UNIVERSITÉ DU QUÉBEC À MONTRÉAL

INTELLIGENCE ARTIFICIELLE AVEC APPRENTISSAGE AUTOMATIQUE POUR L'ESTIMATION DE LA POSITION D'UN AGENT MOBILE EN UTILISANT LES MODÈLES DE MARKOV CACHÉS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

STATISTIQUE

PAR

CÉDRIC BEAULAC

NOVEMBRE 2015

REMERCIEMENTS

D'abord et avant tout, j'aimerais remercier Fabrice Larribe, mon directeur de maitrise. Merci d'avoir été patient, compréhensif et structuré. Merci de m'avoir fait grandir académiquement et professionnellement, de m'avoir fait confiance et d'avoir rendu mes visites à l'université non seulement enrichissantes mais aussi agréables.

Je tiens aussi à remercier Guillaume Racicot et Louis-Alexandre Vallières-Lavoie. Votre travail fut un élément clé du succès de cette recherche.

Je désire également remercier l'ensemble du corps professoral de l'UQAM pour sa passion et son dévouement au travail. J'aimerais particulièrement remercier Sorana Froda, François Watier, René Ferland, Serge Alalouf et Christophe Howhleg pour m'avoir aidé à travers diverses épreuves.

Je tiens aussi à remercier mes amis. Nos discussions ont contribué à faire évoluer mes idées.

J'aimerais également remercier ma famille pour son support et son amour inconditionnel. Finalement, un merci tout particulier à Martine, merci pour ta compréhension, ton amour, ta patience et ta contribution pendant la réalisation de ce mémoire.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

RÉSUMÉ

Dans ce mémoire, nous développons une méthodologie afin d'estimer la position d'un agent mobile dans un environnement borné. Nous utilisons un modèle de semi-Markov caché pour modéliser notre problématique. Dans ce contexte, l'état caché est la position de l'agent. Nous développons les algorithmes et programmons une intelligence artificielle capable de faire le travail de façon autonome. De plus, nous utilisons l'algorithme de Baum-Welch pour permettre à cette intelligence d'apprendre de ses expériences et de fournir des estimations plus précises au fil du temps. Finalement, nous construisons un jeu vidéo pour mettre à l'épreuve cette intelligence artificielle. Le fonctionnement de la méthode est illustré avec plusieurs scénarios, et nous montrons que la méthode que nous proposons est meilleure que d'autres.

MOTS-CLÉS : modèles de Markov cachés, modèles de semi-Markov cachés, intelligence artificielle, apprentissage automatique, jeu vidéo.

INTRODUCTION

Depuis quelques années, grâce à une technologie en constante évolution, la puissance des ordinateurs a connu une importante progression. Cette nouvelle puissance ouvre la porte à l'implémentation d'algorithmes mathématiques bien plus complexes qu'avant. La programmation et l'intelligence artificielle sont des domaines qui vont de pair avec les mathématiques et la statistique. Bien que de nombreux chercheurs en science informatique travaillent au développement de nouvelles intelligences artificielles performantes, le besoin de puissants outils statistiques est présent.

Dans ce mémoire, nous allons proposer une solution à la problématique de l'estimation de la position d'un agent mobile. Bien qu'il s'agisse d'une problématique que l'on retrouve dans de nombreuses situations, nous allons l'approcher avec l'intention de mettre au point une intelligence artificielle. Nous allons modéliser cette problématique, puis la résoudre à l'aide des modèles de semi-Markov cachés, notre méthodologie d'intérêt. Nous allons ensuite mettre au point une intelligence artificielle capable d'effectuer ce travail de façon autonome et programmer les algorithmes nécessaires. Cette intelligence sera dotée de la capacité d'apprendre de ses diverses expériences et d'améliorer la qualité de ses estimations. Bien que cette méthodologie puisse servir dans de multiples situations, nous nous concentrerons sur son efficacité dans l'environnement des jeux vidéo. Il s'agit d'un environnement où cette problématique est très présente.

C'est pourquoi le premier chapitre va introduire quelques concepts entourant les

jeux vidéo ainsi que l'intelligence artificielle. Les deux chapitres suivants décriront comment utiliser nos principaux outils de travail, les modèles de Markov cachés et les modèles de semi-Markov cachés. Par la suite, nous verrons comment modéliser notre problématique et comment utiliser notre méthodologie mathématique pour analyser et résoudre efficacement ce problème. Le chapitre cinq décrira comment la programmation de l'intelligence artificielle et de l'environnement de tests furent élaborés. Ensuite, le chapitre six démontrera nos résultats. Pour terminer, la dernière section présentera une courte liste de quelques améliorations potentielles du modèle.

CHAPITRE I

INTRODUCTION À L'INTELLIGENCE ARTIFICIELLE ET AUX JEUX VIDÉO

Une courte introduction de diverses notions sera effectuée ici. Nous définirons ce qu'est une intelligence artificielle pour nous et introduirons les mécaniques derrière certains jeux vidéo.

1.1 Intelligence artificielle (*IA*)

Définissons d'abord de façon informelle ce qu'est l'intelligence artificielle. L'objectif de la recherche en intelligence artificielle est de doter un système informatique de capacités de réflexion similaires à celles des humains. Il y a donc un défi dans la compréhension du raisonnement humain, mais surtout dans la modélisation et dans la reproduction de celui-ci.

C'est ici que les mathématiques et la statistique entrent en ligne de compte. Comme plusieurs domaines maintenant considérés connexes à la statistique : la biologie, la génétique et la finance par exemple, la recherche en intelligence artificielle nécessite un travail de modélisation statistique rigoureux. Les notions de choix selon certains critères, de maximisation du bonheur et de réflexion au sens plus large peuvent être reproduites par des modèles mathématiques et probabilistes complexes, sans compter que certains évènements aléatoires surviennent et

affectent notre prise de décision. La modélisation du raisonnement humain est très complexe, mais il est réaliste de croire que divers outils mathématiques peuvent être utilisés pour reproduire fidèlement des parcelles de ce complexe réseau de neurones. L'avancement dans ce domaine repose sur un travail auquel des mathématiciens et des statisticiens peuvent contribuer.

1.2 Jeux vidéo

Ce mémoire ne traite pas précisément de jeux vidéo ; néanmoins, nous introduirons ici rapidement quelques notions concernant ceux-ci. Toute connaissance sur le sujet peut aider le lecteur à visualiser certains éléments qui seront mentionnés au cours des prochaines pages. Évidemment, nous énoncerons les notions absolument nécessaires au cours des prochains paragraphes. Il sera tout d'abord question de la définition d'un avatar. Puis, le fonctionnement de certains jeux sera brièvement décrit.

1.2.1 Avatar

Il est important de saisir le principe d'avatar si on veut représenter les jeux vidéo convenablement. L'avatar est la représentation de quelqu'un dans un autre univers. Plus précisément, dans le contexte d'un jeu vidéo, l'avatar du joueur est celui qui le représente à l'intérieur du jeu. Plus précisément, l'avatar d'un joueur obéit aux commandes de ce dernier ; il est ce qui permet au joueur d'interagir dans l'univers du jeu. Par exemple, dans le célèbre jeu vidéo *Super Mario Bros.*, Mario est l'avatar du joueur, puisqu'il est le personnage que contrôle le joueur. Le joueur envoie des commandes à son avatar par l'entremise d'un clavier, d'une souris ou d'une manette, entre autres.

Lorsqu'un avatar est contrôlé par une intelligence artificielle, nous l'appellerons *bot*. Cette désignation est usuelle dans l'univers des jeux vidéos et il est important

de considérer que plusieurs termes relatifs à cet univers sont couramment utilisés sous leur version anglophone. En outre, l'ordinateur contrôle la totalité des *NPC's* dans un jeu, terme provenant de l'anglais *Non-playable character*, c'est-à-dire un personnage «non jouable», en français. Un *NPC* est l'équivalent d'un figurant, mais dans le contexte d'un jeu vidéo. Nous réservons, par contre, le nom de *bot* pour les avatars qui pourraient être contrôlés par un joueur humain mais qui sont contrôlés par l'intelligence du jeu lorsque le nombre de joueurs humains est insuffisant pour combler les besoins en avatars.

1.2.2 Catégories de jeux

Tout comme il existe plusieurs sports, il existe plusieurs types de jeux. Plusieurs d'entre eux seront brièvement décrits dans cette section. Il est à noter que nous utiliserons très peu ces informations dans ce document, mais qu'il est essentiel de les définir afin d'avoir une référence commune concernant le monde des jeux vidéo. Ce faisant, nous nous assurons que la totalité des lecteurs possèdent une base minimale de connaissances dans ce domaine.

1.2.2.1 Jeux compétitifs en ligne *(PvP)*

En ce qui a trait à l'amélioration des intelligences artificielles, les jeux compétitifs sont ceux nécessitant le plus d'effort puisque celle-ci doit imiter un comportement humain réaliste. Ces types de jeux, appelés parfois joueur contre joueur (ou *Player versus player (PvP)* en anglais), sont constitués de deux ou de plusieurs joueurs qui s'affrontent, en équipe ou individuellement. Ces jeux se déroulent dans un environnement borné appelé une *carte de jeu*, de l'anglais *map*. C'est dans cet environnement que les avatars des divers joueurs s'affrontent. Une carte de jeu est donc l'équivalent du terrain pour un sport. Tout comme il existe plusieurs surfaces de jeu au tennis, il existe souvent plusieurs cartes pour le même jeu.

Par contre, imaginons qu'un joueur désire s'adapter aux mécaniques du jeu, apprendre les divers environnements ou simplement s'entraîner. Pour ce faire, il débutera une partie contre l'ordinateur. C'est à ce moment que les jeux compétitifs deviennent importants pour nous, car ici, l'intelligence artificielle devra prendre le contrôle d'un avatar usuellement utilisé par un joueur humain, ce qui n'est pas le cas dans un jeu qui se joue seul. Dans un jeu qui n'est pas compétitif, l'ordinateur contrôle la totalité des éléments de l'environnement du joueur. Dans cette situation, l'intelligence artificielle est programmée au choix du directeur de jeu dans l'optique de créer une expérience intéressante pour le joueur. Par contre, dans notre scénario de jeux compétitifs en ligne, l'intelligence contrôle non seulement des éléments dans l'environnement, mais aussi un avatar possiblement humain lorsque nous débutons une partie contre un *bot*. Dans ce contexte, il serait naturel que le *bot* possède la même information, utilise les mêmes commandes et le même ensemble de règles qu'un vrai joueur possèderait. Voilà pourquoi les jeux à l'étude sont tous des *PvP*, car notre objectif est de donner des outils de prise de décision à l'intelligence artificielle. Ceci permettra au *bot* de se comporter comme un avatar contrôlé par un vrai joueur. Nous élaborerons sur la forme actuelle des *bots* plus tard dans ce chapitre.

1.2.2.2 Jeu de tir en vue subjective (*FPS*)

Communément appelé *FPS*, de l'anglais *First-Person Shooter*, un jeu de tir en vue subjective est un jeu de simulation de conflit armé. Dans ce type de jeu très populaire à l'ordinateur comme sur console, l'avatar du joueur est souvent un être humain ou un humanoïde, détenant une variété d'armes à feu. Le joueur voit à travers l'avatar ; cette perspective étant appelée «vue à la première personne» (voir figure 1.1). De cette manière, le joueur est placé directement à l'intérieur de l'avatar, voyant ses mains devant lui, par exemple, et possédant une vision périphérique similaire à ce qu'un être humain possède.

Figure 1.1 Vue à la première personne, *Counter-Strike*

Dans les *FPS* compétitifs, une équipe l'emporte de manière générale en éliminant tous les membres de l'équipe adverse. Certains de ces jeux ont une série d'objectifs dans le but de stimuler les affrontements. À titre d'exemple, nous décrirons rapidement le jeu *Counter-Strike* puisqu'il s'agit d'un jeu toujours très populaire dans lequel de nombreux joueurs souhaitent s'entraîner pour améliorer leurs performances. De plus, ce jeu est au coeur de l'article de S. Hladky (Hladky et Bulitko, 2008), un document ayant inspiré nos travaux.

Dans le jeu *Counter-Strike*, deux équipes s'affrontent, les «terroristes» et les «anti-terroristes». L'objectif des «terroristes» est de poser une bombe à l'un des deux sites possibles avant l'expiration d'un certain délai de temps, sans quoi ils perdent. S'ils réussissent et que la bombe explose, ils l'emportent. Les «anti-terroristes» doivent empêcher que cela se produise. Si une équipe élimine tous les membres

de l'équipe adverse, elle est victorieuse, à moins que la bombe n'ait été amorcée. Dans ce cas, les «anti-terroristes» possèdent un certain délai de temps pour la désamorcer s'ils veulent gagner. On observe ici que l'objectif entourant la bombe force les affrontements, car si les «terroristes» ne tentent pas de la faire exploser, ils perdent. Nous détaillerons le fonctionnement actuel des *bots* dans la section 1.2.3.

1.2.2.3 Jeu de stratégie en temps réel (*RTS*)

Un autre type de jeu pour lequel certains chercheurs ont tenté d'utiliser les modèles de Markov cachés est le *RTS*, de l'anglais *Real-time strategy* (Southey *et al.*, 2007). Ce jeu se joue d'une vue aérienne en deux dimensions. Il s'agit d'un jeu de simulation de guerre où le joueur ne contrôle pas seulement un personnage, mais bien une base de commande ainsi qu'une armée. Dans ce contexte, il faut gérer les déplacements de son armée, sa composition, sa stratégie ainsi que gérer ses ressources, ses multiples bases de commandement et le travail des ouvriers dans celles-ci. Le joueur voit la scène comme s'il était une sorte de dieu tout-puissant dans le ciel (voir figure 1.2). Par contre, son avatar, c'est-à-dire sa représentation dans le jeu, est la totalité des membres de son armée, ses travailleurs ainsi que les bâtiments qu'il commande.

Figure 1.2 Vue aérienne, *Starcraft 2*

Ce jeu est principalement conçu pour l'ordinateur. Alors que dans un *FPS*, les réflexes et la précision sont les éléments déterminants, c'est la stratégie économique et la stratégie militaire qui sont centrales dans les *RTS*.

Les parties de *Real-time strategy* compétitives opposent 2 ou plusieurs joueurs débutant tous avec le même nombre de ressources et une base similaire. L'objectif est tout simplement de détruire la base de l'adversaire. Pour ce faire, le joueur devra mettre au point une économie, une base de commandement sophistiquée et une armée adéquate pour accomplir le travail, tout en empêchant son adversaire de faire de même. Encore une fois, il est ici possible d'affronter un *bot* pour s'entraîner.

La composante de temps réel (*Real-Time*) signifie que nous sommes dans un univers à temps continu, en opposition aux célèbres jeux tour par tour (*turn-based*)

où les joueurs font une action chacun leur tour comme aux échecs, par exemple.

1.2.2.4 Arène de bataille en ligne multijoueur (*MOBA*)

Le *Multiplayer online battle arena*, *MOBA*, traduit en français par «arène de bataille en ligne multijoueur» est le cadet des jeux décrits dans ce chapitre. Originallement une sous-branche des *RTS*, le *MOBA* est actuellement un des types de jeux compétitifs le plus joué en ligne, par sa simplicité et sa facilité d'accès.

Traditionnellement, un *MOBA* voit deux équipes de 5 joueurs s'affronter. Sur ce terrain de jeu symétrique (voir figure 1.3), de forme carrée, chaque équipe possède un château positionné dans deux coins opposés du carré. Ces derniers sont reliés par trois routes principales, l'une est la droite diagonale reliant ces châteaux, les deux autres longent les cotés de ce terrain carré. Les routes sont gardées par des tours de défense et des soldats contrôlés par l'ordinateur se battent sur ses routes.

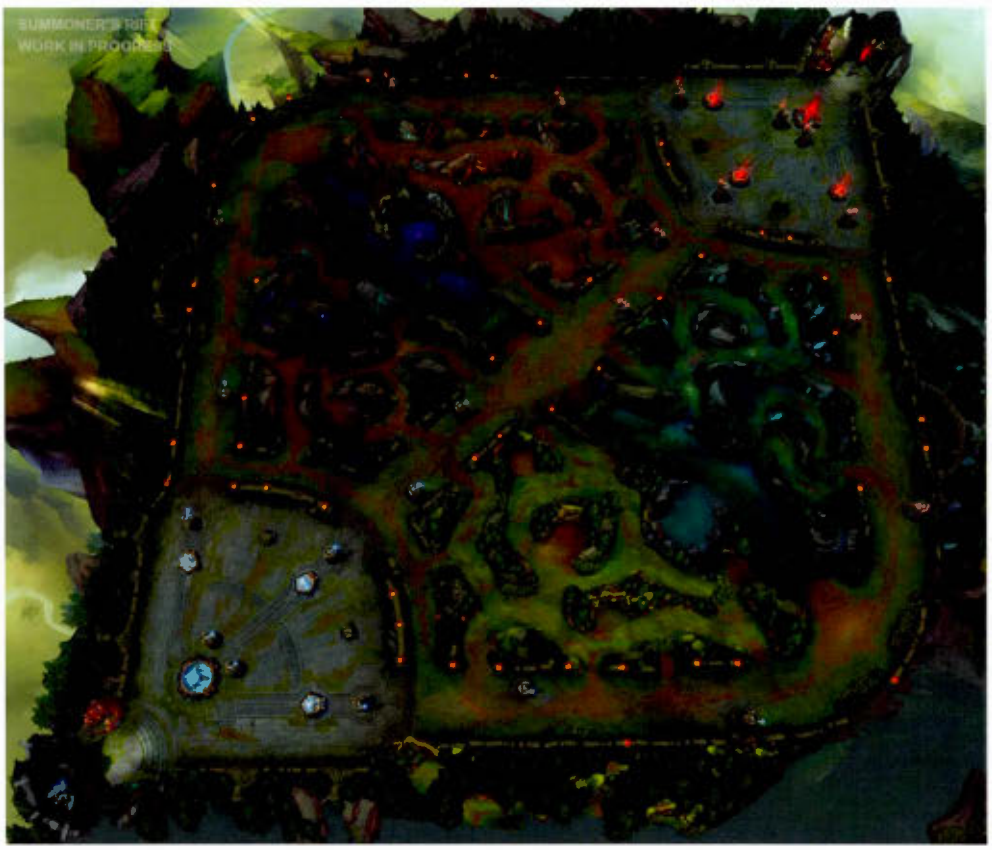

Figure 1.3 Carte de jeu de *League of Legends*

Les joueurs doivent se séparer en sous-groupes pour tenter de progresser, en détruisant les tours sur chacune des routes, avec pour objectif ultime de détruire le château adverse. Plusieurs jeux populaires dont *DOTA 2* et *League of Legends* utilisent le même type de caméra aérienne que l'on retrouve chez leur ancêtre le *RTS*. Malgré tout, certains *MOBA*, comme *Smite*, utilisent une vue à la troisième personne (voir figure 1.4). La caméra est donc située à quelques pieds derrière l'avatar. Ainsi, nous voyons à l'aide d'un cône de vision, mais nous voyons aussi ce qui se déroule derrière notre avatar, sur une certaine distance.

Figure 1.4 Exemple de vue à la troisième personne tiré du jeu *Smite*

Comparativement à un *FPS* où les affrontements sont de courte durée une fois que l'on est en contact avec un adversaire, ici des joueurs opposés s'affrontent constamment dans des batailles de longue haleine. Ces types de jeux sont très intéressants pour nous puisqu'ils ont comme seul objectif d'être des jeux compétitifs. Alors, les besoins de *bots* performants sont criants. De plus, étant le type de jeu le plus populaire, de nombreux nouveaux joueurs débutent leur expérience de jeu et se voient confrontés à des intelligences artificielles décevantes.

1.2.2.5 Autres genres

Plusieurs autres types de jeux peuvent être joués de manière compétitive. Ceux-ci possèdent peut-être des mécaniques de jouabilité, des perspectives ou bien des objectifs différents. Par contre, le besoin d'estimer la position de nos adversaires demeure une priorité dans la majorité de ceux-ci. Bien que nous nous soyons pen-

chés sur certains types de jeu et que notre approche prenne parfois en exemple un jeu en particulier, il est essentiel de comprendre que le développement mathématique ici exprimé sera fait de manière la plus générale possible pour lui permettre de s'adapter à une multitude de types de jeux vidéo, mais aussi à une grande variété de contextes.

1.2.3 État actuel de l'intelligence artificielle dans les jeux

Penchons-nous désormais sur la problématique ayant piquée notre curiosité, la capacité d'une intelligence artificielle à estimer la position d'un agent mobile. Cette problématique est courante dans le contexte des jeux vidéo que nous venons tout juste de décrire. À cet effet, nous expliquerons comment l'intelligence artificielle fut développée dans ces jeux. Il y a bien longtemps que l'on retrouve des *bots* dans les jeux vidéo compétitifs, même à l'époque où les ordinateurs ne possédaient pas les capacités calculatoires qu'ils ont aujourd'hui. Comment les programmeurs ont-ils réussi à mettre au point des intelligences artificielles capables de raisonner une stratégie de guerre dans un *RTS* ou de tendre une embuscade planifiée dans un *FPS* ? Selon Bererton (Bererton, 2004), la réponse est simple : *les intelligences artificielles trichent toujours.*

Nous savons tous ce que signifie tricher ; ne pas respecter un ensemble de règles auxquelles la totalité des joueurs a accepté de se conformer. Dans un contexte de jeux vidéo, lorsque l'intelligence artificielle utilise des paramètres différents du joueur, possède de l'information privilégiée ou un ensemble de règles différentes, on peut affirmer qu'elle triche. Stephen Hladky (Hladky, 2009) nous énumère quelques exemples à l'intérieur de son mémoire. Il mentionne, entre autre, la possibilité qu'un *bot* obtienne un plus grand nombre de ressources dans un *RTS* que ce que reçoit un joueur réel. Il est aussi possible qu'un soldat, dans un contexte de *FPS*, soit doté d'une précision surhumaine.

C'est l'incapacité de créer un intelligence artificielle talentueuse qui a forcé plusieurs directeurs de jeux à mettre au point un *bot* tricheur. Cette incapacité fut causée par deux éléments. Premièrement, l'utilisation insuffisante des outils mathématiques à l'intérieur de ces jeux ; certains outils n'étaient pas assez développés à l'époque alors que d'autres n'ont tout simplement pas été considérés par les grandes entreprises du domaine. L'autre cause était la faible puissance des ordinateurs à l'époque de la naissance des jeux vidéo. Ceci empêchait l'utilisation de nombreux algorithmes qui auraient pu être utiles.

1.3 Présentation du modèle

Notre objectif est de doter notre intelligence artificielle d'outils lui permettant d'être suffisamment performante pour que la nécessité de tricher ne soit plus dans le contexte du jeu vidéo. Nous nous attarderons plus particulièrement à l'estimation de la position de l'adversaire. Dans tout jeu décrit précédemment, estimer la position de notre adversaire est nécessaire et malheureusement les *bots* en sont incapables. Dans le contexte d'un *MOBA*, la totalité des membres d'une équipe pourront, par exemple, avancer sur l'une des trois routes et détruire des tours s'ils savent que leurs adversaires sont suffisamment loin. Dans un *FPS*, le fait de savoir où se situe ses adversaires peut permettre à une équipe de bien se positionner pour ouvrir le feu à son avantage. Bref, il s'agit d'un élément central de plusieurs jeux qui contribue énormément à la victoire. Alors que chaque joueur humain tente de faire du mieux qu'il peut pour estimer la position de l'adversaire, l'*IA* ne le fait jamais. En effet, actuellement, le *bot* voit simplement à travers les murs dans certains jeux, observe la totalité de la carte de jeu et, parfois même, se promène selon une route tracée d'avance sans réellement tenter de savoir où est son adversaire.

Dans le contexte de ce genre de jeu, nous supposons que les déplacements d'un joueur sur la carte de jeu suivent un processus stochastique où l'ensemble des

positions possibles forme l'espace des états. Il serait donc possible de faire de l'inférence sur les déplacements de ce joueur. Dans cette situation, la chaîne de Markov est homogène. Néanmoins, si nous nous mettons dans la position de l'intelligence artificielle et que nous ne voulons pas tricher, mais que nous désirons estimer la position de ce joueur, nous sommes face à une situation où une variable markovienne évolue mais où nous ne connaissons point l'état de la chaîne. Heureusement pour nous, nous obtenons tout de même des observations de l'environnement, tout comme le joueur en possède. Il est raisonnable de considérer que l'intelligence artificielle puisse observer ce qui se situe dans son champ de vision. Ces observations sont aléatoires et dépendent de l'état de la chaîne de Markov. Pour continuer à travailler, nous devrons introduire un nouvel outil, les chaînes de Markov cachées.

CHAPITRE II

MODÈLES DE MARKOV CACHÉS

2.1 Modèles de Markov

Une connaissance de base des chaînes de Markov est essentielle à la description de notre méthodologie. Donc, nous ferons une brève introduction de certaines caractéristiques et des notations utilisées au cours de ce travail. Un processus stochastique est une variable aléatoire indicée. L'indice peut représenter le temps ou une distance, par exemple. Notons ici q_t, le processus stochastique, où q est la variable aléatoire et t est l'indice. À travers ce mémoire, l'indice représentera le temps. Une chaîne de Markov est un processus stochastique où la variable répond à la propriété markovienne, qui s'écrit comme suit :

$$P(q_t = s_i | q_1, q_2, ..., q_{t-1}) = P(q_t = s_i | q_{t-1}). \tag{2.1}$$

La variable aléatoire q_t peut prendre plusieurs valeurs, souvent appelées *états* lorsqu'on parle de chaînes de Markov, avec une certaine probabilité. Néanmoins, cette probabilité n'est pas fixe ; elle dépend de l'état dans lequel se trouvait le processus précédemment. Par cette propriété, nous observons que la probabilité que la chaîne soit à un certain état s_j au temps t dépend seulement de l'état,

s_i, par exemple, où le processus se trouvait au temps $t-1$. Cette probabilité $a_{ij} = P[q_t = s_j | q_{t-1} = s_i]$ est nommée probabilité de transition de s_i vers s_j. Si cette probabilité ne dépend pas de t, la chaîne de Markov est dite homogène.

Pour simplifier certains calculs et certaines notations, il est commun d'entrer ces probabilités de transition dans une matrice de transition notée ici $\mathbf{A}$. Pour un modèle à n états, nous aurons alors une matrice $\mathbf{A}$ de dimension $n \times n$, où les lignes représentent les états de départ et les colonnes, les états d'arrivée. L'élément a_{ij} de la matrice de transition $\mathbf{A}$ est alors la probabilité de passer de l'état s_i vers l'état s_j en une transition. Or, souvenons-nous que la multiplication matricielle nous permet d'obtenir facilement les probabilités de transition de s_i vers s_j en h étapes ; il s'agit simplement de l'élément à la position (i, j) de la matrice $\mathbf{A}^h$. Définissons $\mu_i = P[q_1 = s_i]$, la probabilité que le processus débute à l'état s_i, appelée probabilité initiale. Il est standard de considérer que le modèle markovien possède un vecteur de probabilité initiale $\mu = (\mu_1, \mu_2, ..., \mu_n)$ contenant la totalité de ces probabilités.

On dit que s_j est accessible de s_i si $\sum_{n=1}^{\infty} P[q_{t+n} = s_j | q_t = s_i] > 0$, en d'autres mots, si le processus peut passer de l'état s_i vers l'état s_j éventuellement. Deux états communiquent s'ils sont accessibles l'un à l'autre. Si tous les états sont accessibles entre eux, la chaîne est qualifiée d'irréductible.

Finalement, il existe beaucoup d'autres définitions et résultats connus au sujet des modèles de Markov que nous ne mentionnerons pas ici. Pour plus d'informations, le manuel de Ross (Ross, 2009) est une excellente référence.

2.2 Présentation générale des modèles de Markov cachés

Un modèle de Markov caché peut être décrit en quelques mots comme étant une chaîne de Markov standard dont les états sont cachés. C'est-à-dire qu'il y a un processus markovien dont les probabilités de transition ne dépendent que de l'état actuel du processus, tel que décrit à la section 2.1. Cependant, les données observées ne sont pas les états, mais bien une variable aléatoire fonction des états. Au temps t, la chaîne est à l'état s_i. Par contre, ce qui est observé ici n'est pas l'état s_i, mais plutôt le résultat d'une variable aléatoire qui dépend de l'état dans lequel la chaîne se situe. Lorsque nous travaillons avec des modèles de Markov cachés, le défi est justement d'estimer les paramètres des lois d'observation et les probabilités de transition de la matrice sous-jacente en n'ayant que les résultats de cette variable aléatoire dépendante de l'état comme observations.

Il est important de noter que les observations peuvent être de n'importe quelle loi. Nous pourrions avoir des observations qui sont de loi normale, alors la variable markovienne cachée pourrait influencer la moyenne, la variance ou bien les deux paramètres, par exemple. La variable observée pourrait aussi être de Poisson. Dans ce cas, la variable sous-jacente influencerait le paramètre λ. Cependant, la variable observée pourrait aussi ne suivre aucune loi de probabilité connue. Nos observations doivent seulement être aléatoires et dépendre de l'état de la chaîne.

2.3 Définition formelle

Définissons plus formellement les composantes d'un modèle de Markov caché. Il y a tout d'abord une variable aléatoire markovienne standard, q_t, pouvant prendre n valeurs possibles appelées états. Notons le vecteur d'état comme étant $S = \{s_1, s_2, ..., s_n\}$, duquel nous pouvons établir une matrice $\mathbf{A}$ de taille $n \times n$, où l'élément à la position (i, j), a_{ij}, est la probabilité de transition de l'état s_i

vers l'état s_j, c'est-à-dire $P[q_{t+1} = s_j | q_t = s_i]$ et ce, pour tout t si la chaîne est homogène. Il y a bien entendu un vecteur de probabilité initiale tel que défini à la section 2.1 : $\mu = (\mu_1, \mu_2, ..., \mu_n)$ où $\mu_i = P[q_1 = s_i]$. Dans notre contexte, nous supposerons que la chaîne de Markov est homogène. Bien que la théorie sur les modèles de Markov cachés s'applique sans cette hypothèse, cette dernière est vraie dans notre problématique et simplifie grandement nos calculs.

Néanmoins, ici la variable markovienne est cachée. Conséquemment, l'observation est une variable aléatoire sur celle-ci. Par exemple, supposons que nous avons observé T réalisations de notre variable. Nous avons $q_1, q_2, ...q_T$, la séquence des réalisations de la variable de Markov, les états visités. Toutefois, ce que nous observons est la séquence $x_1, x_2, ..., x_T$ des T observations. Dans un modèle de Markov caché standard, les observations ne dépendent que de l'état de la chaîne sous-jacente. Notons donc $b_i(x_t) = P[X_t = x_t | q_t = s_i]$, la probabilité d'obtenir l'observation x_t sachant que l'état sous-jacent est s_i. La figure 2.1 peut nous aider à visualiser le comportement de ce processus. Ici, b, la loi d'observation, aussi appelée loi de masquage, peut prendre plusieurs formes comme mentionné précédemment. Supposons, par exemple, que la loi d'observation était de Poisson et que l'état sous-jacent influençait λ, un λ_i pourrait être associé à chacun des n états. Dans ce contexte, $b_i(x_t) = e^{-\lambda_i} \lambda_i^{x_t} / x_t!$.

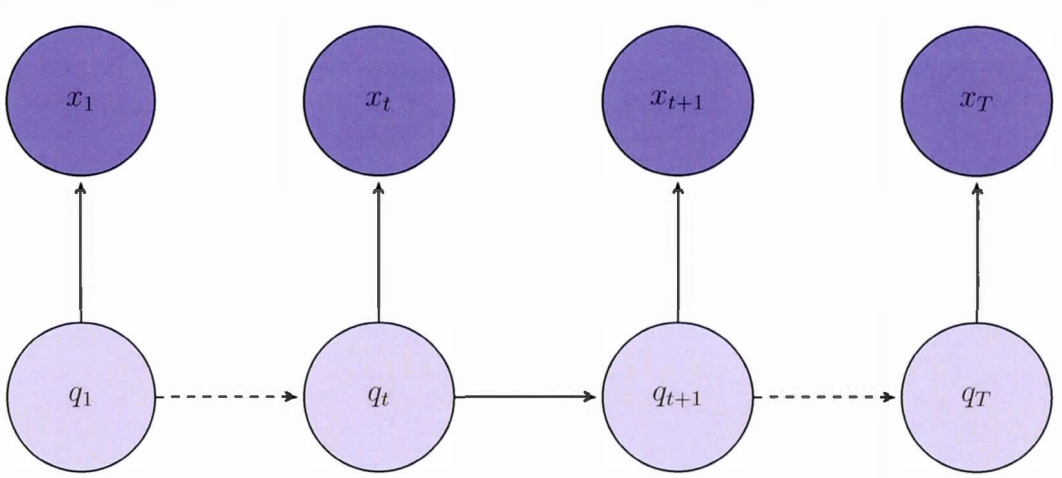

Figure 2.1 Illustration de T réalisations d'une chaîne de Markov cachée. Notons que seuls les x_t sont observés. Les éléments pâles ne sont pas observés.

Pour aider certains lecteurs à mieux visualiser cet outil, nous avons mis au point un petit exemple. Imaginons un modèle très simple où la variable markovienne peut prendre seulement deux valeurs, avec pour matrice de transition :

$$\mathbf{A} = \begin{bmatrix} 0.3 & 0.7 \\ 0.6 & 0.4 \end{bmatrix}$$

Dans ce modèle, les observations seront de Poisson et l'état affectera le paramètre λ. Si l'état de la variable markovienne est 1, $\lambda = 5$, sinon, $\lambda = 20$. À l'aide d'un générateur que nous avons construit, nous avons obtenu la série d'états visités suivante : $\{1, 1, 2, 1, 2\}$. Il a découlé de ces états la série d'observations suivante : $\{5, 6, 18, 5, 17\}$ (voir figure 2.2).

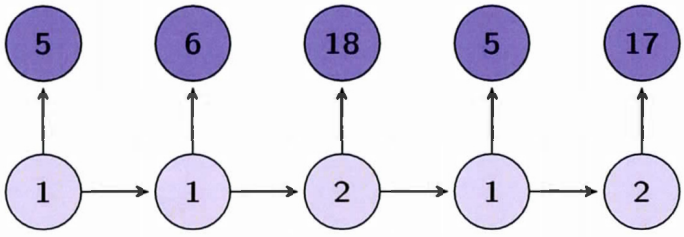

Figure 2.2 Illustration d'une séquence de 5 observations de l'exemple d'une chaîne de Markov cachée énoncée. Les éléments pâles ne sont pas observés.

2.4 Inférence

2.4.1 Algorithme *forward-backward*

2.4.1.1 Algorithme *forward*

Dans les problèmes d'inférence qui nous intéressent, nous cherchons à estimer les paramètres de la loi de masquage et de la matrice de transition sous-jacente à l'aide de nos T observations $x_1, x_2, x_3, ..., x_T$. Face à un problème entourant une chaîne de Markov cachée, il serait intéressant de calculer la probabilité d'être à un certain état à un moment quelconque connaissant nos observations, $P[q_t = s_i | x_1, x_2, x_3, ...x_t, ..., x_T]$. C'est ici que l'algorithme *forward-backward* entre en ligne de compte. Parfois considérés comme deux algorithmes distincts, *forward* et *backward*, ceux-ci sont cependant très similaires dans la logique qui a servi à leur construction. Par contre, ils demeurent très différents et complémentaires dans leur application. Nous commencerons par discuter en profondeur de l'algorithme *forward*, puis nous verrons rapidement comment utiliser ce travail à la construction de l'algorithme *backward*.

Supposons que nous voulons connaître $P[q_t = s_i | x_1, x_2, x_3, ...x_t]$, nous pourrions passer par la probabilité conjointe $P[q_t = s_i \cap x_1, x_2, x_3, ...x_t]$ pour arriver à nos fins.

C'est ce que fait l'algorithme *forward*. Il a pour objectif d'estimer la probabilité conjointe d'avoir observé la totalité des observations obtenues jusqu'au temps t et d'être à l'état sous-jacent s_i et ce pour tout t et pour tout s_i. Ces probabilités sont notées α, donc, $\alpha_t(i) = P[x_1, x_2, ..., x_t \cap q_t = s_i]$

Traditionnellement, cette valeur est calculée de manière récursive en regardant la totalité des états potentiellement visités à l'étape précédente.

Calculons donc $\alpha_{t+1}(j)$:

$$\begin{aligned}
\alpha_{t+1}(j) &= P(x_1, x_2, ...x_t, x_{t+1} \cap q_{t+1} = s_j) \\
&= \sum_{i=1}^{n} P(x_1, x_2, ...x_t, x_{t+1} \cap q_t = s_i \cap q_{t+1} = s_j) \\
&= \sum_{i=1}^{n} P(x_{t+1} \cap q_{t+1} = s_j | x_1, x_2, ...x_t \cap q_t = s_i) P(x_1, x_2, ...x_t \cap q_t = s_i) \\
&= \sum_{i=1}^{n} P(x_{t+1} \cap q_{t+1} = s_j | q_t = S_i) \alpha_t(i) \\
&= \sum_{i=1}^{n} P(x_{t+1} | q_{t+1} = s_j) P(q_{t+1} = s_j | q_t = s_i) \alpha_t(i) \\
&= \sum_{i=1}^{n} b_j(x_{t+1}) a_{ij} \alpha_t(i) \\
&= \sum_{i=1}^{n} \alpha_t(i) a_{ij} b_j(x_{t+1}).
\end{aligned} \tag{2.2}$$

Comme il s'agit d'un algorithme récursif, il est nécessaire de connaitre la valeur initiale. Celle-ci s'obtient facilement grâce au vecteur de probabilité initiale :

$$\begin{aligned}\alpha_1(i) &= P[q_1 = s_i, X_1 = x_1] \\ &= P[X_1 = x_1 | q_1 = s_i] P[q_1 = s_i] \\ &= P[q_1 = s_i] P[X_1 = x_1 | q_1 = s_i] \\ &= \mu_i b_i(x_1).\end{aligned}$$

À l'aide des ces α, nous pouvons calculer de nombreuses probabilités utiles (équations 2.3 et 2.4), par exemple :

$$P[x_1, x_2, ..., x_t] = \sum_{i=1}^{n} P[x_1, x_2, ..., x_t \cap q_t = s_i] = \sum_{i=1}^{n} \alpha_t(i), \tag{2.3}$$

ou bien :

$$P[q_t = s_i | x_1, x_2, ...x_t] = \frac{P[q_t = s_i \cap x_1, x_2, ...x_t]}{P[x_1, x_2, ..., x_t]} = \alpha_t(i) / \sum_{i=1}^{n} \alpha_t(i). \tag{2.4}$$

Dans le calcul des multiples α, une somme d'un produit est réalisée. C'est pourquoi nous pourrions être tentés de voir ceux-ci comme étant des produits matriciels. Posons α_t comme un vecteur défini comme suit : $\alpha_t = [\alpha_t(1), \alpha_t(2), ..., \alpha_t(n)]$. L'écriture que nous introduirons provient de Zucchini (Zucchini et MacDonals, 2009), elle est peut-être moins intuitive, mais beaucoup plus compacte et beaucoup plus simple à programmer. Définissons d'abord $\mathbf{P}(x_t)$. Il s'agit d'une matrice diagonale, où les éléments $p_{i,i}(x_t)$ sont les probabilités d'avoir observé x_t sachant que la chaîne est à l'état s_i, soit : $b_i(x_t)$.

$$\mathbf{P}(x_t) = \begin{bmatrix} b_1(x_t) & 0 & 0 & \cdots & 0 \\ 0 & b_2(x_t) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & b_n(x_t) \end{bmatrix}$$

Souvenons-nous que μ est le vecteur de probabilité initiale et que $\mathbf{A}$ est la matrice de transition. En utilisant ces nouvelles notations, nous pouvons simplement observer qu'il est possible d'écrire le vecteur α_1 de la sorte :

$$\begin{aligned} \alpha_1 &= [\alpha_1(1), \alpha_1(2), ..., \alpha_1(n)] \\ &= [P(x_1, q_1 = s_1), P(x_1, q_1 = s_2), ..., P(x_1, q_1 = s_n)] \\ &= [\mu_1 b_1(x_1), \mu_2 b_2(x_1), ..., \mu_n b_n(x_1)] \\ &= \mu \mathbf{P}(x_1). \end{aligned}$$

De plus, nous pouvons en déduire que :

$$\begin{aligned} \alpha_2 &= \left[\sum_i^N \alpha_1(i) a_{i1} b_1(x_2), \sum_i^N \alpha_1(i) a_{i2} b_2(x_2), ..., \sum_i^N \alpha_1(i) a_{in} b_n(x_2) \right] \\ &= [\alpha_1 \mathbf{A}[,1] b_1(x_2), \alpha_1 \mathbf{A}[,2] b_2(x_2), ..., \alpha_1 \mathbf{A}[,2] b_n(x_2)] \\ &= [\mu \mathbf{P}(x_1) \mathbf{A}[,1] b_1(x_2), \mu \mathbf{P}(x_1) \mathbf{A}[,2] b_2(x_2), ..., \mu \mathbf{P}(x_1) \mathbf{A}[,n] b_n(x_2)] \\ &= \mu \mathbf{P}(x_1) \mathbf{A} \mathbf{P}(x_2). \end{aligned}$$

De manière similaire, nous constatons que : $\alpha_t = \mu \mathbf{P}(x_1) \mathbf{A} \mathbf{P}(x_2) \mathbf{A} \mathbf{P}(x_3) \cdots \mathbf{A} \mathbf{P}(x_t)$. Prenons finalement note que nous pouvons multiplier les vecteurs α par un vecteur colonne de 1 pour en faire la somme. Par exemple, $P[X_1 = x_1, X_0 = x_0, ...X_t = x_t] = \sum_i \alpha_t(i) = \mu \mathbf{P}(x_1) \mathbf{A} \mathbf{P}(x_2) \mathbf{A} \mathbf{P}(x_3) \cdots \mathbf{A} \mathbf{P}(x_t) 1'$.

2.4.1.2 Algorithme backward

Discutons maintenant de la partie *backward* de l'algorithme. Dans le contexte de l'algorithme *forward*, nous utilisions la totalité des observations passées pour faire de l'inférence sur le présent. Dans l'algorithme *backward*, ce sont les observations futures qui sont utilisées. Nous travaillerons à calculer les valeurs $\beta_t(i) = P[X_{t+1} = x_{t+1}, X_{t+2} = x_{t+2}, ..., X_T = x_T | q_t = s_i]$ pour $i = 1, ..., n$ et $t = 1, ..., T$ avec un développement similaire que celui utilisé dans le calcul des α :

$$\begin{aligned}
\beta_t(i) &= P(x_{t+1}, x_{t+2}, ..., x_T | q_t = s_i) \\
&= \sum_{j=1}^{n} P(x_{t+1}, x_{t+2}, ..., x_T \cap q_{t+1} = s_j | q_t = s_i) \\
&= \sum_{j=1}^{n} P(x_{t+1}, x_{t+2}, ..., x_T | q_t = s_i \cap q_{t+1} = s_j) P(q_{t+1} = s_j | q_t = s_i) \\
&= \sum_{j=1}^{n} P(x_{t+1}, x_{t+2}, ..., x_T | q_{t+1} = s_j) P(q_{t+1} = s_j | q_t = s_i) \\
&= \sum_{j=1}^{n} P(x_{t+2}, ..., x_T | q_{t+1} = s_j) P(x_{t+1} | q_{t+1} = s_j) P(q_{t+1} = s_j | q_t = s_i) \\
&= \sum_{j=1}^{n} \beta_{t+1}(j) b_j(x_{t+1}) a_{ij} \\
&= \sum_{j=1}^{n} a_{ij} b_j(x_{t+1}) \beta_{t+1}(j).
\end{aligned} \tag{2.5}$$

Ici, la valeur initiale de l'algorithme récursif est $\beta_T(i) = 1 \ \forall i$. De plus, un développement matriciel similaire pourrait être fait et nous permettrait d'obtenir que $\beta_t = \mathbf{AP}(x_{t+1})\mathbf{AP}(x_{t+2}) \cdots \mathbf{AP}(x_T)$. Finalement, nous dénoterons V_t, la vraisemblance de nos observations. Notons que :

$$\begin{aligned} V_T &= P(X_1 = x_1, X_2 = x_2, ...X_T = x_T) \\ &= (P[x_1, x_2, ...x_t, q_t])(P[x_{t+1}, x_{t+2}, ..., x_T|q_t]) \\ &= \alpha_t \beta'_t \ \forall t. \end{aligned} \tag{2.6}$$

Nous avons donc plusieurs estimations pour la vraisemblance.

À l'aide de l'algorithme *forward* et de l'algorithme *backward*, nous pourrons désormais utiliser les observations passées et futures, et la totalité de l'information pour faire de l'inférence pour tout moment t sur les états sous-jacents. Dans la prochaine section portant sur l'algorithme de *Baum-Welch*, nous verrons comment utiliser ceux-ci en collaboration pour formuler diverses estimations.

2.4.2 Algorithme de *Baum-Welch*

L'algorithme de *Baum-Welch* est la pièce maîtresse de l'inférence sur les modèles markoviens cachés. Il s'agit d'un algorithme EM (*Expectation–maximization*) adapté aux chaînes de Markov cachées. Il s'agit donc d'un algorithme récursif, dans lequel les paramètres sont mis à jour à chaque itération, et qui converge vers les bonnes valeurs à estimer, si nos valeurs initiales sont bien choisies. L'algorithme *forward-backward* fait partie intégrante de cet algorithme. Observons d'abord deux propriétés. Débutons par développer la probabilité d'un état conditionnellement aux observations :

$$\begin{aligned}
\gamma_t(i) &= P[q_t = S_i | x_1, x_2, ..., x_T] \\
&= P[q_t = S_i | x_1, x_2, ..., x_t, x_{t+1}, ..., x_T] \\
&= \frac{P[x_1, x_2, ..., x_t, x_{t+1}, ..., x_T, q_t = s_i]}{P[x_1, x_2, ..., x_T]} \\
&= \frac{P[x_1, x_2, ..., x_t, x_{t+1}, ..., x_T | q_t = s_i] P[q_t = s_i]}{P[x_1, x_2, ..., x_T]} \\
&= \frac{P[x_1, x_2, ..., x_t | q_t = s_i] P[q_t = s_i] P[x_{t+1}, x_{t+2}, ..., x_T | q_t = s_i]}{P[x_1, x_2, ..., x_T]} \\
&= \frac{P[x_1, x_2, ..., x_t, q_t = s_i] P[x_{t+1}, x_{t+2}, ..., x_T | q_t = s_i]}{P[x_1, x_2, ..., x_T]} \\
&= \alpha_t(i)\beta_t(i)/V_T.
\end{aligned} \tag{2.7}$$

Nous aurons ensuite besoin de calculer les probabilités de transition conditionnelles aux observations :

$$\begin{aligned}
\xi_t(i,j) &= P[q_{t-1} = s_i, q_t = s_j | x_1, ..., x_T] \\
&= \frac{P[x_1, ..., x_T, q_{t-1} = s_i, q_t = s_j]}{P[x_1, ..., x_T]} \\
&= \frac{P[x_1, ..., x_{t-1}, q_{t-1} = s_i] P[q_t = s_j | q_{t-1} = s_i] P[x_t, x_{t+1}, ..., x_T | q_t = s_j]}{V_T} \\
&= \frac{P[x_1, ..., x_{t-1}, q_{t-1} = s_i] P[q_t = s_j | q_{t-1} = s_i] P[x_t | q_t = s_j] P[x_{t+1}, ..., x_T | q_t = s_j]}{V_T} \\
&= \alpha_{t-1}(i) a_{i,j} b_j(x_t) \beta_t(j)/V_T.
\end{aligned} \tag{2.8}$$

Avec ces deux outils en main, nous pourrons expliquer comment fonctionne l'algorithme de *Baum-Welch*. Rappelons qu'il s'agit d'un algorithme *EM*, nous utiliserons donc des techniques peut-être connues. Nous allons d'abord obtenir une vraisemblance, avec des données connues et des données manquantes. L'étape

Expectation consiste à calculer l'espérance de la vraisemblance en utilisant l'espérance des variables manquantes, ici les états, conditionnellement aux variables connues, les observations. L'étape *Maximisation* consistera ensuite à maximiser le tout. Puis, nous mettrons les paramètres du modèle à jour avec ce que nous avons calculé à l'étape précédente et nous recommencerons à partir de l'étape *E*. Nous espérons ici converger vers les véritables valeurs des variables manquantes. Calculons la vraisemblance des observations et des états. Pour procéder, nous aurons besoin de deux variables : u et w. Tout d'abord, $u_i(t) = 1$ si et seulement si $q_t = s_i$ et $u_i(t) = 0$ sinon. De plus, $w_{i,j}(t) = 1$ si et seulement si $q_{t-1} = s_i$ et $q_t = s_j$, $w_{i,j}(t) = 0$ sinon. Calculons la vraisemblance à l'aide de ces deux valeurs :

$$\begin{aligned}
V(x,q) = P[x_1, x_2, ..., x_T, q_1, q_2, ...q_T] &= \mu(q_1) b_{q_1}(x_1) a_{q_1,q_2} b_{q_2}(x_2) ... b_{q_T}(x_T) \\
&= \mu(q_1) \prod_{t=2}^{T} a_{q_{t-1},q_t} \prod_{t=1}^{T} b_t(x_t) \\
\Rightarrow \log(P[x_1, x_2, ..., x_T, q_1, q_2, ...q_T]) &= \log(\mu(q_1)) + \sum_{t=2}^{T} \log(a_{q_{t-1},q_t}) + \sum_{t=1}^{T} \log(b_{q_t}(x_t)) \\
&= \sum_{j=1}^{n} u_j(1) \log(\mu(q_1)) \\
&+ \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{t=2}^{T} w_{i,j}(t) \log(a_{q_{t-1},q_t}) \\
&+ \sum_{j=1}^{n} \sum_{t=1}^{T} u_j(t) \log(b_{q_t}(x_t)).
\end{aligned} \tag{2.9}$$

Remplaçons maintenant ces valeurs par leurs valeurs attendues. Souvenons-nous que u et w sont des variables égales à 1 lorsqu'un évènement se réalise, sinon ces dernières sont égales à 0. Dans ce cas, l'espérance est la probabilité de l'évènement parce qu'il s'agit en fait d'une variable de Bernoulli. Dans notre cas :

$$\hat{u}_i(t) = P[q_t = s_i|x_1, x_2, ..., x_T] = \gamma_t(i) = \alpha_t(i)\beta_t(i)/V_T$$
$$\hat{w}_{i,j}(t) = P[q_{t-1} = s_i, q_t = s_j|x_1, x_2, ..., x_T] = \xi_t(i,j) = \alpha_{t-1}(i)a_{i,j}b_j(x_t)\beta_t(j)/V_T.$$

Nous sommes finalement rendus à l'étape de la maximisation. Nous observons que chaque terme de la log-vraisemblance en (2.9) peut être maximisé de manière disjointe. En maximisant chacun des termes, nous obtenons :

$$\begin{aligned}
\hat{\mu}(i) &= \gamma_1(i), \\
\hat{a}_{i,j} &= \frac{\sum_{t=2}^{T} \xi_t(i,j)}{\sum_{j}^{n} \sum_{t=2}^{T} \xi_t(i,j)} \\
&= \frac{\sum_{t=2}^{T} \xi_t(i,j)}{\sum_{t=2}^{T} \gamma_t(i)} \\
&= \frac{\sum_{t=2}^{T} P[q_{t-1} = s_i, q_t = s_j|x_1, x_2, ..., x_T]}{\sum_{t=2}^{T} P[q_t = s_i|x_1, x_2, ..., x_T]}.
\end{aligned} \tag{2.10}$$

Notons que l'estimation de la matrice de transition dans le cas d'un modèle de Markov caché se rapproche beaucoup de l'estimation dans un contexte de modèle de Markov standard, mais bien sûr avec une pondération en probabilité causée par l'incertitude ajoutée du fait que nous ne connaissons pas réellement l'état visité.

La troisième estimation varie selon la fonction d'observation, dans un contexte où celle-ci est de Poisson nous obtenons :$\hat{\lambda}_i = \sum_{t=1}^{T} \gamma_t(i)x_i / \sum_{t=1}^{T} \gamma_t(i)$.

Dans notre situation précise, la fonction d'observation sera particulière, comme nous le verrons plus tard, mais notons que celle-ci ne sera pas réestimée. Elle sera entièrement connue et ne nécessitera pas de mise à jour.

Finalement, nous allons utiliser ces nouvelles valeurs pour calculer les α et les β à nouveau avec l'algorithme *forward-backward* et les étapes énoncées précédemment seront répétées. Ainsi, voici l'algorithme étape par étape :

1. Soumettre les valeurs initiales des paramètres du modèle à l'algorithme.

2. Calculer les α's (équation 2.2)et les β's (équation 2.5) à l'aide des paramètres que nous possédons.

3. Calculer les valeurs attendues tel que détaillé dans les équations 2.7 et 2.8.

4. Estimer les paramètres du modèle tel que décrit dans l'équation 2.10.

5. Utiliser ces paramètres et recommencer au point # 2 jusqu'à ce que l'estimation des paramètres soit suffisamment stable.

2.4.3 Inférence sur le nombre d'états

Nous compléterons ici la section sur les modèles de Markov cachés. Lorsque nous sommes face à une série d'observations provenant d'un modèle de Markov caché, nous nous devons d'avoir des méthodologies nous permettant d'estimer le nombre d'états de la chaîne de Markov sous-jacente.

Le nombre d'états sous-jacents est intimement relié aux nombres de paramètres à estimer. C'est pourquoi nous utiliserons certains outils couramment utilisés en régression, notamment pour vérifier la qualité d'un modèle. Des outils se servant de la vraisemblance et la log-vraisemblance pourront s'appliquer ici. À chaque fois que nous rajouterons un paramètre au modèle, nous augmenterons la vraisemblance. Certaines mesures ont été créées pour considérer une punition lorsque nous ajoutons des paramètres inutiles. Ces outils ont pour objectif de réellement comparer des vraisemblances en considérant qu'il y a un prix à augmenter le

nombre de paramètre, ici des états. Nous visons le modèle qui obtient la meilleure vraisemblance par rapport au nombre de variables qu'il nécessite.

Rappelons que si la vraisemblance L augmente, la valeur $-2log(L)$ diminuera. Nous verrons deux statistiques, tout d'abord, le *Akaike Information Criterion* (AIC) :

$$AIC = -2\log(L) + 2p,$$

où p représente le nombre de paramètres dans le modèle. Nous voulons ici utiliser comme nombre d'états celui qui produit l'AIC le plus petit. Une autre statistique couramment employée en régression est le *Bayesian Information Criterion* (BIC) :

$$BIC = -2\log(L) + p\log(m).$$

Dans ce cas, m représente le nombre d'observations que nous possédons dans notre échantillon. Nous voulons encourager les bonnes vraisemblances qui ne nécessitent pas trop d'observations, pour deux vraisemblances similaires, l'une sera priorisée si celle-ci requiert un échantillon plus petit que l'autre. Bien que ces deux statistiques se ressemblent, l'une incorpore la taille d'échantillon. De plus, le BIC favorise un nombre de paramètres souvent inférieur au AIC.

Bien qu'il ne s'agisse que d'une petite parcelle de toute la théorie en ce qui concerne l'inférence sur les modèles de Markov cachés, nous allons nous contenter de cette théorie dans ce mémoire.

CHAPITRE III

MODÈLES DE SEMI-MARKOV CACHÉS

3.1 Présentation générale

Dans un modèle de semi-Markov caché, la chaîne sous-jacente est semi-Markovienne. Cela signifie que le processus demeure un certain nombre d'unités de temps aléatoire à chaque état qu'il visite. Par le fait même, chaque état sous-jacent visité engendre une série d'observations et non une seule. Il y aura ainsi une fonction de durée, qui est une loi aléatoire, discrète dans notre cas, qui détermine la durée de temps passée à chaque état. De la sorte, le défi ici sera de bien utiliser ces observations additionnelles pour évaluer nos paramètres cachés. Par contre, nous ajouterons aussi de nombreuses estimations à notre modèle puisque nous désirons maintenant estimer cette fonction de durée pour chacun des états sous-jacents.

3.1.1 Processus semi-markoviens

Pour débuter, introduisons brièvement les processus de semi-Markov. Une introduction plus complète à ce processus est présentée dans le livre de Ross (Ross, 2009). Nous pouvons imaginer une chaîne de Markov tel que décrit au chapitre 2 avec n états, par exemple. Dans le cas d'un processus markovien classique, au temps t, le processus serait à l'un des n états de la chaîne, s_i par exemple, et au temps $t+1$, précisément une transition aurait eu lieu et le processus serait

maintenant à l'état s_j avec probabilité $a_{i,j}$. Dans un modèle semi-markovien, le processus effectue un nombre de visites aléatoires à chaque état entre les transitions. Ce dernier demeure ainsi à l'état s_i du temps t au temps $t + l$, où l est une variable aléatoire. Par la suite, une transition aura lieu selon les probabilités contenues dans la matrice $\mathbf{A}$, un peu comme cela aurait été le cas dans le modèle de Markov classique. Il est important de noter que la variable qui décrit cet intervalle de temps aléatoire entre les transitions peut être discrète ou continue. Dans notre contexte, nous allons considérer que cette variable est discrète. Bien entendu, dans un processus markovien standard, nous demeurons un temps aléatoire à un état, temps qui suit une loi géométrique. Les modèles de semi-Markov sont, de cette manière, une généralisation des modèles de Markov classiques permettant l'utilisation d'autres lois de durées que la loi géométrique.

Compte tenu du fait que la variable markovienne prend une certaine valeur pour un intervalle de temps, il est courant de considérer que la probabilité de retour $a_{i,i}$ est nulle. En conséquence, nous avons une matrice de transition $\mathbf{A}$ où la diagonale principale ne contient que des zéros. De la sorte, après avoir passé l unité de temps à un certain état, le processus changera d'état.

3.2 Définition formelle

Dans cette section, nous définirons plus formellement les diverses composantes et les notations utilisées. Il est extrêmement difficile de trouver ou de bâtir une notation logique et cohérente avec la totalité des articles traitant le sujet. Celle utilisée ici se rapproche grandement de celle utilisée par Murphy (Murphy, 2002), mais est légèrement modifiée par souci de cohérence avec ce que nous avons utilisé précédemment. Définissons d'abord G_t comme étant ce que nous appellerons l'état généralisé, au temps t. Cette variable est en fait un couple qui comprend l'état dans lequel est le processus ainsi que le nombre d'unités de temps qui y aura passé. Une

observation de cette variable, g_t, est donc un couple qui comprend la valeur de la variable de Markov obtenue au temps t ; q_t, ainsi que la durée de temps passée dans cet état ; l_t. Souvenons-nous que si le processus demeure l_t unités de temps à l'état s_i, par exemple, nous obtiendrons une séquence d'observations, de longueur l_t en provenance de l'état s_i. Posons $Y(g_t)$, la séquence d'observations émise par l'état généralisé g_t. $Y(g_t^+)$ représentera toutes les observations qui suivront $Y(g_t)$, et $Y(g_t^-)$ représentera la totalité des observations qui ont précédé $Y(g_t)$. Finalement, définissons g_{t_p} comme l'état précédant g_t et g_{t_n} sera l'état généralisé suivant g_t. Dans les modèles de Markov cachés standards, nous avions défini $b_i(x_t) = P[X_t = x_t|q_t = s_i]$, son équivalent dans un modèle de semi-Markov caché est le suivant $b_t(s_i, l_t) = P[Y(g_t)|q_t = s_i, L_t = l_t]$. La figure 3.1 nous aide à visualiser comment fonctionne ce processus.

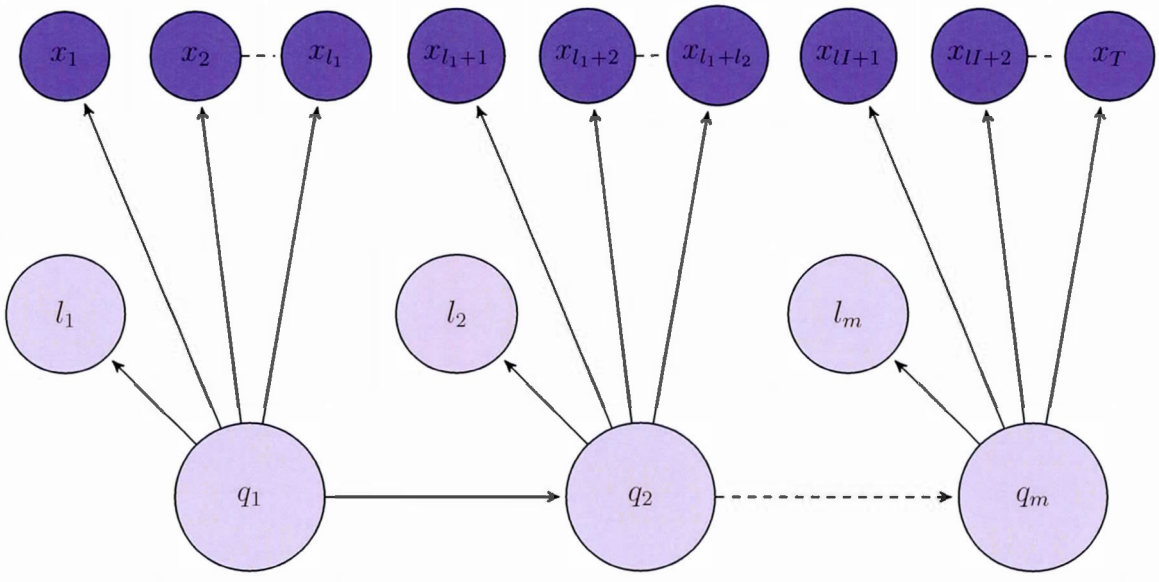

Figure 3.1 Illustration d'une séquence d'observations d'une chaîne de semi-Markov cachée. Ici, m variables markoviennes furent réalisées et T observations furent obtenues. De plus, $lI = \sum_1^{m-1} li$. Les éléments pâles ne sont pas observés.

3.3 Inférence

Voyons ici comment le fait de travailler avec un modèle sous-jacent semi-markovien affecte les algorithmes définis au chapitre 2. Il serait important de pouvoir trouver une équivalence à ceux-ci dans l'objectif de pouvoir faire de l'inférence.

3.3.1 Algorithme *forward*

Souvenons-nous que pour un modèle de Markov caché, α fut défini de la manière suivante :

$$\alpha_t(i) = P(x_1, x_2, ...x_t \text{ et } q_t = s_i).$$

Pour un modèle de semi-Markov caché, nous définirons l'équivalent :

$$\alpha_t(g) = P(Y(g_t^-), Y(g_t) \text{ et } G_t = g).$$

Tentons d'en dégager une forme récursive, comme nous l'avons fait en 2.4.1.

$$
\begin{aligned}
\alpha_t(g) &= P(Y(g_t^-), Y(g_t), G_t = g) \\
&= \sum_{g'} P(Y(g_t^-), Y(g_t), G_t = g, G_{t_p} = g') \\
&= \sum_{g'} P(Y(g_t)|G_t = g, G_{t_p} = g', Y(G_t^-)) P(G_t = g, G_{t_p} = g', Y(g_t^-)) \\
&= \sum_{g'} P(Y(g_t)|G_t = g) P(G_t = g|G_{t_p} = g', Y(G_t^-)) P(G_{t_p} = g', Y(g_t^-)) \\
&= \sum_{g'} P(Y(g_t)|G_t = g) P(G_t = g|G_{t_p} = g') P(G_{t_p} = g', Y(g_t^-)) \\
&= b_t(g) \sum_{g'} P(g|g') \alpha_{t_p}(g').
\end{aligned}
\tag{3.1}
$$

Nous observons une forme très similaire à ce que nous avions obtenu pour un modèle de Markov caché soit $\alpha_{t+1}(i) = b_j(x_{t+1}) \sum_i^N a_{ij} \alpha_t(i)$. Par contre, il peut être ardu de travailler avec G_t puisqu'il s'agit de couples de variables. Pour mieux répondre à nos besoins, nous allons définir un nouveau α de manière à y observer q_t, l'état de la chaîne, et l_t, la longueur du segment de temps durant lequel le processus demeure dans cet état de manière disjointe. Murphy (Murphy, 2002) introduit une nouvelle notation qui ne se retrouve pas dans d'autres articles portant sur le même sujet. Il s'agit de F_t défini comme étant une variable booléenne qui nous indique où en est le processus par rapport à l_t. Formellement, nous poserons $F_t = 1$ si le processus en est à son dernier moment avant une transition, donc si $q_t \neq q_{t+1}$, $F_t = 0$ sinon. Définissons désormais $\alpha_t(q, l)$ comme :

$$
\alpha_t(s_i, l) = P(q_t = s_i, L_t = l, F_t = 1, x_{1:t}).
$$

Dans ce qui suit, nous travaillons alors au dernier instant avant un changement

d'état, puisque $F_t = 1$. De plus, souvenons-nous que $x_{1:t}$ représente les t premières observations, précédemment identifiées comme $Y(g_t), Y(g_t^-)$. Nous pouvons donc utiliser le $\alpha_t(g)$ que nous venons de calculer en (3.1), en gardant en tête que nous venons d'écouler les l_t unités de temps à l'état q_t :

$$\begin{aligned}\alpha_t(s_i, l_t) &= P(q_t = s_i, L_t = l_t, F_t = 1, x_{1:t}) \\ &= P(x_{t-l_t+1:t}|s_i, l_t) \sum_{j \neq i} \sum_{l_{t_p}} P(s_i, l|s_j, l_{t_p}) \alpha_{t_p}(s_j, l_{t_p}).\end{aligned} \tag{3.2}$$

Nous allons finalement énoncer une dernière hypothèse. Supposons que nous travaillons avec un modèle de Markov caché à durée explicite, un cas particulier de semi-Markov caché. Dans ce modèle, le temps demeuré dans un état ne dépend que de l'état lui-même, et la probabilité de faire une transition vers un état ne dépend que de l'état où le processus était précédemment. Cela implique que : $P(g_t|g_{t_p}) = P(s_i, l|s_j, l_{t_p}) = P(s_i|s_j)P(l|s_i)$. Ceci suppose en effet que la probabilité de transition vers un état ne dépend que de l'état dans lequel le processus se situe, et non de la durée de temps qu'il y a passé. Cela suppose également que le temps que ce dernier reste à un état ne dépend que de cet état, et non du passé. Cette hypothèse est acceptable dans les conditions de notre problème. Néanmoins, notons que le prochain développement ne fonctionne pas pour tous les modèles de semi-Markov cachés. Examinons comment cette hypothèse influence les α :

$$\begin{aligned}
\alpha_t(s_i, l_t) &= P(x_{t-l_t+1:t}|s_i, l_t) \sum_{j \neq i} \sum_{l_{t_p}} P(s_i, l|s_j, l_{t_p}) \alpha_{t_p}(s_j, l_{t_p}) \\
&= P(x_{t-l_t+1:t}|s_i, l_t) \sum_{j \neq i} \sum_{l_{t_p}} P(s_i|s_j) P(l|s_i) \alpha_{t_p}(s_j, l_{t_p}) \\
&= P(x_{t-l_t+1:t}|s_i, l_t) P(l|s_i) \sum_{j \neq i} P(s_i|s_j) \sum_{l_{t_p}} \alpha_{t_p}(s_j, l_{t_p}).
\end{aligned} \tag{3.3}$$

Nous définirons finalement une dernière version de α qui ne tient pas compte de l_t, nous allons donc sommer sur cette variable :

$$\begin{aligned}
\alpha_t(s_i) &= P(q_t = s_i, F_t = 1, x_{1:t}) \\
&= \sum_l \alpha_t(q_t, l_t) \\
&= \sum_l P(x_{t-l_t+1:t}|s_i, l_t) P(l|s_i) \sum_{j \neq i} P(s_i|s_j) \sum_{l_{t_p}} \alpha_{t_p}(s_j, l_{t_p}) \\
&= \sum_l P(x_{t-l_t+1:t}|s_i, l_t) P(l|s_i) \sum_{j \neq i} P(s_i|s_j) \alpha_{t_p}(s_j).
\end{aligned} \tag{3.4}$$

C'est cette version qui sera utilisée afin de faire de l'inférence.

3.3.2 Algorithme *backward*

Nous verrons brièvement comment modifier l'algorithme *backward* pour l'adapter à un modèle de semi-Markov. La section sera peu détaillée puisqu'il s'agit d'étapes similaires à ce que nous venons de faire en 3.3.1. Premièrement, nous utiliserons une définition équivalente à celle établie en 2.4.1.2, soit :

$$\begin{aligned}
\beta_t(g) &= P(Y(g_t^+)|G_t = g) \\
&= \sum_{g'} P(Y(g_t^+), G_{t_n} = g'|G_t = g) \\
&= \sum_{g'} P(Y(g_t^+)|G_{t_n} = g', G_t = g) P(G_{t_n} = g'|G_t = g) \\
&= \sum_{g'} P(Y(g_{t_n}), Y(g_{t_n}^+)|G_{t_n} = g', G_t = g) P(g'|g) \\
&= \sum_{g'} P(Y(g_{t_n}^+)|G_{t_n} = g') P(Y(g_{t_n})|G_{t_n} = g') P(g'|g) \\
&= \sum_{g'} b_{t_n}(g') \beta_{t_n}(g') P(g'|g).
\end{aligned} \tag{3.5}$$

Ce qui est, encore une fois, l'équivalent de ce que nous avions calculé précédemment, dans un cas de Markov caché. Tout comme au sujet des α, nous allons redéfinir successivement des β avec l'objectif de pouvoir obtenir un β défini uniquement en fonction de l'état sous-jacent, et non de l'état généralisé. Commençons par observer les deux composantes de l'état généralisé :

$$\begin{aligned}
\beta_t(s_i l_t) &= P(x_{t+1:T}|q_t = s_i, L_t = l_t, F_t = 1) \\
&= \sum_{j \neq i} \sum_{l_{t_n}} P(x_{t+1:t+l_{t_n}}|q_{t_n}, L_{t_n}) \beta_{t_n}(q_{t_n}, l_{t_n}) P(q_{t_n} = s_j, L_{t_n} = l_{t_n}|q_t = s_i, L_t = l_t).
\end{aligned} \tag{3.6}$$

En supposant à nouveau que nous sommes face à un modèle à durée explicite, nous pouvons simplifier cet algorithme de la sorte :

$$\begin{aligned}
\beta_t(q_t, l_t) &= P(x_{t+1:T}|q_t = s_i, L_t = l_t, F_t = 1) \\
&= \sum_{j \neq i} \sum_{l_{t_n}} P(x_{t+1:t+l_{t_n}}|q_{t_n}, L_{t_n}) \beta_{t_n}(q_{t_n}, l_{t_n}) P(l_{t_n}|s_j) P(s_j|s_i).
\end{aligned} \tag{3.7}$$

Les β, étant indépendants de l_t, peuvent être réécrits comme :

$$\beta_t(s_i) = \sum_{j \neq i} P(s_j|s_i) \sum_{l_{t_n}} P(x_{t+1:t+l_{t_n}}|q_{t_n}, L_{t_n}) \beta_{t_n}(s_j) P(l_{t_n}|s_j). \tag{3.8}$$

En somme, nous obtenons un $\beta_t(q_t)$ qui nous indique la probabilité d'une séquence d'observations futures sachant que le processus sort de l'état $q_t = s_i$ au temps t. Par contre, Murphy note qu'il serait plus intéressant de connaître la probabilité d'une séquence d'observations futures sachant que le processus débute son séjour à l'état s_j au temps $t+1$. Nous dénoterons cette valeur comme $\beta_t^*(s_j)$. Donc :

$$\begin{aligned} \beta_t(s_i) =& P(x_{t+1:T}|q_t = s_i, L_t = l_t, F_t = 1) \\ &= \sum_{j \neq i} P(s_j|s_i) \sum_{l_{t_n}} P(x_{t+1:t+l_{t_n}}|q_{t_n}, L_{t_n}) \beta_{t_n}(s_j) P(l_{t_n}|s_j) \\ &= \sum_{j \neq i} P(s_j|s_i) \beta_t^*(s_j), \end{aligned} \tag{3.9}$$

où :

$$\begin{aligned} \beta_t^*(s_j) &= P(x_{t+1:T}|q_{t+1} = s_j, F_t = 1) \\ &= \sum_{l_{t_n}} P(x_{t+1:t+l_{t_n}}|q_{t_n}, L_{t_n}) \beta_{t_n}(s_j) P(l_{t_n}|s_j). \end{aligned} \tag{3.10}$$

3.3.3 Algorithme de *Baum-Welch*

Nous tenterons ici d'utiliser nos connaissances sur les modèles de Markov cachés pour faire l'inférence des modèles de semi-Markov cachés. Au chapitre 3, nous

avons développé l'algorithme de *Baum-Welch*, une version adaptée au modèle de Markov caché de l'algorithme *EM*. Dans cette section, nous expliquerons brièvement le développement des outils d'inférence qui sont maintenant bien plus complexes.

Comme les α et les β utilisent la matrice de transition, la loi d'observation ainsi que la loi de durée, nous devrions estimer ces dernières par leurs maximums de vraisemblance dans un algorithme *EM*. Bien que cela soit faisable, nous n'irons pas aussi loin dans le contexte du problème qui nous préoccupe. Comme nous n'aurons que la matrice sous-jacente à estimer, nous utiliserons l'estimateur de vraisemblance de celle-ci en nous servant de ce que nous avions obtenu au chapitre 2 :

$$\hat{a}_{i,j} = \frac{\sum_{t=2}^{T} \alpha_t(i) a_{i,j} \beta_t^*(j)}{\sum_{j}^{n} \sum_{t=1}^{T} \alpha_t(i) a_{i,j} \beta_t^*(j)}. \tag{3.11}$$

Ce qui est en fait le même estimateur que celui qui serait utilisé pour un modèle qui n'est pas semi-markovien. De manière logique, il s'agit simplement du nombre de fois où nous passons de s_i vers s_j divisé par le nombre de fois où nous partons de s_i, pondéré par la probabilité de ces évènements. Bien entendu, nous utiliserons la matrice de transition, la loi d'observation ainsi que la loi de durée dans le calcul de l'algorithme *foward-backward*, ce qui pourrait être problématique dans de nombreux contextes. Par contre, dans le contexte de notre problème, nous n'aurons pas à estimer la fonction de masquage ni la distribution initiale. De plus, la valeur initiale de la matrice de transition sera tout de même précise. Les paramètres initiaux que nous utiliserons permettront donc d'assurer un minimum d'efficacité à l'algorithme.

CHAPITRE IV

ESTIMATION DE LA POSITION D'UN AGENT MOBILE PAR UNE INTELLIGENCE ARTIFICIELLE

Dans ce chapitre, nous détaillerons la partie centrale de ce mémoire. Bien que la recherche en intelligence artificielle avance d'un bon pas, le développement de celle-ci dans le contexte des jeux vidéo semble plutôt lent. À ce propos, il faut établir qu'il s'agit de la puissance des ordinateurs modernes qui permet de croire que nous pourrons utiliser, un jour, de puissants algorithmes mathématiques à l'intérieur de jeux accessibles à tous. Dans ce travail, nous irons donc légèrement dans le sens contraire de ce qui est pratique courante en recherche puisque nous tenterons d'abord de résoudre une problématique fortement associée aux jeux vidéo, puis de la généraliser pour d'autres fins. Nous utiliserons les contraintes de ce contexte, mais il est à noter que ces résultats pourraient servir pour le développement d'intelligence artificielle avec apprentissage automatique (*machine learning*) dans plusieurs autres environnements. Pour résoudre la problématique d'estimation de la position d'un agent mobile dans un environnement borné, deux outils mathématiques sont utilisés. Entre autres, on retrouve les filtres particulaires, dont nous parlerons très brièvement, et les modèles de semi-Markov cachés, notre principale méthodologie d'intérêt. Dans tous les cas, nous chercherons à estimer une probabilité conditionnelle, soit la probabilité que l'agent soit à une certaine position, connaissant certaines observations diverses.

4.1 Filtres particulaires

4.1.1 Introduction

Nous décrirons brièvement ce que sont les filtres particulaires puisqu'il s'agit de la première méthodologie développée afin d'estimer la position d'un agent mobile dans un environnement borné. Bererton (Bererton, 2004) utilisa pour la première fois cet outil dans l'objectif d'améliorer l'intelligence artificielle dans les jeux vidéo. Il s'agit de l'un des premiers textes qui tente de résoudre le problème d'estimation de la position d'un adversaire dans un jeu. Selon l'auteur, il était auparavant impossible de concevoir de telles intelligences puisque la quasi-totalité du processeur de l'ordinateur de l'utilisateur d'un jeu travaillait exclusivement à faire fonctionner ce même jeu. Considérant que certaines limitations sont similaires, Bereton s'inspire de travaux réalisés en robotique pour travailler les intelligences des jeux vidéo. Il mentionne que, dans les deux cas, des décisions doivent être rapidement prises et ce, dans un environnement qui change en temps réel. Bererton fait part du fait que les algorithmes doivent être efficaces puisqu'en robotique, tout comme dans les jeux vidéo, beaucoup d'énergie est dépensée dans l'analyse des observations et dans le fonctionnement moteur. Sachant cela, il décide d'utiliser les filtres particulaires. Arulampalam (Arulampalam *et al.*, 2002) définit les filtres particulaires comme une méthode de Monte Carlo séquentielle basée sur l'utilisation de point de masse, appelé particule. Cette méthode est une généralisation des filtres de Kalman.

Mettons-nous dans la situation où nous désirons résoudre un problème d'estimation a posteriori à l'aide de la formule de Bayes. Par exemple, si nous désirons estimer $P(x|y)$, où X et Y sont deux variables aléatoires, nous pourrions utiliser le développement suivant : $P(x|y) = \frac{P(y|x)P(x)}{P(y)}$. Par contre, si la marginale de Y est inconnue, nous pourrions nous servir du développement suivant :

$$\frac{P(y|x)P(x)}{P(y)} = \frac{P(y|x)P(x)}{\int P(y|x)P(x)dx}. \tag{4.1}$$

Bien évidemment, cette intégrale ne sera pas nécessairement facile à résoudre. Ce problème fut, pendant de nombreuses années, un frein au développement des estimateurs de Bayes. Depuis quelques années, la simulation Monte Carlo a rendu possible la résolution de ces intégrales. De nombreuses variantes de cet outil ont vu le jour dont la méthode de Monte Carlo séquentielle, aussi appelée filtres particulaires.

4.1.2 Formalisme

De nombreuses excellentes sources sont disponibles au sujet de la simulation par chaînes de Markov Monte Carlo ; de plus, il ne s'agit pas de l'outil que nous préconiserons. Pour ces raisons, nous ne décrirons pas en détails la théorie sur la simulation par Monte Carlo, mais nous ferons malgré tout une introduction suffisament rigoureuse. Dans le but de résoudre la problématique quant à l'intégrale énoncée ci-haut, la définition même de l'espérance fut utilisée. Soit la variable aléatoire X de fonction de densité $f(x)$, alors $E_X[g(x)] = \int g(x)f(x)dx$ où g(x) est une fonction de x quelconque. De plus, nous connaissons un excellent estimateur de l'espérance, la moyenne empirique. Nous pourrions ainsi utiliser la moyenne empirique comme estimateur de notre intégrale de la manière suivante $\frac{1}{N}\sum_{i=1}^{N} g(x_i) \approx E_X[g(x)] = \int g(x)f(x)dx$. Les méthodes d'estimation par chaînes de Markov Monte Carlo sont des techniques d'estimation qui misent sur la simulation. L'objectif est donc de simuler plusieurs observations $g(x)$ où $X \sim f(x)$, et d'en faire la moyenne empirique afin d'approximer notre intégrale.

Dans cette situation, il n'y a aucun problème s'il est simple de simuler des valeurs de $f(x)$. Néanmoins, la distribution de X ne sera pas toujours facilement simu-

lable, surtout pour résoudre notre problématique, soit de déduire le dénominateur de l'estimateur de Bayes. Supposons que, dans notre petit exemple précédent, il soit difficile, voire impossible, de simuler des observations de la distribution $f(x)$. Une méthodologie, appelée échantillonnage préférentiel (*importance sampling*), consiste à déterminer une distribution, $h(x)$ par exemple, beaucoup plus simple à simuler. Nous utiliserons cette dernière pour estimer notre intégrale :

$$\int g(x)f(x)dx = \int g(x)f(x)h(x)/h(x)dx = \int \left(g(x)f(x)/h(x)\right)h(x)dx. \quad (4.2)$$

L'intégrale de l'équation 4.2 est l'équivalent de $E[(g(x)f(x)/h(x)]$, valeur que nous estimerons par sa moyenne empirique à l'aide de simulations tirées de $h(x)$.

Ce qu'on appelle une particule est en fait un échantillon de simulation et son poids est dénoté $w(x) = f(x)/h(x)$. Le filtre particulaire, le Monte Carlo séquentiel, est un algorithme récursif qui tente d'utiliser ces poids pour prioriser les particules, c'est-à-dire les échantillons simulés qui ont le plus de valeur, qui estiment le mieux notre intégrale.

4.1.3 Utilisation en intelligence artificielle

4.1.3.1 Implémentation

Cette méthodologie fut la première à être utilisée pour résoudre notre problème d'estimation de position d'un agent. L'objectif est de déterminer la position d'un agent mobile dans un environnement borné. Dans un contexte de jeu vidéo, il est question d'évaluer la position de son adversaire sur le terrain de jeu. Soit X_t, le processus stochastique modélisant la position du joueur adverse. Nous cherchons donc à estimer x_t, la position de cet agent au temps t en ne possédant qu'un ensemble d'observations au temps t, soit z_t. Ainsi :

$$
\begin{aligned}
p(x_t|z_t) &= \frac{p(x_t \cap z_t)}{p(z_t)} \\
&= \frac{p(z_t|x_t)p(x_t)}{p(z_t)} \\
&= \frac{p(z_t|x_t)\int_{x_{t-1}} p(x_t|x_{t-1})p(x_{t-1})dx_{t-1}}{p(z_t)} \\
&= \frac{p(z_t|x_t)\int_{x_{t-1}} p(x_t|x_{t-1})p(x_{t-1}|z_{t_1})p(z_{t-1})dx_{t-1}}{p(z_t)} \\
&= c \times p(z_t|x_t)\int_{x_{t-1}} p(x_t|x_{t-1})p(x_{t-1}|z_{t_1})dx_{t-1},
\end{aligned}
\tag{4.3}
$$

où c est une constante. Cette intégrale très complexe sera estimée à l'aide des méthodes de Monte Carlo séquentielles.

4.1.3.2 Expérience et résultats

Bererton conçoit son propre jeu afin d'effectuer des tests. Il conçoit un jeu tout simple dans un environnement en deux dimensions où deux équipes de robots s'affrontent. L'objectif du jeu est de lancer des briques sur les adversaires afin de les éliminer. L'auteur considère aussi, dans son modèle, que le déplacement des joueurs suit un mouvement brownien. Selon l'auteur, les résultats sont très satisfaisants. Bien entendu, plus on augmente le nombre de particules, c'est-à-dire le nombre de simulations dans notre modèle de Monte Carlo, plus les estimations sont précises. Par contre, plus de temps et de puissance sont nécessaires pour que le processeur réalise cette tâche. On trouve très peu de statistiques démontrant de manière significative que cette intelligence artificielle est performante dans sa tâche. De cette manière, il nous est difficile de témoigner de la réelle efficacité de son travail.

Nous pouvons tout de même en déduire une série d'améliorations potentielles. Il serait nécessaire de débuter en mettant au point une statistique permettant de

déterminer de manière significative l'efficacité des algorithmes utilisés par l'intelligence artificielle. Cette statistique, si elle est bien construite, pourrait nous permettre d'évaluer si les modifications suggérées ont un effet statistiquement significatif. Dans les travaux actuels de Bererton, le nombre d'échantillons de simulations, de particules, fut déterminé arbitrairement, entre 200 et 500. Comme un trop grand nombre de simulations a un coût en temps et en énergie important et qu'un nombre insuffisant de simulations cause des erreurs d'estimation, il serait important de mettre au point un outil qui permet de déterminer le nombre de particules nécessaires pour atteindre la convergence désirée. De nombreux probabilistes se sont penchés sur la question de la convergence des algorithmes de simulation par chaîne de Markov Monte Carlo. Entre autres, Rosenthal (Rosenthal, 1995) nous offre une méthode rigoureuse pour déterminer le nombre d'itérations requis afin d'obtenir une convergence satisfaisante. Plusieurs autres excellents articles traitent du sujet. Il serait réaliste de croire qu'il est possible d'utiliser l'une de ces méthodes pour déterminer le nombre minimal de particules nécessaires.

Il serait finalement intéressant de considérer d'autres distributions a priori que le mouvement brownien. Celle-ci était simple à manipuler et à simuler. Ainsi, elle offrait un ensemble de qualités. Néanmoins, cette distribution est tout sauf réaliste. Nous sommes dans un contexte où les joueurs choisissent de manière logique leurs déplacements afin de ne pas se faire voir ; nous sommes donc loin d'un mouvement brownien, d'une série de déplacements purement aléatoire uniforme. Bien que nous devrions supposer les déplacements aléatoires pour y faire de l'inférence, certaines distributions pourraient être plus appropriées. Une solution serait peut-être de débuter avec une distribution a priori parfaitement uniforme, puis d'estimer cette dernière au fil des parties à l'aide des informations acquises durant chaque joute.

4.2 Modèles de semi-Markov cachés

Nous discuterons ici du cœur de ce mémoire, soit l'estimation de la position d'un agent mobile par une intelligence artificielle en nous servant des modèles de semi-Markov cachés. Nous verrons comment modéliser cette situation par l'outil que nous avons bâti au cours des derniers chapitres. L'article de Southey, Loh et Wilikinson (Southey *et al.*, 2007) est l'un des premiers que l'on puisse trouver qui utilise les modèles de semi-Markov cachés dans une problématique similaire à celle qui nous intéresse. L'objectif des auteurs était d'estimer la position de départ et d'arrivée d'un certain agent mobile dans un environnement borné en n'utilisant que quelques observations en chemin.

Soit un ensemble de t observations $O_{1:t}$ et soit θ la paire de positions que nous voulons estimer ; les auteurs désirent évaluer $P(\theta|o_{1:t})$. À l'aide d'un développement de Bayes, ils obtiennent : $P(\theta|O_{1:t}) \propto P(O_{1:t}|\theta)P(\theta)$. Une distribution a priori devra être judicieusement sélectionnée pour $P(\theta)$, mais qu'en est-il de $P(O_{1:t}|\theta)$? Supposons qu'il existe un chemin optimal pour aller de A vers B et que tout le monde n'utilise que ce chemin. Si nous observons l'agent sur ce chemin, alors $P(O_{1:t}|(A,B)) = 1$, cette valeur sera égale à 0 si l'individu fait un pas à l'extérieur de ce chemin optimal. Bien entendu, cette supposition semble irréaliste. Dans l'objectif de permettre les chemins non optimaux, les auteurs ont employé les modèles de semi-Markov cachés. Cela permettait un chemin plus aléatoire autour de la paire de positions à estimer. Bien que ces travaux ne soient pas directement utilisés pour le développement de notre méthodologie, nous avons jugé pertinent de rapidement discuter de l'article qui fut le premier à utiliser cet outil mathématique dans un contexte similaire à celui sur lequel nous travaillons.

Le développement de la méthodologie que nous ferons dans ce chapitre est inspiré

du travail de S. Hladky (Hladky, 2009). Son mémoire portait sur les *FPS* et nous utiliserons cet environnement comme référence pour la réalisation de nos travaux. Néanmoins, nous construirons une modélisation plus générale qui pourrait être utilisée dans plusieurs types de jeux et même plus, à l'intelligence artificielle au sens large. Plaçons-nous donc dans le contexte d'un jeu vidéo : commençons par discrétiser la carte de jeux, l'espace borné où s'affrontent les joueurs, à l'aide d'une grille en deux dimensions. Ces cases formeront l'espace des états de notre modèle markovien sous-jacent. Toujours en gardant en tête que nous voulons doter notre intelligence artificielle de capacités similaires à celles des êtres humains, il est important que le *bot* utilise les mêmes observations qu'un joueur. Les observations de notre intelligence artificielle seront donc les observations de l'avatar de l'intelligence artificielle, le cône de vision de l'avatar dans le jeu. Ce seront donc ces observations qui seront utilisées pour estimer la position du joueur adverse, qui est inconnue, mais qui influence les observations.

4.2.1 Formalisme

Imaginons d'abord un univers en deux dimensions. La carte de jeu, l'environnement borné, ici en deux dimensions, sera quadrillée et l'ensemble de ces n cases formera l'espace des états du modèle de Markov sous-jacent. L'état de la chaîne au temps t est en fait la case s_i dans laquelle se trouve le joueur adverse. Cette position n'est pas connue de l'intelligence artificielle, mais cette dernière possède des observations, les cases qu'elle observe, dépendantes de la position du joueur adverse. L'idée est d'utiliser l'algorithme *forward* pour estimer à chaque instant l'état de la chaîne sous-jacente, soit la position du joueur adverse. L'algorithme *forward* nous fournira la probabilité que la chaîne soit à l'état s_i au temps t $\forall t, i$ en fonction des observations passées et présentes. L'algorithme *backward* ne nous est pas utile ici puisque celui-ci se base sur des observations futures, ici encore inconnues. Nous définirons comme estimation de la position de l'adversaire l'état

le plus probable selon l'algorithme *forward*. Pour utiliser cet algorithme, plusieurs éléments sont requis, dont la fonction d'observation, la fonction de durée, la matrice de transition sous-jacente ainsi que le vecteur de probabilité initiale.

Tout d'abord, dans le contexte d'un jeu vidéo, *Counter-Strike* par exemple, il n'est pas nécessaire d'estimer le vecteur de probabilité initiale, il est incrémenté dans le jeu et connu de tous les joueurs. Dans ce genre de jeu, il y a un ensemble de positions initiales potentielles, souvent très petit, et l'une de celles-ci est tirée de manière aléatoire uniformément. Après seulement quelques parties, les joueurs humains connaissent ces positions initiales, il n'est donc pas déraisonnable de fournir le vecteur de probabilité initiale à l'intelligence artificielle. Ceci demeure vrai pour la très grande majorité des jeux compétitifs, le vecteur de probabilité initiale est connu de tous les joueurs. Celui-ci sera donc fourni à notre intelligence artificielle pour l'estimation de la position de son adversaire, et ce, sans qu'elle triche.

La fonction d'observation est un défi supplémentaire de construction. Il faut d'abord créer un ensemble que nous noterons W, représentant la totalité des observations perçues par l'avatar de l'intelligence artificielle. Dans un *FPS*, il s'agit d'un cône de vision en trois dimensions, dans un *RTS*, c'est ce qui est contenu dans l'ensemble de tous les cercles lumineux entourant les unités et les bâtiments. Cette partie représente un grand défi de programmation ; prenons pour acquis ici qu'il est simple d'identifier l'ensemble des cases, des états observables, par l'avatar de l'intelligence artificielle. À l'aide de cet ensemble nous pourrons mettre au point notre vecteur d'observation que nous décrirons plus en détail sous peu. Dans le cas d'un jeu en trois dimensions, une grande partie du problème consiste à gérer les observations en trois dimensions pour évaluer les possibilités qu'un agent se trouve sur une carte en deux dimensions. Traitons premièrement le cas d'un uni-

vers en deux dimensions. Nous discuterons du cas en trois dimensions par la suite. Prenez note que nous n'utiliserons pas initialement les modèles de semi-Markov cachés, mais bien les modèles de Markov cachés standards afin de simplifier les explications et la programmation initiale.

Souvenons-nous que l'objectif premier est d'estimer la position d'un agent mobile dans notre espace borné. Cette position, par conséquent, est inconnue. Dans le cas où l'adversaire est observé, tout ce que nous tentons de construire ne sera pas utilisé : l'estimation de la position de l'agent sera simplement sa position exacte telle qu'observée. De cette manière, nous nous situons présentement dans le cas où une grande quantité de positions est observée mais que l'adversaire ne s'y trouve pas, et ce pour la totalité des développements suivants. Ainsi, la probabilité d'observer qu'une certaine case est vide sachant que l'agent s'y trouve est nulle. Avec la même logique, la probabilité d'observer qu'une certaine case est vide sachant que l'agent est dans l'une des autres cases est alors de 100%. Nous utiliserons, pour ce faire, une fonction de masquage binaire toute simple. Voici la loi d'observation mis au point :

$$P[X_t = x_t | q_t = s_i] = 1 - \mathbf{1}[s_i \in W].$$

De la sorte, nous éliminons, $\forall\ t$, la totalité des états observés en t comme étant des positions potentielles du processus au temps t. De plus, comme les α sont construits de manière récursive, cette fonction d'observation permet aussi l'élimination, pour les temps suivants, des chemins qui seraient passés par n'importe quelle $s_i \in W$ au temps t.

Mettons au point un petit exemple. Si, à l'instant t, nous observons un ensemble de cases vides et que parmi ces cases vides se trouve l'état s_i, alors la probabilité

d'avoir observé cet ensemble de cases vides sachant que l'agent se trouve à la position s_i est nulle i.e. :

$$\begin{aligned}
&\Rightarrow P[\text{de nos observations}|q_t = s_i] = 0 \\
&\Rightarrow b_i(x_t) = 0 \\
&\Rightarrow \alpha_t(i) = b_i(x_t) \sum_j^N \alpha_{t-1}(j) a_{ji} = 0 \\
&\Rightarrow P[q_t = s_i|x_1, x_2, ...x_t] = \alpha_t(i) / \sum_i \alpha_t(i). = 0
\end{aligned} \tag{4.4}$$

Il est bien évidemment souhaitable d'estimer que la probabilité que l'agent soit à la position s_i dans ce contexte est nulle. Ce qui est intéressant, c'est que comme $\alpha_t(i)$ est égal à 0, et que l'algorithme est récursif, nous pourrons alors simplifier le calcule des α_{t+1} en éliminant des éléments des sommations. C'est en quelque sorte comme si nous éliminions dans l'avenir tous les chemins possibles qui passaient par s_i au temps t. Ceci explique en bonne partie pourquoi cette méthodologie est si performante.

Traitons maintenant le cas où l'environnement possède trois dimensions. Reprenons l'ensemble des cases précédemment définies, le quadrillage de la carte. Ces cases en deux dimensions formeront encore l'espace des états de la chaîne sous-jacente. Par contre, il y a désormais une troisième dimension. Nous pouvons, en quelque sorte, imaginer qu'il y a maintenant plusieurs cubes empilés les uns sur les autres, centrés sur chacune de nos cases. Pour s'assurer qu'un adversaire n'est pas à une certaine position, il faut donc s'assurer qu'il ne soit à aucune des hauteurs possibles. En se mettant toujours dans le cas où l'intelligence artificielle ne voit pas directement l'agent, si on observe la totalité des cubes alignés sur une

certaine case, on peut affirmer que l'agent ne se trouve pas à cette position, c'est-à-dire que le processus n'est pas à cet état. Nous devons finalement déterminer quoi faire si l'intelligence artificielle n'observe qu'une partie des cubes alignés sur une certaine case. Parfois, elle observera suffisamment de cubes pour affirmer que son adversaire ne s'y trouve pas, parfois elle observera des cubes ne lui donnant aucune information utile. En posant $X(s_i)$ l'ensemble des cubes visibles alignés sur s_i, Hladky développe l'équation suivante :

$$P(X_t = x_t | q_t = s_i) = 1 - |W \cap X(s_i)|/|X(s_i)|.$$

Cette formule fonctionne très bien pour les cas extrêmes, si cette probabilité est 1 ou 0. Si nous observons la totalité des *hauteurs* associées à une case, nous obtenons 0 et si aucun cube n'est observé, 1. Par contre, cette fonction ne semble pas être très appropriée dans le reste des situations. La probabilité d'avoir eu l'observation obtenue ne devrait pas nécessairement être 50% si nous avons observé la moitié des cubes superposés d'une certaine position. La probabilité qu'un agent soit à une certaine position n'est pas proportionnelle au nombre de *hauteurs* observées puisque certains cubes nous donnent plus d'informations. Le cube le plus bas, par exemple, vaut plus que le cube le plus haut puisqu'un individu marche et ne vole pas. Ce concept n'est pas complètement à rejeter par contre, mais il faudrait déterminer un moyen de pondérer ces cubes pour représenter le fait que certains sont plus informatifs que d'autres. Nous voulions ici soulever qu'il serait possible de travailler dans un univers en trois dimensions. Néanmoins, comme nos améliorations ne porteront pas sur la fonction de masquage, mais plutôt sur l'apprentissage automatique, et qu'il est beaucoup plus simple de travailler en deux dimensions, nous utiliserons à partir de maintenant et jusqu'à la fin du mémoire, la fonction de masquage que nous avons conçue précédemment pour un univers

en deux dimensions (p. 52). Malgré tout, notons que ce que nous développerons pourrait tout de même s'appliquer à un univers en trois dimensions.

La fonction de durée régit le temps qu'un joueur passe à un certain état. L'utilisation des modèles de semi-Markov nous donne de la flexibilité quant à cette fonction. Il est effectivement vrai qu'il peut être stratégique pour un joueur d'attendre un certain nombre d'unités de temps à certains endroits dans une carte de jeu. Il est aussi vrai que de se doter de la capacité d'utiliser n'importe quelle fonction de durée permet de mieux modéliser certains jeux. Néanmoins, d'avoir une modélisation plus réaliste apporte son lot de complexité et augmente l'effort nécessaire lors de l'exécution. Il est important de se poser la question si cette généralisation est nécessaire pour le type de jeu visé avant d'utiliser une fonction de durée complexe.

Afin d'obtenir une matrice de transition et une fonction de durée, la loi de probabilité qui régit le nombre de temps que la chaîne de semi-Markov demeure à un état, Hladky utilise un échantillon de cinquante parties opposant des joueurs professionnels. Il a estimé ces deux composantes de manière empirique en utilisant les journaux (*game logs*) de ces parties. Ces journaux contiennent la totalité de l'information concernant la partie : la position des joueurs, leurs cônes de vision, leurs points de vie et autres, et ce, pour tout temps discret, toute image, communément appelée *frame*. De notre côté, nous ne désirons pas que notre intelligence possède toute cette information privilégiée. Bien qu'un joueur expérimenté finisse par connaître beaucoup d'informations, il doit d'abord apprendre en faisant ses propres expériences, et non se faire donner des d'informations si détaillées.

La matrice de transition est utilisée par l'intelligence artificielle lors du calcul de l'algorithme *forward*. Cette matrice sert d'idée de base des mouvements possibles

de son adversaire. Elle est ce qui guide principalement l'intelligence artificielle. Les observations ne servent qu'à éliminer des chemins au fur et à mesure. De la sorte, la matrice sous-jacente est une composante extrêmement importante de l'estimation dans cette méthodologie. Pour cette raison, il est important que cette matrice soit bien construite pour chaque agent, et ce, sans tricher. C'est pourquoi notre objectif est que cette matrice soit estimée par notre intelligence artificielle au fur et à mesure qu'elle confronte un adversaire.

4.2.2 Expériences réalisées

Nous analyserons ici les résultats obtenus par Hladky de l'implémentation de cette méthodologie à des fins d'estimation de la position d'un joueur adverse sur une carte de jeu. Toujours dans l'objectif de créer une intelligence artificielle efficace, soit un système informatique doté de capacités similaires à celles des êtres humains performants, deux aspects doivent être mis à l'épreuve. Ainsi, il a mis au point deux statistiques : l'erreur de précision de prédiction (*PAE*) et la similitude à l'erreur humaine (*HSE*). La première consiste en une mesure de la distance entre l'estimation de la position de l'agent et sa position réelle, ce qui vise à vérifier si l'intelligence artificielle performe dans sa tâche. L'autre statistique, *HSE*, est une mesure de la distance entre l'estimation de la position de l'agent et des réelles prédictions humaines, recueillies de manière empirique. Cette statistique vise donc à vérifier si l'intelligence artificielle réagit de manière similaire aux processus de réflexion humaine. Ce sont ces deux statistiques qui sont utilisées pour évaluer la qualité des estimations.

En premier lieu, l'objectif était de comparer les deux méthodologies énoncées précédemment, soit les modèles de semi-Markov cachés et les filtres particulaires. En examinant la *HSE* et la *PAE*, il est observé que les modèles de semi-Markov cachés performent mieux que les filtres particulaires en ce qui concerne ces deux statis-

tiques lorsque les paramètres sont optimaux. Non seulement cette méthodologie performe mieux dans ces estimations, mais d'un point de vue d'efficacité, le temps de mise à jour est au minimum 100 fois inférieur à celui des filtres particulaires. Il est donc beaucoup moins exigeant pour l'ordinateur que l'intelligence artificielle utilise cette méthodologie. Finalement, dans cette série d'expériences réalisées, un modèle de Markov d'ordre deux est aussi à l'épreuve. Nous aurions tendance à croire que le fait d'utiliser les deux dernières positions pour estimer la position future devrait grandement améliorer la qualité de cette estimation. Malgré tout, surprenamment, les estimations sont moins bonnes en ce qui a trait aux deux statistiques définies précédemment. Il serait intéressant de travailler cette question pour au moins connaître ce qui cause ce résultat étonnant. Finalement, Hladky mentionne que la prédiction de l'intelligence artificielle est plus précise que celle des joueurs professionnels en ce qui a trait à l'erreur de précision de prédiction et que ces prédictions sont plus humaines que celles du prédicteur parfait. Somme toute, considérant que le temps requis à la mise à jour du prédicteur est moins qu'une milliseconde, ce qui est acceptable dans le contexte d'une implémentation dans un jeu vidéo commercial, et que les estimations sont précises selon les statistiques mises au point, l'auteur fut satisfait de cette méthodologie.

4.3 Apprentissage automatique

Nos connaissances en modèles de semi-Markov cachés nous permettent de faire un constat en ce qui concerne les travaux de Hladky. La matrice sous-jacente, estimée à l'aide des journaux de 50 parties, est ce qui guide majoritairement l'estimation et celle-ci n'est peut-être pas parfaite. Entre autres, elle ne s'adapte pas à tous les adversaires. Quelqu'un jouant de manière hors norme va flouer complètement une intelligence utilisant une matrice estimée à partir de joueurs plus traditionnels. De plus, l'intelligence artificielle se fera berner partie après partie sans jamais changer sa stratégie. Il serait intéressant que l'intelligence artificielle puisse mettre

à jour ses connaissances a priori, la matrice sous-jacente, au fil des duels contre un certain joueur. L'intelligence pourrait donc apprendre sur le long terme le comportement d'un agent en particulier. Les estimations pour cet agent seraient ainsi plus spécialisées et, par le fait même, plus précises.

Nos travaux consistent principalement à implanter une méthodologie permettant à une intelligence artificielle de s'adapter au comportement d'un agent au fil du temps. Si nous observons plusieurs fois le déplacement d'un agent, nous désirons que l'intelligence artificielle utilise cette expérience additionnelle pour adapter ses estimations en fonction de cet agent en particulier. Nous appellerons le fait de conserver l'information du passé concernant un agent pour s'adapter à ce dernier une mémoire adaptative. N'oublions pas que l'objectif premier de la recherche en intelligence artificielle est de reproduire le plus fidèlement possible le processus de la réflexion humaine. Doter notre intelligence programmable d'une mémoire et lui permettre d'apprendre de ces expériences est une avancée importante et nécessaire dans l'objectif d'imiter le raisonnement humain qui utilise ses expériences passées pour tenter de prédire le futur.

Encore une fois, les jeux vidéo nous serviront de cadre d'expérimentation puisqu'il existe une problématique en ce qui a trait à l'intelligence artificielle dans les jeux *PvP*. Malgré tout, les travaux effectués pourraient très bien s'appliquer à l'intelligence artificielle dans d'autres circonstances. Dans le contexte d'un jeu vidéo, plus précisément le *FPS* décrit au chapitre 2, *Counter-Strike*, il est raisonnable de croire qu'après chaque partie, un joueur connaît un peu plus son adversaire et peut mieux prévoir ses déplacements. On souhaiterait donc qu'un *bot* fasse de même.

De la sorte, nous désirons trouver une manière d'utiliser nos observations, par-

ties après parties, dans l'objectif d'accroître nos connaissances sur l'agent, notre adversaire, et ainsi d'améliorer progressivement notre estimation de sa position. Comme nous savons que la matrice de transition sous-jacente utilisée par l'algorithme *foward* représente en quelque sorte cette connaissance que possède l'intelligence artificielle au sujet des habitudes de déplacement de son adversaire, il serait intéressant que celle-ci soit mise à jour au fil du temps. Notre objectif est de se servir de l'algorithme de *Baum-Welch* afin de réestimer cette matrice sous-jacente au fil du temps et améliorer l'estimation fournie par l'algorithme *foward*.

Le défi à utiliser l'algorithme de *Baum-Welch* sur un ensemble de parties, soit sur plusieurs séries d'observations et non une longue série d'observations. Rappelons que l'algorithme détaillé aux chapitres 2 et 3 se rapprochait de sa contrepartie pour des chaînes de Markov standards où l'on estime $\hat{a}_{i,j}$ par la somme du nombre de fois où le processus fait une transition vers j lorsque ce dernier part de i sur le nombre de départs total de i. Dans un contexte de modèle de Markov standard, l'estimation serait exactement la même. Par contre, dans un contexte de modèle caché, comme nous ne savons pas exactement l'état subséquent, les algorithmes mis au point utilisent le maximum d'informations disponibles, dont le vecteur de probabilités initiales. En considérant cela, il y a ainsi une grande différence entre une longue séries d'observations et plusieurs séries d'observations indépendantes.

Nous allons considérer les parties indépendantes, ce qui revient à considérer les séries d'observations comme étant indépendantes. Bien que cela ne soit pas exactement le cas, un agent intelligent favorisera les stratégies gagnantes au fil du temps. Ceci n'est pas exactement faux non plus puisqu'un joueur ne change pas complètement sa stratégie en fonction de la partie précédente. De la sorte, nous pourrons, d'une certaine manière, faire la somme pour toutes les parties de la somme du nombre de fois où le processus fait une transition vers j lorsque ce

dernier part de i sur le nombre de départs total de i comme estimateur de $a_{i,j}$, tout en considérant à chaque fois notre vecteur de probabilités initiales. En ayant joué m parties, nous utiliserons l'algorithme de *Baum-Welch* tel que décrit dans les chapitres précédents mais nous utiliserons cette estimation de la matrice de transition :

$$\hat{a}_{i,j} = \frac{\sum^{m} \sum_{t=2}^{T} \alpha_t(i) a_{i,j} \beta_t^*(j)}{\sum^{m} \sum_{j}^{n} \sum_{t=1}^{T} \alpha_t(i) a_{i,j} \beta_t^*(j)}. \tag{4.5}$$

Cette matrice sera réestimée à la fin de chacun des duels en y ajoutant la nouvelle partie. Considérant que quelques parties sont insuffisantes pour offrir plus d'informations que 50 journaux de matchs professionnels, nous devons déterminer à partir de quand la matrice mise au point sera plus informative.

À partir de ce moment, c'est cette nouvelle matrice qui sera soumise à l'intelligence artificielle lorsque cette dernière utilisera l'algorithme *foward* en temps réel pour estimer la position de son adversaire. Considérant que cette matrice est la composante principale de l'estimation de la position d'un adversaire, nous croyons donc que cette estimation, faite à partir d'une matrice construite suite à une série de matchs contre le même adversaire, sera plus adaptée à cet adversaire. Nous tenterons de démontrer l'efficacité de notre algorithme au chapitre 6.

CHAPITRE V

ALGORITHMIE ET PROGRAMMATION

Dans ce chapitre, nous expliquerons la programmation réalisée afin de mettre au point l'intelligence artificielle, la mettre à l'épreuve et en tirer des conclusions vérifiables statistiquement.

Après avoir choisi et élaboré nos algorithmes, nous avons décidé d'utiliser *C++* comme language de programmation. Ce language est un choix courant dans la recherche en statistique puisque ce dernier offre toute la puissance nécessaire aux calculs d'algorithmes complexes. La programmation fut effectuée en collaboration avec Louis-Alexandre Vallières-Lavoie et Guillaume Racicot, deux étudiants au baccalauréat en génie informatique à l'École de technologie supérieure de Montréal. Alors qu'ils ont surtout travaillé sur la plateforme de tests, nous nous sommes concentrés sur la programmation des divers algorithmes d'inférences. Dans ce chapitre, nous décrirons comment le choix des algorithmes a été effectué et comment ceux-ci furent implémentés. Par la suite, nous décrirons comment fut effectuée la programmation du petit jeu vidéo nous servant de plateforme de tests. Les règles et objectifs seront décrits, en plus de certains détails quant à la programmation concrète de ce jeu et de l'intelligence artificielle servant d'adversaire au joueur.

5.1 Algorithmie

Nous allons discuter ici des algorithmes qui furent implémentés, du choix de ceux-ci ainsi que de certaines techniques utilisées pour assurer une stabilité des algorithmes.

5.1.1 Choix des algorithmes à implémenter

Nous avons choisi d'implémenter les algorithmes de Markov cachés au profit de ceux de semi-Markov cachés. Tout d'abord, le temps mis à la programmation est nettement inférieur. La possibilité de gagner du temps de programmation des algorithmes nous permet de consacrer ce temps de programmation à d'autres aspects du jeu. Nous avons jugé que la plateforme d'expérimentation devait être sans faille pour pouvoir réellement vérifier l'efficacité de notre intelligence artificielle et cette économie de temps de programmation des algorithmes fut profitable pour l'expérience.

Deuxièmement, les algorithmes choisis sont beaucoup plus rapides d'exécution. Comme nous avons fait le choix d'avoir un grand espace des états où exactement chacune des positions possibles est un état, l'utilisation d'un modèle de Markov caché et non de semi-Markov caché a permis d'obtenir des algorithmes beaucoup plus rapides et efficaces. L'algorithme de Baum-Welch pour un modèle de Markov caché est déjà un algorithme qui demande des calculs assez intensifs pour un ordinateur et nous ne voulions pas imposer aux utilisateurs de notre jeu un temps d'attente pénible entre chacune des parties si la version semi-markovienne de cet algorithme avait été utilisée.

Finalement, l'hypothèse que le nombre de visites que le processus stochastique effectue à un certain état soit géométrique n'est pas complètement absurde. En

effet, pour une chaîne de Markov standard, le temps avant que le processus quitte l'état s_i suit une loi géométrique de paramètre $p = 1 - a_{i,i}$. Dans certains jeux vidéo, où de grandes stratégies furent établies à travers le temps, il est commun que des joueurs s'arrêtent et attendent à certaines positions clés. Néanmoins, dans notre jeu, ce n'est pas le cas : il n'y a pas de réel incitatif à rester sur place. Donc, rien ne nous pousse à croire que la distribution de la durée de temps que le processus demeure dans un état est strictement différente d'une géométrique. Comme l'hypothèse de la distribution géométrique était acceptable et parce que nous avions des incitatifs techniques, énoncés précédemment, à utiliser un modèle de Markov caché, c'est ce modèle qui fut programmé pour réaliser nos expériences. Néanmoins, l'utilisation des modèles de semi-Markov pour résoudre notre problématique demeure une alternative viable et, éventuellement, il serait intéressant de tester aussi ces algorithmes.

5.1.2 Transformations de variables

Comme nous avons vu aux chapitres 2 et 3, les valeurs α et β sont des produits de probabilités et tenderont rapidement vers 0. Bien que les ordinateurs puissent manipuler de très petits nombres, leur capacité est limitée quant au nombre de décimales et éventuellement ces valeurs seront arrondies à 0. Bien entendu, certaines bibliothèques permettent de travailler avec de très petites valeurs. Nous avons tout de même utilisé une mise à l'échelle ainsi qu'une transformation logarithmique pour résoudre ce problème dans l'objectif de programmer un outil indépendant des librairies et qui peut fonctionner pour n'importe quel langage.

Discutons maintenant de la mise à l'échelle. Nous expliquerons ces techniques en faisant référence aux coefficients α. Par contre, ce traitement de données est similaire en ce qui concerne les valeurs β. Posons $\hat{\alpha}$, notre variable mise à l'échelle. Nous tenterons de faire en sorte que $\sum_i \hat{\alpha}_t(i) = 1$, nous ferons cela en divisant

les coefficients α initiaux par leur somme. Le tout est clairement expliqué dans l'article de Rabiner (Rabiner, 1989).

Initialisons nos variables de la sorte :

$$\begin{aligned} \ddot{\alpha}_1(i) &= \alpha_1(i), \\ c_1 &= \frac{1}{\sum_i \ddot{\alpha}_1(i)}, \\ \hat{\alpha}_1(i) &= c_1 \ddot{\alpha}_1(i). \end{aligned} \tag{5.1}$$

Comme nous ne voulons qu'aucun résultat de nos calculs s'approche trop près de 0, nous allons utiliser $\hat{\alpha}$ afin de calculer la totalité des itérations de la sorte :

$$\begin{aligned} \ddot{\alpha}_t(i) &= \sum_j^N b_i(x_t) a_{ji} \hat{\alpha}_{t-1}(j), \\ c_t &= \frac{1}{\sum_i \ddot{\alpha}_t(i)}, \\ \hat{\alpha}_t(i) &= c_t \ddot{\alpha}_t(i). \end{aligned} \tag{5.2}$$

Grâce à cette méthode, nous aurons calculé la totalité de nos $\hat{\alpha}$ en nous assurant qu'aucun d'eux ne s'approche trop de 0. Par contre, nos algorithmes nécessitent α et non $\hat{\alpha}$. Nous devrons alors trouver une manière de retrouver α. Analysons $\ddot{\alpha}_2(i)$ dans le but d'observer un résultat intéressant :

$$\begin{aligned}
\hat{\alpha}_2(i) &= c_2\ddot{\alpha}_2(i) \\
&= c_2 \sum_j^N b_2(x_t) a_{1,2} \hat{\alpha}_1(j) \\
&= c_2 \sum_j^N b_2(x_t) a_{1,2} c_1 \ddot{\alpha}_1(i) \\
&= c_2 c_1 \sum_j^N b_2(x_t) a_{1,2} \alpha_1(i) \\
&= c_2 c_1 \alpha_2(i) \\
&= \left(\prod_{k=1}^{2} c_k \right) \alpha_2(i).
\end{aligned} \tag{5.3}$$

De manière similaire que détaillé dans l'équation 5.3, nous pourrions démontrer que : $\hat{\alpha}_t(i) = \left(\prod_{k=1}^{t} c_k\right) \alpha_t(i)$. Néanmoins, nous voulons malgré tout travailler avec le logarithme de nos α pour nous assurer que ceux-ci ne soient pas arrondis à 0.

$$\begin{aligned}
\hat{\alpha}_t(i) &= \left(\prod_{k=1}^{t} c_k \right) \alpha_t(i) \\
\Leftrightarrow \alpha_t(i) &= \left(\prod_{k=1}^{t} \frac{1}{c_k} \right) \hat{\alpha}_t(i) \\
\Leftrightarrow \log(\alpha_t(i)) &= \log\left(\left(\prod_{k=1}^{t} \frac{1}{c_k} \right) \hat{\alpha}_t(i) \right) \\
&= \sum_{k=1}^{t} \log(\frac{1}{c_k}) + \log(\hat{\alpha}_t(i)).
\end{aligned} \tag{5.4}$$

La transformation décrite par les équations 5.1 et 5.2 fut utilisée dans le calcul de l'algorithme *forward* et pour l'algorithme *backward* (ces algorithmes se trouvent en annexe). Bien qu'une bibliothèque des grands nombres aurait probablement

pu régler ce problème, cette transformation est efficace et nous assure que peu importe le langage utilisé, nos valeurs ne convergeront jamais vers 0. Bien que cette transformation ralentisse la vitesse d'exécution, la différence est minime et nous nous assurons d'avoir un code utilisable dans tous les langages sans devoir compter sur une librairie. Nous verrons finalement les transformations nécessaires à l'implémentation de l'algorithme de *Baum-Welch*. Comme les valeurs γ et ξ calculées en 2.3.2 nécessitent la vraissemblance V_T et que celle-ci serait encore une fois arrondie à 0, nous devrons utiliser un autre outil. Celui-ci est mentionné dans le livre de Durbin (Durbin *et al.*, 1998) et consiste à utiliser l'approximation suivante :

$$\log(p+q) = \log(p) + \log(1+\tfrac{q}{p}) = \log(p) + \log(1+\exp(\tilde{q}-\tilde{p})),$$

où $\tilde{q} = \log(q)$ et $\tilde{p} = \log(p)$. Nous voulons calculer la log-vraissemblance, ici nous dénoterons $\alpha_t(\tau) = max_i\{\alpha_t(i)\}$ et $c = \log(\alpha_t(\tau))$:

$$
\begin{aligned}
\log(V) &= \log\left(\sum_i \alpha_t(i)\right) \\
&= \log\left(\sum_{i\neq\tau} \alpha_t(i) + \alpha_t(\tau)\right) \\
&= \log(\alpha_t(\tau)) + \log\left(1 + \exp\left(\log(\sum_{i\neq\tau} \alpha_t(i)) - \log(\alpha_t(\tau))\right)\right) \\
&= c + \log\left(\exp(0) + \exp\left(\log(\sum_{i\neq\tau} \alpha_t(i)) - c)\right)\right) \\
&= c + \log\left(\exp(0) + \exp\left(\log(\sum_{i\neq\tau} \alpha_t(i)\right)\exp(-c)\right) \\
&= c + \log\left(\exp(0) + \left(\sum_{i\neq\tau} \alpha_t(i)\right)\exp(-c)\right) \\
&= c + \log\left(\exp\left(\log(\alpha_t(\tau)) - \log(\alpha_t(\tau))\right) + \sum_{i\neq\tau} \exp(\log(\alpha_t(i)))\exp(-c)\right) \\
&= c + \log\left(\exp\left(\log(\alpha_t(\tau)) - \log(\alpha_t(\tau))\right) + \sum_{i\neq\tau} \exp\left(\log(\alpha_t(i)) - \log(\alpha_t(\tau))\right)\right) \\
&= c + \log\left(\sum_i \exp\left(\log(\alpha_t(i)) - c\right)\right).
\end{aligned}
\tag{5.5}
$$

À l'aide de cette technique (équation 5.5) nous permettant de calculer la log-vraissemblance, nous pourrons dorénavant implémenter l'algorithme de *Baum-Welch* sans difficulté.

5.2 Jeu vidéo

Il sera d'abord question de la mise au point théorique du jeu : quels sont les paramètres, les règlements, les conditions de victoire ou de défaite, et quelles sont les commandes possibles du jeu ? Par la suite, nous aborderons sa programmation

concrète.

5.2.1 Élaboration du jeu vidéo

Afin de tester l'*IA*, il nous fallait concevoir un jeu permettant de mettre à l'épreuve notre algorithme d'estimation de position d'un agent mobile. Conséquemment, nous avons décidé de mettre deux agents mobiles en opposition dans un environnement borné. À ce propos, il est à noter que nous avons mis l'estimation de l'adversaire comme le principal facteur déterminant de la victoire, un peu comme un jeu de cache-cache, surtout pour notre *bot*. De plus, nous devions créer un jeu qui semblait équilibré, c'est-à-dire que les deux joueurs avaient des chances équivalentes de gagner. Pour ce faire, nous désirions créer une carte de jeu qui pousse le joueur à créer et à développer des stratégies fonctionnelles. De la sorte, nous pourrions réellement observer si notre *IA* apprend au fil du temps la stratégie de son adversaire et si elle réagit en conséquence. Il serait peu intéressant de mettre au point un jeu où il est évident que la meilleure stratégie du joueur est de se déplacer de manière aléatoire car, dans ce contexte, il serait inutile que notre *bot* cherche à comprendre les déplacements de son adversaire.

Nous avons mis au point un jeu où les avatars des joueurs se promènent dans une carte de jeu composée de cases clairement définies, comme sur un jeu d'échecs. Il y a des murs, des couloirs et des cases "objectifs", soit les cases où le joueur doit se diriger pour gagner la partie. Bien entendu, un joueur ne peut traverser un mur. Pour gagner, le joueur doit atteindre ces cases "objectifs" alors que son adversaire, le *bot*, devra tenter de l'en empêcher en le trouvant et en le touchant, tout simplement. Nous expliquerons plus en détails les règlements du jeu dans le chapitre portant sur les expérimentations et les résultats. Nous désirons ici mettre nos efforts sur les algorithmes utilisés et la programmation du jeu. Le jeu est programmé pour fonctionner au tour par tour. Comme expliqué brièvement, cela

signifie que le joueur effectuera une action, se déplacer d'une case par exemple, puis son adversaire fera, à son tour, une action, tel qu'un jeu d'échecs où l'on joue tour après tour. Ainsi, notre jeu fut programmé de cette manière. Les actions que peut faire un joueur sont relativement simples : il peut se déplacer vers l'une des quatre cases adjacentes, les déplacements en diagonale n'étant pas permis, ou bien il peut rester sur place. Les commandes sont simplement fournies par le joueur à son avatar par le biais d'un clavier d'ordinateur.

5.2.2 Intelligence artificielle

Notre objectif est toujours de mettre au point une intelligence artificielle capable de battre un joueur, sans tricher. Nous allons tout de même lui donner un ensemble d'informations. Bien entendu, cette information est identique à ce qu'un joueur humain possèderait dans la même situation. L'intelligence artificielle connaîtra les positions de départ potentielles de son adversaire, la position des objectifs et, de plus, il sera au courant de la position de son adversaire à la toute fin du jeu. À l'exception de ces informations, il ne pourra utiliser que ses propres observations pour tenter de gagner.

Après chaque tour, notre programme met au point le vecteur d'observation de l'*IA* pour cet instant grâce à la fonction d'observation définie en 4.2.1 (p. 52). À chaque instant, il va mettre à jour ce que nous appelons la matrice d'observations, qui est en fait une matrice contenant autant de lignes que de tours et où chacune de ces lignes est en fait le vecteur d'observation que nous venons de décrire. Comme son but est de trouver l'adversaire, il va donc, à chaque tour, estimer la probabilité que son adversaire soit à chacune des positions possibles à l'aide de l'algorithme *forward*, tel que décrit dans la section 2.3.1.1 ;

$$P[q_t = s_i | x_1, x_2, ...x_t] = \frac{\alpha_t(i)}{\sum_i \alpha_t(i)}.$$

La position la plus probable sera l'estimation de la position de l'adversaire. Par la suite, il devra déterminer le chemin le plus court vers cette position.

Dans un contexte de recherche de chemin (*pathfinding*), il est courant d'appeler le but d'un trajet *l'objectif*. Ici, l'objectif est l'estimation de la position de l'adversaire, soit sa position la plus probable selon notre modèle. Pour déterminer le chemin le plus court, nous avons considéré chaque position comme un élément d'un graphe et nous avons employé l'algorithme de Dijkstra. De la sorte, l'*IA* peut déterminer, parmi les cases qui l'entourent, celle qui est la plus proche de son objectif. À chaque tour, le *bot* obtient des observations, calcule la position de son adversaire la plus probable, le chemin le plus court et fait le premier pas sur ce chemin.

Une problématique potentielle survient si plusieurs cases adjacentes sont à la même distance de l'objectif. Dans ce cas, pour chacune de ces cases, l'intelligence calculera celle qui révèle les positions les plus probables. Le *bot* se déplacera vers la case dont la somme des probabilités qui l'entoure est maximale.

En ce qui concerne l'algorithme *forward*, nous savons qu'une importante quantité d'informations doit lui être fournie. Nous avons expliqué précédemment d'où provenait la majorité de celles-ci, mais nous détaillerons ici ce qui a trait à la matrice de transition. C'est cette dernière que nous tenterons de perfectionner, partie après partie, dans l'objectif d'avoir une *IA* qui connaît mieux son adversaire. Il faudra néanmoins que le *bot* se débrouille au début sans avoir d'informations concrètes sur son adversaire. Nous demanderons alors à d'honnêtes joueurs de jouer à notre jeu

et nous utiliserons toutes les informations obtenues par ces parties pour créer une matrice de transition empirique qui servira de connaissances générales à notre *bot*. Cette matrice servira donc à l'algorithme *forward* au cours des premières parties.

Par la suite, il faudra construire une matrice de transition tirée des parties jouées contre le joueur. Au fil des parties, les matrices d'observations sont sauvegardées dans des fichiers externes. Ces matrices sont ensuite importées par notre programme pour être utilisées par l'algorithme de *Baum-Welch*. L'intelligence utilisera alors une matrice hybride composée de la matrice représentant une expérience générale qui lui est fournie au début du jeu et une matrice obtenue grâce à l'ensemble des quelques dernières parties jouées contre son adversaire. Cet ensemble de parties devra demeurer petit si l'on veut que notre *bot* demeure sensible au changement soudain de tactique mais il représente une partie importante de la matrice hybride utilisée pour l'estimation en temps continu.

CHAPITRE VI

EXPÉRIMENTATIONS ET RÉSULTATS

6.1 Plateforme de tests

Notre plateforme d'expérimentation est évidemment un jeu vidéo. Tel que mentionné au chapitre 5, un jeu vidéo fut mis au point dans l'objectif précis de mettre à l'épreuve notre *IA*. Pour une meilleure compréhension, nous allons rapidement décrire le jeu, ses règlements et fournir des détails quant à sa conception.

6.1.1 Fonctionnement du jeu

Le jeu oppose deux adversaires : le joueur et l'intelligence artificielle. Le joueur devra atteindre l'un des deux objectifs et y demeurer un certain temps, alors que le *bot* devra trouver son adversaire et le toucher. Le joueur est le premier à jouer et peut décider soit de se déplacer à l'une des cases adjacentes qui n'est pas un mur, soit il peut demeurer sur place. Toutefois, les déplacements en ligne diagonale ne sont pas permis. Après que le joueur ait soumis une commande à son avatar, le *bot* exécutera une action. Il s'agit donc d'un jeu tour à tour, comme le jeu d'échecs. De plus, l'intelligence artificielle ne voit que les positions qui lui sont adjacentes alors que le joueur voit partout autour de lui à une distance de deux cases, lui permettant ainsi de s'enfuir si nécessaire.

6.1.2 Carte de jeu

Bien que le jeu soit simple en apparence, nous avons créé une carte de jeu asymétrique contenant des pièges dans l'objectif d'inciter les joueurs à mettre au point une tactique qu'ils jugent optimale. L'idée est de motiver les joueurs à inventer des stratégies de jeu afin d'évaluer l'efficacité de notre méthodologie à identifier cette stratégie. Dans le même ordre d'idées, les objectifs présents dans la carte ne sont pas à la même distance du joueur : l'un d'eux est nettement plus proche du joueur, mais est aussi plus proche de son adversaire. Aussi, nous avons rajouté une mécanique de case piégée. Sept cases piégées sont situées à certaines positions clés sur la carte. Lorsque le joueur se déplace sur cette case, le *bot* est alerté que le joueur s'y trouve. En plus d'être une source d'informations importante pour l'intelligence artificielle, lui permettant d'être beaucoup plus efficace dans ses estimations, ces cases ajoutent beaucoup de profondeur au jeu en permettant au joueur de les utiliser pour berner son adversaire. Certaines tactiques pourront être développées autour de ces cases : certains joueurs voudront déclencher intentionnellement un des pièges, puis se déplacer dans la direction opposée dans le but de déjouer le *bot*. Bien entendu, la carte a été conçue pour qu'il soit impossible au joueur de gagner avant que son adversaire ait la possibilité de l'atteindre. De plus, le joueur peut s'enfuir s'il voit son adversaire venir à lui, dans la plupart des cas. Or, ce n'est pas le cas des cases «objectifs» : effectivement, si le joueur décide d'y aller, c'est pour gagner, puisqu'il s'agit d'une position vulnérable qui ne lui permettra pas de s'enfuir, donnant ainsi la possibilité au *bot* de gagner la partie. La figure 6.1 représente notre carte de jeu.

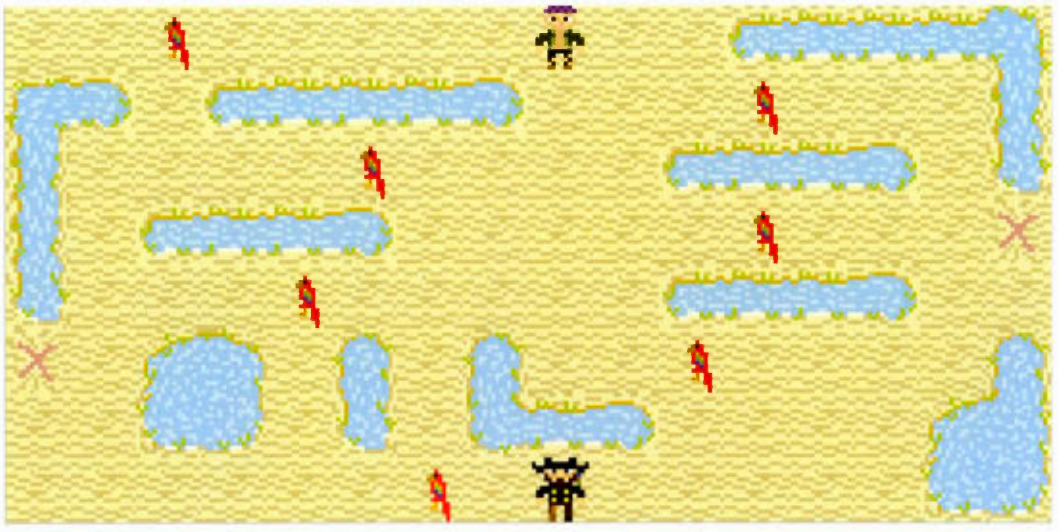

Figure 6.1 Image de l'état initial de notre jeu. Le joueur contrôle l'homme en haut de la carte, l'*IA* celui d'en bas. Le joueur peut se déplacer sur le sable mais ne peut pas traverser les étangs. Les objectifs sont représentés par les deux «X» rouges alors que les cases piégées sont représentées par un perroquet.

6.1.3 Équité des chances

Aucun des membres de notre équipe de recherche n'avait jamais conçu de jeu vidéo précédemment et l'un des défis était de mettre au point un jeu qui semblait équitable, soit un jeu où le joueur et son adversaire semblaient avoir autant de chances de gagner. Pour ce faire, il fallait d'abord s'assurer que chaque joueur puisse gagner. Par exemple, s'il n'y avait qu'un seul objectif, l'intelligence artificielle n'aurait qu'à y aller et y demeurer pour s'assurer de gagner. Par contre, en donnant une meilleure vision au joueur, nous permettons à celui-ci de réagir s'il voit son adversaire. Non seulement ceci ajoute une source de plaisir au joueur, mais celui-ci n'a pas le sentiment qu'il s'agit d'un simple jeu de chance. Néanmoins, en donnant cet outil au joueur, il faut s'assurer que ceci ne lui assure pas

la victoire. Dans cette optique, comme il voit plus loin que son adversaire, il peut potentiellement s'enfuir et ne jamais se faire prendre. C'est pourquoi nous avons conçu les cases objectifs afin que, si le joueur s'y aventure, il soit en position vulnérable et ne puisse s'enfuir même s'il voit son adversaire venir à lui.

Sachant maintenant que chaque joueur a la possibilité de gagner, il faut que ses possibilités soient du même ordre. Il est primordial que les deux joueurs aient le sentiment d'avoir autant de chances de gagner. Bien que pour nous, l'un des joueurs sera un *bot*, il faut une équité de gagner pour que le jeu soit représentatif d'un vrai jeu de joueur contre joueur. Bien qu'il soit très difficile de mettre au point un jeu où la probabilité de gagner est exactement 50 %, nous avons doté notre jeu d'un paramètre nous permettant d'ajuster les probabilités de gagner des joueurs. Il s'agit du nombre de tours que le joueur devra demeurer à l'une des cases objectifs. Si le joueur n'a qu'à y rester deux tours, il aura beaucoup plus de chances de gagner que s'il doit y rester dix tours. Après plusieurs parties, nous avons fixé de manière heuristique le nombre de tours pour que le joueur gagne à six. Par contre, il est à noter qu'un ajustement peut être effectué si nous réalisons que le jeu n'est pas équitable. De plus, cet outil pourrait nous servir pour créer plusieurs niveaux de difficulté.

6.1.4 Mise en situation du jeu

Nous avons créé une mise en situation à notre jeu pour plusieurs raisons. Tout d'abord, il était plus agréable d'y œuvrer pendant le processus de création. De plus, la mise en contexte aide beaucoup les joueurs à comprendre les objectifs tout en augmentant, espérons-le, l'appréciation de l'expérience. Finalement, la vulgarisation de certains éléments du jeu se voit facilitée puisque l'on peut effectuer des références concrètes.

Notre jeu opposera deux pirates, un matelot mutiné et le capitaine. Les deux sont sur une île, cette île représente la carte de jeu, c'est notre environnement borné. Le matelot devra tenter de voler un trésor au capitaine. Ce faisant, il prouvera qu'il est digne d'être le nouveau capitaine. Dans notre contexte, les trésors représentent les cases «objectifs» et le nombre de tours que l'on doit y passer peut s'expliquer par le temps que cela prend pour déterrer le trésor. Nous pouvons justifier la moins longue portée de la vision du capitaine par son cache-œil. Malgré tout, il a vécu plus d'une mutinerie et s'il attrape le matelot, il met fin à celle-ci immédiatement. Finalement, les cases piégées s'expliquent par les fidèles perroquets du capitaine qui crieront à tue-tête s'ils aperçoivent le matelot en cavale.

6.2 Statistique de tests

Nous devrons mettre au point une statistique de tests nous permettant de vérifier l'efficacité de nos algorithmes. Bien que notre environnement d'expérience soit un jeu vidéo, notre objectif principal est d'estimer la position d'un agent mobile dans un environnement borné. Nous allons donc construire une statistique qui évalue l'efficacité de nos outils à effectuer précisément cette tâche.

À l'intérieur même de notre programme, nous avons inclu le calcul de la distance entre l'estimation faite par l'intelligence artificielle et la position réelle de l'agent et ce, pour chaque tour. Par contre, la précision de l'estimation ponctuelle en elle-même n'est pas très enrichissante. En début de partie et après chaque fois que le joueur touche à un piège, elle sera très précise. C'est pourquoi nous la regarderons à travers une partie complète. Notre statistique d'intérêt sera donc la distance moyenne entre l'estimation de la position de l'agent et sa position réelle au cours d'une partie.

Nous allons utiliser cette statistique de plusieurs manières. Nous allons, premiè-

rement, observer l'évolution de cette statistique de partie en partie. En ce qui concerne notre *IA*, nous devrions observer une diminution de cette statistique à mesure qu'elle joue contre le même agent alors qu'une *IA* sans apprentissage performera toujours de la même manière. De plus, nous allons calculer cette statistique pour notre *IA* et la comparer avec cette même statistique obtenue pour l'*IA* de Hladky.

6.3 Expérimentation

6.3.1 Paramètres d'expérimentation

Nous avons conçu plusieurs expériences pour mettre à l'épreuve notre *IA* dans plusieurs scénarios. Notre statistique d'intérêt était celle décrite à la section précédente. Ce que nous avions constaté dans les précédents travaux, sur le même sujet, était le manque d'adaptation à certains scénarios précis. Si un joueur développe une stratégie gagnante, il pourra la répéter sans arrêt et toujours gagner contre une *IA* qui n'apprend pas. De plus, un joueur qui hors de l'ordinaire, jouant d'une manière très différente de celle des joueurs échantillonnés pour former la connaissance générale, surprendra toujours une *IA* sans mémoire adaptative.

C'est pourquoi les trois scénarios mis au point auront comme objectif de vérifier précisément cette capacité d'adaptation. Premièrement, nous avons déterminé une stratégie gagnante, puis nous l'avons utilisée pour dix parties consécutives. Les *bots* sans apprentissage devraient se faire flouer chaque fois tandis que le nôtre devrait s'adapter, soit estimer plus précisément la position de son adversaire, de partie en partie, et finalement le vaincre. C'est pourquoi notre première expérimentation consistait à opposer plusieurs *IA* différentes à un joueur répétant exactement la même stratégie pour dix parties de suite.

Notre premier scénario fut conçu pour nous assurer qu'un apprentissage était possible. Par contre, la deuxième expérimentation vérifie qu'après un apprentissage, notre intelligence artificielle saura s'adapter de nouveau si nécessaire. Pour ce faire, nous avons trouvé une stratégie gagnante et l'avons réutilisée pour sèpt autres parties. Par la suite, nous avons changé pour une nouvelle tactique gagnante et l'avons réutilisée encore sept fois. De la sorte, nous pouvons évaluer la capacité de l'*IA* à s'adapter à un premier parcours, puis à s'adapter suite à un changement drastique de stratégie.

Finalement, nous voulions nous assurer que notre intelligence pouvait apprendre simultanément plusieurs stratégies. Nous avons donc déterminé deux stratégies gagnantes et nous avons joué douze parties en alternant entre ces deux stratégies. Bien que notre intelligence ne décèle pas l'alternance, elle devrait néanmoins identifier des indices qui témoignent de la stratégie utilisée et réagir en conséquence.

Dans ces divers scénarios, trois intelligences furent mises à l'épreuve. Toutes trois utilisent les algorithmes décrits précédemment pour estimer la position de leur adversaire. Par contre, ces *bots* se servent tous d'une matrice de transition différente pour produire ces estimations. Nous avons tout d'abord créé le *bot* le moins intelligent possible. Ce dernier utilise une matrice de transition uniforme : il considère que les déplacements de son adversaire sont distribués de manière uniformément aléatoire. Par la suite, nous avons aussi mis au point une *IA* similaire à celle de Hladky. Cette dernière se sert d'un échantillon de cinquante parties jouées par de bons joueurs pour construire sa matrice de transition. Finalement, notre *IA* utilise aussi ces mêmes cinquante parties pour construire des connaissances de base, mais utilise aussi les dix dernières parties jouées contre son adversaire pour construire, à l'aide de l'algorithme de *Baum-Welch*, une matrice de transition qui s'adapte aux comportements de ce dernier.

6.3.2 Premier scénario

En premier lieu, nous avons développé un outil pour bien démontrer visuellement nos résultats. Il s'agit de carte de chaleur (*heatmap*) des probabilités. En fait, ces cartes sont une représentation de la carte de jeu sur laquelle nous affichons les probabilités estimées par l'intelligence artificielle, probabilités représentées par un dégradé de rouge. Plus la case est rouge, plus la probabilité estimée par le *bot* que le joueur s'y trouve est grande. À l'opposé, une case blanche signifie que l'intelligence associe une probabilité nulle à cette position. Ces cartes nous permettent d'abord de visualiser ce que fait l'intelligence artificielle en temps réel. Nous y observons comment elle estime la probabilité à chacune des cases et comment elle se déplace en conséquence. Ces cartes furent d'une grande utilité pour observer le travail de l'intelligence tour par tour. De plus, ces cartes nous aident aussi à visualiser graphiquement l'apprentissage. Nous pouvons comparer ces graphiques pour plusieurs parties afin d'y observer l'évolution des diverses estimations.

Dans ces graphiques, les cases noires représentent les murs. Le matelot contrôlé par le joueur est représenté par le carré bleu, le carré bourgogne représente le capitaine, c'est-à-dire l'avatar contrôlé par l'*IA*. Aux cases objectifs, les trésors, on retrouve des carrés jaunes. Pour terminer, la position des perroquets est identifiée par des triangles verts.

Nous allons d'abord observer (figures 6.2 et 6.3) les cartes de chaleur d'une partie complète ; il s'agit du dixième affrontement entre notre *IA* dotée d'une mémoire adaptative et un joueur qui utilise le même trajet depuis la première partie.

Analysons le contenu de ces graphiques. Tout d'abord, la toute première image

de la figure 6.2 représente l'état initial du jeu. On constate que les deux joueurs sont à la même position que dans la figure 6.1. Comme dans ce jeu les joueurs connaissent la position de départ de leurs adversaires respectifs, on observe qu'une seule case est teintée de rouge dans cette image ; il s'agit de la position de départ du joueur. De plus, cette coloration est très foncée. Ceci indique que le *bot* estime bel et bien que la seule position possible au premier tour est la position de départ. Nous observons, dans les cartes suivantes, que les probabilités se dissipent graduellement autour du point de départ. Comme l'intelligence savait où se trouvait son adversaire au premier tour, elle sait qu'au deuxième tour il devrait se trouver autour de la case de départ. Conséquemment, on note cette dissipation progressive des probabilités estimées pour les images subséquentes.

À la septième image de la figure 6.2, nous constatons un phénomène particulier : nous observons les cases piégées à l'oeuvre. Souvenons-nous que lorsque le joueur touche à l'un des perroquets, le capitaine contrôlé par l'intelligence est alerté et apprend la position du joueur à cet instant précis. C'est pourquoi nous observons que pour cette carte, une seule case est rouge, celle où se situe le joueur. De la même manière qu'au départ du jeu, on note dans les cartes suivantes que les probabilités se dissipent à partir de cette position. Il est intéressant d'observer dans plusieurs graphiques, dont ceux de la troisième ligne de la figure 6.3, la situation des cases piégées. Comme l'intelligence sait si le joueur s'y trouve, elle sait, en quelque sorte, que le joueur ne s'y trouve pas le reste du temps. Ceci explique pourquoi ces cases sont toujours blanches sauf si le joueur s'y trouve. Nous voyons aussi, dans cette ligne, et dans plusieurs autres cartes, le blanc qui entoure le petit carré bourgogne. Ceci est causé par les observations du *bot* qui élimine, tour après tour, les cases qui l'entourent, leur associant ainsi une probabilité nulle.

Finalement, ces cartes témoignent aussi de l'apprentissage. Souvenons-nous que les

images des figures 6.2 et 6.3 proviennent de la dernière partie contre un adversaire qui répète la même stratégie. Dans les premières images de la figure 6.2, on voit que l'intelligence semble avoir compris que le joueur commençait la partie en empruntant le couloir de gauche. Cela s'observe par la couleur qui colore la case du joueur à chaque tour. En effet, nous observons que la case où se trouve le joueur est toujours colorée en rouge vif. Nous constatons ce phénomène durant la majorité de la partie, nous permettant ainsi de constater que les estimations de l'intelligence sont assez précises. On observe aussi que l'*IA* poursuit le joueur efficacement, étant positionnée toujours très près de ce dernier.

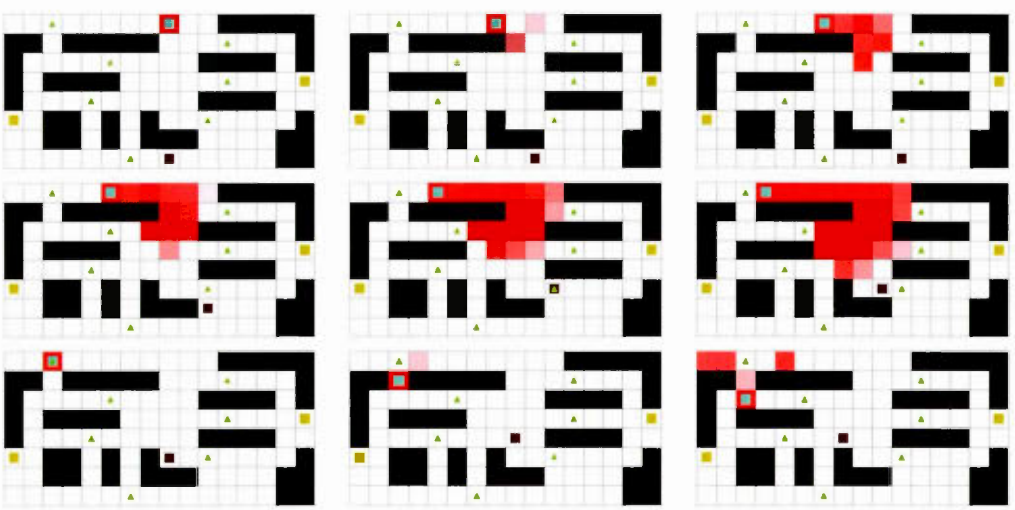

Figure 6.2 Cartes de chaleur de la dixième partie contre un joueur utilisant la même stratégie. Tours 1 à 9. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de gauche à droite, puis de haut en bas).

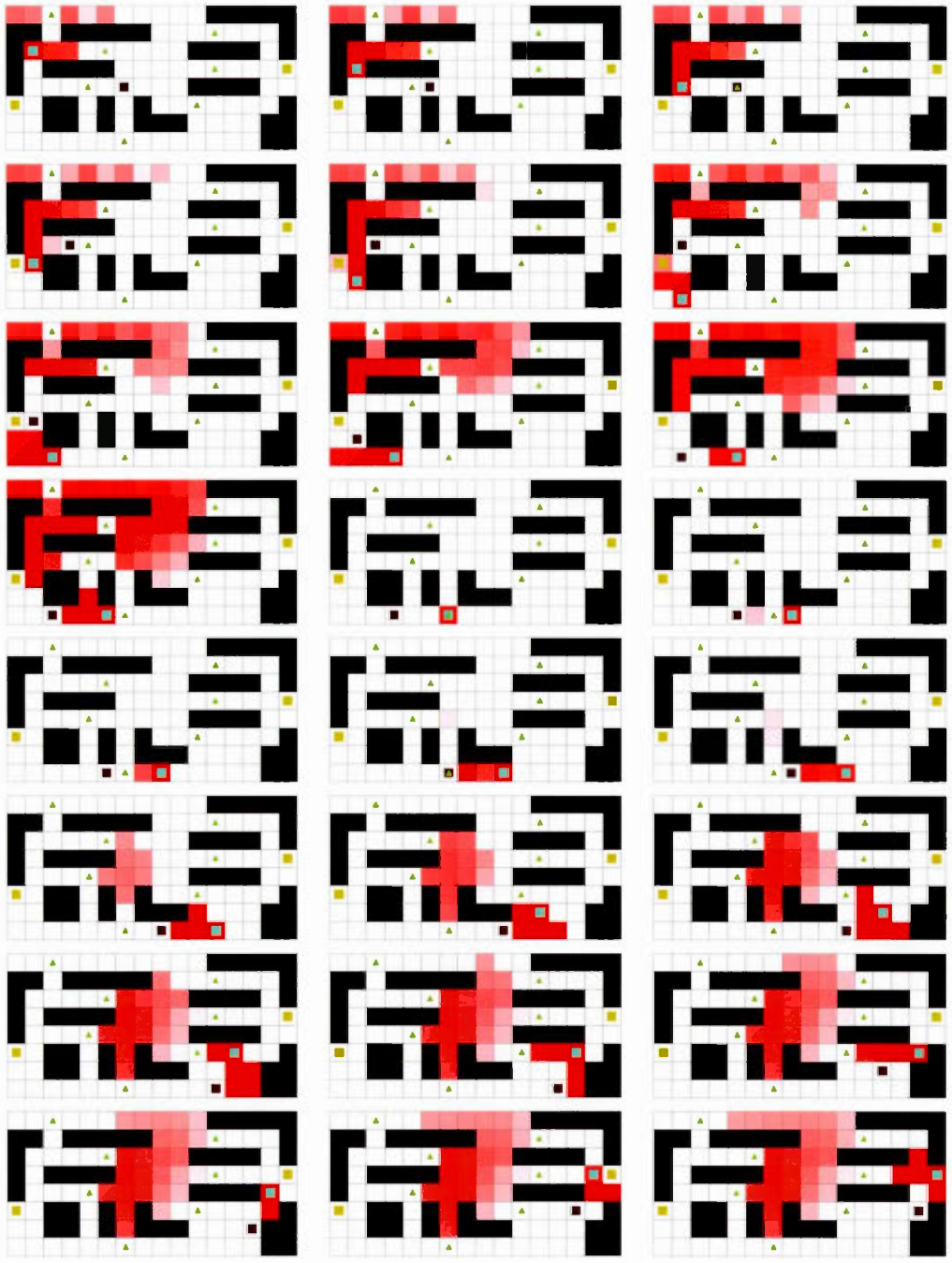

Figure 6.3 Cartes de chaleur de la dixième partie contre un joueur utilisant la même stratégie. Tours 10 à 33. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de gauche à droite, puis de haut en bas).

Nous avons répété l'expérience d'effectuer dix fois le même parcours contre l'*IA* avec quatre parcours différents. Ces stratégies furent respectivement gagnantes lors de leur première utilisation. Nous avons mis au point des graphiques pour représenter la statistique de tests détaillée à la section 6.2. Souvenons-nous que nous avons calculé cette statistique pour différentes intelligences artificielles. La figure 6.4 contient les graphiques de l'évolution de notre statistique au fil de dix parties contre les trois *bots* décrits à la section 6.3.1 pour quatre stratégies différentes notées A,B,C et D.

Les graphiques de la figure 6.4 nous aident à observer la composante d'apprentissage et son effet sur la qualité de l'estimation contre des adversaires qui répètent la même stratégie pour un grand nombre de fois. Bien que nous ayons un peu exagéré l'expérience en utilisant exactement le même parcours dix parties consécutives, cette stratégie permettait au joueur de gagner contre les deux *IA* sans apprentissage. Il aurait été naturel que le joueur effectue sans cesse ce trajet étant donné qu'il n'aurait jamais perdu de cette manière. D'un point de vue comportemental, la victoire agit comme un renforcement positif qui encourage le joueur à répéter l'expérience. En effet, il se voit récompensé en utilisant la stratégie gagnante car il remporte la partie.

Ces graphiques représentent donc la distance moyenne entre la véritable position du joueur et l'estimation de cette position faite par l'intelligence artificielle pour une partie. La ligne noire fut calculée à partir de duels contre une *IA* sans apprentissage et sans connaissance de base. La ligne verte provient des parties jouées contre le *bot* développé par Hladky. Pour terminer, la ligne rouge représente notre *IA* avec une mémoire adaptative.

Analysons plus en détails ces graphiques. Dans les quatre images de la figure 6.4,

nous observons que les courbes noires et vertes sont, en fait, des constantes. Ceci s'explique par le fait qu'elles ont été calculées à partir de parties jouées contre des *IA* sans apprentissage, comme Hladky suggérait. Pour ces intelligences, chaque partie se déroule exactement de la même manière : elles estiment toujours la position du joueur comme étant la même et effectuent toujours le même parcours. La distance moyenne d'une partie entre la position réelle du joueur et l'estimation de ces deux intelligences est donc constante à chaque confrontation. Cela entre en contraste avec la courbe rouge qui est globalement décroissante. Cette dernière représente la statistique de tests, partie après partie, calculée à partir de confrontations contre l'intelligence que nous avons mise au point. Le fait qu'elle diminue implique qu'au fur et à mesure que l'*IA* affronte un joueur qui effectue la même stratégie, la distance moyenne entre la position réelle et l'estimation diminue. En d'autres mots, l'estimation de la position de l'agent est de plus en plus précise à chaque partie. De plus, dans les quatre graphiques de la figure 6.4, on constate qu'à la dernière confrontation la distance moyenne est extrêmement proche de zéro, nous laissant croire que l'intelligence a estimé correctement la position du joueur humain presque à chaque tour.

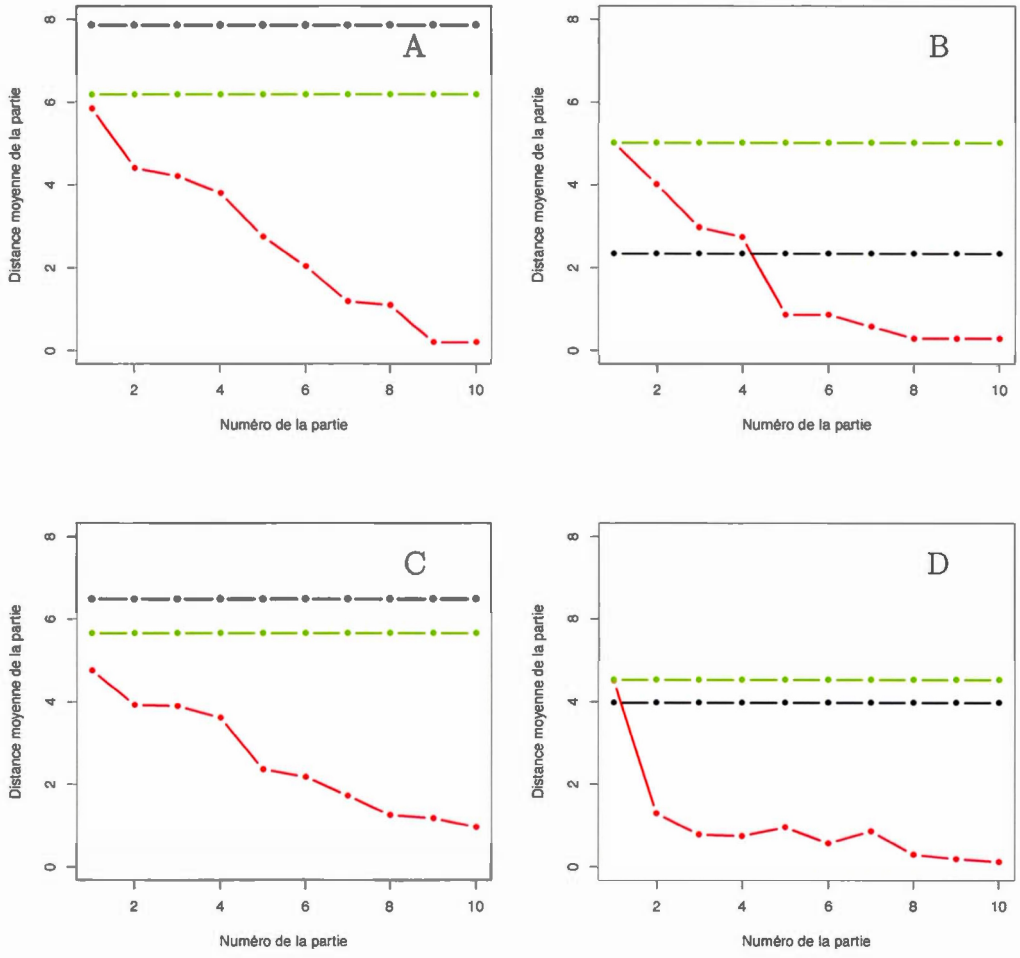

Figure 6.4 Évolution de la statistique de tests pour quatre stratégies du premier scénario. La ligne noire fut obtenue en jouant contre une *IA* sans connaissance, la verte provient des parties jouées contre le *bot* développé par Hladky. Finalement, la ligne rouge représente notre *IA* avec une mémoire adaptative.

Par la suite, nous voulions nous assurer qu'à la dixième partie le *bot* doté d'une mémoire adaptative performerait plus efficacement de manière statistiquement

significative. Nous avons choisi la dixième partie puisqu'il s'agit de la dernière, c'est donc à cette partie que l'intelligence a le plus appris. Le tableau 6.1 contient les résultats de ces tests.

Nous observons (tableau 6.1) que la distance moyenne est significativement inférieure pour notre *IA* avec apprentissage, et ce,pour quatre parcours complètement différents. Cela signifie qu'elle estime plus précisément la position de son adversaire que l'intelligence mise au point par Hladky, après apprentissage.

Stratégie	Partie #10	
	Valeurs t	Valeurs p
A	-7.3049	6.225e-09
B	-3.2673	0.00619
C	-5.7857	4.376e-07
D	-9.7531	4.356e-12

Tableau 6.1 Résultats des tests pour les quatre stratégies du premier scénario.

Voyons finalement une démonstration visuelle de l'apprentissage grâce aux cartes de chaleur. Voyons la première et la dixième parties opposant notre *IA* à un joueur utilisant la même stratégie au fil de ces dix parties. L'objectif est d'observer la différence dans l'estimation de la probabilité relative à chacune des positions par l'utilisation de ces cartes. Les figures 6.5, 6.6, 6.7 et 6.8 présentent les cartes de chaleur pour les tours impairs de ces deux parties. La colonne de gauche représente la première partie et celle de droite la dixième. La première ligne de la figure 6.5 représente le premier tour.

En analysant attentivement ces figures, nous observons que, dans la quasi-totalité des cartes, la case sous le joueur est teintée d'un rouge plutôt pâle dans la colonne

de gauche, ce qui contraste avec le fait que la position du joueur est toujours colorée d'un rouge foncé dans la colonne de droite. C'est-à-dire que l'*IA* estime la position réelle du joueur comme étant improbable lors de leur première confrontation, mais cette même position est estimée comme étant très probable lors de la dixième partie. Ce phénomène est évidemment dû à la composante d'apprentissage. De plus, ces graphiques nous permettent aussi de témoigner du changement de stratégie de l'intelligence en fonction de ses nouvelles estimations. En effet, comme les probabilités associées à chaque position sont différentes à la dixième partie, le *bot* change ainsi de parcours afin de toucher son adversaire. Nous assistons donc, à la cinquième ligne de la figure 6.5, à un changement important de stratégie de la part de l'intelligence. On observe, dans la colonne de gauche, qu'elle a décidé de continuer sa montée vers le haut, alors qu'elle a tourné à gauche dans la colonne de droite. Ce changement de parcours peut s'observer dans les figures 6.5, 6.6, 6.7 et 6.8 où l'on observe que l'*IA* est constamment plus près de son adversaire durant la dixième partie. Nous pouvons ainsi affirmer que l'estimation plus précise de la position de son adversaire a permis à l'intelligence de mieux performer dans la poursuite du joueur humain et ainsi de remporter la partie.

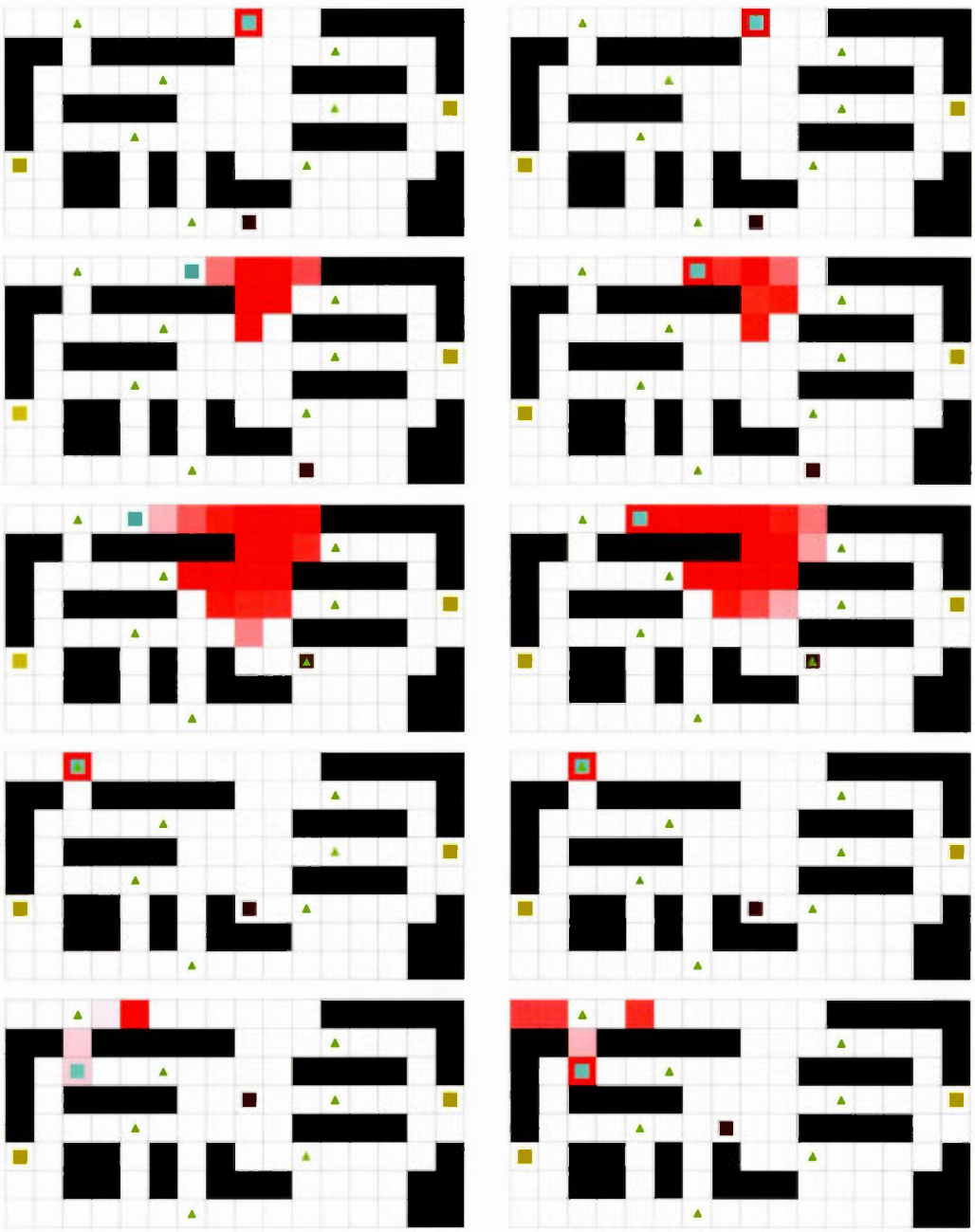

Figure 6.5 La colonne de gauche représente la première partie, celle de droite la dixième. Tours impairs de 1 à 9. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de haut en bas).

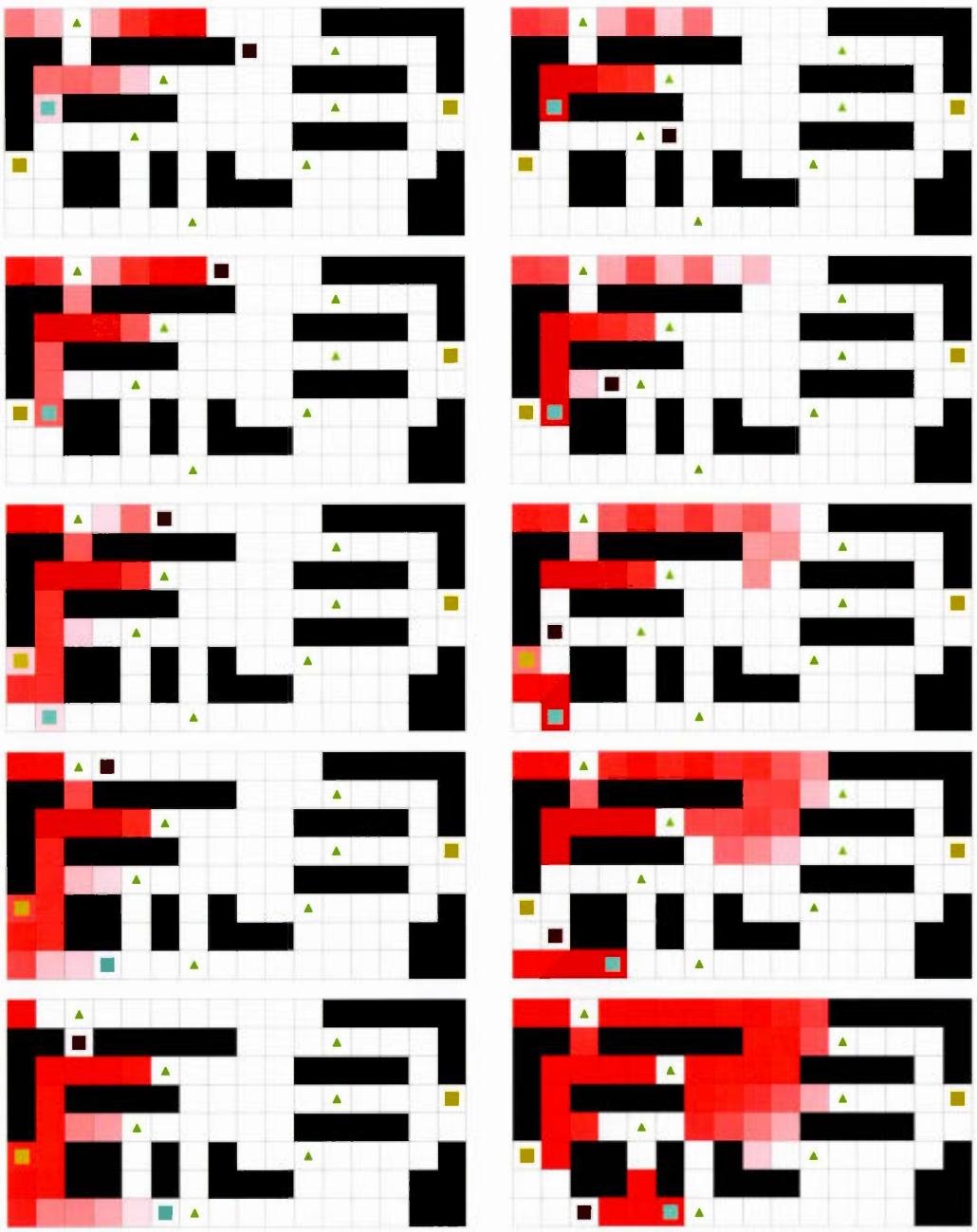

Figure 6.6 La colonne de gauche représente la première partie, celle de droite la dixième. Tours impairs de 11 à 19. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de haut en bas).

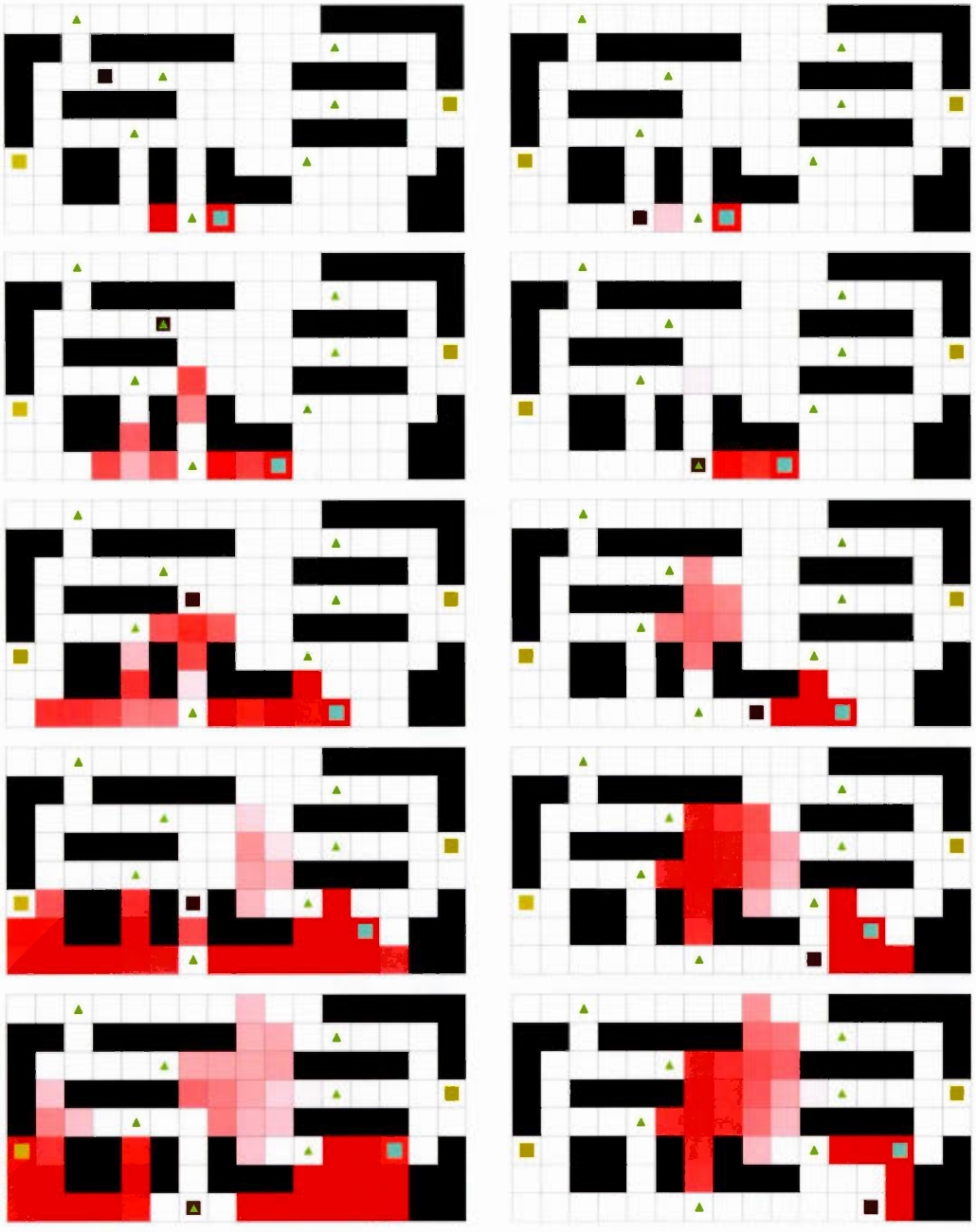

Figure 6.7 La colonne de gauche représente la première partie, celle de droite la dixième. Tours impairs de 21 à 29. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, puis de haut en bas).

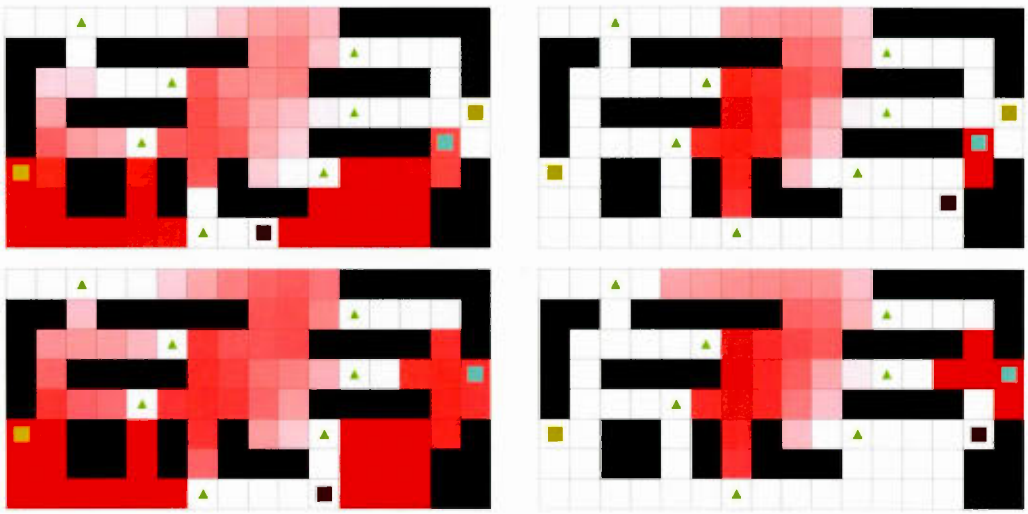

Figure 6.8 La colonne de gauche représente la première partie, celle de droite la dixième. Tours 31 et 33. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de haut en bas).

6.3.3 Deuxième scénario

Rappelons que notre deuxième expérimentation consistait à changer de stratégie après huit parties et à observer l'adaptation de notre intelligence à cette nouvelle tactique. Nous voulions ainsi nous assurer que notre intelligence était capable d'apprendre une stratégie, mais surtout d'en apprendre une autre par la suite. Nous avons fait cette expérimentation quatre fois, utilisant ainsi huit stratégies différentes. La figure 6.9 présente les graphiques d'évolution de la distance moyenne.

Dans ces graphiques, nous observons, encore une fois, que les courbes vertes et noires sont des fonctions constantes. Il y a, par contre, une cassure à la neuvième partie, causée par le changement de stratégie : par la suite la fonction redevient constante. En ce qui a trait à la courbe rouge, celle calculée à partir des confron-

tations contre l'intelligence artificielle avec mémoire adaptative, nous observons d'abord une diminution de la distance moyenne jusqu'à la huitième partie. On note ensuite une brisure et une importante distance moyenne pour la neuvième partie. Ceci s'explique par le changement de stratégie de la part du joueur qui a surpris l'intelligence, cette dernière estimant que son adversaire effectuerait un chemin similaire que celui effectué lors des huit parties précédentes. Malgré tout, on observe que l'*IA* apprend à nouveau progressivement. On peut constater ce nouvel apprentissage par la diminution de la distance moyenne que l'on observe de la neuvième à la seizième partie. Somme toute, nous sommes satisfaits de la capacité de l'intelligence à s'adapter à une nouvelle stratégie après en avoir déjà appris une.

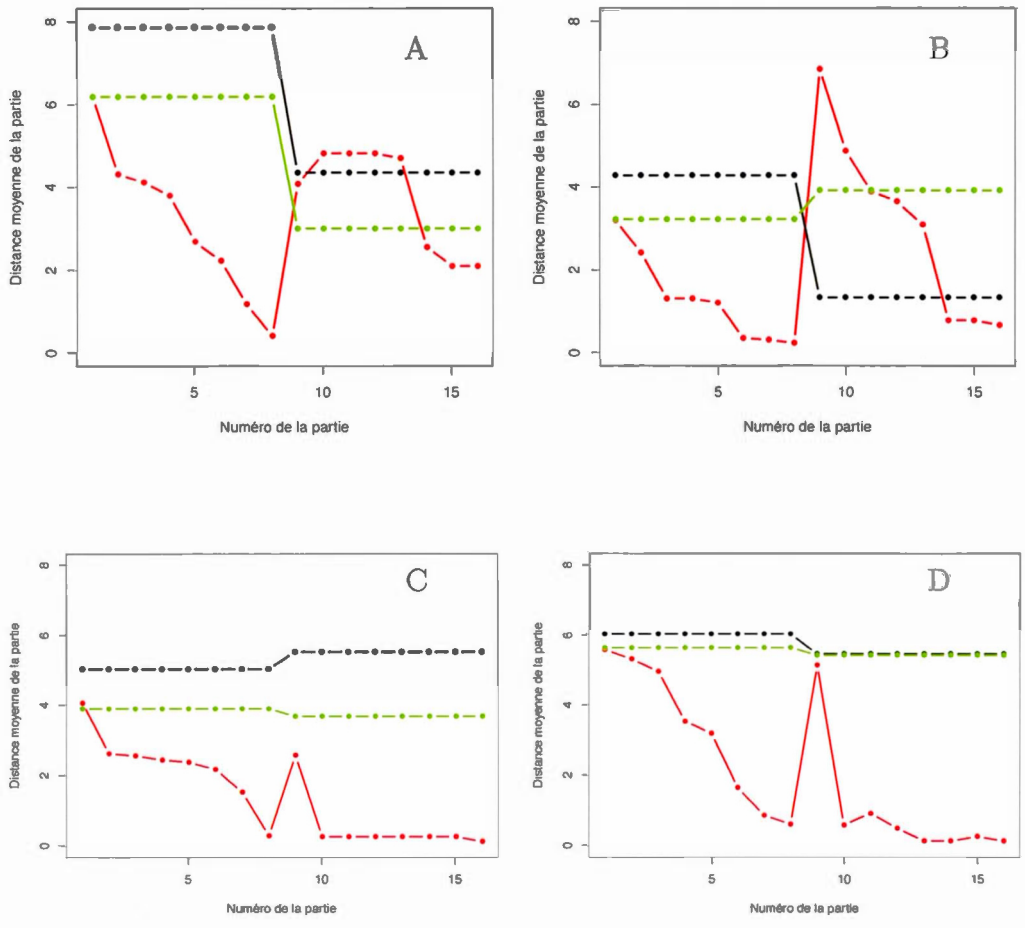

Figure 6.9 Évolution de la statistique de tests pour quatre expériences du deuxième scénario. La ligne noire fut obtenue en jouant contre une *IA* sans connaissance, la verte provient des parties jouées contre le *bot* développé par Hladky. Finalement, la ligne rouge représente notre *IA* avec une mémoire adaptative.

Encore une fois, ici, nous voulions vérifier que notre intelligence performait mieux que celle de Hladky après l'apprentissage, c'est-à-dire à la huitième confrontation ainsi qu'à la seizième. Malgré tout, nous désirions aussi nous assurer qu'elle ne

performait pas significativement moins bien que les autres tout juste après le changement de stratégie, soit à la neuvième rencontre. Le tableau 6.2 contient les résultats de ces tests.

Selon nos expériences, notre intelligence a estimé avec plus de précision la position de son adversaire de manière statistiquement significative sept fois sur huit après l'apprentissage. De plus, elle a performé aussi bien que les deux autres intelligences immédiatement après un changement de stratégie lors de trois expériences.

Changement	Partie #8		Partie #9		Partie #16	
	Valeurs t	Valeurs p	Valeurs t	Valeurs p	Valeurs t	Valeurs p
A	-6.9473	1.42e-08	0.7686	0.4492	-0.7168	0.2419
B	-6.4082	9.993e-08	2.7693	0.007986	-4.5768	4.114e-05
C	-6.3822	9.97e-08	-1.1967	0.2386	-5.0932	2.241e-05
D	-7.2901	4.362e-08	-0.2683	0.7894	-8.0063	1.387e-09

Tableau 6.2 Résultats des tests pour les quatre changements de stratégie du deuxième scénario.

6.3.4 Troisième scénario

Finalement, le troisième scénario que nous avons mis au point consiste à alterner entre deux stratégies gagnantes. Souvenons-nous que nous voulions nous assurer que notre *IA* pouvait apprendre plus d'une stratégie à la fois. Il aurait été problématique qu'un joueur puisse alterner entre deux stratégies gagnantes et que cette alternance soit suffisante pour empêcher l'intelligence de bien s'adapter à son adversaire. En tant que joueur, nous avons décidé d'utiliser exactement le même parcours à toutes les parties impaires puis une autre stratégie pour toutes les parties paires. Nous avons ainsi réalisé deux expériences mettant au point quatre

stratégies différentes. Les graphiques de l'évolution de notre statistique de tests sont visibles dans la figure 6.10.

Analysons rapidement ces graphiques. Les lignes vertes et noires oscillent légèrement, performant respectivement toujours de la même manière lors des parties paires et impaires. Pour ces intelligences sans apprentissage, l'alternance ne change rien. Étant donné qu'elles ne sont pas dotées d'une mémoire quelconque, leurs estimations sont toujours les mêmes et, ainsi, la distance moyenne au cours d'une partie entre la position réelle et l'estimation de la position de l'agent est toujours la même. En ce qui a trait à l'intelligence avec mémoire adaptative que nous avons mise au point, représentée par la ligne rouge, on observe une décroissance de cette courbe à long terme. Bien qu'elle oscille légèrement et qu'on observe parfois des remontées, cette courbe est généralement décroissante. Les remontées sont causées par les changements de stratégies répétitifs qui trompent l'intelligence lors de certaines parties. L'adaptation semble être légèrement plus difficile dans cette expérience. Néanmoins, la distance moyenne entre la position réelle de l'agent mobile et l'estimation de notre *bot* est nettement inférieure pour les deux stratégies après l'apprentissage, soit aux onzième et douzième parties.

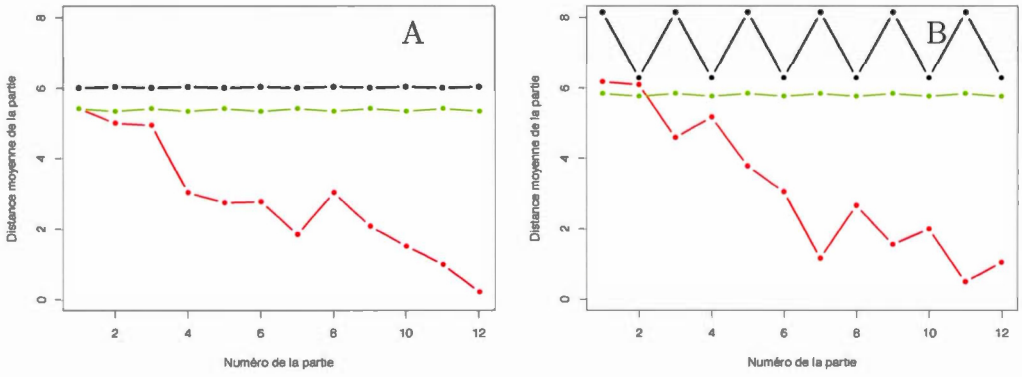

Figure 6.10 Évolution de la statistique de tests pour quatre expériences du troisième scénario. La ligne noire fut obtenue en jouant contre une *IA* sans connaissance, la verte provient des parties jouées contre le *bot* développé par Hladky. Finalement, la ligne rouge représente notre *IA* avec une mémoire adaptative.

Nous allons finalement nous assurer que l'estimation de la position de l'agent mobile est plus précise pour notre intelligence pour les stratégies différentes après l'apprentissage. Nous allons donc vérifier si la différence est significative pour la onzième et la douzième parties, puisqu'il s'agit des dernières parties contre chacune des stratégies respectives. Les résultats de ces tests se trouvent dans le tableau 6.3. Dans ces deux expériences, la distance moyenne entre la véritable position du joueur et l'estimation de cette position de notre intelligence avec mémoire adaptative est significativement inférieure à cette même statistique pour l'*IA* de Hladky. Dans le tableau 6.3, on observe que les estimations de l'intelligence artificielle que nous avons mise au point sont plus précises pour les quatre stratégies que les estimations d'une intelligence sans apprentissage. Nous pouvons affirmer que l'*IA* que nous avons mise au point est capable d'apprendre plus d'une stratégie simultanément.

Alternance	Partie #11		Partie #12	
	Valeurs t	Valeurs p	Valeurs t	Valeurs p
A	-5.3526	6.451e-06	-7.8525	1.783e-09
B	-6.8703	1.983e-08	-6.2715	2.553e-07

Tableau 6.3 Résultats des tests pour les deux alternances de stratégies du troisième scénario.

Finalement, nous allons à nouveau analyser deux séries de cartes de chaleur. Nous analyserons ici la première partie, la colonne de gauche, et la onzième partie, celle de droite. Souvenons-nous que le joueur a utilisé exactement le même parcours durant les parties # 1, # 3, # 5, # 7, # 9 et # 11. Encore une fois, dans les cartes des figures 6.11 et 6.12, nous observons que la case sur laquelle se trouve le joueur est généralement colorée d'un rouge relativement pâle dans la colonne de gauche, contrairement au rouge très foncé de cette même case dans les cartes de la colonne de droite. Cela indique que l'*IA* considère, à la onzième partie, que la case sur laquelle est situé le joueur est une position très probable, ce qui témoigne en quelque sorte de son apprentissage. Il est intéressant d'observer que, dans la carte de droite de la deuxième ligne de la figure 6.11, nous pouvons y trouver deux cases teintées de rouge foncé à des positions relativement éloignées. Ceci est causé par l'apprentissage de deux stratégies de manière simultanée. Dans cette expérience, l'autre stratégie utilisée consistait à emprunter le chemin de droite. Dans ce contexte, on observe que l'intelligence est hésitante en début de partie. Cela change lorsque le joueur touche au perroquet, ce qu'on peut observer à la quatrième ligne de la figure 6.11. Cette information aide l'intelligence à saisir laquelle des deux stratégies est utilisée par le joueur et les estimations suivantes sont nettement plus précises (voir la colonne de droite de la figure 6.12). On observe

finalement, dans la colonne de droite, que l'avatar de l'intelligence artificielle utilise un chemin qui le positionne beaucoup plus proche du joueur que lors de la première partie.

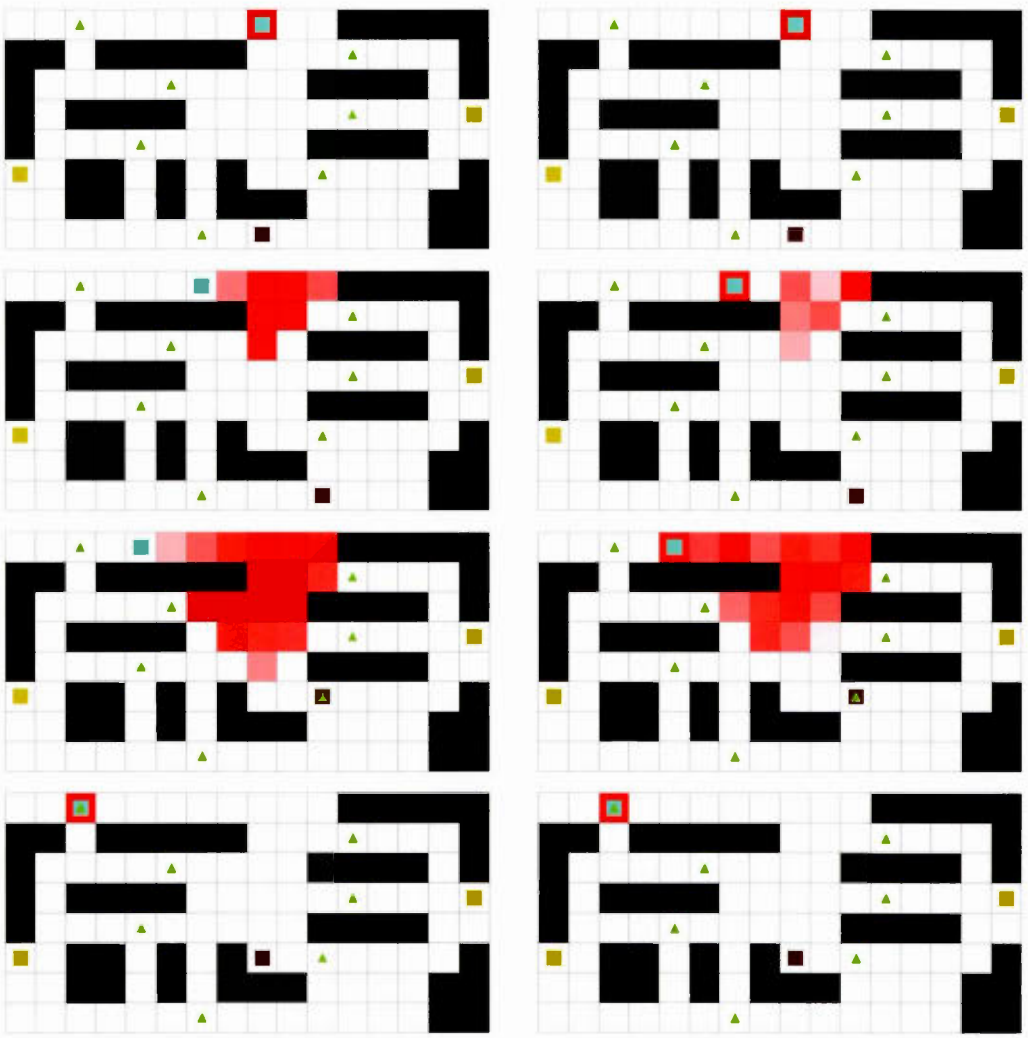

Figure 6.11 La colonne de gauche représente la première partie, celle de droite la onzième. Tours impairs de 1 à 7. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de haut en bas).

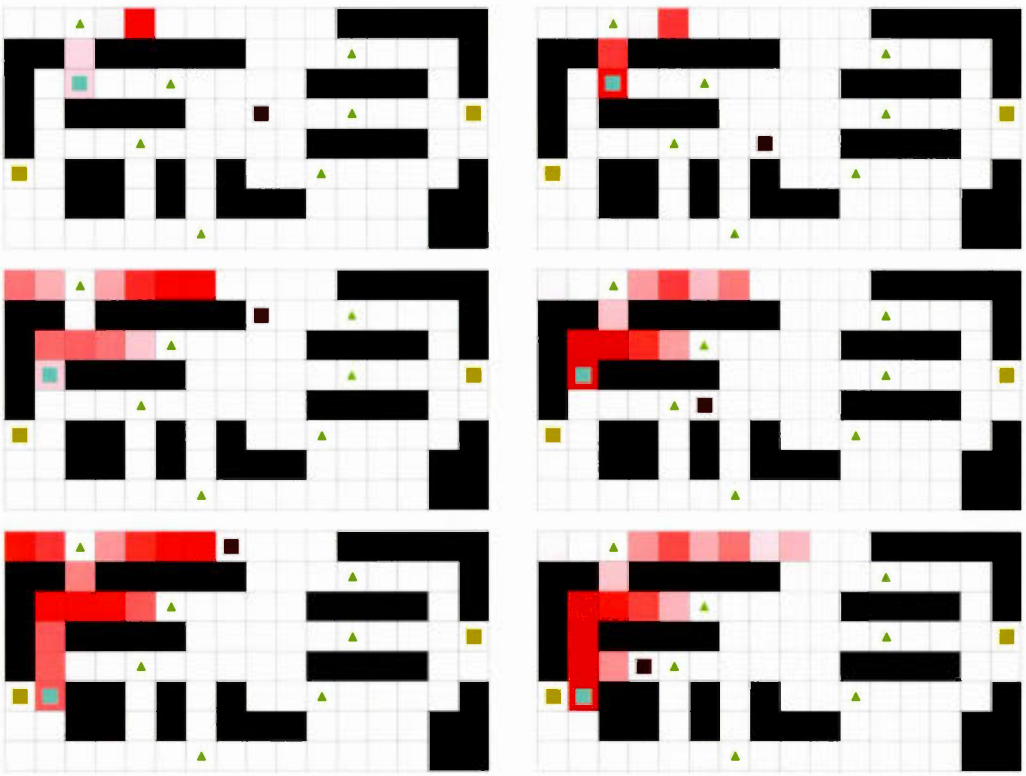

Figure 6.12 La colonne de gauche représente la première partie, celle de droite la onzième. Tours impairs de 9 à 13. Le carré bleu est le joueur, le carré bourgogne l'*IA* (À lire traditionnellement, de haut en bas).

En résumé, de nombreux outils nous ont permis d'observer l'apprentissage et de vérifier son efficacité. Nous sommes capables d'affirmer que cette méthodologie semble efficacement accomplir ce pour quoi elle fut conçue.

CHAPITRE VII

AMÉLIORATIONS POTENTIELLES

Dans cette section, nous introduirons différentes modifications qui pourraient améliorer l'efficacité de notre *IA*. Nous décrirons, en rafale, diverses améliorations potentielles. Nous allons également tenter de fournir des pistes de solutions pour permettre l'utilisation de ces modifications.

Tout d'abord, nous avons décidé de développer les algorithmes et de programmer une méthodologie plus simple, en utilisant un modèle qui n'était pas semi-markovien. Il aurait été tout de même intéressant de programmer et de vérifier l'efficacité des algorithmes de semi-Markov cachés pour effectuer cette tâche ; il s'agissait de notre idée initiale. L'utilisation de ces algorithmes aurait permis une meilleure modélisation de certaines cases, comme les objectifs, par exemple. Utiliser un modèle plus complexe permet souvent une modélisation plus réaliste, mais nous avons considéré que le gain en réalisme ne valait pas le coût de la complexité y étant relié. Le calcul de l'estimation en temps réel peut se faire tout de même très rapidement, mais l'algorithme d'apprentissage est beaucoup moins précis dans un modèle de semi-Markov. Néanmoins, nous avons observé que notre *IA* avait un comportement étrange en ce qui concernait les cases objectifs et l'utilisation d'un modèle semi-markovien aurait assurément permis à l'intelligence de mieux saisir le comportement de l'agent en ce qui a trait à ces cases.

Dans un autre ordre d'idées, modifier la notion de position aurait pu améliorer grandement la rapidité des différents algorithmes. Actuellement, une position, un état de notre chaîne de Markov, est représentée par une seule et unique case. Nous aurions pu regrouper un certain nombre de cases ensemble comme étant une seule position, réduisant ainsi le nombre d'états. Ceci permettrait d'accélérer la vitesse d'exécution tout en créant des positions plus logiques. Il n'est pas toujours intéressant de savoir la position exacte de l'agent, mais plutôt de connaître la région dans laquelle il se trouve. Pour ce faire, la solution aurait été de créer des agglomérations de positions, pour former des positions logiques, comme les couloirs ou les grands espaces rectangulaires. Cela aurait non seulement lissé la courbe de notre statistique de tests mais aurait contribué à aider l'apprentissage de stratégies plus globales. Actuellement, l'intelligence tente d'apprendre comment se déplace son adversaire, case par case. Ce processus est peut-être trop exigeant. Il serait suffisant, assurément plus simple et probablement préférable d'apprendre des déplacements plus généraux. Par exemple, au lieu de se questionner sur le chemin exact utilisé, l'*IA* pourrait se concentrer sur quel couloir l'agent emprunte et quel virage il effectue.

Introduire cette amélioration dans le modèle engendrerait un ensemble de défis. Le plus complexe serait de devoir modifier la fonction d'observation b en fonction de notre nouvelle définition d'un état. Dans notre modèle actuel, nous éliminons les cases que nous observons en leur désignant une valeur de zéro dans le vecteur d'observations. Si nous voulions utiliser des agglomérations de cases, l'*IA* pourrait simplement observer une portion d'une position. Toutefois, nous ne pourrions pas simplement éliminer cette case ; il faudrait considérer que nous avons observé une partie de celle-ci. Une solution simple serait d'utiliser une fonction d'observations similaire à celle-ci :

$$b_i(x_t) = 1 - \frac{|N_i \cap W|}{|N_i|}, \tag{7.1}$$

où N_i est le nombre de cases qui compose l'état s_i. Ici, W est défini comme à la section 4.2.1 (p. 51). Cette structure (équation 7.1) fait en sorte que si nous observons la moitié des cases qui forme l'état s_i nous éliminons, en quelque sorte, la moitié de cette position. Ce faisant, nous réduisons la probabilité que l'agent s'y trouve de moitié. Cette définition revient à supposer que si l'agent se trouve à la position s_i, il a autant de chances d'être à chacune des N_i cases qui forment cet état. Par contre, supposons que la case objectif fait partie de l'état s_j. Dans ce cas, il serait envisageable de croire que l'agent a plus de chances d'être à la case objectif qu'aux autres cases qui forment la position s_j. Nous pourrions associer un poids $P_{s_i}(l)$ à chacune des N_j cases qui forment l'état s_j tel que $\sum_{l=1}^{N_j} P_{s_i}(l) = 1$. De la sorte, nous pourrions utiliser la fonction d'observation suivante :

$$b_i(x_t) = 1 - \sum_{l=1}^{N_j} P_{s_i}(l) \times \mathbf{1}[l \in W]. \tag{7.2}$$

De cette manière, nous pourrions éliminer les positions proportionnellement à ces poids.

La prochaine amélioration potentielle provient d'une problématique que l'on retrouve dans les jeux vidéo compétitifs. Comme la plupart de ces jeux sont joués en équipe, il serait intéressant que cette méthodologie soit applicable lorsqu'il y a plusieurs adversaires et plusieurs individus les pourchassant. Pour faire cela, il faudra que nos états soient maintenant un vecteur contenant la position de chacun des joueurs. Si nous avions quadrillé notre carte en n cases, nous aurions obtenu,

dans le cas d'un seul joueur, n états. Maintenant que nous avons k joueurs, chaque joueur peut être dans une des n cases. Notre modèle aurait donc n^k états, ce qui peut être gigantesque. Pour cette raison, Hladky (Hladky, 2009) a supposé que la position des joueurs d'une même équipe était indépendante. Cette supposition est fausse puisque le positionnement de chaque joueur est extrêmement dépendant de la position des autres. En effet, dans un jeu d'équipe, les stratégies sont établies en mettant en mouvement tous les membres de l'équipe. Au sens plus large, il s'agit de la problématique d'estimation de la position de multiples agents mobiles dépendants dans un environnement borné. La suggestion précédente d'agglomération d'états pourrait ici aider à réduire le nombre d'états et peut-être rendre le modèle utilisable. Il serait sûrement possible de mettre au point une fonction proportionnelle entre le nombre de joueurs k et le nombre de cases N_i que devrait contenir chacun des états s_i. Dans ce but, plus il y aurait de joueurs, plus nous diminuerions le nombre d'états. Ceci diminuerait la précision de nos estimations, mais diminuerait aussi la valeur n^k, ce qui pourrait permettre aux algorithmes de s'exécuter efficacement.

Il serait aussi intéressant de tenter de modifier nos observations pour inclure plusieurs formes d'observations. Actuellement, nous utilisons des observations visuelles. Cependant, dans une optique plus large, nous pourrions aussi obtenir des informations sonores. Il est d'ailleurs courant, dans le contexte des jeux vidéo, d'utiliser le son pour estimer la position de ses adversaires. Une piste pour inclure le son dans le vecteur d'observations serait de créer un vecteur similaire pour le son, où l'on attitrerait une valeur de 1 par défaut aux positions et de 0 si le son nous permettrait d'éliminer avec assurance la position de l'agent. Supposons qu'un joueur entende des bruits de pas à sa droite. Il ne peut pas estimer avec précision la position de son adversaire, mais il peut éliminer un ensemble de positions dans un cône à sa gauche. Un 0 serait donc attitré à ces positions dans le

vecteur d'observations sonores. Par la suite, notre *IA* pourrait utiliser un produit termes à termes entre le vecteur d'observations visuelles et le vecteur d'observations sonores pour créer un vecteur d'observations confondues. C'est ce vecteur que l'intelligence utiliserait pour l'estimation en temps réel.

Finalement, nous croyons qu'utiliser un processus markovien d'ordre deux ou plus aurait pu être bénéfique pour notre méthodologie. Hladky (Hladky, 2009) avait conclu que cette modification n'avait pas d'impact réel sur la précision des estimations. Néanmoins, nous pensons qu'avec un ajustement approprié des paramètres, nous pourrions obtenir un énorme gain en précision. Actuellement, dans le jeu que nous avons développé, plusieurs cases servent de pivot entre deux stratégies. Certaines positions, souvent des intersections entres deux couloirs, sont très déterminantes dans la stratégie. Considérer un modèle de Markov d'ordre deux nous permettrait ainsi d'évaluer des directions plutôt que de simples positions.

En considérant les deux dernières positions, nous pouvons observer des directions grâce à nos observations. Sachant que nous désirons que notre *IA* apprenne des stratégies, le fait d'estimer la direction de l'agent et non seulement sa position ne peut être qu'informatif. Ceci ne peut qu'augmenter la précision de ses estimations. Il faut, bien entendu, revoir la définition des divers paramètres du modèle. Notons tout de même que la littérature au sujet des modèles de Markov cachés d'ordre deux existe et pourrait nous servir de base pour construire cette modification.

Quelques modifications d'ordre mathématique seraient aussi bénéfiques. Les algorithmes *EM*, étant dépendants des valeurs initiales utilisées, ne convergent pas assurément vers les bonnes valeurs. Il serait important de vérifier que la convergence s'effectue toujours correctement dans notre jeu. Nous avons tenté de guider intelligemment l'algorithme en lui fournissant des valeurs initiales logiques. Or,

la convergence n'est pas assurée. En outre, il est nécessaire de s'assurer que la matrice estimée par l'algorithme de *Baum-Welch* n'estime pas à 0 des évènements possibles. Nous considérons que les déplacements du joueur adverse sont aléatoires puisqu'il sont inconnus pour nous. Malgré tout, un joueur ne prend pas ces décisions de manière purement aléatoire : il pourrait décider de se comporter d'une certaine manière durant cent expériences puis de changer drastiquement son comportement. Il est important, dans cette situation, que nos algorithmes n'aient pas considéré que ce nouveau scénario était impossible puisqu'il ne s'est pas produit lors des cent premières expériences. La gestion de la convergence de l'algorithme de *Baum-Welch* pourrait donc faire l'objet d'améliorations. Il faudrait d'abord s'assurer que nos valeurs initiales assurent une convergence vers les bonnes valeurs, mais aussi prévenir une convergence trop drastique vers certaines valeurs absorbantes comme 0 ou 1.

Bien que cette section ne contienne que quelques modifications possibles, elle permet de mettre en évidence l'énorme potentiel de cette méthodologie. Les améliorations énumérées dans ce chapitre permettrait toutes d'améliorer l'intelligence artificielle que nous avons mise au point. Certains de ces changements amélioreraient l'efficacité des algorithmes, alors que d'autres permettraient d'augmenter la précision des estimations. Notons finalement que plusieurs autres modifications pourraient aussi être effectuées ; il ne s'agit que d'une courte liste parmi un grand nombre d'améliorations potentielles.

CONCLUSION

L'objectif de ce mémoire était de mettre au point une intelligence artificielle capable d'estimer la position d'un agent mobile dans un environnement borné. Après avoir introduit quelques concepts en lien avec les jeux vidéo et avec l'intelligence artificielle, nous nous sommes concentrés sur l'explication des outils mathématiques que nous désirions utiliser. Notre technique consistait à utiliser les chaînes de semi-Markov cachées, en considérant l'état de la chaîne comme la position de l'agent, pour effectuer cette tâche.

Ensuite, nous avons analysé et travaillé les algorithmes nécessaires pour nous assurer qu'ils pouvaient effectuer le travail nécessaire puis nous les avons programmés efficacement. Cela nous a permis de mettre au point une intelligence artificielle qui pourrait utiliser ces algorithmes dans le contexte d'un jeu vidéo. Afin de vérifier l'efficacité de notre travail, nous avons mis au point notre propre jeu vidéo. Nous avons testé notre intelligence artificielle en la confrontant à de vrais joueurs. De la sorte, nous avons obtenu une grande quantité de données.

Nous avons finalement mis au point une statistique de tests et de nombreux graphiques afin de visualiser nos résultats. Nous avons constaté une efficacité de l'intelligence artificielle bien au-delà de nos attentes. Non seulement les calculs s'exécutent assez rapidement pour pouvoir utiliser cette *IA* dans de multiples environnements, mais la précision des estimations est plus que satisfaisante. De plus, malgré la nécessité d'ajuster divers paramètres, l'apprentissage dont a été dotée l'intelligence est un puissant outil. Nous avons été surpris par la force de cette

capacité d'adaptation qui a fait de cette intelligence une adversaire de taille à tout joueur dans le contexte du jeu vidéo.

Finalement, ce travail de recherche a permis de démontrer la puissance de divers modèles de Markov dans la programmation d'intelligence artificielle. Nous espérons que ce mémoire ouvrira la voie à une implication plus grande des chercheurs en statistique dans le domaine de l'intelligence artificielle.

APPENDICE A

ALGORITHMES

A.1 Algorithme *forward*

```
#include <vector>
#include <boost/numeric/ublas/matrix.hpp>
#include <numeric>

#include "operationVector.h"

using namespace std;
using namespace boost::numeric::ublas;

matrix<double> Alpha(matrix<double> Obs, matrix<double> M
, std::vector<double> pie) {

//  Obs est la matrice des observations
//  M est la matrice de transition
//  pie est le vecteur de probabilite initiale

const int m = Obs.size2(); // Nombre detat, de cases
```

```
const int n = Obs.size1(); // Nombre dobservation, duree de la partie


matrix<double> Alpha(n, m);
auto alph = std::vector<double>(m, 0);
d'un vecteur de longueur m remplie de 0

for (int i = 0; i < m; ++i) {
  alph[i] = pie[i] * Obs(0, i);
}

double log1surc = log(sum(alph));

alph = scalarMultiplication(alph, 1.0 / sum(alpha));

for (int i = 0; i < m; ++i) {
    Alpha(0, i) = log(alph[i]) + log1surc;
}

auto Alph = matrix<double>(1, m);

for (int i = 0 ; i < m ; ++i) {
   Alph(0, i) = alph[i];
}

for (int i = 1; i < n ; ++i) {
  Alph = prod(Alph, M);
```

```
  for (int j = 0; j < m ; ++j) {
    Alph(0, j) = Alph(0, j) * Obs(i, j);
  }

  log1surc += log(sum(Alph));
  Alph /= sum(Alph);

  for (int j = 0; j < m ; ++j) {
      Alpha(i, j) = log(Alph(0, j)) + log1surc;
  }

}
return Alpha;
}
```

A.2 Algorithme *backward*

```
#include <vector>
#include <boost/numeric/ublas/matrix.hpp>
#include <numeric>

#include "operationVector.h"

using namespace std;
using namespace boost::numeric::ublas;

matrix<double> Beta(matrix<double> Obs, matrix<double> M,
std::vector<double> pie) {

//  Obs est la matrice des observations
//  M est la matrice de transition
//  pie est le vecteur de probabilite initiale


const size_t m = Obs.size2(); // Nombre detat, de cases
const size_t n = Obs.size1(); // Nombre dobservation, duree de la partie


matrix<double> Beta(n, m);
matrix<double> bet(1,m);

fill(bet, [&](int l, int c){
```

```
  return 1.0/m;
});

for (int i = 0; i < m; ++i) {
  Beta(n-1, i) = 0;
}

double log1surc = log(sum(bet));
matrix<double> Bet{1, m};

matrix<double> Tligne = trans(ligne(Obs,n-1));
Bet = prod(M,Tligne);
Bet = trans(Bet);

for (int i = 0; i < m; ++i) {
  Beta(n-2, i) = log(Bet(0,i))+log1surc;
}

log1surc = log1surc + log(sum(Bet));

Bet /= sum(Bet);

for (int i = 3; i < n+1 ; ++i) {
  Tligne = trans(ligne(Obs,n-i+1));
  for (int j = 0; j < m ; ++j) {
    Tligne(j, 0) = Tligne(j, 0) * Bet(0, j);
  }
  Bet = prod(M, Tligne);
```

```
  Bet = trans(Bet);
  for (int j = 0; j < m ; ++j) {
    Beta(n-i, j) = log(Bet(0, j)) + log1surc;
  }

  log1surc += log(sum(Bet));

  Bet /= sum(Bet);
}
return Beta;
}
```

A.3 Algorithme de *Baum-Welch*

```
#include "bw.h"

#include <vector>
#include <boost/numeric/ublas/matrix.hpp>
#include <numeric>

#include "operationVector.h"
#include "alpha.h"
#include "beta.h"

using namespace std;
using namespace boost::numeric::ublas;

matrix<double> BW(std::vector<matrix<double>> C, matrix<double> Mini
, std::vector<double> pie, int iter) {

//  C est la les des matrices d<observations
//  Mini est la matrice de transition initiale
//  pie est le vecteur de probabilite initiale
//  m est le nombre detat
//  iter, le nombre diteration desire
if (C.size() > 0) {
  matrix<double> theRealMini = Mini;
  const int m = C[0].size2();
```

```
const int nbTours = C.size(); // Nombre de parties joués
auto nouvMm = std::vector<matrix<double>>(nbTours, matrix<double>(m,m));

for (int k=0 ; k < iter; ++k) {
  matrix<double> nouvM = ( 0.999 * Mini + 0.001 * theRealMini ) ;
  for (int o=0; o < nbTours; ++o) {

  int n = C[o].size1();
  auto lalpha = Alpha(C[o],nouvM,pie);
  auto lbeta = Beta(C[o],nouvM,pie);
  double p = getMax(ligne(lalpha,n-1));
  double logl = p + log(sum(expMat(ligne(lalpha,n-1)-p)));

  for (int i=0; i <m; ++i) {
    for (int j=0; j<m; ++j) {
      double grostruc = 0;
      for (int ind=0; ind <n-1; ++ind) {
        grostruc = grostruc+ exp(lalpha(ind,i)
        +C[o](ind+1,j)+lbeta(ind+1,j)-logl);
      }
      nouvMm[o](i,j) = Mini(i,j) * grostruc;
    }
  }

}

nouvM = nouvMm[0];
```

```
  for (int o=1; o< nbTours; ++o) {
    nouvM = nouvM + nouvMm[o];
  }

  for (int j=0; j<m; ++j) {
    double sumLigne = sum(ligne(nouvM,j));
    if (sumLigne == 0) {
      for (int i = 0; i <m; ++i) {
        nouvM(j,i) = Mini(j,i);
      }
    } else {
      for (int i = 0;  i <m; ++i) {
        nouvM(j,i) = nouvM(j,i) / sumLigne;
      }

    }

  }
  Mini = nouvM;
}
}
return Mini;
}
```

RÉFÉRENCES

Arulampalam, S., Maskell, S., Gordon, N. et Clapp, T. (2002). A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking. *IEEE transactions on signal processing*, 50(2), 174-188

Baum, L., Petrie, T., Soules, G. et Weiss, N. (1970). A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1), 164-171

Bererton, C. (2004). *State Estimation for Game AI Using Particle Filters : Proceedings of the AAAI 2004 Workshop on Challenges in Game AI.* Actes du colloque, 25-26 juillet 2004, San Jose, États-Unis. (p. 36-40) Pittsburgh (Penn.) : AAAI Press.

Durbin, R., Eddy, S., Krogh, A. et Mitchison, G. (1998). *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids.* Cambridge (U.K.) : Cambridge University Press.

Hladky, S. et Bulitko, V. (2008). *An Evaluation of Models for Predicting Opponent Positions in First-Person Shooter Video Games : IEEE Symposium on Computational Intelligence and Games (CIG'08).* Actes du colloque, 15-18 décembre 2008, Perth, Australia. Récupéré de http ://www.csse.uwa.edu.au/cig08/Proceedings/papers/8071.pdf

Hladky, S. (2009). *Predicting Opponent Locations in First-Persone Shooter Video Games.* (Mémoire de maitrise). Edmonton. University of Alberta.

Murphy, K. P. (2002). *Hidden semi-Markov models (HSMMs).* [Document non publié]. University of California at Berkley.

Rabiner, L. R. (1989). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2), 257-286

Rosenthal, J. (1995). Minorization Conditions and Convergence Rates for Markov Chain Monte Carlo. *Journal of the American Statistical Association*, 90, 558-566

Ross, S. (2006). *A first course in probability* (7^{e} éd.). Upper Saddle River (N.J.) : Prentice Hall

Ross, S. (2009). *Introduction to Probability Models* (10^{e} éd.). San Diego (Cal.) : Academic Press

Shen, D. (2008). *Some Mathematics for HMM.* [Tutoriel]. Récupéré de http ://courses.media.mit.edu/2010fall/mas622j/ProblemSets/ps4/tutorial.pdf

Southey, F., Loh, W. et Wilkinson, D. (2007). *Inferring complex agent motions from partial trajectory observations : International Joint Conference on Artificial Intelligence (IJCAI)* Actes du colloque, 6-12 janvier 2007, Hyderabad, India. (p. 2631-2637). Menlo Park (Cal.) : AAAI Press

Zucchini, W. et MacDonald I. (2009). *Hidden Markov Models for Time Series : An Introduction Using R* (2^{e} éd.). Boca Raton (Flo.) : Chapman and Hall/CRC