement [Dysband 00] [Zarozinski 01], nous devons spécifier, pour chaque état, une transition vers les autres états, sous la forme d'une condition logique sur les variables sensorielles.

Cela implique que le programmeur doit discriminer parmi les $\Pi_i m_i$ combinaisons sensorielles possibles pour décrire les transitions d'état à état. Cela pose le problème de déterminer les transitions appropriées, mais aussi celui de les représenter formellement en pratique. Cette approche peut conduire à plusieurs implémentations, la plus commune étant celle que nous avons exposée pour *Unreal Tournament*, basée sur un script. Nous avons vu que ce genre de script est difficile à écrire, et à maintenir.

Par contraste, notre approche consiste à donner, pour chaque variable sensorielle, pour chaque tâche possible, une distribution (m_i nombres se sommant à 1). En pratique, nous écrivons des tables telles que la table 3.2, qui représente $P(\text{Santé}|T^t)$. Les valeurs de Santé sont énumérées dans la première colonne, celles de T^t dans la première ligne. Les cellules marquées x sont calculées de façon à ce que chaque colonne se somme à 1.

	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.001	0.1	x	0.1	0.7	0.1
Moyen	0.1	x	0.01	x	0.2	x
Haut	x	x	0.001	x	0.1	x

TAB. 3.2 – $P(\text{Santé}|T^t)$

De plus, au lieu de spécifier les conditions qui font changer le personnage d'une tâche à l'autre, nous spécifions les (distributions de probabilité sur les) valeurs sensorielles quand le personnage exécute une tâche donnée. Cette façon de spécifier un capteur en supposant la tâche connue – alors qu'à l'exécution le capteur est connu, et la tâche recherchée – est ce qui nous fait nommer cette méthode "programmation inverse".

Nous lisons la première colonne de la table 3.2 de la façon suivante : sachant que le personnage sera dans l'état Attaque, nous savons qu'il a une très faible probabilité (0.001) d'avoir un niveau de santé faible, une probabilité moyenne (0.1) d'avoir un niveau de santé moyen, et une forte chance ($x = 1 - 0.001 - 0.1 = 0.899$) d'avoir un niveau de santé élevé.

Bien qu'elle puisse être troublante au premier abord, cette manière de procéder est au cœur de notre méthode de spécification d'un séquencement. En effet, elle nous permet de décrire séparément l'influence de chaque capteur sur la tâche exécutée par le personnage, et d'ainsi réduire la quantité d'information à manipuler. De plus, il devient très facile d'ajouter une nouvelle

variable sensorielle à notre modèle : cela nécessite seulement d'écrire une nouvelle table, sans modifier celles existantes.

Enfin, le nombre de valeurs que nous devons spécifier pour décrire un comportement est $s^2 + s \times n \times m$, où s est le nombre de tâches, n le nombre de variables sensorielles, et m le nombre moyen de valeurs possibles pour les variables sensorielles. Si m est constant, ce nombre de valeurs augmente linéairement avec le nombre de variables sensorielles.

Spécifier et ajuster des comportements

Un comportement peut être spécifié en écrivant les tables correspondant à $P(T^t \mid T^{t-1})$ et $P(\text{Capteur} \mid T^t)$ (pour chaque variable sensorielle Capteur).

Cette forme de spécification nous permet de formaliser simplement les contraintes que nous voulons imposer sur le comportement, de manière condensée, séparément pour chaque variable sensorielle. Par exemple, la table 3.2 formalise notre idée intuitive de la relation du niveau de santé du personnage à sa tâche : s'il attaque, alors son niveau de santé est probablement haut. Les autres tables $P(\text{Capteur} \mid T^t)$ disent : s'il part chercher une recharge de santé, sa santé est probablement faible ; s'il fuit, sa santé est probablement faible, avec un fort degré d'incertitude.

Toutes les tables concernant les variables sensorielles sont construites sur le même modèle, mais celle donnant $P(T^t \mid T^{t-1})$ est spéciale. Elle répond de manière probabiliste à la question : ne sachant rien que la tâche courante, quelle sera ma tâche à l'instant suivant ?

Nous décrivons en parallèle la construction de deux comportements différents : un personnage prudent et un personnage très agressif.

$P(T^t \mid T^{t-1})$ **(table 3.3)** Cette table décrit la distribution de la tâche courante sachant la tâche précédente, sans se préoccuper des variables sensorielles.

Pour le combattant prudent, elle correspond à l'hypothèse suivante :

- quand le personnage effectue une tâche, il a de grandes chances de garder cette même tâche au pas de temps suivant (cases bleues : $x = 1 - 0.01 \times 5 = 0.95$).

Pour le combattant agressif, l'hypothèse est :

- quand le personnage effectue une tâche, il a de très grandes chances de garder cette même tâche (cases bleues) ou de passer dans la tâche d'attaque (cases rouges).

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	x	0.01	0.01	0.01	0.01	0.01
RechArme	0.01	x	0.01	0.01	0.01	0.01
RechSanté	0.01	0.01	x	0.01	0.01	0.01
Exploration	0.01	0.01	0.01	x	0.01	0.01
Fuite	0.01	0.01	0.01	0.01	x	0.01
DétDanger	0.01	0.01	0.01	0.01	0.01	x
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	x	x	x	x	x	x
RechArme	10^{-5}	x	10^{-5}	10^{-5}	10^{-5}	10^{-5}
RechSanté	10^{-5}	10^{-5}	x	10^{-5}	10^{-5}	10^{-5}
Exploration	10^{-5}	10^{-5}	10^{-5}	x	10^{-5}	10^{-5}
Fuite	10^{-5}	10^{-5}	10^{-5}	10^{-5}	x	10^{-5}
DétDanger	10^{-5}	10^{-5}	10^{-5}	10^{-5}	10^{-5}	x

TAB. 3.3 – $P(\mathrm{T}^{\mathrm{t}}|\mathrm{T}^{\mathrm{t}-1})$

P(Santé | T^{t}) **(table 3.4)** Cette table décrit la distribution du niveau de santé (bas, moyen, haut) selon la tâche suivie par le personnage.

Pour le personnage prudent, nous faisons les hypothèses suivantes :

- quand il attaque, son niveau de santé est sans doute haut, peut-être moyen, mais très probablement pas bas (cases bleues) ;
- quand il recherche une arme, explore l'environnement, ou détecte le danger, son niveau de santé n'est probablement pas bas : il est dangereux d'exécuter ces tâches avec peu de santé (cases vertes : probabilité 0.1) ;
- quand il cherche un bonus de santé, nous sommes presques sûrs qu'il a un niveau de santé faible (cases jaunes) ;
- quand il fuit, il est probable qu'il ait un niveau de santé faible, mais possible cependant que celui-ci soit moyen ou fort (cases rouges).

Pour le personnage agressif, notre seule hypothèse est :

- le niveau de santé n'influe pas sur le choix de la tâche (toutes cases grises : distributions uniformes).

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.001	0.1	x	0.1	0.7	0.1
Moyen	0.1	x	0.01	x	0.2	x
Haut	x	x	0.001	x	0.1	x
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	x	x	x	x	x	x
Moyen	x	x	x	x	x	x
Haut	x	x	x	x	x	x

TAB. 3.4 – $P(\text{Santé}|\text{T}^t)$

$\text{P}(\text{Arme} \mid \text{T}^t)$ **(table 3.5)** Cette table décrit la distribution de l'arme du personnage sachant la tâche qu'il choisit.

Pour le combattant prudent, nous faisons les hypothèse suivantes :

- quand le personnage attaque, il possède une arme (case verte : probabilité 10^{-5}) ;
- quand le personnage recherche une arme, la probabilité qu'il ait une arme moyenne est faible, et celle qu'il ait une arme forte, très faible (cases jaunes : probabilités 0.1 et 10^{-5}) ;
- quand le personnage fuit, la probabilité est faible qu'il ait une arme moyenne, et très faible qu'il ait une arme forte (cases bleues : probabilités 0.1 et 0.01) ;
- quand le personnage effectue les autres tâches, nous ne savons rien de son arme (cases grises : distributions uniformes).

Pour le combattant agressif les hypothèses faites sont similaires – elles varient dans le degré de certitude que nous leur assignons :

- quand le personnage attaque, il est improbable qu'il n'ait pas d'arme, mais moins improbable que pour le personnage prudent (case verte : probabilité 0.01) ;
- quand le personnage recherche une arme, il est très improbable qu'il ait une arme moyenne, et extrêmement improbable qu'il n'ait pas d'arme (cases jaunes : probabilités 0.001 et 10^{-7}) ;
- quand le personnage fuit, la probabilité est très faible qu'il ait une arme moyenne, et extrêmement faible qu'il ait une arme forte (cases bleues : probabilités 0.001 et 10^{-5}) ;
- quand le personnage effectue les autres tâches, nous ne savons rien de son arme (cases grises : distributions uniformes).

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	10^{-5}	x	x	x	x	x
Moyenne	x	0.1	x	x	0.1	x
Forte	x	10^{-5}	x	x	0.01	x
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	0.01	x	x	x	x	x
Moyenne	x	0.001	x	x	0.001	x
Forte	x	10^{-7}	x	x	10^{-5}	x

TAB. 3.5 – $P(\text{Arme}|\text{T}^t)$

P(ArmeE | T^t) **(table 3.6)** Cette table décrit la distribution de l'arme de l'adversaire sachant la tâche choisie.

Pour le combattant prudent, les hypothèse faites sont les suivantes :

- quand le personnage effectue une autre tâche que l'attaque, nous ne savons rien de l'arme de l'adversaire (cases grises) ;
- quand le personnage est en attaque, l'arme de l'adversaire n'est très probablement pas forte (case verte : probabilité 0.01).

Les hypothèses faites pour le combattant agressif sont presque identiques :

- quand le personnage effectue une autre tâche que l'attaque, nous ne savons rien de l'arme de l'adversaire (cases grises) ;
- quand le personnage est en attaque, l'arme de l'adversaire n'est probablement pas forte (case bleue : probabilité 0.1), mais avec moins de certitude que pour le personnage prudent.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	x	x	x	x	x	x
Moyenne	x	x	x	x	x	x
Forte	0.01	x	x	x	x	x
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	x	x	x	x	x	x
Moyenne	x	x	x	x	x	x
Forte	0.1	x	x	x	x	x

TAB. 3.6 – $P(\text{ArmeE}|\text{T}^t)$

P(Bruit | T^t) **(table 3.7)** Cette table décrit la distribution du bruit (variable booléenne) selon la tâche choisie.

Pour le combattant prudent, nous faisons les hypothèses suivantes :

- quand le personnage recherche une arme ou un bonus de santé, il est plutôt improbable qu'il entende du bruit (cases vertes : probabilité 0.1), puisque le bruit intervient principalement dans les situations d'affrontement ;

– pour la même raison, quand le personnage explore son environnement, il est très improbable qu'il entende du bruit (cases bleues : probabilité 10^{-5}) ;
– quand le personnage essaye de détecter un danger, il entend du bruit (cases jaunes : probabilité 10^{-5} de ne pas entendre de bruit) ;
– quand le personnage attaque ou fuit, nous avons choisi de ne pas exprimer de connaissance sur le bruit (cases grises : distributions uniformes).

Nous faisons des hypothèses identiques pour le personnage agressif.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	10^{-5}
Vrai	x	0.1	0.1	10^{-5}	x	x
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	10^{-5}
Vrai	x	0.1	0.1	10^{-5}	x	x

TAB. 3.7 – $P(\text{Bruit}|\text{T}^{\text{t}})$

P(NbE|T$^{\text{t}}$) **(table 3.8)** Cette table décrit la distribution du nombre d'ennemis proches (aucun, un seul, ou deux et plus) suivant la tâche choisie.

Pour le personnage prudent, nous faisons les hypothèses suivantes :
– quand le personnage attaque, nous sommes sûrs qu'il a au moins un ennemi (case rouge : probabilité nulle), et nous pensons qu'il est plutôt improbable qu'il ait deux ennemis, puisqu'il est prudent (case verte : probabilité 0.1) ;
– quand le personnage cherche une arme, explore l'environnement, ou détecte le danger, c'est sans doute qu'il ne perçoit pas d'ennemi (cases bleues : probabilité faible d'avoir des ennemis proches) : en présence d'ennemis, ces tâches sont inefficaces ;
– quand il cherche un bonus de santé, nous ne savons rien du nombre d'ennemis (cases grises : distribution uniforme) : il peut être intéressant de rammasser un bonus de santé pendant le combat ;
– quand il fuit, nous sommes sûrs qu'il a au moins un ennemi, et nous pensons qu'il en a sans doute deux ou plus (cases jaunes).

Pour le personnage agressif, nous faisons les hypothèses suivantes :
– quand le personnage attaque, nous somme sûrs qu'il a au moins un ennemi (case rouge : probabilité nulle), et il est légèrement plus probable qu'il soit en présence de deux ennemis ou plus ;
– quand le personnage recherche un bonus de santé ou détecte le danger, nous sommes certains qu'il n'a pas d'ennemi (cases cyan : probabilités très faibles) ;
– quand le personnage fait une autre tâche, nous ne savons rien du nombre d'ennemis (cases grises : distributions uniformes).

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucun	0	x	x	x	0	x
Un	x	0.1	x	0.001	0.1	0.001
DeuxOuPlus	0.1	0.01	x	0.0001	x	0.0001
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucun	0	x	x	x	x	x
Un	0.2	x	x	10^{-9}	x	10^{-9}
DeuxOuPlus	0.8	x	x	10^{-10}	x	10^{-10}

TAB. 3.8 – $P(\text{NbE}|\text{T}^t)$

P(ArmeProche | T^t) **(table 3.9)** Cette table décrit la probabilité qu'une arme soit proche (variable booléenne), sachant la tâche choisie.

Pour le personnage prudent, nous faisons les hypothèses suivantes :

- quand le personnage recherche un bonus de santé, explore son environnement, ou détecte les dangers, il n'est généralement pas à proximité d'une arme (cases bleues : probabilité 10^{-5}) – en pratique, ces tâches auront donc très peu de chances de s'exécuter quand une arme est proche ;
- quand il exécute une autre tâche, nous ne savons rien de la proximité ou non d'une arme (cases grises : distributions uniformes).

Les hypothèses pour le personnage agressif sont identiques.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	x	10^{-5}	10^{-5}	x	10^{-5}
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	x	10^{-5}	10^{-5}	x	10^{-5}

TAB. 3.9 – $P(\text{ArmeProche}|\text{T}^t)$

P(SantéProche | T^t) **(table 3.10)** Cette table décrit la probabilité qu'un bonus de santé soit à proximité, selon la tâche exécutée.

Pour le personnage prudent, nous faisons les hypothèses suivantes :

- quand il recherche une arme, explore l'environnement ou cherche à détecter le danger, il n'est généralement pas à proximité d'un bonus de santé (cases bleues : probabilité 10^{-5}) – ces tâches ne sont donc pas appropriées quand un bonus de santé est proche ;
- quand il attaque, recherche un bonus de santé ou fuit, nous ne savons rien de la proximité d'un bonus de santé (cases grises : distributions uniformes).

Pour le personnage agressif, les hypothèses sont les mêmes.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	10^{-5}	x	10^{-5}	x	10^{-5}
agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	10^{-5}	x	10^{-5}	x	10^{-5}

TAB. 3.10 – $P(\text{SantéProche}|\text{T}^{\text{t}})$

Résultats

On peut observer et comprendre ces stratégies en observant le jeu du personnage. Par exemple, le personnage agressif est excité par la présence de plusieurs ennemis, alors que cette situation fait fuir le personnage prudent. Nous avons mesuré une différence de performance entre les comportements. Cette différence de performance est normale pour ce type de jeu (match à mort) où un personnage qui meurt n'encourt pas de pénalité particulière (comme un délai avant de réapparaître), et où les stratégies agressives sont généralement les meilleures.

Nous nous sommes basés sur des tâches de base programmées de manière classique (non bayésienne, dans un langage de programmation standard) – dont la qualité n'est pas au niveau des tâches de base natives d'*Unreal Tournament*. Ces tâches de base sont un facteur limitant de la performance pure de nos comportements. Les personnages natifs du jeu ne trichent pas (par exemple, ils ne perçoivent pas les autres personnages à travers les murs), mais disposent de tâches élémentaires très efficaces. Le jeu ajuste le niveau des personnages natifs par rapport au niveau maximal en réduisant leur vitesse et leur précision au tir.

La table 3.11 compare les performances au combat de quatre personnages :

- un personnage programmé à la main pour être agressif, et
- un personnage programmé à la main pour être prudent (dont nous venons de détailler les tables) ;
- un personnage dont la table de transitions est celle du personnage prudent (table 3.3), mais toutes les tables liées aux capteurs sont uniformes ;
- un personnage natif du jeu *Unreal Tournament*, de niveau moyen (3/8).

Ces personnages ont joué dix parties de match à mort, les uns contre les autres. Le premier personnage atteignant 100 ennemis tués remporte la partie, et obtient un score de 0. Pour les autres, le score pour cette partie correspond alors à la différence entre 100 et le nombre d'ennemis tués. La table 3.11 présente la moyenne de ce score pour chaque personnage, sur les 10 parties. Un personnage ayant remporté toutes les parties recevrait donc un score final de 0.

Cette table montre que nous avons pu créer des personnages aussi performants qu'un personnage natif du jeu. Elle montre qu'il est possible de créer plusieurs personnages aux stratégies différentes : un personnage agressif qui a tendance à attaquer, un personnage prudent qui fuit et recherche plus souvent à se soigner.

spécification manuelle, agressive	8.0
spécification manuelle, prudente	12.2
spécification manuelle, uniforme	43.2
personnage natif UT (niveau 3/8)	11.0

TAB. 3.11 – Comparaison des performances de trois personnages programmés à la main, et d'un personnage natif d'*Unreal Tournament* (score entre 0 et 100, un score faible correspond à un personnage performant au combat).

3.5 Comparaison avec les comportements d'*Unreal Tournament*

Nous pouvons résumer les caractéristiques de notre méthode de construction de comportements, en la comparant aux automates d'Unreal Tournament à l'aune de nos questions de problématique.

– **(Q1a : structure des paramètres)**

Les paramètres des comportements en programmation bayésienne sont faits de deux parties :

– une partie fixe, programmée à la main une fois pour toutes : le choix des variables, la décomposition de la distribution conjointe, et le choix des formes des facteurs de celle-ci ;
– une partie ajustable : les paramètres des facteurs de la décomposition.

Le premier bloc décrit le squelette du modèle, et repose sur une définition formelle sur le papier. Son implémentation peut prendre une variété de formes (bibliothèque plus ou moins générique, implémentation spécifique), qui ne reflète pas nécessairement le formalisme. La définition formalisée met des noms et un domaine clair pour les paramètres. Cependant, changer le domaine d'une variable ou la forme d'un facteur peut, selon l'implémentation, entraîner des modifications dans une grande partie de celle-ci : c'est un problème pratique de localité. De plus, changer certains des paramètres peut avoir des effets sur plusieurs aspects du comportements : c'est un problème d'identification. Enfin, il est difficile de manipuler automatiquement les paramètres de notre squelette : le nombre de décompositions possibles est très grand, le choix des formes des facteurs de la décomposition dépend d'une expertise du domaine d'application.

Ce premier bloc concentre en fait les éléments issus de l'expertise du programmeur. Il a l'avantage d'etre clairement formalisé, et bien délimité.

Le second bloc regroupe les paramètres des facteurs de la décomposition. Pour les modèles que nous avons décrits en exemple, ce sont les valeurs de tables de probabilités. Ces paramètres sont des nombres réels bien localisés et identifiés : chacun se trouve à un endroit unique du programme, et n'influence qu'une distribution. Leur structure est très simple.

Par rapport aux machine d'états finis d'Unreal Tournament, nos modèles ont l'avantage de séparer les paramètres utiles pour l'apprentissage et l'ajustement de comportements, de ceux issus de la structure du modèle, dépendant fortement de l'expertise du programmeur.

– **(Q1b : paramètres et intuition)**

Tous les paramètres d'un programme bayésien ont un sens intuitif. Nous avons vu comment les tables de probabilités de nos modèles de séquencement peuvent être construites, et interprétées. Un non-spécialiste peut les ajuster lui-même, moyennant une compréhension élémentaire de ce qu'est une distribution de probabilités conditionnelle.

L'avantage sur la méthode classique d'Unreal Tournament est principalement dû au fait que les paramètres modifiables sont clairement isolés, et étiquetés.

– **(Q2 : complexité en mémoire et temps de calcul)**

Dans le cas général, l'espace mémoire nécessaire pour représenter une tâche simple en programmation bayésienne n'est limité que par la taille de l'espace des variables, qui peut être très grande. Cette limite est abaissée en faisant des hypothèses d'indépendance conditionnelle dans la décomposition de la distribution conjointe. Il en est de même pour le temps de calcul – le problème de l'inférence bayésienne exacte ou approchée est NP-dur [Cooper 90] – qui doit être limité par des hypothèses appropriées.

Pour notre modèle de séquencement, l'espace nécessaire augmente linéairement avec le nombre moyen de valeurs pour les variables, et linéairement avec le nombre de variables. Il reste donc gérable.

En termes pratiques, notre modèle de séquencement, utilisant les tâches simples que nous avons présentées, est capable de prendre 10 décisions par seconde sur un *Pentium 4* 1.7 Ghz, en parallèle avec l'exécution d'Unreal Tournament.

Ces réponses aux questions montrent que notre méthode apporte une manière condensée et maniable pour définir un comportement structuré comme une machine d'états finis.

Pour un choix de variables, un comportement se présente comme un ensemble de tables de probabilités. Ces tables sont par essence des données qui peuvent être copiées, sauvegardées, modifiées, chargées simplement. C'est un point fort en termes de programmation.

De plus, la structure de machine à états est préservée dans les tables de probabilités : il est simple de modifier les transitions et d'ajouter des tâches. Cela n'était pas le cas dans l'implémentation scriptée utilisée par *Unreal Tournament*.

Nous n'avons pas abordé la question de l'apprentissage : est-il possible d'apprendre un comportement par démonstration ? C'est ce à quoi nous nous consacrons dans la partie suivante.

Chapitre 4

Apprendre un comportement

Notre but est maintenant d'enseigner un comportement à un personnage, au lieu de spécifier toutes les distributions à la main. Un enseignant humain pilote le personnage à la souris et au clavier, et nous récoltons des données sur ce pilotage, de façon à déterminer nos tables de probabilités.

Apprendre un comportement, c'est ajuster ses paramètres au regard :

– des connaissances *a priori* que nous avons sur les valeurs de ces paramètres ;

– de données expérimentales.

Les paramètres que nous apprenons sont les éléments des tables de probabilités qui sont au cœur de notre modèle. Par exemple, pour le modèle de séquencement de notre personnage *Unreal Tournament* nous voulons apprendre tous les nombres composant la table P(Santé | T^t), dont une version programmée à la main est rappelée sur la table 4.1.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.001	0.1	x	0.1	0.7	0.1
Moyen	0.1	x	0.01	x	0.2	x
Haut	x	x	0.001	x	0.1	x

TAB. 4.1 – P(Santé|T^t)

4.1 Sur ce que nous apprenons, et n'apprenons pas

Les paramètres auxquels nous nous intéressons sont ceux des facteurs de la décomposition de la distribution conjointe. Il est cependant envisageable d'apprendre d'autres éléments de nos programmes bayésiens : le choix des variables pertinentes parmi toutes celles disponibles [Dangauthier 03] [Dangauthier 05b] [Dangauthier 05a], le choix de la décomposition parmi toutes celles possibles [Diard 03a], ou le choix des formes paramétriques correspondant aux facteurs de la décomposition.

Cette restriction de l'apprentissage à certains paramètres est en premier lieu due au fait que l'apprentissage des autres éléments d'un programme bayésien représente un problème largement non résolu. De plus, dans le cadre restreint des modèles de comportements, il est possible de laisser l'établissement de ces éléments difficilement apprenables au programmeur sans que cela lui soit une charge insurmontable. Au contraire même, ces éléments représentent souvent la partie d'un comportement sur laquelle il est le plus facile d'exprimer des connaissances *a priori*. Cela est dû à leur nature largement structurelle : choix des variables, de leurs dépendances, et de leurs types de relations.

4.2 Principe de l'apprentissage des valeurs des paramètres

Le principe de l'apprentissage bayésien de paramètres est résumé sur la figure 4.1.

Deux questions sont possibles pour réaliser l'apprentissage. La première reflète que notre intérêt est de prédire la distribution des futures données Δ' sachant les données d'apprentissage Δ ; elle marginalise donc sur l'ensemble des paramètres Θ :

$$\begin{aligned} & \mathrm{P}(\Delta' \mid \Delta) \\ & = \frac{\sum_\Theta \mathrm{P}(\Theta)\, \mathrm{P}(\Delta' \mid \Theta)\, \mathrm{P}(\Delta \mid \Theta)}{\sum_\Theta \mathrm{P}(\Theta)\, \mathrm{P}(\Delta \mid \Theta)} \end{aligned} \tag{4.1}$$

Dans certains cas, cette question de prédiction peut être résolue analytiquement. C'est notamment le cas pour les tables de probabilités que nous utilisons dans nos comportements.

La seconde possibilité est d'estimer les paramètres dont la probabilité est maximale, sachant les données expérimentales observées pendant l'apprentissage :

$$\theta^* \;=\; \mathrm{ArgMax}_\theta\, \mathrm{P}(\theta \mid \delta) \tag{4.2}$$

Cette maximisation est appelée calcul du *maximum a posteriori* (MAP).

Quand la distribution *a priori* sur les paramètres se réduit à une distribution uniforme, il suffit pour trouver θ^* de maximiser la vraisemblance :

$$\text{si } \mathrm{P}(\Theta) \;=\; \mathrm{U},\; \theta^* \;=\; \mathrm{ArgMax}_\theta\, \mathrm{P}(\delta \mid \theta) \tag{4.3}$$

- Pour certains modèles simples munis d'un *a priori* $\mathrm{P}(\Theta)$ particulier, θ^* peut être connu analytiquement.
- Pour certains modèles $\mathrm{P}(\Delta \mid \Theta)$, il est possible d'utiliser un *a priori* particulier, appelé *distribution conjuguée* (en Anglais : *conjugate prior*) [Gelman 03], de sorte que $\mathrm{P}(\Theta \mid \Delta)$ soit calculable analytiquement. L'*a priori* sur les paramètres n'est plus alors seulement l'expression d'une connaissance préalable, puisque sa forme est contrainte par celle du modèle.
- Dans les autres cas, une intégration numérique est nécessaire. Le domaine de Θ étant souvent très grand, il est généralement nécessaire de recourir à des méthodes approchées.

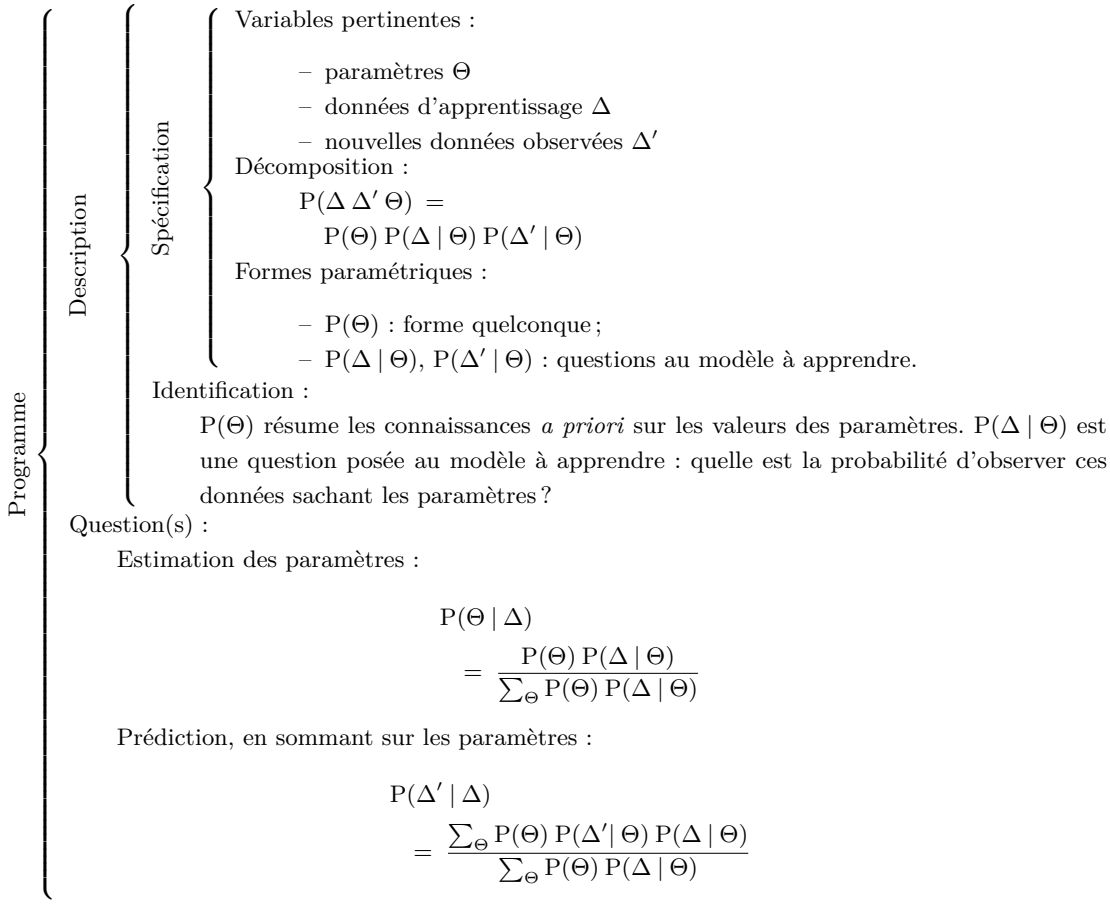

FIG. 4.1 – Apprentissage bayésien des paramètres d'un modèle

4.3 Apprendre une tâche simple

Dans une tâche simple, toutes les variables sont observées lors de l'apprentissage. De plus, nous n'utilisons que deux formes de distributions de probabilités :

- la table de probabilités non conditionnelle de la forme P(A) (où A peut éventuellement être une conjonction de variables) ;
- la table de probabilités conditionnelle de la forme P(A | B) (où A et B peuvent être des conjonctions de variables).

4.3.1 Table de probabilités, distribution de Laplace

Une table de probabilités sur une variable A de cardinal |A| est définie comme |A| nombres θ_i tels que :

- $\forall\, i,\ \theta_i \in [0, 1]$

– $\sum_i \theta_i = 1$
– $\forall\, i,\ P([A = a_i]) = \theta_i$

Nous faisons n observations de la variable A. Nous notons n_i le nombre de fois où le cas $[A = a_i]$ a été observé. Avec un *a priori* uniforme sur l'ensemble des paramètres θ_i, l'apprentissage par maximum *a posteriori* (équivalent ici au maximum de vraisemblance) nous donne les paramètres les plus probables :

$$\theta_i^* = \frac{n_i}{n}$$

Ces paramètres appris correspondent donc à l'histogramme des fréquences. Ce résultat peut être démontré en annulant la dérivée de la vraisemblance $P(\delta \mid \theta_1 \ldots \theta_{|A|})$.

Nous pouvons cependant faire mieux que cette estimation par maximum *a posteriori* : il est possible de résoudre analytiquement la question de prédiction $P(\Delta'|\Delta)$. Avec un *a priori* uniforme sur les paramètres, la probabilité d'observer le cas $[A = a_i]$ alors qu'il a été observé n_i fois pendant l'apprentissage parmi n observations suit une loi de succession de Laplace [Jaynes 03] :

$$P([A = a_i] \mid \Delta) = \frac{1 + n_i}{n + |A|} \tag{4.4}$$

Quand les observations d'apprentissage deviennent très nombreuses, cette expression tend vers celle de l'histogramme de maximum de vraisemblance. Mais la loi de succession de Laplace a les avantages suivants sur l'histogramme des fréquences :

– elle est bien définie quand aucune donnée n'a encore été observée ;
– quand peu d'observations ont été faites, les cas pas encore observés sont prédits avec une probabilité non-nulle.

Ces deux propriétés permettent d'utiliser la loi de succession de Laplace dans le cadre d'un schéma d'apprentissage incrémental, où la loi apprise au pas de temps t est utilisée pour apprendre au pas de temps $t + 1$.

En termes pratiques, apprendre une loi de succession de Laplace est équivalent à apprendre l'histogramme des fréquences, en introduisant avant l'apprentissage une donnée virtuelle pour chaque valeur de la variable.

En guise d'exemple, nous reprenons le comportement de détection du danger que nous avons présenté lors de notre introduction à la programmation de comportements simples. Ce comportement est rappelé sur la figure 4.2. Nous cherchons à apprendre le facteur $P(V_{rot})$, au lieu de le spécifier à la main.

Lors de l'apprentissage, nous faisons $n = 100$ observations de la variable $A = V_{rot}$ (de cardinal $|A| = |V_{rot}| = 7$), qui se répartissent comme sur le tableau 4.2.

v_{rot}	-3	-2	-1	0	1	2	3
occurrences	31	4	0	0	3	10	52

TAB. 4.2 – Répartition des occurrences de chaque valeur de la variable V_{rot} lors de l'apprentissage du comportement de détection du danger.

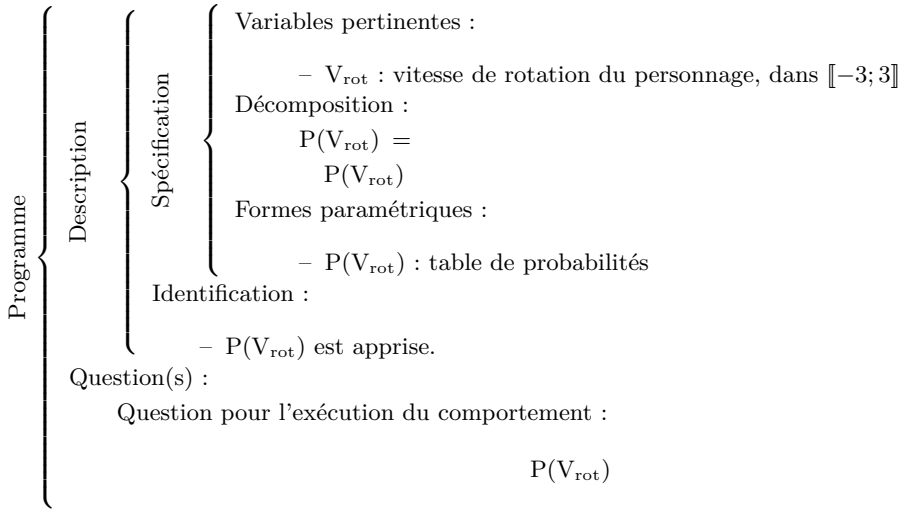

FIG. 4.2 – Comportement de détection du danger

Avec ces donnés d'apprentissage, la table de probabilités de maximum de vraisemblance est celle des fréquences. Nous la donnons dans la table 4.3.

v_{rot}	-3	-2	-1	0	1	2	3
$P([V_{rot} = v_{rot}])$	0.31	0.04	0	0	0.03	0.10	0.52

TAB. 4.3 – Table de probabilités de maximum de vraisemblance pour les données d'apprentissage de la table 4.2.

Avec ces même données, la loi de succession de Laplace donne l'histogramme de la table 4.4. On peut remarquer que les cas jamais observés ($[V_{rot} = -1]$, $[V_{rot} = 0]$) correspondent à une probabilité faible, mais non-nulle (cases jaunes : probabilité 0.00935).

v_{rot}	-3	-2	-1	0	1	2	3
$P([V_{rot} = v_{rot}])$	0.299	0.0467	0.00935	0.00935	0.0374	0.103	0.495

TAB. 4.4 – Table de probabilités $P(V_{rot})$ apprise avec la loi de succession de Laplace pour les données d'apprentissage de la table 4.2.

4.3.2 Table de probabilités conditionnelle

Nous avons vu comment apprendre une table de probabilités non conditionnelle, de la forme P(A), avec une loi de succession de Laplace. Nos modèles incluent des tables conditionnelles de la forme $P(A \mid B)$. Pour les apprendre, nous nous ramenons au cas non conditionnel :

$$
\begin{aligned}
&\mathrm{P}([\mathrm{A}=\mathrm{i}] \mid [\mathrm{B}=\mathrm{j}]\ \Delta) \\
&\quad = \frac{\mathrm{P}([\mathrm{A}=\mathrm{i}]\ [\mathrm{B}=\mathrm{j} \mid \Delta])}{\sum_{\mathrm{i}} \mathrm{P}([\mathrm{A}=\mathrm{i}]\ [\mathrm{B}=\mathrm{j} \mid \Delta])} \\
&\quad = \frac{\frac{1+\mathrm{n}_{\mathrm{i,j}}}{\mathrm{n}+|\mathrm{A}| \times |\mathrm{B}|}}{\sum_{\mathrm{i}} \frac{1+\mathrm{n}_{\mathrm{i,j}}}{\mathrm{n}+|\mathrm{A}| \times |\mathrm{B}|}} \\
&\quad = \frac{1+\mathrm{n}_{\mathrm{i,j}}}{\mathrm{n}+|\mathrm{A}|\ \times\ |\mathrm{B}|} \times \frac{\mathrm{n}+|\mathrm{A}|\ \times\ |\mathrm{B}|}{|\mathrm{A}|+\mathrm{n}_{\mathrm{j}}} \\
&\quad = \frac{1+\mathrm{n}_{\mathrm{i,j}}}{|\mathrm{A}|+\mathrm{n}_{\mathrm{j}}}
\end{aligned} \tag{4.5}
$$

Dans cette formule, $\mathrm{n}_{\mathrm{i,j}}$ est le nombre d'occurrences observées du cas $[\mathrm{A}=\mathrm{a}_{\mathrm{i}}]$, $[\mathrm{B}=\mathrm{b}_{\mathrm{j}}]$, et n_{j} est le nombre d'occurrences de $[\mathrm{B}=\mathrm{b}_{\mathrm{j}}]$.

Nous pouvons ainsi envisager d'apprendre le comportement de suivi de cible que nous avons décrit plus haut, et que nous rappelons sur la figure 4.3. Au lieu de spécifier le facteur $\mathrm{P}(\mathrm{V}_{\mathrm{rot}} \,|\, \mathrm{Dir})$ à la main, nous l'apprenons.

Programme
- Description
  - Spécification
    - Variables pertinentes :
      - $\mathrm{V}_{\mathrm{rot}}$: vitesse de rotation du personnage, dans $[\![-3;3]\!]$
      - Dir : direction relative de la cible, dans $[\![-4;4]\!]$
    - Décomposition :
      $\mathrm{P}(\mathrm{V}_{\mathrm{rot}}\ \mathrm{Dir}) =$
      $\mathrm{P}(\mathrm{Dir})\ \mathrm{P}(\mathrm{V}_{\mathrm{rot}} \mid \mathrm{Dir})$
    - Formes paramétriques :
      - $\mathrm{P}(\mathrm{Dir})$: inconnu
      - $\mathrm{P}(\mathrm{V}_{\mathrm{rot}} \mid \mathrm{Dir})$: table de probabilités
  - Identification :
    - $\mathrm{P}(\mathrm{V}_{\mathrm{rot}} \mid \mathrm{Dir})$: table de probabilité apprise.
- Question(s) :
  Question pour l'exécution du comportement :
  $\mathrm{P}(\mathrm{V}_{\mathrm{rot}} \mid \mathrm{Dir})$

FIG. 4.3 – Comportement de suivi de cible

Nous faisons $\mathrm{n} = 100$ observations du couple $(\mathrm{A}, \mathrm{B}) = (\mathrm{V}_{\mathrm{rot}}, \mathrm{Dir})$, qui sont résumées dans la table 4.5.

La table de probabilité associée à ces données, calculée avec la loi de Laplace, est donnée sur la table 4.6.

Nous pouvons remarquer que :

$v_{rot} \setminus$ dir	−4	−3	−2	−1	0	1	2	3	4
−3	6	14	11	0	0	0	0	0	0
−2	1	1	1	4	0	0	0	0	0
−1	0	0	0	5	1	0	0	0	0
0	0	0	0	0	11	0	0	0	0
1	0	0	0	0	2	3	0	0	0
2	0	0	0	0	0	4	1	1	0
3	0	0	0	0	0	0	13	14	7

TAB. 4.5 – Répartition des occurrences de chaque valeur du couple (V_{rot}, Dir) lors de l'apprentissage du comportement de suivi de cible.

$V_{rot} \setminus$ Dir	−4	−3	−2	−1	0	1	2	3	4
−3	0.5	0.682	0.632	0.0625	0.0476	0.0714	0.0476	0.0455	0.0714
−2	0.143	0.0909	0.105	0.312	0.0476	0.0714	0.0476	0.0455	0.0714
−1	0.0714	0.0455	0.0526	0.375	0.0952	0.0714	0.0476	0.0455	0.0714
0	0.0714	0.0455	0.0526	0.0625	0.571	0.0714	0.0476	0.0455	0.0714
1	0.0714	0.0455	0.0526	0.0625	0.143	0.286	0.0476	0.0455	0.0714
2	0.0714	0.0455	0.0526	0.0625	0.0476	0.357	0.0952	0.0909	0.0714
3	0.0714	0.0455	0.0526	0.0625	0.0476	0.0714	0.667	0.682	0.571

TAB. 4.6 – Table de probabilités P(V_{rot}|Dir) apprise avec la loi de successions de Laplace, d'après les données d'apprentissage de la table 4.5.

– l'observation de zéro donnée dans une case entraîne une probabilité faible mais non-nulle ;
– le calcul de la table de probabilités revient simplement à ajouter 1 dans chaque case du tableau des observations, et à normaliser chaque colonne de façon à ce qu'elle se somme à 1.

4.3.3 Conclusion

Nous savons donc apprendre des tables de probabilités directes et conditionnelles : c'est tout ce qui est nécessaire pour ajuster par apprentissage les paramètres de nos tâches simples.

Nos formules d'apprentissage sont basées sur le comptage d'échantillons (n_i, $n_{i,j}$). Elles sont très rapides à évaluer, et la mémoire nécessaire pour apprendre une distribution est équivalente à celle nécessaire pour le stockage de la distribution elle-même (le produit des tailles des variables de la distribution).

Les comportements séquencés incluent une variable interne non mesurable : la tâche du personnage. Celle-ci n'est pas mesurable directement, et il nous faut une méthode spécifique pour réaliser l'apprentissage de ce type de comportement.

4.4 Apprendre un séquencement de tâches

Nous devons donc à chaque instant mesurer les variables sensorielles et motrices du personnage contrôlé. Les variables sensorielles (niveau de santé, arme, nombre d'ennemis...) nous sont directement accessibles. Mais la tâche T^t ne l'est pas, et nous devons la déterminer à chaque instant. Cela peut être fait en laissant le joueur la spécifier lui-même en temps réel alors qu'il joue, ou en inférant la tâche d'après les mesures bas niveau disponibles. Cette inférence peut être faite de manière *ad hoc*, en utilisant des heuristiques propres au jeu ; elle peut aussi être faite de manière générique si les tâches élémentaires sont implémentées dans le cadre de la programmation bayésienne.

4.4.1 Apprentissage avec sélection explicite de la tâche

Contrôle d'un personnage par sélection de la tâche

Cette forme d'apprentissage présente une interface simple au joueur, comme sur la figure 4.4. Le joueur contrôle son personnage au moyen d'un sélecteur qui lui permet de choisir la tâche à appliquer (Attaque, RechArme, RechSanté, Exploration, Fuite, DétDanger), en temps réel avec la souris. Il voit son personnage évoluer dans l'environnement dans une fenêtre séparée. Il n'a pas de contrôle plus spécifique que ce sélecteur de tâche : il ne peut pas choisir de direction de déplacement particulière.

FIG. 4.4 – Interface d'apprentissage par démonstration, avec sélection explicite de la tâche (à gauche). La fenêtre de droite affiche une vue à la troisième personne du personnage contrôlé.

Résultats

Nous connaissons donc à chaque instant les valeurs de chacune des variables de notre modèle : la tâche précédente T^{t-1}, la tâche courante T^t, et les variables sensorielles (Santé, Arme, ArmeE, Bruit, NbE, ArmeProche, SantéProche).

Nous pouvons donc ajuster chacune de nos tables aux données de l'apprentissage, suivant la loi de succession de Laplace. Par exemple, pour la table de transition $P(T^t \mid T^{t-1})$, nous avons :

$$\begin{aligned} & P([T^t = i] \mid [T^{t-1} = j]) \\ &= \frac{P([T^t = i] \mid [T^{t-1} = j])}{P([T^{t-1} = j])} \\ &= \frac{1 + n_{i,j}}{|T|^2 + \sum_{i,j} n_{i,j}} \\ &\times \frac{|T|^2 + \sum_j n_j}{1 + n_j} \\ &= \frac{1 + n_{i,j}}{|T| + n_j} \end{aligned} \tag{4.6}$$

Dans cette formule, $|T|$ est le nombre de valeurs possibles pour T^t (ici, 6) et $n_{i,j}$ est le nombre d'occurrences de l'observation $T^t = i$, $T^{t-1} = j$.

Pour évaluer le comportement produit, nous mesurons les performances au combat du personnage. Celui-ci est mis en compétition avec d'autres personnages autonomes dans une série de matches à mort. Chaque match prend fin quand l'un des combattants atteint le score de 100 – le score correspondant au nombre d'adversaires tués, moins le nombre de fois où le personnage

s'est tué lui-même par accident (chute, explosion trop proche d'une roquette...). Pour chaque match, la performance d'un personnage est sa différence de score à la fin du match par rapport au vainqueur. La performance globale du personnage est la moyenne de ses performances sur 10 matches. Une hypothétique performance globale de 0 correspond donc à un personnage qui a gagné les 10 matches ; une performance globale de 100 correspond à un personnage qui n'a tué aucun adversaire pendant les matches.

La table 4.7 montre les performances du comportement appris par sélection de la tâche, comparées à celles de trois comportements programmées à la main, et d'un personnage natif du jeu *Unreal Tournament*. Ce personnage natif est du niveau d'un joueur humain moyen. Nous avons présenté auparavant les comportements "prudent" et "agressif" programmés à la main. Le comportement "uniforme" programmé à la main correspond à un modèle de séquencement dont les tables $P(\text{Capteur} \mid T^t)$ sont uniformes, et dont la table $P(T^t \mid T^{t-1})$ est une diagonale forte, comme pour le personnage prudent.

Cette table montre que l'apprentissage par sélection de la tâche produit des comportements du niveau d'un séquencement quasi-aléatoire. Cela est sans doute dû à la difficulté de contrôler un personnage à travers une interface non naturelle. C'est pourquoi nous nous consacrons à l'utilisation de l'interface native de contrôle des personnages.

apprentissage par sélection de la tâche, agressive	45.7
spécification manuelle, agressive	8.0
spécification manuelle, prudente	12.2
spécification manuelle, uniforme	43.2
personnage natif UT (niveau 3/8)	11.0

TAB. 4.7 – Comparaison des performances d'un comportement appris par sélection de la tâche, de personnages programmés à la main, et d'un personnage natif d'*Unreal Tournament* (score entre 0 et 100, un score faible correspond à un personnage performant au combat).

4.4.2 Apprentissage avec reconnaissance de tâche par une heuristique

Heuristique de reconnaissance de tâche

L'expertise d'un joueur pour démontrer un comportement s'exprime mieux avec les contrôles natifs du jeu ; pour *Unreal Tournament*, il s'agit d'un mode de déplacement et d'action au clavier et à la souris, avec un retour visuel 3D à la première personne.

Nous utilisons [Le Hy 04] [Arrigoni 03] une heuristique programmée d'une manière impérative classique pour reconnaître la tâche que le joueur humain démontre le plus probablement à un instant donné.

Un programme de reconnaissance est construit pour chaque tâche élémentaire. Chacun de ces programmes de reconnaissance est fait sur l'un de quatre schémas de reconnaissance. Ces schémas sont exécutés dans l'ordre, jusqu'à ce que l'un d'eux arrive à reconnaître la tâche.

(1) RechSanté et RechArme sont à **validation retardée**. Lorsque le personnage atteint un point de navigation caractéristique (portant une recharge de santé ou une arme), les tâches précédentes inconnues sont identifiées à RechSanté ou RechArme.

(2) DétDanger est à **validation fixe**. Si dans les deux dernières secondes le joueur a fait un demi-tour sur lui-même, DétDanger est affecté aux 20 points de données précédents.

(3) Fuite et Attaque sont à **validation instantanée**. Si le personnage se tient face à un ennemi, et si sa vitesse par rapport à cet ennemi est en rapport avec l'arme qu'il porte et sa distance, la tâche Attaque est identifiée. Si le personnage s'éloigne de l'ennemi le plus proche et qu'il n'attaque pas (pas de cible, ou vitesse incohérente), la tâche Fuite est identifiée.

(4) Si aucun des programmes de reconnaissances ci-dessus n'est déclenché pour un point de données, la tâche pour ce point est marquée inconnue. Quand il n'est plus possible que cette tâche soit identifiée *a posteriori* par validation retardée (1) ou validation fixe (2), elle est traitée comme la **tâche par défaut** : Exploration.

Cette procédure vise à décider de manière univoque de la tâche qu'est en train d'exécuter le joueur. Il peut apparaître des ambiguïtés. Par exemple, quand le joueur court dans une pièce sans ennemis, nous ne pouvons pas savoir s'il explore l'environnement, part chercher un bonus de santé ou une arme, ou même s'il poursuit un ennemi qu'il a entendu dans la pièce d'à côté. Une partie de cette ambiguïté est levée en considérant une fenêtre de temps pour la recherche de bonus de santé ou d'arme, et en propageant vers l'arrière l'information du ramassage d'un objet. Cependant, toute incertitude ne peut être levée (en particulier à cause du problème de la taille des fenêtres temporelles). Cette méthode de reconnaissance de tâche fait l'hypothèse qu'il n'est pas nécessaire de prendre en compte l'incertitude restante, et qu'une décision univoque sur la tâche peut être prise sans compromettre l'apprentissage.

Résultats

apprentissage par reconnaissance automatique de la tâche, agressive	4.4
apprentissage par reconnaissance automatique de la tâche, prudente	13.9
apprentissage par sélection de la tâche, agressive	45.7
spécification manuelle, agressive	8.0
spécification manuelle, prudente	12.2
spécification manuelle, uniforme	43.2
personnage natif UT (niveau 3/8)	11.0

TAB. 4.8 – Comparaison des performances de comportements appris par reconnaissance automatique de la tâche, par sélection de la tâche, spécifiés manuellement, et d'un personnage natif d'*Unreal Tournament* (score entre 0 et 100, un score faible correspond à un personnage performant au combat).

10 à 15 minutes de démonstration permettent d'apprendre un comportement. La table 4.8 reprend nos résultats précédents de programmation manuelle et d'apprentissage par sélection de la tâche. Sont ajoutés (en gris) deux comportements appris par l'heuristique de reconnaissance de

la tâche. L'un est appris par un joueur jouant prudemment, qui s'attache à rechercher des armes fortes et à conserver un niveau de santé important ; l'autre par un joueur jouant agressivement.

Cette méthode d'apprentissage permet clairement de produire des comportements efficaces au combat. Malgré les limites de nos tâches élémentaires, le comportement enseigné par un joueur agressif est significativement meilleur que personnage natif de niveau 3.

Commentaires sur l'apprentissage par reconnaissance avec heuristique

La table 4.9 correspond à $P(T^t \mid T^{t-1})$ tel qu'elle est apprise avec la méthode d'apprentissage avec reconnaissance par heuristique. Elle peut être comparée à la table 4.10, qui donne la version agressive que nous avons développée à la main.

La diagonale forte (cases bleues) se retrouve dans les deux tables. La table apprise présente des probabilités de transition d'une tâche à une tâche différente (cases hors diagonale) sensiblement plus élevées que celles spécifiées à la main (toutes égales à 0.01). Dans la table apprise, certaines transitions hors de la diagonale sont favorisées (cases vertes) : Attaque vers Exploration (probabilité 0.097), Exploration vers Attaque (probabilité 0.107), Fuite vers Exploration (probabilité 0.122).

agressif	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	0.865	0.06	0.011	0.107	0.019	0.060
RechArme	0.04	0.968	0.004	0.003	0.007	0.030
RechSanté	0.012	0.008	0.962	0.002	0.018	0.030
Exploration	0.097	0.014	0.019	0.846	0.122	0.030
Fuite	0.022	0.003	0.003	0.041	0.832	0.30
DétDanger	0.001	0.002	0.002	0.002	0.001	0.818

TAB. 4.9 – $P(T^t|T^{t-1})$ apprise avec la méthode d'apprentissage avec reconnaissance par heuristique

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	x	0.01	0.01	0.01	0.01	0.01
RechArme	0.01	x	0.01	0.01	0.01	0.01
RechSanté	0.01	0.01	x	0.01	0.01	0.01
Exploration	0.01	0.01	0.01	x	0.01	0.01
Fuite	0.01	0.01	0.01	0.01	x	0.01
DétDanger	0.01	0.01	0.01	0.01	0.01	x

TAB. 4.10 – $P(T^t|T^{t-1})$: version prudente écrite à la main

La table 4.11 correspond à la distribution $P(\text{Santé}|T^t)$, telle qu'apprise à l'aide de l'heuristique de reconnaissance de tâche. Il est intéressant de la rapprocher de la table 4.12, qui rappelle notre version manuelle de $P(\text{Santé} \mid T^t)$ pour le personnage prudent.

Entre les niveaux de santé Bas et Moyen, nous retrouvons dans la table apprise certaines des spécifications de la table écrite à la main :

- quand le personnage attaque, il a peu de chances d'avoir un niveau de santé faible (cases bleues) ;
- quand le personnage explore, il a peu de chances d'avoir un niveau de santé faible (cases vertes) ;
- quand le personnage fuit, il est plutôt probable que son niveau de santé soit faible, et un peu moins probable que son niveau de santé soit moyen (cases rouges) ;
- quand le personnage détecte le danger, il est très peu probable que son niveau de santé soit faible (cases jaunes).

Pour le niveau de santé Haut, les valeurs de la table apprise sont toutes inférieures à celles décrite dans la table programmée à la main. Cela est dû au fait que le style de jeu du joueur humain l'a conduit à n'avoir un niveau de santé Haut que très rarement. La table décrite à la main ne prenait pas cet élément en compte, et faisait implicitement l'hypothèse que les niveaux de santé seraient uniformément couverts lors du jeux.

Une différence importante est observée pour la tâche RechSanté (cases grises). Dans le comportement appris, le niveau de santé Bas n'est pas privilégié. Nous pouvons l'expliquer en remarquant qu'un joueur humain a généralement un comportement opportuniste pour ramasser des bonus de santé : il le fait principalement quand il aperçoit un bonus proche, sans tenir compte de son état de santé courant. Par conséquent, la table apprise $P(\text{Santé} \mid [T^t = \text{RechSanté}])$ est proche de P(Santé).

Santé \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.179	0.342	0.307	0.191	0.457	0.033
Moyen	0.478	0.647	0.508	0.486	0.395	0.933
Haut	0.343	0.011	0.185	0.323	0.148	0.033

TAB. 4.11 – $P(\text{Santé} \mid T^t)$ apprise par l'heuristique de reconnaissance de tâche

Santé \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.001	0.1	x	0.1	0.7	0.1
Moyen	0.1	x	0.01	x	0.2	x
Haut	x	x	0.001	x	0.1	x

TAB. 4.12 – $P(\text{Santé} \mid T^t)$ spécifiée à la main pour le personnage prudent

Les tables 4.13 et 4.14 présentent $P(\text{Bruit} \mid T^t)$, telle qu'elle est apprise avec l'heuristique de reconnaissance de tâche, et telle que nous l'avons spécifiée à la main. Nous présentons deux tables apprises : dans la première, le joueur humain disposait du son quand il a réalisé l'apprentissage ; dans la seconde, il n'en disposait pas.

Ces tables décrivant $P(\text{Bruit} \mid T^t)$ montrent des informations très différentes entre la spécification manuelle et l'apprentissage pour toutes les tâches. Cependant, quand le personnage détecte

Bruit \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	0.418	0.535	0.659	0.526	0.483	0.909
Vrai	0.582	0.465	0.340	0.474	0.517	0.091
Bruit \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	0.227	0.374	0.259	0.274	0.228	0.138
Vrai	0.772	0.626	0.741	0.726	0.772	0.862

TAB. 4.13 – P(Bruit | T^t) apprise par l'heuristique de reconnaissance de tâche. En **haut** : avec le son ; en **bas** : sans le son.

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	10^{-5}
Vrai	x	0.1	0.1	10^{-5}	x	x

TAB. 4.14 – P(Bruit | T^t) spécifiée à la main pour le personnage prudent

le danger, la distribution apprise sur le bruit est proche de celle spécifiée à la main dans le cas où le joueur humain n'entendait pas le son en jouant (cases vertes) ; la version apprise quand le joueur humain entendait le son (cases rouges) est opposée à celle spécifiée à la main.

Ces différences et cette similarité mettent en relief la différence de perception qui sépare le joueur humain du personnage autonome. Le personnage autonome ne perçoit qu'un indicateur booléen indiquant si un son est entendu ; ce son peut provenir d'un tir, mais aussi de l'impact d'un projectile sur un mur, de la chute d'un personnage, du ramassage d'un objet... Le joueur humain perçoit chacun de ces sons de manière différente et sait les identifier ; de plus, la stéréophonie et certains indices (éclats des balles sur le mur, mémoire de la position d'objets proches) lui permet d'identifier la direction approximative de leur source. Essayer de mettre en correspondance ces deux informations sensorielles est vain : nous obtenons des tables apprises sans rapport avec celles spécifiées à la main.

Cependant, en supprimant le son au joueur, nous avons rapproché les perceptions du joueur et du personnage autonome. Le joueur doit alors se fier aux indices visuels qui sont associés au son : impacts sur les murs, trajectoires des projectiles. Pour la tâche de détection du danger, cela est suffisant pour faire correspondre table apprise et table spécifiée à la main. Nous tirons donc la leçon suivante : un capteur simplifié à l'excès (comme Bruit) conduira à des tables apprises difficiles à interpréter.

En résumé, les tables apprises ont de nombreux points en commun avec celles établies à la main. Par exemple, pour P(Santé | T^t = Fuite), la probabilité d'avoir un niveau de santé faible ou moyen est forte devant celle d'avoir un niveau de santé élevé.

Des différences dans le comportement du professeur influencent le comportement appris : l'agressivité et la prudence s'y retrouvent, et se traduisent en variations de performance au combat. Dans notre expérience particulière, les comportements agressifs semblent en général plus efficaces.

Des différences entre les tables spécifiées à la main et les tables apprises peuvent être remarquées. Elles peuvent être expliquées comme :

- **des comportements spécifiques du professeur humain**, que nous n'avions pas imaginé en écrivant nos tables à la main. Par exemple, un joueur humain se replie rarement, et préfère généralement attaquer. Un autre est la faible probabilité de $P([\text{Santé} = \text{Haut}] \,|\, T^t)$ dans la table 4.11 ; elle peut être expliquée par le fait qu'un joueur humain maintient rarement un niveau de santé élevé, et attaque plutôt que de chercher une arme quand c'est le cas.
- **des informations supplémentaires dans les données issues de l'apprentissage**, que nous ne savions pas prédire en programmant les tables à la main. Certaines de nos tables écrites à la main contiennent des distributions uniformes : il nous était impossible de spécifier *a priori* un lien entre certains évènements, comme les différentes valeurs de l'arme de l'adversaire sachant que notre personnage explore son environnement. Il n'est pas surprenant que les tables apprises ne reproduisent pas ces distributions uniformes.
- **des différences de perception**. En particulier, un joueur humain et un personnage synthétique ont une perception différente des sons. L'humain perçoit la direction du son, son origine (un impact de balle sur un mur, le bruit du tir lui-même...) et sa qualité (la distance approximative, le genre d'arme...) et prend en compte les autres modalités perceptives (la vision est importantes pour évaluer les caractéristiques d'un tir) ; le personnage synthétique ne prend en compte qu'une direction. Cela a une influence importante sur la réaction possible que le joueur, humain ou synthétique peut mettre en œuvre quand il entend un bruit.
- **des biais introduits par les quantités de données et la discrétisation des variables**. Par exemple, notre choix de discrétisation pour le niveau de santé n'est pas optimal, et il se trouve qu'un joueur humain se trouve la plupart du temps au niveau de santé 'Moyen' ; cela explique en partie les valeurs importantes sur la ligne Santé = Moyen dans la table 4.11. Trop fine, la discrétisation rend difficile la spécification à la main des comportements ; elle ralentit l'inférence, augmente la taille des données en mémoire, et la quantité de données nécessaire pour apprendre le modèle. Trop grossière, elle coagule des états sensoriels différents, ce qui réduit l'expressivité du modèle. Au vu de la distribution du niveau de santé dans les expériences d'apprentissage, celui-ci pourrait être discrétisé de la façon suivante : *très bas* sous 70, *bas* entre 71 et 95, *moyen* entre 96 et 110, *haut* entre 111 et 130, *très haut* au-dessus de 131.

Les différences entres tables apprises et tables spécifiées à la main sont les signes d'un modèle difficile à interpréter sur certains points. Ces problèmes d'interprétation n'entraînent pas de problèmes de performance pour les personnages appris ; par contre, ils peuvent être sources de difficultés dans la programmation manuelle.

4.4.3 Apprentissage avec reconnaissance bayésienne de tâche basée sur les modèles de tâche

Nous cherchons à présent à réaliser la reconnaissance des tâches non pas par la programmation d'heuristiques *ad hoc*, mais de manière systématique. Cela est possible en nous appuyant sur des tâches basées sur la programmation bayésienne, et en considérant le modèle de séquencement par programmation inverse comme un modèle de Markov caché.

Modèle de Markov caché : définition

Un modèle de Markov caché (figure 4.5) lie une variable d'observation et une variable d'état, à une série d'instants 0, 1 ... n.

La figure 4.5 décrit la forme générale d'un modèle de Markov caché [Rabiner 89]. L'idée essentielle de ce programme est de condenser l'état interne du personnage dans une variable T nommée variable d'état. Notre personnage est également pourvu d'une variable d'observation O. Le programme prend en compte l'état et l'observation à une série d'instants consécutifs 0, 1, ..., n.

Les hypothèses essentielles de ce modèle sont :

- sachant l'état T^i, les états T^{i+1}, T^{i+2}, ... ne dépendent pas des états T^0, ... T^{i-1} (hypothèse de Markov d'ordre 1 [Markov 6]) ;
- sachant l'état T^i, l'observation O^i ne dépend pas de T^0, ... T^{i-1}, T^{i+1}, T^{i+2}, ...

Programme

- Description
  - Spécification
    - Variables pertinentes :
      états T^0 ... T^n, observations O^0 ... O^n
    - Décomposition :
      $P(T^0 \dots T^n\ O^0 \dots O^n) =$
      $P(T^0)\, P(O^0 \mid T^0)\, \prod_{i=0}^{1} P(T^i \mid T^{i-1})\, P(O^i \mid T^i)$
    - Formes paramétriques :
      - $P(O^0)$: table de probabilités
      - $P(T^i \mid T^{i-1})$: table de probabilités
      - $P(O^i \mid T^i)$: table de probabilités
  - Identification :
    Les tables peuvent être connues au départ, ou apprises. Elles constituent les paramètres du modèle, notés θ.
- Question(s) :
  Parmi les questions usuelles :
  - dernier état sachant les observations : $P(T^n \mid O^0 .. O^n)$
  - dernier état sachant les observations et les états précédents : $P(T^n|O^0..O^nT^0..T^{n-1})$
  - ensemble des états sachant les observations : $P(T^0 ..T^n \mid O^0 .. O^n)$

FIG. 4.5 – Modèle de Markov caché : définition générale.

La distribution conjointe de ce programme présente d'un bloc toutes les variables du temps 0 au temps n. En pratique, la nature récursive de ce modèle permet pour certaines questions de

travailler de manière glissante, à la manière d'un filtre.

L'algorithme de Baum-Welch pour apprendre un modèle de Markov caché

Principe de l'algorithme de Baum-Welch L'algorithme de Baum-Welch [Baum 72] [Rabiner 89] permet d'apprendre les paramètres d'un modèle de Markov caché. Ces paramètres sont les tables de probabilité décrivant le modèle de transition d'état à état $P(T^i \mid T^{i-1})$ et le modèle d'observation $P(O^i \mid T^i)$. Nous notons Θ l'ensemble de ces paramètres numériques.

Nous avons vu comment apprendre ces tables comme des lois de Laplace si nous sommes capables d'évaluer l'état (notre tâche) à un instant donné à l'aide d'une heuristique.

L'algorithme de Baum-Welch propose d'évaluer une distribution sur l'état sachant les observations, et d'utiliser cette distribution pour établir les tables de probabilité faisant intervenir l'état. Ces tables sont donc construites comme des histogrammes, à ceci près que les échantillons qui y sont injectés sont proportionnels à la probabilité inférée sur leurs variables.

Cet algorithme va donc, sur le principe d'un algorithme *Expectation-Maximization* (EM) [Dempster 77], alterner entre :

- l'évaluation de la distribution de probabilité des variables non-observées sachant une estimation des paramètres, et
- la maximisation des paramètres sachant la distribution des variables non observées, et les valeurs des variables observées.

Il est garanti que cet algorithme converge vers un maximum local de la vraisemblance, c'est-à-dire de la probabilité d'observation des variables observées, sachant les paramètres des distributions apprises.

Algorithme de Baum-Welch L'algorithme de Baum-Welch considère l'ensemble des observations disponibles d'un coup ; il n'est pas incrémental. Il consiste à réestimer itérativement les paramètres du modèle, jusqu'à convergence (que l'on peut détecter numériquement). L'essentiel est donc de savoir passer d'une estimation courante des paramètres du modèle θ à une meilleure estimation $\bar{\theta}$.

La structure générale de l'algorithme d'apprentissage est la suivante :

initialiser les paramètres θ;
convergence $\leftarrow$ faux;
tant que *non* convergence **faire**
  réestimer $\bar{\theta}$ en fonction de θ et $o^0 ... o^n$;
  convergence $\leftarrow |\bar{\theta} - \theta| < \epsilon$;
  $\theta \leftarrow \bar{\theta}$;
fin

Algorithme 1 : Algorithme de Baum-Welch

Réestimer les paramètres θ signifie réestimer la table de transition $P([T^t = j] \mid [T^{t-1} = i])$ et le modèle d'observation $P([O^t = k] \mid [T^t = j])$.

Nous notons (transitions $i \rightarrow j$) le nombre espéré de transitions de l'état i vers l état j, et (transitions $i \rightarrow \cdot$) le nombre espéré de transitions à partir de l'état i. Notons (observations k en j)

le nombre espéré d'observations de la valeur k dans l'état j.

La formule de réestimation de la table de transition est :

$$\begin{aligned} & \mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, \bar{\theta}) \\ & \quad = \frac{(\text{transitions i } \rightarrow \text{ j})}{(\text{transitions i } \rightarrow \cdot)} \qquad (4.7) \\ & \quad = \frac{\sum_{\mathrm{t}=1}^{\mathrm{n}} \mathrm{P}([\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, [\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid \mathrm{o}^0 \ldots \mathrm{o}^{\mathrm{n}}\, \theta)}{\sum_{\mathrm{t}=1}^{\mathrm{n}} \mathrm{P}([\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}] \mid \mathrm{o}^0 \ldots \mathrm{o}^{\mathrm{n}}\, \theta)} \qquad (4.8) \end{aligned}$$

La formule de réestimation du modèle d'observation est :

$$\begin{aligned} & \mathrm{P}([\mathrm{O}^{\mathrm{t}} = \mathrm{k}] \mid [\mathrm{T}^{\mathrm{t}} = \mathrm{j}]\, \bar{\theta}) \\ & \quad = \frac{(\text{observations k en j})}{(\text{transitions j } \rightarrow \cdot)} \qquad (4.9) \\ & \quad = \frac{\sum_{\mathrm{t}=0}^{\mathrm{n}} \mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid \mathrm{o}^0 \ldots \mathrm{o}^{\mathrm{t}-1}\, [\mathrm{O}^{\mathrm{t}} = \mathrm{k}]\, \mathrm{o}^{\mathrm{t}+1} \ldots \mathrm{o}^{\mathrm{n}}\, \theta)}{\sum_{\mathrm{t}=0}^{\mathrm{n}} \mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid \mathrm{o}^0 \ldots \mathrm{o}^{\mathrm{n}}\, \theta)} \qquad (4.10) \end{aligned}$$

Les éléments de ces formules peuvent être évalués efficacement au moyen du programme 4.6.

Dans le modèle 4.6, $\alpha_{\mathrm{t}}(\mathrm{T})$ et $\beta_{\mathrm{t}}(\mathrm{T})$ peuvent être calculés récursivement. α est la propagation vers l'avant des connaissances sur l'état :

$$\alpha_{\mathrm{t}}(\mathrm{T}^{\mathrm{t}}) \;=\; \mathrm{P}([\mathrm{O}^0 = \mathrm{o}^0] \ldots [\mathrm{O}^{\mathrm{t}} = \mathrm{o}^{\mathrm{t}}]\, \mathrm{T}^{\mathrm{t}} \mid \theta) \qquad (4.11)$$

Nous pouvons calculer α récursivement :

$$\begin{cases} \alpha_0(\mathrm{T}^0) \;=\; \mathrm{P}(\mathrm{O}^0\, \mathrm{T}^0 \mid \theta) \\ \alpha_{\mathrm{t}}(\mathrm{T}^{\mathrm{t}}) \;=\; \sum_{\mathrm{T}^{\mathrm{t}-1}} \left(\alpha_{\mathrm{t}-1}(\mathrm{T}^{\mathrm{t}-1})\, \mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1}\, \theta)\, \mathrm{P}(\mathrm{o}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1}\, \theta)\right) \end{cases} \qquad (4.12)$$

β est la propagation vers l'arrière des connaissances sur l'état :

$$\beta_{\mathrm{t}}(\mathrm{T}^{\mathrm{t}}) \;=\; \mathrm{P}([\mathrm{O}^{\mathrm{t}+1} = \mathrm{o}^{\mathrm{t}+1}] \ldots [\mathrm{O}^{\mathrm{n}} = \mathrm{o}^{\mathrm{n}}] \mid \mathrm{T}^{\mathrm{t}}\, \theta) \qquad (4.13)$$

Il peut être calculé récursivement de la manière suivante :

$$\begin{cases} \beta_{\mathrm{n}}(\mathrm{T}^{\mathrm{n}}) \;=\; 1 \\ \beta_{\mathrm{t}-1}(\mathrm{T}^{\mathrm{t}-1}) \;=\; \sum_{\mathrm{T}^{\mathrm{t}}} \left(\mathrm{P}(\mathrm{T}^{\mathrm{t}} \mid \mathrm{T}^{\mathrm{t}-1}\, \theta)\, \mathrm{P}(\mathrm{o}^{\mathrm{t}+1} \mid \mathrm{T}^{\mathrm{t}}\, \theta)\, \beta_{\mathrm{t}}(\mathrm{T}^{\mathrm{t}})\right) \end{cases} \qquad (4.14)$$

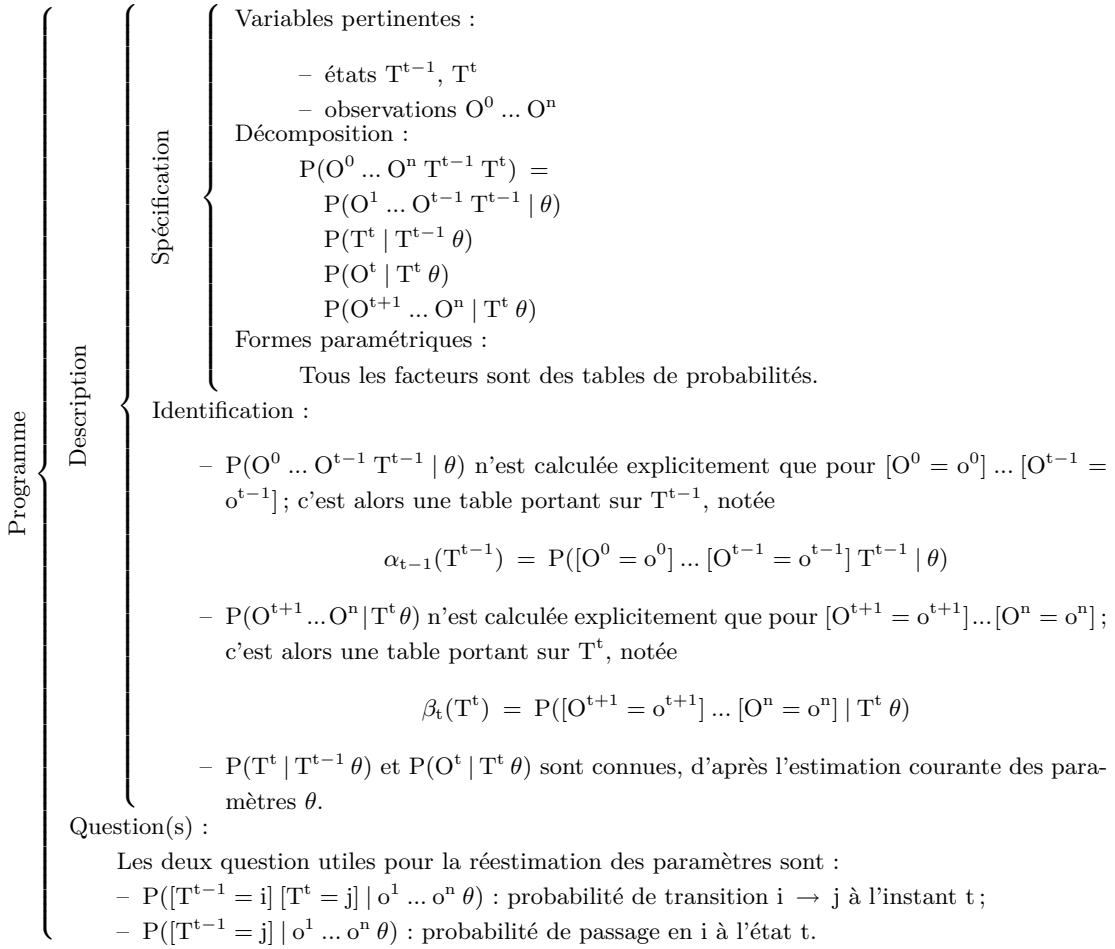

FIG. 4.6 – Modèle pour la réestimation des paramètres dans l'algorithme de Baum-Welch.

Algorithme de Baum-Welch incrémental [Florez-Larrahondo 05] propose une version incrémentale de l'algorithme de Baum-Welch pour l'apprentissage de modèle de Markov cachés discrets. Celle-ci consiste à réestimer les paramètres du modèle après chaque nouvelle observation, au lieu de considérer toute les observations d'un bloc.

Nous notons θ^n les paramètres du modèle à l'issue des observations de 0 à n. Les formules de réestimation consistent à exprimer θ^n en fonction de θ^{n-1} et o^n.

La structure générale de l'algorithme d'apprentissage incrémental est la suivante :

initialiser les paramètres θ^0;
pour $t \leftarrow 1$ **à** n **faire**
  observer $[O^t = o^t]$;
  réestimer θ^t en fonction de θ^{t-1} et o^t;
fin

Algorithme 2 : Algorithme de Baum-Welch incrémental : structure générale

La réestimation incrémentale de la table de transition se fait selon la formule :

$$\begin{aligned} &\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\ \theta^{\mathrm{n}}) \\ &\quad = \frac{1}{\mathrm{Z}} \left(\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\ \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\ [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\ \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])} \right) \end{aligned} \tag{4.15}$$

Dans cette formule, l'indice t correspond au temps d'utilisation du modèle de séquencement alors que n correspond au l'indice temporel de la dernière observation. Ainsi, les tables que nous apprenons portent sur $\mathrm{T}^{\mathrm{t}-1}$, T^{t} et O^{t}, alors que nous maintenons au cours de l'apprentissage des connaissances (observées ou inférées) sur $\mathrm{T}^{\mathrm{n}-1}$, T^{n} et O^{n}.

Z est un facteur de normalisation ne dépendant pas de j. $\widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])$ correspond à l'estimation incrémentale de la probabilité d'être dans l'état i, au vu des observations O^0 à $\mathrm{O}^{\mathrm{n}-1}$:

$$\widetilde{\mathrm{P}}^{\mathrm{n}}([\mathrm{T} = \mathrm{i}]) \ = \ \frac{\sum_{\mathrm{t}=0}^{\mathrm{n}} \mathrm{P}([\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}] \mid \theta^{\mathrm{t}})}{\mathrm{n}+1} \tag{4.16}$$

$$= \ 1 + \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}} = \mathrm{i}] \mid \theta^{\mathrm{n}-1})}{\widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])} \tag{4.17}$$

Ce motif de mise à jour correspond à l'ajout à l'histogramme $\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}])$ d'un échantillon virtuel de poids $\frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1}=\mathrm{i}]\ [\mathrm{T}^{\mathrm{n}}=\mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T}=\mathrm{i}])}$, alors que n échantillons ont déjà été pris en compte.

Cette formule peut être dérivée de la manière suivante, en notant $(\text{transitions i} \ \rightarrow \ \mathrm{j})^{\mathrm{n}}$ le

nombre espéré de transitions depuis l'état i vers l'état j, au vu des données de 0 à n.

$$
\begin{aligned}
&\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, \theta^{\mathrm{T}}) \\
&= \frac{(\text{transitions i } \rightarrow \text{ j})^{\mathrm{n}}}{(\text{transitions i } \rightarrow \cdot)^{\mathrm{n}}} \\
&= \frac{(\text{transitions i } \rightarrow \text{ j})^{\mathrm{n}-1} + \mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\, [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{(\text{transitions i } \rightarrow \cdot)^{\mathrm{n}-1} + \mathrm{P}([\mathrm{T}^{\mathrm{n}} = \mathrm{i}] \mid \theta^{\mathrm{n}-1})} \\
&= \frac{(\text{transitions i } \rightarrow \cdot)^{\mathrm{n}-1}}{(\text{transitions i } \rightarrow \cdot)^{\mathrm{n}-1} + \mathrm{P}([\mathrm{T}^{\mathrm{n}} = \mathrm{i}] \mid \theta^{\mathrm{T}-1})} \\
&\times \left(\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\, [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{(\text{transitions i } \rightarrow \cdot)^{\mathrm{n}-1}} \right) \\
&= \frac{\mathrm{n}\, \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])}{\mathrm{T}\, \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}]) + \mathrm{P}([\mathrm{T}^{\mathrm{n}} = \mathrm{i}] \mid \theta^{\mathrm{n}-1})} \\
&\times \left(\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\, [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\, \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])} \right) \\
&= \frac{1}{\mathrm{Z}} \left(\mathrm{P}([\mathrm{T}^{\mathrm{t}} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]\, \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\, [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\, \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])} \right)
\end{aligned}
$$

La réestimation incrémentale du modèle d'observation se fait selon la formule :

$$
\begin{aligned}
&\mathrm{P}([\mathrm{O}^{\mathrm{t}} = \mathrm{k}] \mid [\mathrm{T}^{\mathrm{t}} = \mathrm{j}]\, \theta^{\mathrm{n}}) \\
&= \frac{1}{\mathrm{Z}} \left(\mathrm{P}([\mathrm{O}^{\mathrm{t}} = \mathrm{k}] \mid [\mathrm{T}^{\mathrm{t}} = \mathrm{j}]\, \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{O}^{\mathrm{n}} = \mathrm{k}]\, [\mathrm{T}^{\mathrm{n}} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\, \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{j}])} \right)
\end{aligned}
\tag{4.18}
$$

Ce motif de mise à jour correspond à l'ajout à l'histogramme $\mathrm{P}([\mathrm{O}^{\mathrm{t}} = \mathrm{k}] \mid [\mathrm{T}^{\mathrm{t}} = \mathrm{j}]\, \theta^{\mathrm{n}})$ d'un échantillon virtuel de poids $\frac{\mathrm{P}([\mathrm{O}^{\mathrm{n}}=\mathrm{k}]\, [\mathrm{T}^{\mathrm{n}}=\mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T}=\mathrm{i}])}$, alors que n échantillons ont déjà été pris en compte. Le facteur de normalisation Z est identique à celui de la formule de réestimation de la table de transition, ci-dessus. La dérivation de cette formule peut être faite de manière similaire à celle de mise à jour de la table de transition.

Apprentissage du modèle de séquencement

Que savons-nous, que cherchons-nous ?

Rappel du programme de séquencement Nous pouvons reprendre notre modèle de Markov caché, et l'instancier pour le problème particulier de notre personnage *Unreal Tournament* (figure 4.7). Nous retrouvons le programme de séquencement dont nous avons déjà décrit la programmation à la main.

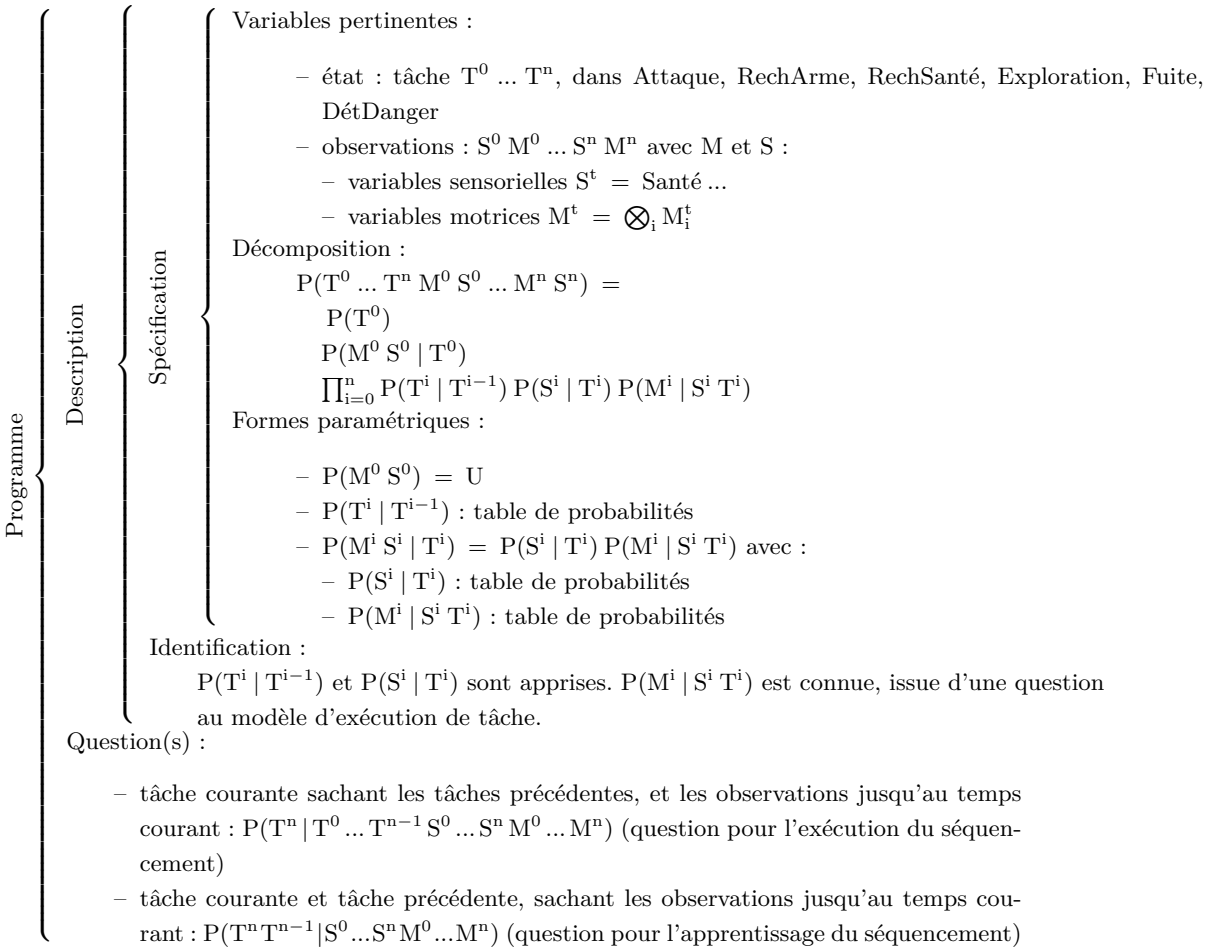

FIG. 4.7 – modèle de séquencement comme un HMM

Paramètres à apprendre

– **Le modèle de tâche est connu.** Il s'agit du modèle qui permet d'exécuter une tâche donnée. Il répond à la question $P(M^i \mid S_0^i\, T^i)$, qui est utilisée dans notre modèle complet d'apprentissage. Il peut être décrit comme un ensemble de sous-modèles $P(M\, S_0 \mid t)$, pour chaque tâche t.

– **Le modèle de séquencement est cherché.** Il s'agit d'apprendre les tables $P(T^i \mid T^{i-1})$, $P(S_0^i \mid T^i)$ et $P(S_1^i \mid T^i)$. Nous considérons que ces tables sont indépendantes du temps, et nous pouvons aussi les écrire en omettant les indices temporels :

$$P(T \mid T^{-1}),\ P(S_0 \mid T) \text{ et } P(S_1 \mid T)$$

– **Les tâches sont inconnues.** Elles ne peuvent pas être observées. Puisque nos variables de tâche $T^0 \dots T^n$ apparaissent dans les tables de séquencement que nous voulons apprendre,

nous devrons cependant obtenir de l'information sur ces tâches. Le modèle de tâche contribuera à nous fournir cette information, puisqu'il est connu, et relie les variables observées (M, S_0) à la tâche.

Algorithme final d'apprentissage

Structure générale L'algorithme d'apprentissage a la structure suivante :

$P(\mathrm{T^t}|\mathrm{T^{t-1}}) \leftarrow U$;
pour *chaque* i **faire**
  $P(S_i^t|T^t) \leftarrow U$;
fin
$P(T^0) \leftarrow U$;
pour $t \leftarrow 1$ **à** n **faire**
  observer les variables motrices M et S;
  inférer $Q = P(\mathrm{T^t T^{t-1}}|MS)$;
  $P(\mathrm{T^t}|\mathrm{T^{t-1}}) \leftarrow P(\mathrm{T^t}|\mathrm{T^{t-1}}) + \Delta^n P([\mathrm{T^t}|\mathrm{T^{t-1}})$;
  pour *chaque* i **faire**
    $P(S_i^t|T^t) \leftarrow P(S_i^t|T^t) + \Delta^n P(S_i|T^t)$;
  fin
fin

Algorithme 3 : Algorithme de Baum-Welch incrémental : structure générale

Le cœur de cet algorithme est le calcul des incréments $\Delta^{\mathrm{n}}\,\mathrm{P}([\mathrm{T^t} \mid \mathrm{T^{t-1}})$ et $\Delta^{\mathrm{n}}\,\mathrm{P}(\mathrm{S_i} \mid \mathrm{T^t})$ en fonction de la question Q. Nous le détaillons dans la section suivante.

Réestimation incrémentale des paramètres du modèle de séquencement Les facteurs à réestimer sont :

- $\mathrm{P}(\mathrm{T^t} \mid \mathrm{T^{t-1}})$;
- $\mathrm{P}(\mathrm{S_i^t} \mid \mathrm{T^t})$.

L'algorithme de Baum-Welch incrémental nous donne directement la formule de réestimation de la table de transition de tâche à tâche :

$$\forall\, \mathrm{i,\ j,}$$
$$\mathrm{P}([\mathrm{T^t = j}] \mid [\mathrm{T^{t-1} = i}]\,\theta^{\mathrm{n}}) \tag{4.19}$$
$$= \frac{1}{\mathrm{Z}} \left(\mathrm{P}([\mathrm{T^t = j}] \mid [\mathrm{T^{t-1} = i}]\,\theta^{\mathrm{n-1}}) + \frac{\mathrm{P}([\mathrm{T^{n-1} = i}]\,[\mathrm{T^n = j}] \mid \theta^{\mathrm{n-1}})}{\mathrm{P}([\mathrm{T^{n-1} = i}] \mid \theta^{\mathrm{n-1}})}\, \mathrm{n}\, \widetilde{\mathrm{P}}^{\mathrm{n-1}}([\mathrm{T = i}]) \right) \tag{4.20}$$

La réestimation des tables liant les variables sensorielles à la tâche est faite selon la formule de réestimation du modèle d'observation :

$$\forall\, \mathrm{i},\ \mathrm{j},\ \mathrm{k},$$

$$\mathrm{P}([\mathrm{S_i} = \mathrm{k}] \mid [\mathrm{T^t} = \mathrm{j}]\ \theta^{\mathrm{n}}) \tag{4.21}$$

$$= \frac{1}{\mathrm{Z}} \left(\mathrm{P}([\mathrm{S_j} = \mathrm{k}] \mid [\mathrm{T^t} = \mathrm{j}]\ \theta^{\mathrm{n}-1}) + \frac{\mathrm{P}([\mathrm{S_i} = \mathrm{k}]\ [\mathrm{T^n} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\ \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{j}])} \right) \tag{4.22}$$

Pour représenter ces distributions en pratique, nous utilisons des matrices non normalisées comptabilisant des échantillons (non entiers). Nous pouvons maintenir de manière incrémentale les facteurs de normalisation pour chacune d'entre elles, et ainsi en dériver les distributions de probabilité associées quand cela est nécessaire.

Initialement, nous n'avons pas d'information particulière sur les tables de probabilités, et nous voulons donc qu'elles soient uniformes. Cela est réalisé simplement en insérant un échantillon virtuel de poids 1 dans chaque case des trois matrices.

Cette façon de procéder correspond exactement à traiter nos tables de probabilités comme des lois de Laplace. Nos tables partent d'un état initial (*a priori*) uniforme, et au fur et à mesure que des échantillons sont ajoutés, convergent vers les histogrammes correspondants.

Ces échantillons virtuels sont :

$$\Delta^{\mathrm{n}}\, \mathrm{P}([\mathrm{T^t} = \mathrm{j}] \mid [\mathrm{T}^{\mathrm{t}-1} = \mathrm{i}]) = \frac{\mathrm{P}([\mathrm{T}^{\mathrm{n}-1} = \mathrm{i}]\ [\mathrm{T^n} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\ \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{i}])} \tag{4.23}$$

$$\Delta^{\mathrm{n}}\, \mathrm{P}([\mathrm{S_i} = \mathrm{k}] \mid [\mathrm{T^t} = \mathrm{j}]) = \frac{\mathrm{P}([\mathrm{S_i} = \mathrm{k}]\ [\mathrm{T^n} = \mathrm{j}] \mid \theta^{\mathrm{n}-1})}{\mathrm{n}\ \widetilde{\mathrm{P}}^{\mathrm{n}-1}([\mathrm{T} = \mathrm{j}])} \tag{4.24}$$

Résultats

L'algorithme de Baum-Welch incrémental, dans notre implémentation non optimisée et sur un Pentium IV 1.7 Ghz, traite environ dix mesures sensori-motrices par seconde.

Nous avons réalisé une démonstration de jeu d'une durée de 350 secondes, soit 3500 mesures sensori-motrices. Nous avons appliqué l'algorithme d'apprentissage à cette démonstration.

Nous discutons dans un premier temps les tables issues de cet apprentissage. Nous présentons ensuite les résultats au combat d'un personnage autonome utilisant ces tables.

Tables de probabilités apprises La table 4.15 est la table de transitions de tâche à tâche : $\mathrm{P}(\mathrm{T^t} \mid \mathrm{T}^{\mathrm{t}-1})$. Elle correspond à la diagonale (cases bleues) décrite pour les tables programmées à la main (table 4.16). Les tâches – et en particulier celle de détection du danger – apparaissent cependant moins stables que dans le programme manuel. Il est normal que la tâche de détection du danger soit moins stable, dans la mesure où elle s'exécute généralement pendant moins de temps que les autres : elle consiste principalement à se retourner.

$T^t \setminus T^{t-1}$	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	0.78	0.0024	0.023	0.028	0.052	0.048
RechArme	0.014	0.96	0.012	0.016	0.017	0.017
RechSanté	0.16	0.02	0.94	0.041	0.06	0.16
Exploration	0.017	0.0049	0.0078	0.86	0.038	0.048
Fuite	0.016	0.0047	0.0090	0.018	0.78	0.068
DétDanger	0.017	0.0059	0.0068	0.038	0.048	0.66

TAB. 4.15 – $P(T^t \mid T^{t-1})$

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Attaque	x	0.01	0.01	0.01	0.01	0.01
RechArme	0.01	x	0.01	0.01	0.01	0.01
RechSanté	0.01	0.01	x	0.01	0.01	0.01
Exploration	0.01	0.01	0.01	x	0.01	0.01
Fuite	0.01	0.01	0.01	0.01	x	0.01
DétDanger	0.01	0.01	0.01	0.01	0.01	x

TAB. 4.16 – $P(T^t|T^{t-1})$: version prudente écrite à la main

La table 4.17 décrit $P(\text{ArmeE} \mid T^t)$: la distribution de l'arme de l'adversaire le plus fort sachant la tâche courante. Nous utilisons une variable d'arme de l'ennemi modifiée par rapport à nos expériences précédentes : celle-ci comporte une valeur spéciale pour indiquer qu'aucun ennemi n'est présent (dans les tables précédentes, ce cas donnait ArmeE = Aucune).

Nous observons la similarité suivante entre la table apprise et la table spécifiée à la main (table 4.18) :

– quand le personnage attaque, l'arme de l'adversaire est plutôt faible (colonne Attaque dans les deux tables).

La table écrite à la main spécifie toutes les colonnes autre que celle de la tâche Attaque comme uniformes (cases grises) : en tant que concepteurs du comportement, nous avons exprimé une absence de connaissance sur les distributions concernées. La table apprise contient de l'information dans ces colonnes :

– quand le personnage recherche une arme, il est très probable qu'il n'ait pas d'ennemi ;
– quand le personnage recherche de la santé, il a souvent un ennemi dont l'arme est faible ou bonne ;
– quand le personnage explore, deux cas sont communs : n'avoir aucun ennemi, et avoir un ennemi avec une arme inexistante ou faible ;
– quand le personnage fuit, il a forcément un ennemi, et la probabilité que son arme soit forte est relativement importante (probabilité 0.12) ;
– quand le personnage détecte le danger, il y a une probabilité importante que l'ennemi n'ait pas d'arme, ou seulement une arme faible.

ArmeE \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	0.081	0.00096	0.079	0.22	0.12	0.23
Faible	0.56	0.026	0.54	0.36	0.46	0.56
Bonne	0.3	0.0021	0.35	0.11	0.29	0.1
Forte	0.001	0.0003	0.022	0.0027	0.12	0.0047
PasDEnnemi	0.053	0.97	0.011	0.3	0.00013	0.1

TAB. 4.17 – P(ArmeE | T^t)

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	x	x	x	x	x	x
Moyenne	x	x	x	x	x	x
Forte	0.01	x	x	x	x	x

TAB. 4.18 – P(ArmeE|T^t) : version prudente écrite à la main

La table 4.19 décrit P(Arme | T^t) telle qu'elle a été apprise ; la table 4.20 correspond à cette même distribution, programmée à la main.

Entre la table apprise et la table programmée à la main, nous observons la similarité suivante :

– quand le personnage attaque, la probabilité est très faible qu'il n'ait pas d'arme (case verte).

Nous observons les disparités suivantes :

– quand le personnage recherche une arme ou un bonus de santé, ou fuit, il a probablement une arme, dont la force est à peu près uniformément répartie : cela contredit les hypothèses que nous avions faites pour les tâches RechArme et Fuite, qui spécifiaient qu'il serait plus probable que le personnage ait une arme faible ;

– quand le personnage explore ou détecte le danger, il a de plutôt fortes chances d'avoir une arme forte (case bleue) ;

Arme \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	0.00025	0.012	0.00049	0.0014	0.011	0.00095
Faible	0.49	0.47	0.35	0.32	0.37	0.29
Bonne	0.37	0.31	0.37	0.2	0.33	0.17
Forte	0.13	0.2	0.28	0.48	0.28	0.53

TAB. 4.19 – P(Arme | T^t)

La table 4.21 décrit P(Bruit | T^t) telle qu'elle a été apprise ; sa version programmée à la main est donnée dans la table 4.22.

Deux colonnes sont communes aux tables :

– quand le personnage recherche une arme, la probabilité est plutôt faible qu'il entende du bruit (case bleue) ; cela coïncide avec le fait que pendant la recherche d'arme il n'est généralement en présence d'aucun ennemi ;

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Aucune	10^{-5}	x	x	x	x	x
Moyenne	x	0.1	x	x	0.1	x
Forte	x	10^{-5}	x	x	0.01	x

TAB. 4.20 – $P(\text{Arme}|\text{T}^t)$: version prudente écrite à la main

– quand il détecte le danger, la probabilité est forte qu'il entende du bruit (case verte).

Deux colonnes sont notablement différentes (tâches rouges) :

– pour le comportement appris, quand le personnage recherche un bonus de santé ou explore, la probabilité est plutôt forte qu'il entende du bruit – la table écrite à la main faisait l'hypothèse inverse.

Enfin, la table apprise contient des informations que nous n'avions pas spécifiées à la main (cases grises) :

– quand le personnage attaque ou fuit, il est probable qu'il entende du bruit – cela semble normal, dans la mesure où le bruit est associé aux situations de combat.

Bruit \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
faux	0.17	0.68	0.26	0.36	0.079	0.19
vrai	0.83	0.32	0.74	0.64	0.92	0.81

TAB. 4.21 – $\text{P}(\text{Bruit} \mid \text{T}^t)$

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
faux	x	x	x	x	x	10^{-5}
vrai	x	0.1	0.1	10^{-5}	x	x

TAB. 4.22 – $P(\text{Bruit}|\text{T}^t)$: version prudente écrite à la main

La table 4.23 décrit $\text{P}(\text{NbE} \mid \text{T}^t)$ dans sa version apprise ; la distribution spécifiée à la main se trouve sur la table 4.24.

Nous observons les similarités suivantes entre ces tables :

– quand le personnage attaque, il a au moins un ennemi, et la probabilité est faible qu'il ait de nombreux ennemis (cases bleues) ;

– quand le personnage recherche une arme ou explore, la probabilité est forte qu'il n'ait pas d'ennemi (cases verte).

Nous observons les différences suivantes (tâches rouges) :

– quand le personnage fuit, la probabilité qu'il ait un seul ennemi reste importante (probabilité 0.52), même si celle qu'il ait 2 ennemis ou plus est forte également (probabilité $0.34 + 0.14 = 0.48$) ;

– quand le personnage détecte le danger, la probabilité est forte qu'il ait au moins un ennemi, contrairement à ce que nous avons spécifié à la main.

La table apprise spécifie la colonne concernant la recherche de santé :

– quand le personnage recherche un bonus de santé, il est très probable qu'il ait un ou deux ennemis (probabilité $0.51 + 0.38 = 0.89$).

NbE \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0.053	0.97	0.011	0.7	0.0013	0.1
1	0.54	0.018	0.51	0.14	0.52	0.62
2	0.34	0.0047	0.38	0.11	0.34	0.24
3 ou plus	0.07	0.007	0.098	0.049	0.14	0.037

TAB. 4.23 – P(NbE | T^t)

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0	x	x	x	0	x
1	x	0.1	x	0.001	0.1	0.001
2 ou plus	0.1	0.01	x	0.0001	x	0.0001

TAB. 4.24 – P(NbE|T^t) : version prudente écrite à la main

La table 4.25 décrit P(ArmeProche | T^t) telle qu'elle a été apprise ; la table 4.26 décrit cette distribution telle que nous l'avons spécifiée à la main. Nous avons ajouté une valeur ∞ à la variable ArmeProche, dénotant qu'aucune arme n'est visible, ou que celle visible est trop loin pour être pertinente pour agir. De plus, nous avons séparé l'observation ArmeProche = vrai en 5 valeurs, selon que l'arme est plus ou moins proche.

Les trois colonnes que nous avions spécifiées se retrouvent dans la table apprise :

– quand le personnage recherche de la santé, explore ou détecte le danger, la probabilité est forte à très forte qu'il n'ait pas d'arme à proximité (cases bleues).

Pour les colonnes spécifiées comme uniformes (cases grises), la distribution apprise est similaire : la probabilité qu'aucune arme ne soit présente est très grande. La variable ArmeProche est donc quasiment indépendante de la tâche effectuée. Cependant, les tâches d'exploration et de fuite conduisent à plus de proximité avec des armes.

ArmeProche \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0.00025	0.0003	0.00028	0.00027	0.00027	0.00027
1	0.00027	0.0082	0.0054	0.017	0.011	0.0072
2	0.00026	0.021	0.018	0.11	0.052	0.016
3	0.003	0.019	0.025	0.12	0.072	0.024
4	0.00027	0.036	0.033	0.13	0.11	0.034
∞	1	0.91	0.92	0.62	0.75	0.92

TAB. 4.25 – P(ArmeProche | T^t)

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	x	10^{-5}	10^{-5}	x	10^{-5}

TAB. 4.26 – $P(\text{ArmeProche}|\text{T}^\text{t})$: version prudente écrite à la main

La table 4.27 décrit P(SantéProche | T^t) telle qu'elle a été apprise ; la table 4.28 correspond à cette même distribution, spécifiée à la main. Nous avons ajusté la discrétisation de SantéProche : celle-ci est décomposée en cinq valeurs réparties selon une échelle logarithmique, et une sixième qui indique que le bonus est trop loin pour être pertinent dans le comportement courant, ou qu'aucun n'est visible.

Comme pour la table P(ArmeProche | T^t), la distribution sur la SantéProche est similaire pour toutes les tâches, et ressemble à ce que nous avons spécifié pour les tâches RechArme, Exploration et DétDanger : il est très probable qu'aucun bonus de santé ne soit proche (cases bleues). La tâche de fuite est cependant plus nuancée : la probabilité qu'un bonus de santé soit proche reste importante ($1 - 0.53 = 0.47$). Cela peut indiquer que le joueur humain a tendance à fuire dans la direction de bonus de santé.

SantéProche \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0.00025	0.0003	0.00028	0.0013	0.01	0.00027
1	0.00025	0.008	0.0013	0.0065	0.11	0.0075
2	0.00025	0.045	0.012	0.018	0.11	0.018
3	0.0069	0.047	0.011	0.028	0.12	0.038
4	0.036	0.088	0.034	0.091	0.13	0.12
∞	0.96	0.81	0.94	0.86	0.53	0.82

TAB. 4.27 – P(SantéProche | T^t)

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Faux	x	x	x	x	x	x
Vrai	x	10^{-5}	x	10^{-5}	x	10^{-5}

TAB. 4.28 – $P(\text{SantéProche}|\text{T}^\text{t})$: version prudente écrite à la main

La table 4.29 correspond à P(Santé | T^t) telle qu'elle a été apprise ; la version spécifiée à la main de cette distribution est rappelée sur la table 4.30.

Le niveau de santé le plus haut n'a pas été atteint pendant l'apprentissage ; sa probabilité d'apparence est donc résiduelle pour chacune des tâches.

Entre les deux tables, les similarités suivantes peuvent être notées :

- quand le personnage recherche une arme, il est probable qu'il ait un niveau de santé important (cases bleues) ;
- quand le personnage fuit, il est plutôt probable que sa santé soit faible (cases vertes).

Les différences suivantes sont observées (tâches rouges) :

– dans les tâches Attaque, RechSanté, Exploration et DétDanger, la distribution sur la santé est proche de l'uniforme sur les quatre premiers niveaux de santé, avec une préférence pour le niveau le plus haut ; cela contredit en particulier notre spécification pour la tâche de recherche de santé, pour laquelle nous pensions que le personnage aurait un niveau de santé plutôt bas ; cela est cependant explicable en considérant que le comportement de recherche de santé d'un joueur humain est souvent plus opportuniste que planifié, et que celui-ci ramasse principalement les bonus de santé près desquels il passe.

Santé \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
très faible	0.16	0.098	0.12	0.23	0.23	0.23
faible	0.19	0.17	0.23	0.2	0.3	0.19
moyen	0.29	0.14	0.18	0.16	0.37	0.17
haut	0.35	0.6	0.47	0.42	0.1	0.41
très haut	0.00025	0.0003	0.00028	0.00029	0.00027	0.00027

TAB. 4.29 – P(Santé | T^t)

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
Bas	0.001	0.1	x	0.1	0.7	0.1
Moyen	0.1	x	0.01	x	0.2	x
Haut	x	x	0.001	x	0.1	x

TAB. 4.30 – P(Santé|T^t) : version prudente écrite à la main

Résultats au combat du comportement appris La table 4.31 décrit les résultats au combat d'un personnage autonome utilisant les tables apprises à l'aide de l'algorithme de Baum-Welch incrémental. En termes d'efficacité au combat, ce personnage se situe au même niveau que le personnage natif Unreal Tournament de référence, et que les autres personnages agressifs.

Ce résultat montre qu'il est possible de construire un comportement compétitif entièrement basé sur la programmation bayésienne. De plus ce comportement peut être appris sur la base de quelques minutes de démonstration. Il faut cependant éviter d'accorder trop de poids aux scores relatifs des différentes méthodes dans cette table. En effet, le comportement appris avec l'algorithme de Baum-Welch incrémental repose sur des tâches élémentaires différentes des autres comportements. Une variation dans l'efficacité de la tâche d'attaque peut ainsi à elle seule expliquer une partie des différences de performance.

apprentissage bayésien, démonstration agressive	8.5
apprentissage par reconnaissance automatique de la tâche, agressive	4.4
apprentissage par reconnaissance automatique de la tâche, prudente	13.9
apprentissage par sélection de la tâche, agressive	45.7
spécification manuelle, agressive	8.0
spécification manuelle, prudente	12.2
spécification manuelle, uniforme	43.2
personnage natif UT (niveau 3/8)	11.0

TAB. 4.31 – Comparaison des performances de comportements appris par l'algorithme de Baum-Welch incrémental, par reconnaissance automatique de la tâche, par sélection de la tâche, spécifiés manuellement, et d'un personnage natif d'*Unreal Tournament* (score entre 0 et 100, un score faible correspond à un personnage performant au combat).

Commentaires sur l'apprentissage bayésien

L'algorithme de Baum-Welch incrémental permet de réaliser l'apprentissage automatique de comportements basés sur un séquencement. Par contraste avec la méthode d'apprentissage par reconnaissance de la tâche, nous n'avons pas besoin de programmer une heuristique de reconnaissance en plus des modèles de tâches. En réalité, dans le cadre de l'apprentissage bayésien, les modèles de tâches élémentaires jouent le double rôle de modèle pour l'action et de modèle pour la reconnaissance.

De plus, cette reconnaissance, par sa nature bayésienne, fournit une information riche, prenant en compte l'incertitude de reconnaissance. Cette incertitude n'est pas réduite, mais propagée d'un pas de temps à l'autre,

Dans la perspective de l'intégration de l'apprentissage et de la programmation bayésienne dans la construction de comportements, ces deux aspects sont de grands avantages.

Cependant, les modèles des tâches simples doivent impérativement être construits à la lumière de leur double rôle : ils doivent être bons pour agir, et pour reconnaître. En particulier, les tâches doivent générer des motifs sensori-moteurs les plus distincts possibles. Par exemple, si en l'absence d'ennemis la tâche de fuite produit les mêmes commandes motrices que la tâche d'exploration, ces deux tâches seront créditées de la même probabilité de reconnaissance ; cela va conduire à un programme appris tout aussi efficace, mais difficile à analyser, puisqu'il produira une tâche de fuite en l'absence même d'ennemis. Une solution – celle que nous avons adoptée – est de laisser la tâche de fuite produire des commande motrices uniformément aléatoire en l'absence d'ennemis.

Une difficulté du séquencement par programmation inverse apparaît clairement dans les tables apprises. Considérons l'apprentissage d'une table $P(S|T^t)$, et la distribution obtenue $P(S|[T^t = i])$ pour i donné. Si la tâche i n'influe pas sur la distribution du capteur S, nous avons :

$$P(S \mid [T^t = i]) \;=\; P(S) \tag{4.25}$$

et non pas :

$$P(S \mid [T^t = i]) \;=\; U \tag{4.26}$$

Le cas 4.26 correspond à ce que nous avons posé dans notre modélisation à la main. Cela rend plus délicate la comparaison entre tables apprises et tables spécifiées à la main. De plus, cela pose question quant à l'interprétation des tables. Une façon de s'affranchir de cet obstacle serait d'utiliser à la place du facteur $P(S \mid T^t)$ un facteur faisant apparaître T^t et S en partie droite, au moyen d'une variable de cohérence M_S :

$$P(M_S \mid T^t\, S)$$

4.4.4 L'apport de l'apprentissage : comparaison avec les comportements d'*Unreal Tournament*

Nous pouvons reprendre les questions de la problématique de construction de comportements pour éclairer les apports de l'apprentissage automatique.

– **(Q1a : structure des paramètres)**, **(Q1c : apprentissage)**

Le succès des méthodes d'apprentissage que nous proposons fait la preuve de la structuration des paramètres de nos comportements. Nous pouvons apprendre un comportement à l'aide d'un algorithme générique qui s'appuie directement sur la description des tâches élémentaires. Au prix d'un effort de modélisation bayésienne, nous gagnons la possibilité d'enseigner comment se comporter à un personnage. Cela peut être exploité au cours du développement, ou comme un élément du jeu.

– **(Q2a : complexité en mémoire)**, **(Q2b : complexité en temps de calcul)**

L'algorithme de Baum-Welch incrémental fonctionne en temps constant par rapport aux nombre d'échantillons d'apprentissage (c'est-à-dire par rapport à la durée de la démonstration). Notre implémentation naïve permet de l'exécuter au même rythme que les données sont observées.

Chapitre 5

Bilan : la programmation bayésienne pour les personnages de jeux vidéos

5.1 Problématique abordée : programmation et apprentissage de comportements pour personnages de jeux vidéos

Nous avons essentiellement traité deux problèmes :

- Comment programmer le comportement d'un personnage, de manière simple et puissante ?
- Comment donner les moyens au joueur d'enseigner des comportements à ses personnages ?

5.1.1 La programmation de comportements

La problématique de programmation est celle de la construction d'un comportement crédible, donc de la description et de la gestion d'un modèle complexe. Les effets attendus pour le joueur sont :

- des comportements peu répétitifs, difficilement prédictibles, flexibles ;
- des adversaires dont l'habileté est adaptée, ni trop forts ni trop faibles.

Pour le développeur, ces effets peuvent être atteints si les critères suivants sont satisfaits :

- la simplicité de description du comportement ;
- la possibilité de construire des comportements puissants ;
- l'efficacité du comportement produit en terme de ressources mémoire et processeur ;
- la possibilité de comprendre et modifier un comportement existant pour l'ajuster.

5.1.2 L'appprentissage de comportements

Les techniques d'apprentissage représentent en premier lieu un point d'attraction fort pour le joueur. L'apprentissage par démonstration consiste à apprendre un comportement d'après l'observation de celui du joueur. Il permet à celui-ci d'entraîner des personnages autonomes, amis ou ennemis, pour l'assister ou le remplacer. Il devient un élément du jeu, en proposant au joueur de devenir vraiment maître de ses créatures, et d'échanger les comportements qu'il a enseigné avec d'autres joueurs.

Pour le développeur, l'apprentissage par démonstration est un outil de programmation. Il permet d'ajuster les paramètres des comportements de manière naturelle et intuitive.

5.1.3 Approche pratique

Nous avons ramené nos deux problèmes à une série de questions pratiques, nous permettant d'évaluer plus précisément notre méthode de construction de comportements.

- **(1)** *En quoi consiste la description d'un comportement ?*
  - **(Q1a : structure des paramètres)** *Les paramètres d'un comportement sont-ils localisés, identifiés et structurés ?*
  - **(Q1b : paramètres et intuition)** *Les paramètres d'un comportement sont-ils accessibles à l'intuition ?*
  - **(Q1c : apprentissage)** *Est-il possible d'apprendre les paramètres du comportement par démonstration ?*
- **(2)** *Peut-on exécuter un comportement complexe dans des limites fixées de ressources matérielles ?*
  - **(Q2a : complexité en mémoire)** *Un comportement complexe peut-il être exécuté dans les limites typiques de mémoire disponible ?*
  - **(Q2b : complexité en temps de calcul)** *Un comportement complexe peut-il être exécuté dans les limites typiques de temps processeur disponible ?*

Nous avons assis la portée de ces questions pratiques sur l'étude de la construction de comportements dans un jeu commercial : *Unreal Tournament*. Les comportements dans *Unreal Tournament* sont basés sur un langage de programmation spécialisé pour la description de machines d'états finis. Cette méthode a les caractéristiques essentielles suivantes :

- les paramètres des comportements y sont dilués et mal structurés, mais compréhensibles ;
- il n'est pas possible d'apprendre les comportements ;
- ceux-ci sont économes en temps de calcul et en mémoire.

Notre point de référence a donc été dans la suite cette méthode classique basée sur les machines d'états finis, qui répond à nos deux problèmes de la façon suivante :

- il est simple de programmer des comportements qui s'exécutent rapidement, mais parfois difficile de les étendre ;
- il n'est pas possible d'apprendre ces comportements.

5.2 Comportements avec la programmation bayésienne

Nous proposons trois techniques basées sur la programmation bayésienne des robots, qui permettent de construire et apprendre des comportements complexes de manière simple :

- la fusion par cohérence améliorée ;
- le séquencement par programmation inverse ;
- l'apprentissage de ce séquencement avec l'algorithme de Baum-Welch incrémental.

5.2.1 La fusion par cohérence améliorée

La fusion par cohérence (figure 5.1) a été introduite sous le nom de fusion avec diagnostic par [Pradalier 03a]. Sa motivation originale était d'exprimer dans le cadre de la programmation bayésienne une fusion de consignes comme un simple produit de distributions. Elle repose sur l'introduction d'une variable artificielle booléenne qui exprime la cohérence de la consigne et de la commande. En s'assurant que cette variable est positive lors de la question, il est possible d'obtenir du modèle une commande qui est compatible avec la consigne. De plus, si une commande et une consigne sont observées, il est possible de déterminer la variable de cohérence, qui fait alors office de diagnostic pour indiquer si la commande est cohérente avec la consigne.

Sur cette base, nous avons utilisé la fusion par cohérence non comme un artifice de calcul, mais comme un outil de modélisation pour le contrôle. Pour ce faire, nous y avons apporté deux améliorations :

- nous avons pris en compte des consignes exprimées comme des distributions, et non plus simplement comme des valeurs connues ;
- nous avons montré comment utiliser la variable de cohérence en positif et en négatif, de façon à pouvoir prendre en compte chaque consigne comme prescriptive ou proscriptive.

Cela nous permet d'avancer la fusion par cohérence améliorée comme un mécanisme modulaire de fusion dans le cadre de la programmation bayésienne des robots :

- il est possible de fusionner un nombre arbitraire de consignes ;
- chaque consigne est une simple distribution sur la variable de fusion, qui peut être le résultat d'une question à un sous-modèle.

Avec ces améliorations, la fusion par cohérence se présente comme une brique de programmation plus intuitive et plus versatile que les modes de fusion souvent utilisés en programmation bayésienne (fusions sensorielle et motrice). Nous avons montré comment l'utiliser pour construire des comportements élémentaires pour un personnage autonome.

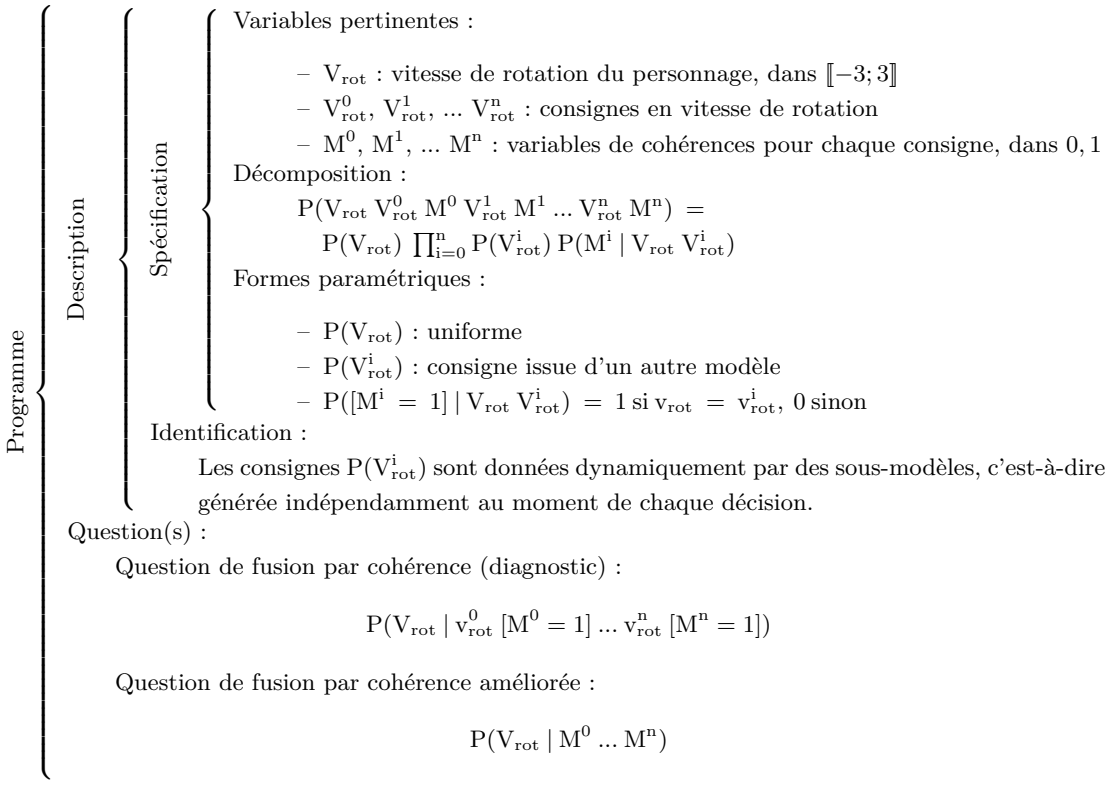

FIG. 5.1 – Fusion par cohérence : exemple de fusion de consignes en vitesse de rotation

5.2.2 Le séquencement de tâches par programmation inverse

La technique de fusion bayésienne naïve consiste à fusionner les informations d'un ensemble de variables en faisant l'hypothèse qu'elles sont toutes indépendantes sachant la variable de fusion.

Un modèle de Markov caché ([Rabiner 89]) met en jeu une variable d'état à deux instants successifs, et une variable d'observation. Il est le plus souvent utilisé pour estimer l'état (ou une succession d'états).

Nous avons introduit un modèle de séquencement de tâches qui s'appuie sur la technique du modèle de Markov caché. La variable d'état correspond à une tâche, c'est-à-dire à un comportement élémentaire connu du personnage autonome. La fusion des informations sensorielles est faite selon une fusion bayésienne naïve. Nous nommoms le modèle résultant "séquencement par programmation inverse".

Nous avons démontré que cette façon de séquencer des tâches :

– repose sur un formalisme simple et solide, qui conduit à des modèles se prêtant bien aux manipulations automatiques ;
– permet la description d'un modèle simple, qui peut être rapproché d'une machine d'états finis ;

– se prête – mieux qu'une machine d'états finis – à l'extension par ajout de tâches ou de variables sensorielles ;
– est paramétré par un ensemble de tables numériques munies d'une sémantique claire, qui peuvent être programmée à la main.

5.2.3 L'aprentissage avec l'algorithme de Baum-Welch incrémental

L'algorithme de Baum-Welch [Baum 72] permet d'apprendre un modèle de Markov caché. Le séquencement par programmation inverse s'appuie sur un tel modèle, et peut donc être appris. La variante incrémentale [Florez-Larrahondo 05] de l'algorithme de Baum-Welch autorise l'apprentissage en ligne du séquencement, c'est-à-dire au fur et à mesure que le joueur fait sa démonstration.

Nous avons montré comment adapter l'algorithme de Baum-Welch incrémental à un modèle de séquencement par programmation inverse. Alors que cet algorithme permet d'apprendre d'une part le modèle de transition, et d'autre part le modèle d'observation, le séquencement par programmation inverse divise le modèle d'observation : une partie en est connue et fixée (le modèle moteur), et l'autre est apprise incrémentalement (le modèle de choix de tâche).

Il devient possible avec cette technique d'apprendre un comportement au fur et à mesure de sa démonstration. Le comportement produit peut être inspecté, modifié, et comparé avec les comportements décrits à la main. Il est possible simplement de sauver, charger et échanger les paramètres d'un comportement appris, puisque ceux-ci sont un ensemble de tables numériques.

5.3 Perspectives

5.3.1 Perspectives techniques

Programmation inverse et fusion par cohérence

Les tables 5.1 et 5.2 rappellent la distribution du nombre d'ennemis sachant la tâche, telle qu'elle a été apprise avec l'algorithme de Baum-Welch incrémental, et telle que nous l'avions auparavant spécifiée à la main. Dans chacune de ces tables, la colonne grise correspond à $P(\text{NbE} \mid [\text{T}^\text{t} = \text{RechSanté}])$.

NbE \ T^t	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0.053	0.97	0.011	0.7	0.0013	0.1
1	0.54	0.018	0.51	0.14	0.52	0.62
2	0.34	0.0047	0.38	0.11	0.34	0.24
3 ou plus	0.07	0.007	0.098	0.049	0.14	0.037

Tab. 5.1 – $P(\text{NbE} \mid \text{T}^\text{t})$: version apprise par reconnaissance de tâche bayésienne

Nous avons voulu spécifier que sachant que le personnage est en train de rechercher un bonus de santé, le nombre d'ennemis n'est pas perturbé, c'est-à-dire qu'il n'est pas différent de ce qu'il

prudent	Attaque	RechArme	RechSanté	Exploration	Fuite	DétDanger
0	0	x	x	x	0	x
1	x	0.1	x	0.001	0.1	0.001
2 ou plus	0.1	0.01	x	0.0001	x	0.0001

TAB. 5.2 – $P(\mathrm{NbE}|\mathrm{T}^{\mathrm{t}})$: version prudente écrite à la main

est en général. Ne connaissant pas *a priori* sa distribution, nous avons opté pour une distribution uniforme.

L'apprentissage nous montre que cette distribution n'est pas uniforme. Cela peut être dû à deux causes :

- en recherche de santé, le nombre d'ennemis n'est pas différent de ce qu'il est en général, mais sa distribution pendant l'apprentissage n'est pas uniforme ;
- en recherche de santé, la distribution sur le nombre d'ennemis est différente de ce qu'elle est en général.

En outre, si la discrétisation de nos variables est mauvaise, il est possible que certaines valeurs d'une variable sensorielle ne soient jamais observées durant l'apprentissage. Nous avons vu que cela conduit à des probabilités quasi-nulles dans les cases correspondantes, dont la faiblesse est le résultat d'une erreur dans le choix des variables plutôt que dans le modèle de séquencement.

Enfin, le calcul montre que la distribution uniforme que nous avons spécifiée à la main influe sur le choix de la tâche. Elle ne se simplifie pas, comme nous pourrions le vouloir (puisqu'elle est le reflet de notre ignorance).

Nous sommes donc confrontés à deux difficultés :

- nous ne savons pas précisément interpréter la différence entre ces deux distributions, ce qui est un problème pour pouvoir ajuster la table écrite à la main, et comprendre la table apprise ;
- la spécification d'une colonne uniforme par défaut a un effet imprévisible sur le comportement.

Si elles n'empêchent pas de réaliser des programmes fonctionnels, ces deux difficultés doivent être affrontées. Une solution possible serait de remplacer chaque facteur $\mathrm{P}(\mathrm{Capteur} \mid \mathrm{T}^{\mathrm{t}})$ par un facteur tel que celui utilisé en fusion par cohérence :

$$\mathrm{P}(\mathrm{M}_{\mathrm{capteur}} \mid \mathrm{Capteur}\ \mathrm{T}^{\mathrm{t}})$$

Il nous semble – sans avoir les moyens d'en apporter la preuve ici – que l'utilisation de la fusion par cohérence jusque dans notre modèle de séquencement aurait les avantages suivants :

- l'interprétation et la comparaison entre tables serait plus aisée ;
- les probabilités très faibles à cause des problèmes de discrétisation seraient évitées ;
- l'espace mémoire et le temps de calcul nécessaires seraient identiques à ceux du schéma de programmation inverse.

Enrichissement du modèle de séquencement de tâches

Dans la perspective de construire des comportements plus complexes, comme ceux pour les jeux en équipe, plusieurs directions sont ouvertes pour augmenter la complexité de notre modèle de séquencement.

La première est de construire des modèles séquencés hiérarchiques, où chaque tâches est elle-même issue d'un modèle de séquencement. Cela permettrait de conserver la maîtrise de la spécification de la relation tâches-capteurs quand le nombre de capteurs et le nombre de tâches deviennent grands. Le principal point bloquant pour développer des modèles hiérarchiques est l'adaptation de la technique d'apprentissage par démonstration.

La deuxième direction est d'enrichir l'état du modèle de séquencement au-delà de la simple tâche. Il est envisageable d'utiliser des informations supplémentaires, comme un objectif d'équipe, ou une seconde tâche parallèle à la première. Le verrou ici est d'une part l'identification des informations à ajouter à l'état pour un objectif donné ; d'autre part, l'adaptation de la méthode d'apprentissage à l'extension de l'état.

La troisième direction est la prise en compte de plus d'un pas de temps pour le séquencement. Cela pourrait consister à prendre en compte les tâches

$$\mathrm{T}^{\mathrm{t}-\mathrm{n}}\ \mathrm{T}^{\mathrm{t}-\mathrm{n}+1}\ ...\ \mathrm{T}^{\mathrm{t}}$$

au lieu de

$$\mathrm{T}^{\mathrm{t}-1}\ \mathrm{T}^{\mathrm{t}}$$

Celà apporterait plus d'expressivité pour la spécification manuelle d'un comportement. Les défis soulevés seraient :

- de montrer que cela conduit à de nouvelles possibilités intéressantes pour la spécification de comportements ;
- de montrer que ce gain d'expressivité ne compromet pas la simplicité de programmation manuelle du modèle de séquencement réellement
- de montrer que la complexité ajoutée ne pose pas un problème de calcul important pour l'exécution (*a priori* sans difficulté) et l'apprentissage (plus difficile).

Planification en programmation bayésienne

La plus grande perspective qui reste ouverte est l'intégration d'une technique de planification. Les tâches que nous avons présentées sont purement réactives, et leur séquencement n'est fait que d'un pas de temps au suivant. Il reste à développer une technique de planification qui s'intégrerait pleinement au formalisme de la programmation bayésienne. Notre personnage pourrait en tirer avantage pour la recherche d'armes ou de bonus de santé, alors qu'il réalise celles-ci de manière complètement réactive et locale.

5.3.2 Perspectives industrielles

Notre étude fait la démonstration de faisabilité de l'utilisation de comportements bayésiens pour la construction de comportements pour personnages de jeux vidéos autonomes.

La première perspective est liée à la relative pauvreté de l'environnement sur lequel nous nous sommes appuyés (*Unreal Tournament*). Celui-ci est un jeu de combat individuel, ou l'habileté motrice est cruciale (viser juste, courir vite, sauter souvent...), et où les aspects stratégiques sont très peu représentés (tenir une zone de l'environnement, contrôler l'accès aux armes et aux munitions...). Les comportements fortement agressifs y sont clairement les plus efficaces, et il est difficile d'y illustrer plusieurs approches de combat différentes et efficaces.

Sans abandonner le domaine des jeux de combat à la première personne, il est possible de se replacer sur une plateforme faisant place au jeu en équipes, où les aspects tactiques et stratégiques sont mis en valeur pour chaque joueur, comme au niveau des équipe. Un exemple d'une telle plateforme est le jeu commercial *Ennemy Territory*.

La seconde perspective est le passage à l'échelle industrielle. Nous avons construit un comportement d'une complexité limitée : il comprend 6 tâches, 6 variables sensorielles, et une unique variable motrice. L'application à une plateforme plus riche impliquerait plusieurs variables et tâches, et potentiellement un besoin pour des tâches hiérarchiques, c'est-à-dire des tâches constituées elles-mêmes de séquencement d'autres tâches. Nous avons montré comment les besoins en temps et en mémoire resteraient maîtrisés, mais le problème de l'apprentissage hiérarchique reste en suspens.

Bibliographie

[Arrigoni 03] A. Arrigoni. *Apprentissage Bayésien de Comportements pour Personnages de Jeux Vidéos*. Mémoire de diplôme d'etudes approfondies, Inst. Nat. Polytechnique de Grenoble, June 2003.

[Baum 72] L.E. Baum. *An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes*. Inequalities, vol. 3, pages 1–8, 1972.

[Cassandra 94] Anthony R. Cassandra, Leslie Pack Kaelbling & Michael L. Littman. *Acting Optimally in Partially Observable Stochastic Domains*. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), volume 2, pages 1023–1028, Seattle, Washington, USA, 1994. AAAI Press/MIT Press.

[Cooper 90] Gregory F. Cooper. *The computational complexity of probabilistic inference using Bayesian belief networks (research note)*. Artif. Intell., vol. 42, no. 2-3, pages 393–405, 1990.

[Dangauthier 03] P. Dangauthier. *Sélection automatique de variables pertinentes*. Dea, INPG, Grenoble (FR), June 2003.

[Dangauthier 05a] Pierre Dangauthier. *Feature Selection For Self-Supervised Learning*. Aaai spring symposium series, AAAI (American Association for Artificial Intelligence), 445 Burgess Drive, Menlo Park, California 94025-3442 USA, March 2005.

[Dangauthier 05b] Pierre Dangauthier, P. Bessière & A. Spalanzani. *Auto-supervised learning in the Bayesian Programming Framework*. In Proc. of the IEEE Int. Conf. on Robotics and Automation, Barcelona (ES), April 2005.

[Dempster 77] A.P. Dempster, N.M. Laird & D.B. Rubin. *Maximum-likelihood from incomplete data via the EM algorithm*. Journal of the Royal Statistical Society, vol. 39, no. 1, pages 1–38, 1977.

[Diard 03a] J. Diard. *La carte bayésienne : un modèle probabiliste hiérarchique pour la navigation en robotique mobile*. Thèse de doctorat, Inst. Nat. Polytechnique de Grenoble, January 2003.

[Diard 03b] J. Diard. *La carte bayésienne : un modèle probabiliste hiérarchique pour la navigation en robotique mobile.* PhD thesis, INPG, January 2003.

[Diard 03c] J. Diard, P. Bessière & E. Mazer. *Combining probabilistic models of space for mobile robots : the Bayesian Map and the Superposition operator.* In M. Armada & P. González de Santos, editeurs, Proc. of the Int. Advanced Robotics Programme, pages 65–72, Madrid (ES), October 2003. Int. Workshop on Service, Assistive and Personal Robots. Technical Challenges and Real World Application Perspectives.

[Diard 03d] J. Diard, P. Bessière & E. Mazer. *Hierarchies of probabilistic models of space for mobile robots : the bayesian map and the abstraction operator.* In Proc. of the Reasoning with Uncertainty in Robotics Workshop, Acapulco (MX), August 2003.

[Diard 04] J. Diard, P. Bessière & E. Mazer. *Hierarchies of probabilistic models of navigation : the Bayesian Map and the Abstraction operator.* In Proc. of the IEEE Int. Conf. on Robotics and Automation, New Orleans, LA (US), April 2004.

[Diard 05] J. Diard, P. Bessière & E. Mazer. *Merging probabilistic models of navigation : the Bayesian Map.* In Proc. of the IEEE-RSJ Int. Conf. on Intelligent Robots and Systems, pages 668–673, 2005.

[Dysband 00] E. Dysband. *A Finite-State Machine Class.* In M. Deloura, editeur, Game Programming Gems, pages 237–248. Charles River Media, 2000.

[Ferber 95] J. Ferber. Les systï¿½es multi-agents, vers une intelligence collective. InterEditions, 1995.

[Florez-Larrahondo 05] G. Florez-Larrahondo. *Incremental Learning of Discrete Hidden Markov Models.* PhD thesis, Mississipi State University, August 2005.

[Gelman 03] Andrew Gelman, John B. Carlin, Hal S. Stern & Donald B. Rubin. Bayesian data analysis, second edition. Chapman & Hall/CRC, July 2003.

[Harnad 90] S. Harnad. *The symbol grounding problem.* Physica D, vol. 42, pages 335–346, 1990.

[Howard 60] R. A. Howard. Dynamic programming and markov processes. The MIT Press, 1960.

[Jaynes 03] E. T. Jaynes. Probability theory : The logic of science. Cambridge University Press, April 2003.

[Koike 03] C. Koike, C. Pradalier, P. Bessière & E. Mazer. *Proscriptive Bayesian Programming Application for Collision Avoidance.* 2003.

[Koike 05] C. Koike. *Bayesian Approach to Action Selection and Attention Focusing. Application in Autonomous Robot Programming.* Thèse de doctorat, Inst. Nat. Polytechnique de Grenoble, Grenoble (FR), November 2005.

[Le Hy 04] R. Le Hy, A. Arrigoni, P. Bessière & O. Lebeltel. *Teaching Bayesian Behaviours to Video Game Characters*. Robotics and Autonomous Systems, vol. 47, pages 177–185, 2004.

[Lebeltel 99] O. Lebeltel. *Programmation Bayésienne des Robots*. PhD thesis, Institut National Polytechnique de Grenoble, FRANCE, 1999.

[Lebeltel 00a] O. Lebeltel, P. Bessière, J. Diard & E. Mazer. *Bayesian Robots Programming*. Rapport technique 1, Les Cahiers du Laboratoire Leibniz, Grenoble (FR), May 2000.

[Lebeltel 00b] O. Lebeltel, J. Diard, P. Bessière & E. Mazer. *A Bayesian framework for robotic programming*. In Proc. of the Int. Workshop on Bayesian Inference and Maximum Entropy Methods in Science and Engineering, Paris (FR), July 2000.

[Lebeltel 04a] O. Lebeltel, P. Bessière, J. Diard & E. Mazer. *Bayesian Robots Programming*. Autonomous Robots, vol. 16, no. 1, pages 49–79, 2004.

[Lebeltel 04b] O. Lebeltel, P. Bessière, J. Diard & E. Mazer. *Programmation bayésienne des robots*. Revue d'Intelligence Artificielle, vol. 18, no. 2, pages 261–298, 2004.

[Lorieux 03] J. Lorieux. *Détection et suivi de véhicules par télémétrie laser en robotique autonome*. Dea, INPG, Grenoble (FR), June 2003.

[Maes 91] P. Maes. *A bottom-up mechanism for behavior selection in an artificial creature*. In J.-A. Meyer & S. Wilson, editeurs, From Animals to Animats : The First International Conference on Simulation and Adaptive Behavior. The MIT Press, Cambridge, Massachusetts, 1991.

[Markov 6] A. A. Markov. Dynamic probabilistic systems, volume 1, chapitre Extension of the limit theorems of probability theory to a sum of variables connected in a chain. John Wiley and Sons, 1971 (orig. 1906).

[Mazer 98] E. Mazer, J.-M. Ahuactzin & P. Bessiï¿½e. *The Ariadne's Clew Algorithm*. Journal of Artificial Intelligence Research, vol. 9, pages 295–316, 1998.

[Petti 05] S. Petti & Th. Fraichard. *Partial Motion Planning Framework for Reactive Planning Within Dynamic Environments*. In Proc. of the IFAC/AAAI Int. Conf. on Informatics in Control, Automation and Robotics, Barcelona (SP), September 2005.

[Pradalier 03a] C. Pradalier, F. Colas & P. Bessière. *Expressing Bayesian Fusion as a Product of Distributions : Application to Randomized Hough Transform*. In Proc. of the Conf. on Bayesian Methods and Maximum Entropy in Science and Engineering, Jackson Hole, WY (US), August 2003.

[Pradalier 03b] C. Pradalier, F. Colas & P. Bessière. *Expressing Bayesian Fusion as a Product of Distributions : Applications in Robotics*. In Proc. of the

IEEE-RSJ Int. Conf. on Intelligent Robots and Systems, Las vegas, NV (US), October 2003.

[Pradalier 04] C. Pradalier. *Navigation intentionnelle d'un robot mobile*. Thèse de doctorat, Inst. Nat. Polytechnique de Grenoble, Grenoble (FR), September 2004.

[Rabiner 89] L.R. Rabiner. *A tutorial on hidden Markov models and selected applications inspeech recognition*. In Proceedings of the IEEE, volume 77, pages 257–286, February 1989.

[Simonin 05] E. Simonin, J. Diard & P. Bessière. *Learning Bayesian models of sensorimotor interaction : from random exploration toward the discovery of new behaviors*. In Proc. of the IEEE-RSJ Int. Conf. on Intelligent Robots and Systems, pages 1226–1231, 2005.

[Sutton 98] R. Sutton & A. Barto. Reinforcement learning : An introduction. MIT Press, 1998.

[Tinbergen 51] N. Tinbergen. The study of instinct. Clarendon Press, Oxford, England, 1951.

[Turing 50] A. M. Turing. *Computing Machinery and Intelligence*. Mind, vol. 59, pages 433–460, 1950.

[Tyrrell 93] T. Tyrrell. *Computational Mechanisms for Action Selection*. PhD thesis, University of Edinburgh, 1993.

[Zarozinski 01] M. Zarozinski. *Imploding Combinatorial Explosion in a Fuzzy System*. In M. Deloura, editeur, Game Programming Gems 2, pages 342–350. Charles River Media, 2001.