CINQ-MARS PATRICK

APPRENTISSAGE D'UNE POLITIQUE DE GESTION DE RESSOURCES EN TEMPS RÉEL : APPLICATION AU COMBAT MARITIME

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès sciences(M.Sc.)

Faculté des sciences et de génie
UNIVERSITÉ LAVAL
QUÉBEC

2010

Résumé

Dans le secteur de la défense, la majorité des processus de combat soulèvent des problèmes complexes, comme l'allocation de ressources. Le projet NEREUS cherche des méthodologies de planification et d'exécution de stratégies pour le centre de commandement et de contrôle (C2) d'une frégate canadienne.

L'approche par apprentissage proposée dans ce mémoire permet de constater que de nouvelles stratégies découvertes par expérimentation peuvent se comparer à des doctrines préalablement conçues. Les processus de decision de Markov ont été utilisés pour construire un cadre de développement et modéliser des agents capables d'agir dans des environnements en temps réel. Un agent basé sur l'apprentissage par renforcement a donc été évalué dans un environnement de simulation de combat maritime et un formalisme issu du Weapon-Target Assignment a été utilisé pour définir le problème en temps réel.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Chapitre premier

1.1 Introduction

Instinctivement, l'Homme s'applique dans la recherche de méthodes et de concepts afin d'améliorer l'efficacité de son travail. C'est par son désir de toujours mieux accomplir les tâches essentielles que l'esprit d'innovation est né chez lui. Doté d'une faculté de raisonnement unique, il a peu à peu développé des processus permettant d'optimiser le savoir faire et ainsi réaliser de grandes innovations.

Dans pratiquement tous les domaines, l'ingéniosité humaine permet de mettre au point des méthodologies intelligentes, repoussant chaque jour les limites de la technologie. L'évolution technologique a tôt fait de voir naître des systèmes capable de réaliser des tâches que les hommes eux-même ne sauraient réaliser aussi efficacement.

Seule l'intelligence peut engendrer autant d'exploits. L'intelligence est si fascinante que plusieurs rêvent que l'Homme puisse un jour créer une intelligence artificielle à son image.

1.1.1 Historique

On doit le nom de Intelligence Artificielle (IA) à John McCarthy qui, en 1959, rassembla plusieurs chercheurs du domaine de l'IA à une conférence de deux mois sur le sujet. Une dizaine des plus importantes figures dans le domaine ont répondu à l'appel et se sont rencontrés à Dartmouth pendant l'été 59. Bien que l'intelligence artificielle

date de la Deuxième Guerre mondiale, ce n'est qu'après cette rencontre qu'elle sera baptisée telle qu'on la connaît aujourd'hui, sous le nom de IA.

L'intelligence artificielle est une science toute jeune, fondée sur des concepts provenant de plusieurs autres sciences telles que la philosophie, la psychologie, l'économie, la théorie du contrôle, les neurosciences, les mathématiques, etc. Issus de secteurs très variés, les chercheurs n'ont pas tous exactement la même vision ou définition de l'IA. Certains proposent que celle-ci soit vue comme la science voulant reproduire le comportement humain, d'autres cherchent plutôt à s'approcher de la rationalité, que ce soit au niveau du raisonnement ou des actions.

Les champs d'études de l'intelligence artificielle se regroupent sous plusieurs branches, dont l'apprentissage automatique. Les autres branches telles que le langage naturel, les jeux, les diagnostiques ou la robotique ne seront pas couvertes par ce mémoire.

En IA, l'apprentissage consiste à faire évoluer le raisonnement ou le comportement d'un système dans le temps. Les systèmes pourvus d'une notion d'apprentissage utilisent les informations fournies pour établir ou améliorer leur manière d'agir. L'apprentissage se divise en plusieurs catégories, différenciées par la manière dont l'information est transmise au système et ensuite utilisée. Ce sont l'*apprentissage supervisé* et l'*apprentissage par renforcement* qui sont les plus étudiés. Parmi les autres méthodes, se trouvent l'apprentissage non-supervisé et semi-supervisé.

Pour utiliser une technique d'apprentissage supervisé, on fournit à l'algorithme un échantillon de valeurs en entrée et on donne la correspondance de ces valeurs avec une classe (ou valeur) en sortie. Dans cette catégorie d'apprentissage, on retrouve donc majoritairement des problèmes de classification. L'algorithme devra alors construire une fonction de classification permettant d'associer les valeurs en entrée avec leur classe respective en sortie. Pour y arriver, l'algorithme aura besoin de s'entraîner sur un large échantillon de données. La séance d'apprentissage est suivie d'une évaluation où on ne fournit que les valeurs en entrée et on compare alors la classification donnée par l'algorithme avec les valeurs de sortie espérées.

L'apprentissage semi-supervisé est très similaire. La différence réside dans le fait que certaines valeurs d'entrée ne sont pas accompagnées d'une classe correspondante en sortie. Dans ce cas, on utilise généralement un échantillon composé majoritairement de valeurs ayant une classe inconnue. L'apprentissage non-supervisé est à l'extrême, n'ayant aucune valeur de sortie spécifiée.

Pour l'apprentissage par renforcement, il ne s'agit plus de classes, mais d'actions.

Au lieu d'un ensemble de classes auxquelles doivent correspondre les entrées, il s'agit de chercher l'action qu'il serait souhaitable de prendre. Cependant, l'action ainsi choisie s'insère dans une séquence qui produit alors un comportement. Le principe associé à cette catégorie d'apprentissage est que l'on renforce un comportement souhaité par une récompense. L'apprentissage par renforcement s'effectue donc par essais-erreurs, où le comportement s'ajuste en fonction des récompenses au fur et à mesure que l'agent prend de l'expérience.

1.1.2 Agents

La notion d'interaction ainsi introduite dans l'apprentissage nécessite un cadre. L'algorithme d'apprentissage devient alors le processus de décision d'un agent faisant partie d'un environnement, qu'il perçoit et dans lequel il agit.

Un agent est une entité interagissant dans un environnement et pouvant effectuer une tâche qu'on lui a assignée. Un agent se charge donc d'effectuer des actions en conjonction avec un but fixé. Par exemple, un agent immobilier est un agent ayant pour but de vendre ou acheter des maisons. Sa tâche est donc d'interagir avec les acheteurs et vendeurs concernés, ou encore d'autres agents, afin de guider son client vers une transaction souhaitable. Il en est de même pour l'agence de voyage. Plus encore, un thermostat est un agent en ce sens qu'il a pour tâche d'assurer le maintient de la chaleur dans une pièce. Il interagit donc avec son environnement à l'aide d'un capteur de température et d'un circuit électrique qu'il a le loisir d'ouvrir ou fermer. Un agent logiciel effectue des tâches informatiques pour un utilisateur ou un autre agent logiciel. Par exemple, un moteur de recherche comme Google peut être vu comme un agent ; il effectue l'indexation et la recherche des pages Web correspondant à la requête de l'internaute.

Certains agents informatiques sont des logiciels qui peuvent également être utilisés pour résoudre des problèmes mathématiques. Le problème en question devient l'environnement de l'agent et celui-ci a pour tâche de lui trouver une solution.

L'*autonomie* est ce qui caractérise un agent. Un agent n'en est pas un s'il ne peut pas effectuer des tâches seul. Bien entendu, une telle caractéristique est nécessaire à l'apprentissage par renforcement. L'agent est donc laissé à lui-même et le résultat de ses actions est ensuite observé. L'apprentissage de l'agent se fait au moyen du renforcement. En effet, sans une notion du "bien" et du "mal", l'agent ne peut faire la part des choses entre de *bonnes* et de *mauvaises* actions et conclure sur la légitimité de celles qu'il effectue. En lui acheminant une récompense plus ou moins grande selon la pertinence

de l'action, l'agent peut alors identifier si son comportement est bon ou mauvais.

La figure 1.1 montre la décomposition en composantes d'un agent pouvant apprendre dans un environnement donné. On peut voir sur cette figure que l'agent interagit avec l'environnement. D'un côté l'agent reçoit un signal x provenant de l'environnement tandis que ce dernier reçoit le signal a correspondant à l'action prise par l'agent.

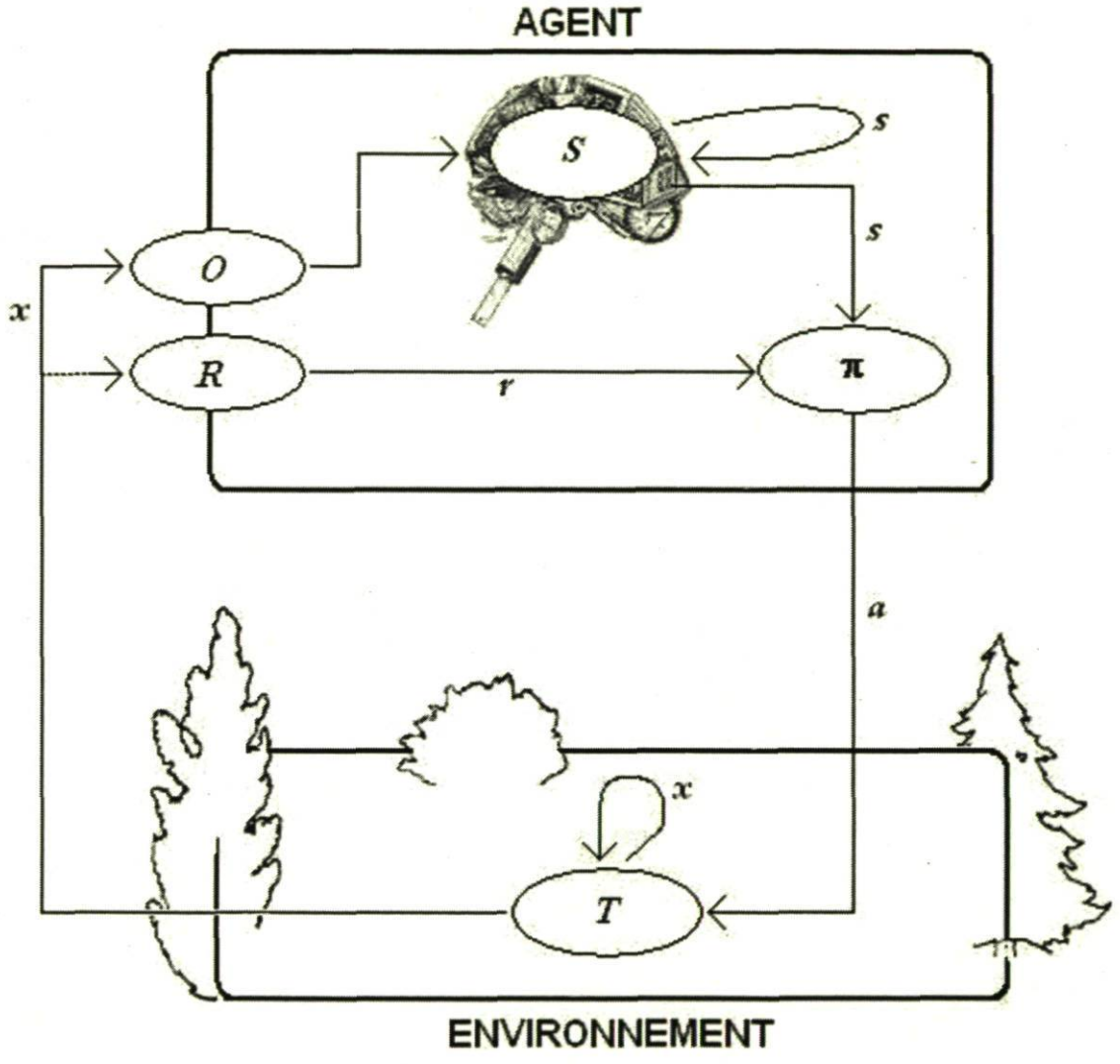

FIGURE 1.1 – Vue d'ensemble d'un agent dans son environnement.

Le signal x est un vecteur de caractéristiques décrivant l'environnement. Ce signal est perçu par la fonction d'observation $O()$, qui transforme ce signal en une observation o. L'observation o est la perception de l'agent. Elle est passée ainsi que l'état courrant à la fonction $S()$, générant l'état de croyance de l'agent.

L'agent recoit un signal de renforcement via la fonction $R()$. Ce signal provient

généralement de l'environnement, mais peut être attribué par une tierce personne ou être défini implicitement par le concepteur.

Ensuite, l'état courant s et la récompense immédiate r sont utilisés par l'algorithme de l'agent pour produire une action. Le processus de décision de l'agent est représenté ici par la fonction $\pi()$, retournant le signal a correspondant à l'action choisie.

1.1.3 Environnement

Une action a est produite dans l'environnement, lequel est régi par des règles et normes. Par exemple, notre environnement naturel est régi, entre autres, par les règles de la physique. L'environnement passe d'un état x_t à un état x_{t+1} selon un modèle de transition quelconque, que nous représentons par $T()$.

Il existe plusieurs environnements possibles pour un agent. Ceux-ci sont tous différents et suivent une dynamique différente également. Un échiquier en est un exemple, de même qu'un terrain de soccer ou un environnement virtuel tel que l'Internet.

L'environnement a des caractéristiques ; on peut le qualifier selon sa dynamique dans le temps. Prenons l'exemple de l'échiquier : le jeu reste le même tant et aussi longtemps qu'un joueur n'a pas joué. On dira que cet environnement est *statique*. D'un autre coté, si les joueurs d'un terrain de soccer ne font pas d'action, le ballon, lui, continue de rouler. L'environnement est alors *dynamique*.

L'environnement peut être stochastique ou déterministe. Il est dit *déterministe* lorsque les actions ont un impact bien défini dans l'environnement et que l'agent connaît et peut prévoir avec certitude le prochain état peu importe l'action exécutée. Dans ce cas, la fonction de transition $T()$ peut retourner un seul état en sortie par couple état-action en entrée. D'autre part, si $T()$ peut retourner plusieurs états différents en sortie pour une même valeur d'entrée, l'agent ne peut prévoir le prochain état, car plusieurs résultats sont possibles suite à l'action dans l'état courant. Il y a donc une certaine probabilité pour chaque état que ce dernier soit le suivant. Lorsque les états suivant sont incertains et suivent une certaine distribution de probabilité, nous dirons que l'environnement est *stochastique*.

Du point de vue de l'agent, l'observabilité de l'environnement est un facteur important. Elle peut être totale ou partielle. Dans le premier cas, l'agent détient implicitement dans chaque observation obtenue de $O()$ tous les éléments nécessaires pour identifier l'état x correctement. En bref, un jeu d'échecs est *totalement observable* parce que

l'agent connaît l'emplacement de toutes les pièces du jeu et c'est tout ce qu'il a besoin de savoir pour jouer. Quant au second cas, il implique que certains éléments ne soient pas observés à chaque instant. C'est donc dire que l'agent n'est pas capable d'avoir à la fois toutes les informations requises pour identifier clairement l'état dans lequel il se trouve.

Imaginez un homme ou une femme que l'on place quelque part dans un labyrinthe. La personne en question ne connaît pas son emplacement, mais dispose d'un plan du labyrinthe. Les murs du labyrinthe sont trop haut pour qu'il lui soit possible de voir autre chose que la configuration du couloir dans lequel elle se trouve. Même si cette personne peut observer tout son entourage, il lui manque des éléments pour connaître exactement une position exacte sur le plan. Afin d'identifier son emplacement, elle devra se déplacer dans le labyrinthe en comparant les murs observés avec ceux sur le plan. Après avoir fait quelques détours, la personne pourra certainement avoir une certaine idée de son emplacement ; du moins, elle aura ciblé sur la carte quelques endroits où les murs concordent avec ceux observés et où il est probable qu'elle se trouve. Ceci est un bon exemple d'environnement *partiellement observable.*

Lorsqu'un environnement est totalement observable, l'observation obtenue de la fonction d'observation $O()$ a une relation bijective avec un état de S ; on observe alors directement l'état. Dans un environnement partiellement observable, on a une probabilité d'observer o :

$$Pr(O(x, s) = o)$$

Tout comme la personne dans le labyrinthe, l'agent a alors des états dans lesquels il est probable qu'il se trouve. L'agent utilise une croyance sur les états, que l'on appelle *belief state*, pour représenter son état courrant. Il s'agit d'un vecteur de probabilité par état. La fonction S() générant l'état suivant devient alors beaucoup plus complexe.

1.1.4 Complexité

La complexité est une notion bien définie en informatique. On mesure la complexité d'un problème par le temps qu'il faut pour le résoudre, relativement à la taille de l'instance à traiter. Par exemple, lorsqu'il s'agit de trier une liste d'éléments, la taille de l'instance est le nombre d'éléments que contient cette liste. Généralement, plus la taille de l'instance est grande, plus le temps de calcul est long. Le temps de calcul ne se mesure pas en secondes, mais en instructions. En effet, les processeurs dans nos ordinateurs ont des fréquences de calcul différentes. Lorsque cette fréquence est élevée,

un processeur peut effectuer plus d'instructions dans un même lapse de temps qu'un autre processeur moins puissant. Un même algorithme sera exécuté en plus ou moins de secondes selon la vitesse de calcul du processeur. Il ne convient donc pas d'utiliser les secondes pour représenter le temps d'exécution. Les instructions, pour leur part, sont toujours nécessaire. Cependant, d'un langage de programmation à un autre et également d'un programmeur à l'autre, plus ou moins d'instructions peuvent être utilisées pour effectuer le même traitement. C'est pourquoi nous ne tenons compte que de l'ordre de grandeur de la complexité d'un algorithme.

On note $O()$ (grand-ô) le temps d'exécution au pire cas d'un algorithme. Pour $O(n)$, il faut 10 fois plus de temps pour traiter 100 valeurs qu'il n'en faut pour 10 valeurs. L'algorithme a alors une complexité linéaire en n. Un algorithme en $O(n^2)$ prend 100 fois plus de temps à traiter 100 valeurs que 10 valeurs. Pour notre exemple de tri d'une instance de n éléments, le temps d'exécution le plus court possible est de l'ordre de $O(n\log n)$, c'est-à-dire qu'on doit, dans le pire des cas, exécuter $n\log n$ instructions pour compléter le tri. Les constantes ont été enlevées, car il se peut que le véritable temps soit $3n\log n + C$, mais puisqu'on cherche un ordre de grandeur sur n, ces constantes sont négligeables.

Il y a plusieurs degrés de complexité. Ainsi, $O(3^n)$ est plus complexe que $O(n^3)$, si n s'approche asymptotiquement de l'infini. Dans le premier cas, le temps d'exécution croît de façon exponentiel si n augmente. Dans le second cas, l'algorithme est de complexité polynomiale. Les algorithmes dont la complexité est un polynôme de n ont un temps d'exécution que l'on qualifie de satisfaisant. En règle générale, les algorithmes de complexité exponentielle deviennent vite très long à exécuter, car le temps d'exécution croît de façon déraisonnable.

En analyse d'algorithmes, les problèmes polynomiaux appartiennent à la classe *P*. La classe *NP* est celle qui regroupe les problèmes dont on ne connaît pas encore d'algorithme polynomial, mais pour lesquels nous pouvons vérifier en temps polynomial si une solution au problème est exacte. Parmi les problèmes de la classe *NP*, il y a les *NP*-complets.

Les problèmes *NP*-complets sont les problèmes les plus difficiles. La notion de *NP*-complétude a été introduite par Steven Cook[Cook(1971)]. Ce dernier a fait la preuve que le problème de satisfiabilité, communément appelé **SAT**, fait partie de la classe des problèmes *NP*-complets.

Pour prouver qu'un problème est dans la classe des *NP*-complets, on doit réduire celui-ci et déterminer qu'il est équivalent à un autre algorithme *NP*-complet [Karp(1972)].

Tous les problèmes officiellement *NP*-complets ont probablement tous été réduits jusqu'au problème **SAT**, prouvé par Cook.

Par exemple, une des méthodes de cryptographie les plus utilisées, soit le RSA, est un algorithme qui encode l'information avec deux clés, une clé publique et une clé privée. Le message envoyé entre deux sujets est décodable par un algorithme simple. Cependant, l'intercepteur du message, bien qu'il puisse détenir tous les éléments pour décoder le message, est confronté à un algorithme *NP*-complet. Il s'agit en fait de trouver les facteurs premiers d'un nombre pour pouvoir décoder le message. Le problème, c'est que la factorisation est une opération si laborieuse que même un ordinateur doit mettre plusieurs années pour trouver les facteurs premiers d'un grand nombre. Si une personne pouvait résoudre ce problème de factorisation en un temps raisonnable, elle pourrait alors frauder n'importe quelle banque sécurisée par cryptographie RSA. Heureusement, la banque a le temps de changer ses codes plusieurs fois avant que n'importe quel ordinateur puisse en décoder un seul.

Il convient donc de trouver des alternatives pour les problèmes *NP*-complets. Parfois, il s'agit de trouver une solution moins bonne, mais très près de la solution optimale. En autant que cette solution soit satisfaisante. Parfois, il est préférable d'insérer des heuristiques dans la résolution du problème. Par exemple, pour la recherche d'une route entre deux points, un expert sur le terrain pourrait dire que tel et tel secteurs ne doivent pas être considérés. Malgré qu'il puisse exister un passage optimal traversant ces secteurs, l'espace de recherche pourrait être limité de façon à ce que l'agent trouve plus rapidement une route satisfaisante. Dans certains contextes, on pourrait changer les paramètres d'un algorithme ou construire un modèle qui sera un peu plus spécifique a un problème, dans le but d'améliorer la solution ou le temps de calcul.

Chapitre 2

Combat maritime

Le secteur de la défense militaire est présent dans la plupart des grands pays. Il fait face à des défis technologiques importants, touchant une multitude de facette du combat. La défense militaire recherche des moyens de contrer les hostilités et menaces provenant de l'ennemi. Les chercheurs dans ce domaine sont constamment confrontés aux dernières technologies et à des méthodes de combat qui peuvent évoluer à tout moment.

2.1 Contexte

Dans plusieurs situations de combat, lorsqu'on se trouve sous les feux de l'ennemi, les décisions doivent être prises en peu de minutes, parfois même en quelques secondes. Les scénarios peuvent être complexes et il faut souvent tenir compte de plusieurs facteurs pour établir un bon plan de défense. Dans certaines situations, les possibilités de réponse à une attaque sont assez nombreuses pour qu'il soit impossible à un cerveau humain de toutes les considérer. Dans de pareils scénarios, il est alors envisageable de faire appel à des systèmes d'aide à la décision. C'est donc un algorithme qui devra trouver le meilleur plan d'actions de la défensive.

Dans le combat maritime, les navires militaires sont susceptibles d'être la cible de missiles en provenance d'avions, d'hélicoptères, de sous-marins, ou encore de la côte. Bien entendu, ces navires possèdent des moyens de défense. Ils ont un certains nombre de ressources qu'ils peuvent déployer pour contrer les menaces. Cependant, les ressources sont limitées et il peut y avoir plusieurs menaces à la fois. Le centre de commandement

et de contrôle (C2, *Command and Control*) du navire est chargé de choisir les ressources qu'il convient d'allouer à chacune des menaces.

Ce problème d'allocation des ressources est complexe. Le C2 est contraint par le temps et les ressources, et celui-ci doit livrer une solution permettant d'assurer la survie du navire. Les navires militaires et les ressources à bord de ces plates-formes sont très coûteux. Puisque les ressources sont limitées, il est possible qu'une plate-forme navale manque de ressources si elle est attaquée à plusieurs reprises, surtout si les ressources sont utilisées sans réserve et que la mission est longue. Il faut donc non seulement minimiser les dommages des navires, mais également l'utilisation des ressources.

Le centre de commandement et de contrôle d'une plate-forme maritime est un noyau décisionnel qui doit gérer plusieurs processus interreliés, lui permettant de décider et de planifier les actions défensives à adopter au cours des scénarios de combat. Le schéma relationel des processus d'un C2 pour la planification des engagements est représenté dans la figure 2.1. Comme le montre cette figure, le fonctionnement d'un C2 sous-tend plusieurs processus, touchant l'acquisition, l'engagement, l'allocation des ressources, etc. Les relations entre les processus sont illustrées par les flèches.

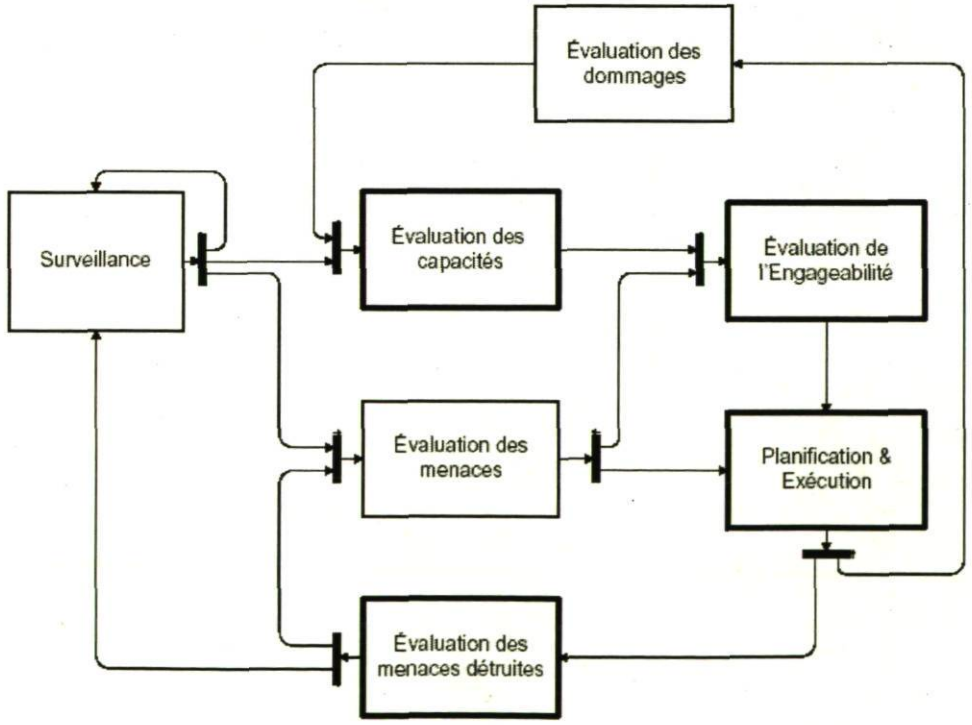

FIGURE 2.1 – Les processus composant le schéma décisionel d'un C2.

Le premier processus de la figure 2.1 concerne la surveillance. L'étape initiale d'une prise de décision est sans doute la prise de conscience d'une menace. La surveillance est le processus par lequel le C2 prend contact avec l'environnement pour trouver des indices annonçant qu'il faut amorcer les processus suivants, soit l'évaluation des capacités et l'évaluation des menaces. L'évaluation des capacités cherche à établir la liste des ressources disponibles pour une éventuelle défense. L'évaluation des menaces touche l'identification et la description des forces qui menacent la plate-forme. Ces informations sont entre autres cruciales pour déterminer la dangerosité des menaces. Le processus d'évaluation de l'engageabilité étudie la faisabilité des assignations que peut faire une plate-forme. L'engageabilité ainsi obtenue limite le nombre d'actions possibles et peut influencer grandement le processus de planification. La planification et l'exécution forment le processus par lequel le C2 ordonne et assigne les ressources afin que l'exécution soit la plus efficiente possible. L'excution du plan d'action du C2 donne des résultats qui seront finalement constatés dans les processus d'évaluation des dommages et d'évaluation des menaces détruites avant que le cycle décisionel ne se continue.

Chaque processus comporte ses particularités et ses problèmes. Les algorithmes voulant modéliser le C2 d'une plate-forme de combat couvrent rarement la totalité du processus décisionnel. La plupart du temps, un seul processus est évoqué à la fois, ne ciblant géréralement qu'un type de problème particulier. Le contenu de ce mémoire se concentre principalement sur le processus de *Planification & Exécution.*

Le processus de Planification et Exécution englobe des problèmes difficiles. Parmi eux, l'allocation des ressources est un problème fort difficile qui revient dans plusieurs domaines. Dans le domaine de la défense, il est mieux connu sous le nom de *Weapon-Target Assignment* (WTA).

2.2 Weapon-Target Assigment

Le WTA consiste à trouver les meilleures assignations de ressources aux menaces qui risquent d'atteindre la plate-forme. Il s'agit donc d'un problème d'allocation de ressources.

Avant d'être formalisé concrètement sous le nom de *Weapon-Target Assignment*, ce problème faisait déjà l'objet de plusieurs recherches dans les années soixante. La plupart des différents modèles qui ont été sujet d'articles sont expliqués et détaillés dans un résumé de Matlin paru en 1970 [Matlin(1970)]. Ces modèles abordent l'allocation des

ressources sous un angle propre à chacun, avec des hypothèses parfois simples, ou parfois complexes. Plus tard, plusieurs autres chercheurs se sont penchés sur ce problème qui reste d'actualité, intéressant encore aujourd'hui bon nombre de personnes.

Ainsi que mentionné précédemment, le C2 doit tenir compte de plusieurs facteurs dans sa prise de décision. De la même manière, les algorithmes ne peuvent ignorer ces facteurs. Cependant, tous les modèles ne considèrent pas ces facteurs de la même façon. Effectivement, certains seront plus axés sur la probabilité de détruire une menace, d'autres les types de ressources, ou encore les temps d'assignation, etc. Autant de paramètres différents peuvent changer la vision du problème WTA. Ainsi, plusieurs façons de représenter et résoudre le problème ont été présentées. Les modèles les plus simples représentent les assignations de façon déterministe, de même que les impacts. Bref, une ressource qu'on utilise contre une menace est assurée de toucher la cible et de la détruire. De façon plus réaliste, il est naturel d'utiliser un modèle stochastique pour représenter l'incertitude de l'environnement. Il convient donc d'utiliser une probabilité P_{ij} qu'une menace i soit éliminée par une ressource j lors d'une assignation.

Lors d'une conférence à *Princeton University*, Mr. Flood décrit le problème d'assignation de ressource similairement au problème d'assignation de personnel. Le problème consiste à minimiser :

$$\sum_{i=1}^{i=m} v_i \sum_{j=1}^{j=n} (1 - P_{ij} A_{ij})$$

où v_i est la valeur de la cible i. A_{ij} est la probabilité que l'arme j soit utilisé contre la menace i. L'assignation en tant que telle n'est pas probabiliste, alors la valeur de A_{ij} est soit 0 ou 1. C'est donc dire que A est la matrice d'assignation.

Bien entendu, la nature non linéaire de ce problème le rendait impraticable à la programmation linéaire. Peu de temps après, Manne simplifia légèrement le modèle afin de le rendre admissible à une méthode de programmation linéaire déjà appliquée à l'époque pour un problème de transport [Manne(1958)]. Pour que cette simplification fonctionne, il faut assumer que la probabilité de détruire une menace i est la même pour tous les types d'armement, c'est-à-dire que $P_i = P_{ij}$ pour tout j. Ceci dit, la fonction à minimiser est la suivante sachant que $A_{ij} \in \{0, 1\}$:

$$\sum_{i=1}^{i=m} v_i \, (1 - P_i)^{\sum_{j=1}^{j=n} A_{ij}}$$

De son coté, Den Broeder emploie les mêmes suppositions pour appliquer son algorithme

[G. G. DenBroeder and Emerling(1958)]. Celui-ci consiste à minimiser la différence entre le nombre d'assignations pour chaque menace.

En 1986, Lloyd et Witsenhausen réussissent à faire la preuve que le problème de WTA est un problème *NP*-complet [Lloyd and Witsenhausen(1986)]. Ceci s'applique même dans le cas spécial où P_{ij} est indépendant de i, ainsi qu'appliqué par Manne et den Broeder. Le Weapon Target Assignement est donc un problème complexe, qui ne peut être résolut en temps polynomial.

2.2.1 WTA Statique

Les algorithmes discutés à la section précédente ont été conçus pour résoudre la version statique du problème. Dans cette version, un certain nombre de menaces et de ressources sont concernées dans une situation donnée, mais à un temps fixe. Avant d'aborder l'évolution du problème, il est intéressant de visualiser le formalisme du WTA statique, soit :

M	:	l'ensemble des menaces
W	:	l'ensemble des ressources
m	:	le nombre de menaces, ou $\|M\|$
n	:	le nombre de ressources, ou $\|W\|$
i	:	indice d'une menace, $i \in \{1, 2, ..., m\}$
j	:	indice d'une ressource, $j \in \{1, 2, ..., n\}$
P_{ij}	:	la probabilité de détruire i avec j
V_i	:	la valeur de la menace i
A_{ij}	:	l'assignation de la ressource j sur la menace i.

La version statique du WTA consiste à minimiser :

$$\sum_{i=1}^{i=m} v_i \prod_{j=1}^{j=n} (1 - P_{ij})^{A_{ij}}$$

Les assignations peuvent également être restreintes tel que clairement décrit dans une revue concise du problème faite par Murphey [Murphey(1999a)]. Cette notion sera mise de coté pour l'instant, mais réintroduite dans un prochain chapitre. Certains auteurs ont reformulé le problème pour qu'il soit centré sur les entités à défendre. Dans cette version du problème, les menaces sont dirigées vers une entité et chaque entité a une valeur.

K	:	l'ensemble des entités
l	:	le nombre d'entités dans K
k	:	indice d'une entité, $k \in \{1, 2, ..., l\}$
V_{ik}	:	la valeur de la menace i pour l'entité k

Metler et Preston ont étudié plusieurs heuristiques pour résoudre des modèles de ce genre [Metler and Preston(1990)]. Dans le cas ou chaque entité est également pourvue de ressources, il conviendrait également d'appliquer des algorithmes de collaboration. La notation des entités n'est pas toujours utilisée puisque dans le cas où une seule plate-forme est concernée, il sera toujours que $k = l = 1$.

2.2.2 WTA Dynamique

Dans un article de Patrick A. Hosein et al. [P. A. Hosein and Athans(1988)], une version dynamique du WTA est introduite. Dans cette version, le problème est étendu à plusieurs points dans le temps, auxquels il est possible de faire des assignations. Dans le cas du WTA dynamique, il y a donc une matrice A_t d'assignations A_{ijt} pour chaque temps t. Bien sur, la probabilité qu'une ressource j détruise une menace i peut être dépendente du temps également. Les probabilités P_{ij} deviendront alors P_{ijt} en fonction du temps :

τ	:	l'ensemble des points dans le temps.
t	:	l'indice de temps, $t \in \tau$
h	:	l'horizon de temps
P_{ijt}	:	la probabilité P_{ij} au temps t.
A_{ijt}	:	l'assignation A_{ij} au temps t.

Lorsque plusieurs intervalles de temps sont utilisés, le problème est alors un WTA à périodes multiples. Les périodes multiples peuvent être abordées de deux façons, qui ne sont pas nécessairement mutuellement exclusives. La première façon est d'assumer que toutes les menaces sont connues à priori et qu'il est possible d'observer si une menace est détruite ou non à la fin d'un intervalle. La nature stochastique provient du fait que les menaces sont détruites avec une probabilité P. Dans cette manière d'engager les menaces, on emploie généralement une doctrine *Shot-Look-Shot*, c'est-à-dire qu'après avoir complété un engagement, les résultats sont observés avant qu'un autre engagement ne soit effectué. La seconde façon de gérer plusieurs périodes a été proposée par Murphey [Murphey(1999b)] et consiste à rendre déterministe le résultat des assauts contre les

menaces. De cette manière, on sait que les menaces qui ont été assignées seront effectivement détruites. Cependant, Murphey introduit une incertitude différente dans son modèle : le nombre et l'emplacement des menaces du prochain intervalle ne sont pas connus. Il propose d'ailleurs un algorithme de programmation stochastique pour résoudre approximativement ce problème. En décomposant une version déterministe du problème, une solution pour la période 1 est trouvée et ensuite utilisée pour résoudre la période 2, de laquelle la solution est utilisée pour borner la période 1. Ces étapes sont répétées jusqu'à ce que la solution de la première période de temps ne soit qu'à ϵ de sa borne. Cependant, cet heuristique n'est applicable que pour la version à deux périodes.

Une des meilleures descriptions du problème WTA fut faite par Toet et de Waard [Toet and de Waard(1995)]. Dans leur article, ils formalisent le problème en incluant presque tous les aspects dont nous avons parlé précédemment. Leur formalisme inclut notamment les entités, le temps (ou les intervalles) et les matrices d'allocation. Ils nomment l'ensemble des matrices d'assignations un *Firing plan* (traduit ici par Plan d'action). De plus, une notion de durée d'engagement est intégrée. Leur fonction d'optimisation est assujeti à une contrainte sur les durées ; la somme des durées d'une ressource ne doit pas dépasser un temps limite D (*deadline*). Ces notions de durée et de limite de temps sont très utiles lorsque la longueur des périodes n'est pas connue d'avance.

Il est possible de pousser le concept de durée d'engagement un peu plus loin. Le temps limite présenté jusque maintenant était un temps D bien défini. Puisque, dans un scénario réaliste, les menaces se dirigeant vers le défenseur ne voyagent pas nécessairement à la même vitesse, ni ne sont à la même distance de la plate-forme ciblée, il y aura alors un temps d'impact différent pour chaque menace. En ayant un temps limite D égal au premier temps d'impact, on peut forcer l'algorithme à effectuer ses manoeuvres avant le temps D. Par contre, dans la réalité, cette menace peut être détruite dans les premiers intervalles et ainsi repousser le temps limite D selon la seconde menace la plus dangereuse en terme de temps. Pour l'instant, il est suffisant d'ajouter que le temps limite D est en fonction des menaces. La contrainte devient donc que les actions effectuées contre i doivent être prises avant le temps D_i.

D_i : le temps limite d'action avant que la menace i touche sa cible.
S_{ijt} : la durée d'engagement de la ressource j sur la menace i au temps t.

Dans plusieurs formalisations, la valeur des menaces V_i est considérée dans la fonction à optimiser. Cette valeur peut également être vue comme le potentiel de dommages de cette menace pour l'entité qu'elle tente d'atteindre. Lorsqu'on considère la survie

d'une plate-forme, il faut minimiser les dommages, maximiser la valeur des menaces détruites ou minimiser la valeur des menaces pouvant atteindre leur cible. Dans tous ces cas, il s'agit optimiser la somme des valeur V_i. Tel que discuté en début de chapitre, lorsqu'on tient compte de la possibilité qu'une plate-forme subisse plusieurs attaques successives au cours d'une mission de longue durée, le coût des ressources utilisées devient plus important. À ce propos, il convient donc d'ajouter une variable au problème : le coût.

C_{ijt} : le coût d'utilisation de la ressource j contre la menace i au temps t

Le coût pourrait également être en fonction du temps, mais puisque dans la plupart des cas le coût des ressources n'est pas dépendant du temps, il vaut mieux laisser tomber cette notation pour des raisons de simplification. Une approche similaire concernant le coût a été employée dans les dernières années, dans un article de Rosenberger, où l'auteur et ses collègues présentent deux algorithmes optimaux pouvant résoudre de petites instances du problème [J. M. Rosenberger and Brungardt(2005)]. Dans cet article, la formulation du WTA diffère de celle présentée jusque maintenant. Les auteurs utilisent une fonction à maximiser sur la somme des bénéfices que rapporte chacune des assignations. Sans utiliser cette formulation du problème, il est intéressant de retenir l'idée d'établir un ensemble d'assignations possibles. Cette notion réfère au processus d'*Engageabilité* présenté au chapitre 1.

Les résultats les plus impressionnants sur le WTA sont probablement ceux de Ahuja et al. Leur article [R. K. Ahuja and Jha(2003)], bien que sur la version statique du problème, reste néanmoins efficace pour des instances du WTA comportant jusqu'à 200 ressources et 200 menaces. Ils utilisent et comparent trois bornes minimales avec un algorithme de *branch and bound* pour trouver une solution optimale au problème avec des instances moyennement grandes. Pour les instances plus larges, les auteurs proposent un algorithme de construction heuristique utilisant une séquence de problèmes de type *minimum cost flow*. Par la suite, un algorithme de recherche VLSN (*very large-scale neighborhood*) trouve une meilleure solution que celle obtenue par construction en cherchant dans le voisinage élargi de cette dernière. Plusieurs améliorations successives seront faites jusqu'à l'obtention d'une solution localement optimale. Les résultats de l'heuristique sont souvent optimaux et, dans le cas contraire, l'algorithme de recherche VLSN rapproche ceux-ci de l'optimalité ne laissant qu'une marge d'erreur relativement négligeable.

2.2.3 Formulation

La formulation utilisée dans le mémoire :

M	:	l'ensemble des menaces
W	:	l'ensemble des ressources
τ	:	l'ensemble des points dans le temps.
m	:	le nombre de menaces, ou $\|M\|$
n	:	le nombre de ressources, ou $\|W\|$
h	:	horizon de temps
i	:	indice d'une menace, $i \in \{1, 2, ..., m\}$
j	:	indice d'une ressource, $j \in \{1, 2, ..., n\}$
t	:	indice de temps, $t \in \tau$
P_{ijt}	:	la probabilité de détruire i avec j au temps t
V_i	:	la valeur de la menace i
D_i	:	deadline pour la menace i
S_{ijt}	:	durée d'engagement de la ressource j sur la menace i au temps t.
C_{ijt}	:	le coût d'utilisation de la ressource j contre la menace i au temps t
A_{ijt}	:	la matrice d'assignation.

Fonction à optimiser :

$$\sum_{i \in M} \left(V_i \prod_{j \in W, t \in \tau} (1 - P_{ijt})^{A_{ijt}} + \sum_{j \in W, t \in \tau} C_{ijt}^{A_{ijt}} \right)$$

Contraintes :

$$\begin{array}{ll} \max_{j,t} (S_{ijt} + t)^{A_{ijt}} \leq D_i & \forall i \\ \sum_i A_{ijt} \leq 1 & \forall j, t \end{array}$$

Sans oublier les contraintes spécifiques à l'instance du problème, notamment sur les ressources et le temps.

2.2.4 WTA Événementiel

Dans un environnement dynamique tel qu'un combat maritime, le découpage de l'affrontement en périodes, ou intervalles, fixes d'une durée prédéfinie ne permet pas d'obtenir une gestion de la défense capable de réagir à tous les nouveaux éléments de l'environnement. Par exemple, l'apparition de nouvelles menaces est imprévisible et

peut survenir à n'importe quel moment. Ainsi, lorsque les temps de début et de fin de périodes sont définis avant la résolution du problème, une menace peut surgir au milieu d'un intervalle en cours. Si une menace n'est pas gérée dès son apparition et qu'elle peut progresser grandement avant que l'algorithme ne la considère dans l'intervalle suivant, les performances en seront affectées assurément. Supposons que ceci survient un court instant après que le plus long intervalle ait commencé, la menace ne sera prise en ligne de compte par l'algorithme que lorsque celle-ci sera beaucoup plus près et plus dangereuse. Cette observation amène donc à se demander en quelle répartition des périodes est-il préférable de diviser le temps du scénario afin d'assurer une gestion optimale des menaces surgissant au hasard. Le problème lorsque l'agent agit en temps réel dans un environnement dynamique, c'est que l'instance du problème évolue dans le temps et on ne connaît donc pas l'instant exact de l'apparition des menaces.

Dans la formulation finale adoptée dans la section précédente, des assignations par intervalle de temps t permettent d'identifier les engagements des ressources sur les menaces. Le plan d'action contient des assignations A_{ij} indicés par t pour définir le temps auquel l'engagement est effectué. Une fois lancés, les engagements ne sont alors plus influencés par les périodes suivantes et sont plutôt régis par le deadline D_i. Donc, une fois qu'une période est commencée, la séquence d'apparition des prochaines périodes n'influence en rien les périodes précédentes. Il est alors possible de générer des intervalles supplémentaires qui ne faisaient pas partie de l'instance originale ou précédente. L'ajout de périodes modifie les données de l'instance et peut, en quelque sorte, rendre le plan en cours moins bon qu'il ne l'était. Dans ces conditions, l'algorithme devra procéder à une reconstruction du plan ou encore une réparation. Les algorithmes réactifs ne faisant pas de planification sont avantagés en ce sens qu'en l'absence de plan, ils n'ont pas à le maintenir valide.

Sachant que les évènements futurs n'étaient pas connus ni prévisibles au moment où les actions ont été prises, celles-ci demeurent valides jusqu'à ce que l'instance change. Au cours des prériodes suivantes, les nouvelles informations permettront sans doute d'identifier des actions qui aurait pu être meilleures pour les intervalles passés. Néanmoins, en l'absence d'information sur l'avenir, celles-ci restent encore les meilleures et ce même si les évènements futurs sont tous en défaveur des décisions prises. Dans un environnement dynamique où l'agent perçoit l'information au fur et à mesure que le temps avance, soit en temps réel, il est souhaitable de formaliser le problème du WTA dans une version orientée évènement.

La WTA évènementiel respecte le formalisme décrit à la section précédente, sauf pour ce qui est de l'ensemble τ et l'horizon de l'instance. L'horizon h prend la valeur de l'infini et l'ensemble τ devient infini à son tour. Cet ensemble n'est pas fixe, la cardina-

lité de l'ensemble croît avec le temps, selon l'ajout de nouvelles périodes. L'apparition d'un intervalle au temps t est un évènement. Chaque évènement change l'instance du problème par $\tau_t = \tau_{t-1} \cup \{t\}$. La dynamique du problème respecte alors la dynamique de l'environnement.

Voici quelques suggestions d'évènements :
- apparition d'une nouvelle menace
- destruction/disparition d'une menace
- libération d'une ressource
- etc...

2.3 Considérations

La formulation du problème donne lieu à beaucoup d'incertitudes quant au modèle. En effet, plusieurs éléments sont inconnus ou incertains, principalement en ce qui concerne le futur.

Tel que discuté au chapitre 1, la perception d'un agent de son environnement peut être partielle ou complète et absolue. Dans un environnement simulé, des modèles de radar et d'acquisition d'information dans l'environnement sont utilisés pour transmettre la description des menaces au C2. Puisque ces modèles ne sont pas nécessairement précis et exacts, il conviendra de dire que cette description est bruitée.

L'incertitude sur l'avenir au-delà de la période en cours rend la planification beaucoup plus ardue. Les éléments à priori imprévisibles sont les suivants :

1. Le résultat d'une assignation est stochastique et la distribution dans le temps de cette variable aléatoire peut être inconnue
2. Des évènements peuvent être ajoutés à n'importe quel temps et ceux-ci sont susceptibles de changer l'instance connue du problème.
3. L'horizon du problème est indéterminé ; de nouveaux intervalles peuvent s'ajouter infiniment.

Le premier point implique que les distributions de probabilités du modèle doivent être estimées ou apprises. Si elles sont estimées, il est fort probable que l'estimation donnée ne soit pas exacte. En présence d'un environnement artificiel, il est possible d'intégrer directement la fonction de transition de l'environnement dans le modèle. Bien entendu, elle doit être connue du concepteur de l'agent. Dans ce cas, il serait alors possible pour l'algorithme de trouver une solution optimale. Dans le cas contraire,

une estimation humaine de la fonction peut être intégrée. Lorsque les probabilités sont estimées selon l'expertise humaine, l'algorithme peut trouver une solution optimale pour le modèle ainsi fourni, mais ce ne sera probablement pas une solution optimale dans l'environnement. Certes, si l'estimé est parfait et que les probabilités exactes de la fonction de transition de l'environnement sont introduites manuellement, alors l'estimé fait par l'humain n'en est plus un et la solution peut alors être optimale. Cependant, dans la majorité des cas, et particulièrement dans les environnements naturels, ces probabilités seront rarement connues parfaitement ou alors il est très fastidieux de les énumérer toutes.

Si les distributions sont apprises, il n'y a qu'en testant le résultat dans l'environnement et en répétant ces tests à l'infini qu'un modèle exact sera obtenu. Bien que les méthodes Monte Carle offrent des résultats estimant assez précisément un modèle lorsqu'on dispose d'un échantillon très grand, l'optimalité n'est atteinte que lorsqu'on peut peaufiner les résultats infiniment.

Que les distributions soient estimées ou apprises, il faut retenir que si le modèle de probabilité ne reflète pas exactement la réalité de l'environnement, alors une solution optimale pour un agent n'est pas forcément une solution optimale dans l'environnement. Il faut bien distinguer la différence.

Le second point concernant l'incertitude empêche théoriquement d'appliquer des équations de programmation dynamique, car l'instance des périodes suivantes est inconnue. Plus encore, la probabilité qu'une instance donnée survienne au prochain intervalle est également inconnue initialement. Ceci dit, on ne peut optimiser le sous-problème suivant car il manque les paramètres nécessaires à la récursion.

Le fait de ne pas connaître les détails du prochain intervalle pose un dilemme. Doit-on agir immédiatement de façon réactive avec les données du moment, ou alors attendre au dernier moment pour faire les actions dans l'espoir de faire des engagements plus éclairés, au détriment de la marge de manoeuvre ? Le choix d'une doctrine d'engagement *au plus tôt* ou *au plus tard* devient généralement une assomption de départ et la doctrine choisie affecte tous les engagements. Dans ce mémoire, une doctrine d'engagement *au plus tôt* est utilisée.

Le troisième point énonce la difficulté d'établir le maximum ou le minimum d'une fonction qui n'est pas bornée. Lorsqu'il y a possiblement une infinité d'intervalles, la fin de la dernière période n'est pas définie et il est difficile d'appliquer la programmation dynamique dans sa forme originale. Il faut donc travailler dans un intervalle restreint sur le temps, dans lequel il est possible d'optimiser localement.

2.3.1 Choix d'une approche

Les points de la section 2.3 incitent à l'utilisation de l'apprentissage par renforcement, car il convient d'obtenir une politique capable de généraliser le comportement. Les algorithmes de recherches de plan ne seront pas étudiés, principalement parce que l'horizon est infini et que ces algorithmes doivent replanifier ou réparer leur plan chaque fois qu'un évènement survient. Une politique issue de l'apprentissage tient compte du futur tout en conservant un comportement réactif.

L'apprentissage par renforcement est utilisé pour ces raisons :

1. Cette approche permet d'apprendre le modèle.
2. Le résultat de l'apprentissage inclut les probabilités de transition entre les états.
3. Les résultats générés par l'algorithme sont avant tout un plan d'action pour chaque moment donné.
4. Si l'instance du problème change, seuls les états changent et le plan reste valide.
5. Les actions sont choisies de façon à maximiser à long terme la qualité de la solution.

Un algorithme d'apprentissage requiert une période d'entraînement au cours de laquelle l'agent construira son modèle par expérimentation. Cette période d'exploration de l'environnement est très exhaustive et sa durée dépend du nombre d'états possibles que peut avoir l'agent. Donc, un désavantage est que cette approche peut prendre beaucoup de temps pour établir un plan et que l'apprentissage se fait par essais-erreurs. D'un autre côté, grâce au renforcement, on peut démarrer l'apprentissage à partir d'un plan de départ donné et faire évoluer ce plan vers un plan d'action proche de l'optimal. Dans certains cas, il est aussi possible pour l'algorithme de s'adapter à de nouveau paramètres d'environnement à partir de l'expérience déjà acquise lors d'une autre séance d'exploration d'un environnement semblable.

L'utilisation de l'apprentissage par renforcement est justifiée dans un premier temps par le fait que l'algorithme peut graduellement s'adapter, mais aussi parce que le temps de réponse hors des sessions d'apprentissage est très rapide. En effet, il suffit à l'agent d'identifier son état et l'algorithme fournira, selon la politique, l'action à prendre immédiatement.

Chapitre 3

Apprentissage par renforcement

Les concepts les plus répendus dans l'apprentissage par renforcement sont les réseaux de neuronnes et les processus de décision markoviens. Dans ce mémoire, seul le second concept sera étudié. Cependant, pour une application des réseaux de neuronnes au problème d'allocation de ressouces, voir [Wacholder(1989)].

3.1 Processus de décision markoviens

Ce chapitre présente les méthodes d'apprentissage par renforcement basées sur les processus de décision de Markov, ou plutôt *Markov Decision Processes* (MDP).

3.1.1 Mise en contexte

Andrei Markov était un mathématicien russe fameux pour ses travaux touchant les processus stochastiques, en particulier les chaînes de Markov.

En statistique, une variable aléatoire est une variable pour laquelle on connaît une probabilité pour chaque valeur possible. Ainsi, une variable aléatoire représentant un lancer de dé possède 6 valeurs possibles $\{1, 2, 3, 4, 5, 6\}$ avec une probabilité de $1/6$ pour chacune des valeurs. Si chaque lancer de dé mène à un état correspondant différent, alors l'ensemble des valeurs de 1 à 6 du dé pourrait identifier chacun des états résultants possibles. L'ensemble des états possibles est appelé l'espace d'état. Le jeu de dé est donc un processus stochastique avec un espace d'état de 6 valeurs (ou états) différentes.

Les chaînes de Markov sont des séquences de variables aléatoires dont chacune respecte la Propriété de Markov. Cette propriété identifie une variable comme étant indépendante des variables précédentes ; son résultat n'est pas influencé par le passé. Un état respectant la propriété de Markov est un état pour lequel le prochain état à survenir ne dépend pas des états passés, mais seulement de l'information contenue dans l'état présent.

Dans un processus de décision de Markov, il y a un ensemble d'états (les états de l'agent) et un ensemble d'actions. Un modèle de transitions définit la probabilité de passer d'un état à un autre en effectuant une action donnée. Ce modèle respecte la propriété de Markov, car les probabilités de transition ne dépendent que de l'état courrant ; elles ne dépendent pas des états précédents de l'agent.

3.1.2 Formalisme

Un MDP est un tuple $< S, A, T, R >$

S : l'ensemble d'états
A : l'ensemble d'actions
T : la fonction de transition
R : la fonction de renforcement

Les ensembles S et A sont respectivement l'ensemble des états possibles de l'agent dans l'environnement et l'ensemble des actions que l'agent peut effectuer dans ce même environnement. Ce sont ces ensembles qui caractérisent l'agent dans un processus de décision markovien.

La fonction de transition T reflète la dynamique de l'environnement. Elle donne la probabilité d'atteindre l'état s' en effectuant l'action a lorsque l'agent se trouve dans l'état s. Cette fonction concorde avec la fonction de transition décrite à la section 1.1.2. La somme des probabilités est définie comme suit : $\sum_{s' \in S} T(s, a, s') = 1$

La fonction de renforcement $R()$, ou fonction de récompense, donne une récompense immédiate pour une transition donnée. Lorsque $R()$ ne dépend que de l'état courant, $R(s) = R(s, a, s')\ \forall a, s'$. Par contre, si l'action influence la récompense indépendamment de la transition, $R(s, a) = R(s, a, s')\ \forall s' \in S$. Puisque cette fonction donne les récompenses et que l'agent cherche à les maximiser, c'est donc la nature de $R()$ qui façonnera les objectifs de l'agent. Le choix d'une juste fonction de récompense est alors essentiel, car

celle-ci influence directement l'apprentissage. Il faut garder en tête que cette fonction est la seule référence pour l'agent sur ce qui est bon ou mauvais.

Dans un problème comme celui étudié dans ce mémoire, on cherche à maximiser les récompenses à long terme. Résoudre un MDP revient donc à trouver, pour chacun des états, l'action qui permet d'obtenir la plus grosse somme de récompenses au final. La solution associant les états à leur action suggérée est appelée une politique.

Une politique π est une fonction sur S, $\pi : S \longrightarrow A$, donnant, pour chaque état $s \in S$, l'action qu'il convient d'effectuer dans cet état. L'exécution de la politique dans l'environnement ne se terminera pas toujours avec les mêmes résultats. Puisque les transitions sont stochastiques, les actions effectuées feront changer l'état de l'agent selon le modèle T. Une politique est optimale si l'exécution de cette politique permet en moyenne d'obtenir la somme des récompenses la plus élevée. Cette politique est optimale et est notée π^*. Elle est la politique la plus utile parmi toutes les solutions possibles, d'où la condition :

$$\pi^* = \text{argmax}_\pi E[\sum_{t=1}^{t=\infty} \gamma^t R(s_t)|\pi]$$

Le γ sert à amenuiser l'impact des récompenses futures anticipées au temps t. Ainsi, plus les récompenses qui composent la valeur d'utilité de l'état suivant sont issues d'états éloignés, plus elles seront discomptées.

3.1.3 Itération de Valeurs

Pour calculer la politique optimale, il existe plusieurs méthodes. La plus intuitive est sans doute l'*Itération de Valeurs* (Value Iteration). Comme son nom l'indique, cet algorithme fonctionne par itérations. Il s'agit de calculer itérativement une valeur pour chaque état jusqu'à ce que ces valeurs ne changent presque plus. Les valeurs sont considérées stables lorsqu'une itération n'aura modifié aucune valeur d'une différence plus élevée que ϵ, seuil choisi qui est généralement un très petit nombre. La valeur à calculer est communément appelée *Utilité* de l'état. L'utilité d'un état est la valeur des récompenses que l'on espère obtenir à partir de cet état, en suivant une politique optimale pour les actions subséquentes. On note par $U(s)$ l'utilité de l'état s. L'utilité des états et la politique optimale sont donc intimement reliées. Sachant la vraie utilité de chacun des états, la politique optimale peut être construite en définissant $\pi(s)$ comme suit :

$$\pi^*(s) = \text{argmax}_a \sum_{s' \in S}[T(s, a, s')U(s')]$$

Plus encore, il existe une relation entre les utilités d'états subséquents, c'est l'équation de Bellman [Bellman(1957)] :

$$U(s) = R(s) + \gamma \max_a \sum_{s' \in S} [T(s, a, s')U(s')]$$

Cependant, il est rare que la vrai utilité d'un état soit connue. Au contraire, elle est l'objectif à atteindre. Parce qu'elle génère la solution, il est d'usage de chercher à trouver la fonction d'utilité sur S, ou encore l'estimer. La valeur estimée de l'utilité d'un état s est notée $V(s)$. Cette valeur peut s'approcher de près ou de loin de la vraie utilité : $V \approx U$. Quoi qu'il en soit, s'il est possible de faire converger $V(s)$ vers $U(s)$, une solution sera obtenue. La plupart du temps, l'estimation est satisfaisante si V s'approche à ϵ près de U.

Pour effectuer l'Itération de Valeurs, il faut boucler sur chaque état et appliquer l'équation de Bellman en utilisant les valeurs estimées $V(s)$ de l'utilité. Pour la $i$ème itération, la formule est la suivante :

$$V_i(s) \leftarrow R(s) + \gamma \max_a \sum_{s' \in S} [T(s, a, s')V_{i-1}(s')]$$

L'application de cette fonction uniformément en tant qu'opérateur sur toutes les valeurs $V(s)$ pour tout s est appelée *Mise à jour de Bellman*. Dans l'Itération de Valeurs, on effectue donc une suite de mises à jour de Bellman jusqu'à ce qu'aucune mise à jour sur s n'aie généré une différence plus grande que ϵ sur la valeur $V(s)$ correspondante. En d'autres termes, lorsque qu'une itération n'apporte qu'une amélioration minime, les valeurs V produisent une estimation satisfaisante sur U. L'estimation optimale est notée V^*.

Algorithm 3.1 Value Iteration

1: $V \leftarrow$ valeurs aléatoires
2: **répéter**
3: $\quad \Delta \leftarrow 0$
4: $\quad$ **pour** $\forall s \in S$ **faire**
5: $\quad\quad v \leftarrow V(s)$
6: $\quad\quad V(s) \leftarrow \max_a \sum_{s' \in S} T(s, a, s') \left[R(s, a, s') + \gamma V(s')\right]$
7: $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
8: **jusqu'à** $\Delta < \epsilon$

L'algorithme 3.1 réflète la boucle de l'Itération de Valeurs. Lors d'une itération, $V(s)$ est changée de façon à ce que l'estimé se rapproche strictement de $U(s)$. Bref, puisque $U(s)$ est l'optimum, $V(s)$ s'approche constamment de la vraie utilité $U(s)$. Avec un nombre infini d'itérations, $V(s)$ converge vers $U(s)$ et on obtient la solution optimale.

3.1.4 Itération de politiques

L'algorithme d'*Itération de Politiques* (Policy Iteration) est présenté par Ron Howard dans un livre paru en 1960 [Howard(1960)]. Cet algorithme constitue une manière fiable d'obtenir une solution. L'idée est de débuter avec une politique, que l'on améliore par la suite de façon itérative. Policy Iteration fonctionne donc en deux étapes. La première consiste à évaluer la politique et la seconde se veut une suite d'améliorations de la politique.

Algorithm 3.2 Policy Iteration

1: $V \leftarrow$ valeurs aléatoires
2: $\pi \leftarrow$ politique aléatoire
3: **répéter**
4: **répéter**
5: $\Delta \leftarrow 0$
6: **pour** $\forall s \in S$ **faire**
7: $v \leftarrow V(s)$
8: $V(s) \leftarrow \sum_{s' \in S} T(s, \pi(s), s') [R(s, \pi(s), s') + \gamma V(s')]$
9: $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
10: **jusqu'à** $\Delta < \epsilon$
11: $c \leftarrow faux$
12: **pour** $\forall s \in S$ **faire**
13: $b \leftarrow \pi(s)$
14: $\pi(s) \leftarrow \text{argmax}_a \sum_{s' \in S} T(s, a, s') [R(s, a, s') + \gamma V(s')]$
15: **si** $b \neq \pi(s)$ **alors**
16: $c \leftarrow vrai$
17: **jusqu'à** $c = vrai$

Dans la phase d'évaluation, il s'agit de calculer les valeurs $V(s)$ pour chaque état s. La différence avec le value itération, c'est qu'on ne calcule pas $V(s)$ de la même façon. En effet, puisqu'une politique est donnée, l'utilité est calculée pour les actions de cette politique seulement. Il convient d'appliquer alors une version simplifiée de la mise à jour de Bellman :

$$V_i(s) \leftarrow R(s) + \gamma \sum_{s' \in S} [T(s, \pi_{i-1}(s), s') V_{i-1}(s')]$$

Dans l'algorithme 3.2, cette mise à jour est effectuée à la ligne 8. La première phase se répète tant que la variation Δ de la valeur n'est pas plus petit que le seuil ϵ (lignes 9 et 10).

Ensuite, la seconde phase permet de comparer chacune des actions d'un état avec l'action suggérée par la politique. Si une autre action que celle de la politique semble plus utile, la politique est modifiée en conséquence. En effet, l'amélioration de la politique faite à la ligne 14 sélectionne l'action la plus prometteuse à long terme.

3.2 Apprendre le contrôle

Les algorithmes présentés jusqu'à maintenant permettent de résoudre une instance bien définie d'un MDP. Mais qu'en est-il lorsque le MDP n'est pas connu complètement ? Que fait-on si la fonction de transition T ou la fonction de récompense R est manquante ? Si les probabilités de transition et les récompenses suite aux actions ne sont pas connues de l'agent, alors celui-ci devra les apprendre.

L'apprentissage par renforcement utilise les concepts de programmation dynamique et les applique en temps réel dans un environnement. En effet, ne connaissant pas les fonction T et R du MDP, l'agent doit procéder par essais-erreurs pour explorer son environnement afin de déterminer l'utilité de ses actions. Pour ce faire, il fait appel aux méthodes EXPLORATION et EXECUTER décrites à la section 3.2.9.

Comme l'aurait fait l'Itération de Valeurs, les algorithmes de contrôle effectuent une suite de mises à jour sur les valeurs $V(s)$ afin de raffiner la politique de l'agent. Cependant, celui-ci ne peut utiliser la mise à jour de Bellman telle que définie précédemment, car cette équation nécessite la connaissance au minimum de T. La connaissance de R ici n'est pas obligatoire, car la récompense immédiate d'une action est reçue à chaque itération. Il est assumé que $R_i(s, a)$ de l'itération i est constant $\forall i$. Dans ce cas, une approche par approximation statistique aurait vite fait d'estimer cette fonction si celle-ci n'était pas fournie. Contraint par l'absence de T, il faut donc planifier les actions en intégrant une nouvelle dimension à $V(s)$, qui devient alors la fonction $Q(s, a)$. On note par Q^* la fonction Q optimale donnant la valeur totale des récompenses escomptées pour une action a dans un état s :

$$Q^*(s, a) = R(s, a) + \gamma E\left[U(s')|T(s, a, s')\right]$$

Les valeurs de Q sont appelées les Q-valeurs. Malheureusement, les valeurs optimales Q^* ne sont pas connues. Les valeurs $Q^*(s, a)$ doivent alors être estimées par $Q(s, a)$. Donc, pour optenir un algorithme de décision ou de contrôle, l'apprentissage se fait sur les Q-valeurs. Conséquemment, $V(s)$ devient alors la fonction donnant la meilleure Q-valeur pour un couple état-action, c'est-à-dire la valeur maximale de l'uti-

lité espérée obtenue par une action, étant donné l'état s, soit $V(s) = \max_a Q(s,a) \equiv V^*(s) = \max_a Q^*(s,a)$.

La qualité de l'estimation de U par V dépend donc essentiellement des Q-valeurs. Pour obtenir des valeurs satisfaisantes pour Q, plusieurs algorithmes ont été proposés. Les algorithmes d'apprentissage couverts dans ce chapitre se situent quelque part entre ces deux orientations principales :

1. apprendre/estimer le modèle et générer Q à partir de cet approximation ;
2. oublier le modèle et estimer directement Q à partir de l'expérience immédiate.

3.2.1 Monte Carlo

Une façon très simple d'aborder le problème est d'explorer l'environnement et construire un estimé pour $Q(s,a)$ à partir des récompenses obtenues lorsque l'action a a été prise dans l'état s. Les méthodes *Monte Carlo* consistent à construire un échantillon assez grand de données pour que l'erreur de l'estimation par rapport à la réalité soit considérée comme minimale, selon un certain seuil de confiance.

Algorithm 3.3 Monte Carlo

1: **pour** $\forall s \in S$ et $\forall a \in A$ **faire**
2: $\quad Q(s,a) \leftarrow$ valeur aléatoire
3: $\quad \hat{R}(s,a) \leftarrow 0$
4: $\quad D(s,a) \leftarrow 0$
5: **pour** chaque épisode **faire**
6: $\quad s \leftarrow$ état de départ
7: $\quad$ **répéter**
8: $\quad\quad a \leftarrow$ EXPLORATION(s)
9: $\quad\quad x \leftarrow$ EXECUTER(a)
10: $\quad\quad s \leftarrow O(x)$
11: $\quad\quad r \leftarrow R(s,a)$
12: $\quad\quad \hat{R}(s,a) \leftarrow \hat{R}(s,a) + r$
13: $\quad\quad D(s,a) \leftarrow D(s,a) + 1$
14: $\quad\quad Q(s,a) \leftarrow \frac{\hat{R}(s,a)}{D(s,a)}$
15: $\quad$ **jusqu'à** s est terminal

L'algorithme Monte Carlo présenté dans l'algorithme 3.3 maintient en mémoire la somme des récompenses pour chaque couple état-action (ligne 12) ainsi que le diviseur de cette somme (ligne 13) donnant la moyenne des récompenses obtenues (ligne 14).

Les Q-valeurs ainsi obtenues sont alors issues de la fonction suivante pour la $i$ème mise à jour :

$$Q_i(s,a) = \frac{1}{i}\sum_{j=1}^{j=i} R_j(s,a)$$

Cette estimation ne tient cependant pas compte du futur à long terme de l'agent. En effet, les Q-valeurs calculées sont des prédictions sur la récompense immédiate d'une transition.

3.2.2 Bootstrapping

Puisque la donnée statistique obtenue estimant la valeur de $Q(s,a)$ n'est basée que sur les récompenses obtenues pour l'action a dans l'état s, l'algorithme Monte Carlo présenté précédemment perd la notion de planification qui se trouvait dans les méthodes de programmation dynamique. Par exemple, si une transition mène à un état ayant une meilleure récompense espérée, ou encore un plus grand potentiel de récompense à long terme, les méthodes utilisant une mise à jour basée sur l'équation de Bellman trouveront une Q-valeur pour cette action qui tiendra compte implicitement du futur de l'agent. Le concept par lequel on construit une valeur en utilisant également d'autres valeurs estimées est appelé *bootstrapping.*

Lorsqu'on utilise le bootstraping avec les Q-valeurs des états atteignables à partir de l'action a pour calculer $Q(s,a)$, et ce récursivement, c'est alors que survient la planification. En effet, sachant que la politique est basée sur l'action correspondant à l'utilité la plus élevée, on peut donner une valeur courante à un état qui tient compte du comportement de l'agent dans l'avenir. Une façon de faire du Bootstrapping avec une approche de type Monte Carlo est d'estimer statistiquement T par $\hat{q}$, où $\hat{q}(s,a,s') \approx T(s,a,s')$.

3.2.3 Différences temporelles

Contrairement à une approche Monte Carlo où l'on ajuste les valeurs seulement en fonction de R, le *Temporal Differencing* (TD) utilise plutôt à la fois la récompense immédiate et l'estimation de celles futures, qui est basée sur les expériences précédentes. Avec le TD, les mises à jour sur les valeurs d'utilité ne peuvent s'effectuer en connaissance du modèle de l'environnement. Ce sont des mises à jour basées sur une transition unique, contrairement à la programmation dynamique, avec laquelle toutes les valeurs possibles sont considérées et pondérées selon le modèle de transition.

Voici une mise à jour de TD(0) démontrant bien le fonctionnement des mises à jour basées sur la transition $< s, a, s' >$ courante :

$$V(s) \leftarrow V(s) + \alpha\,[r + \gamma V(s') - V(s)]$$

Où α est le taux d'apprentissage (*learning rate*) et r est une récompense obtenue de l'environnement : $r = R(s, a, s')$. Le taux d'apprentissage gère le degré de variation des Q-valeurs. Il représente en quelque sorte l'affectation d'une probabilité de transition sur l'utilité du prochain état. Plus α est grand, plus la Q-valeur sera influencée par la transition courante. C'est un peu comme le degré de confiance que lorsqu'une transition survient, c'est que la probabilité de cette transition est susceptible d'être élevée. Parfois, le taux d'apprentissage sera variable, en fonction du nombre de visites dans l'état courrant ou de l'avancement d'un épisode. En effet, plus une Q-valeur a été mise à jour, plus on considère que l'estimation est juste, et donc il ne faut plus la varier trop brusquement.

Ce type d'approche est basé sur le concept du bootstrapping utilisé pour les méthodes de programmation dynamique. Les mises à jour tiennent compte du futur de l'agent puisque la différence calculée dans l'équation se situe entre le prochain état et le courrant. Dans les prochaines sections, les deux principaux algorithmes de contrôle dérivant de la méthode de TD seront présentés, soient les algorithmes Sarsa et Q-Learning.

3.2.4 Sarsa

Algorithm 3.4 Sarsa

1: $Q \leftarrow$ valeurs aléatoires
2: **pour** chaque épisode **faire**
3: $s \leftarrow$ état de départ
4: $a \leftarrow$ EXPLORATION(s)
5: **répéter**
6: $x \leftarrow$ EXECUTER(a)
7: $s' \leftarrow O(x)$
8: $r \leftarrow R(s, a, s')$
9: $a' \leftarrow$ EXPLORATION(s')
10: $Q(s, a) \leftarrow Q(s, a) + \alpha\,[r + \gamma Q(s', a') - Q(s, a)]$
11: $s \leftarrow s'$
12: $a \leftarrow a'$
13: **jusqu'à** s est terminal

Pour appliquer le TD à un problème de contrôle, il convient d'utiliser les actions au même titre que les états. La fonction de mise à jour adaptée est la suivante :

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$$

Notez que dans cette fonction de mise à jour, toutes les variables disponibles composant une transition sont utilisées, c'est-à-dire tous les éléments du quintuple $< s, a, r, s', a' >$, d'où le nom de l'algorithme 3.4.

3.2.5 Q-Learning

Le Q-Learning diffère de l'algorithme Sarsa au niveau de la façon de faire l'approximation des Q-valeurs. Dans le Q-Learning, la fonction Q est une estimation directe de Q^*, indépendamment de la politique suivie. Effectivement, lorsqu'une mise à jour est faite dans l'algorithme de Sarsa, elle utilise les états qui sont actuellement visités et donc les mises à jours respectent la politique exécutée. Lors de l'apprentissage, l'agent effectue de l'exploration et n'utilise donc pas forcément la politique estimée à chaque itération de l'épisode. Cependant, Watkins a présenté dans sa thèse [Watkins(1989)] une équation effectuant une mise à jour sur une politique différente que celle actuellement exécutée. On dit que le Q-Learning est un algorithme *off-policy* tandis que Sarsa est *on-policy*. Les deux concepts sont bien définis et expliqués dans le livre de Sutton et Barto [Sutton and Barto(1998)].

Le Q-Learning présenté par Watkins utilise l'équation de mise à jour suivante :

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Watkins et Dayan ont prouvé en 1992 que le Q-Learning converge vers la solution optimale avec une infinité de mises à jour [Watkins and Dayan(1992)].

3.2.6 Descente de Gradient

Si les Q-valeurs sont représentées par une table de valeurs, alors la dimension de cette table va croître proportionnellement au nombre d'états et actions que connaît l'agent. Dans un MDP, il est bien connu que l'espace d'états est susceptible d'être très grand. La taille de la mémoire nécessaire pour maintenir la table des Q-valeurs devient alors déraisonnablement grande. Lorsque $Q(s,a)$ est représenté par une approximation

$Q_\theta(s,a)$, fonction d'un vecteur de paramètres θ, il est alors possible de ne maintenir en mémoire que les valeurs de ces paramètres en question. L'objectif d'apprentissage serait alors de trouver les valeurs des θ_i du vecteur θ rapprochant Q_θ le plus près possible de Q^*. Ceci est possible en utilisant la technique de descente de gradient. Un algorithme de descente de gradient consiste à minimiser l'erreur de l'approximation.

Afin de trouver l'erreur sur l'approximation, on calcule la différence entre la valeur espérée et la valeur obtenue. L'erreur est définie par la moitié du carré de cette différence :

$$e = [Q^*(s,a) - Q(s,a)]^2/2.$$

Le gradient de e est noté $\nabla_\theta e$, il dénote le vecteur des dérivées partielles de e par rapport à θ. Puisque Q^* n'est pas connu, nous utiliserons une approximation de e en utilisant un estimateur $\hat{Q}$, c'est-à-dire

$$\hat{e} = [\hat{Q}(s,a) - Q(s,a)]^2/2.$$

Pour que les garanties de convergeance soient valides pour l'algorithme de descente de gradient, $\hat{Q}$ doit être un estimé non biaisé de Q^*, donc il faut que $E[\hat{Q}] = Q^*$.

À chaque itération, le vecteur de paramètres θ est ajusté proportionnellement au gradient inverse de l'erreur. C'est la mise à jour qui réduit l'erreur le plus favorablement. La fonction de mise à jour deviendra alors :

$$\theta_i \leftarrow \theta_i + \alpha[R(s,a) + \gamma \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)] \frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

Chaque mise à jour modifie les paramètres de la fonction approximative. Ces paramètres sont les mêmes pour toute paire (s,a) au moment t. On assiste alors à une généralisation de Q sur l'ensemble des couples état-action. La méthode de descente de gradient permet donc de faire une anticipation des Q-valeurs pour des états encore inexplorés.

La fonction approximative Q_θ utilisée est cependant soumise à une condition cruciale : elle doit être linéaire relativement aux paramètres. La non-linéarité en fonction des caractéristiques des états ou autres est possible, du moment que la première condition est respectée. Certains auteurs ont appliqué ce type de technique avec succès et la convergence est expliquée entre autres par Tsitsiklis [Tsitsiklis(1994)], [Bertsekas and Tsitsiklis(2000)].

3.2.7 Dyna-Q

Pour un apprentissage basé sur le modèle, lorsque les mises à jour des Q-valeurs d'un algorithme sont incrémentales sur des itérations rapprochés, il existe alors la possibilité d'interrompre le processus d'exploration pendant un moment pour aider l'agent à planifier ses actions futures en faisant des mises à jour sur des transitions simulées à partir d'un modèle appris.

Algorithm 3.5 Dyna-Q

$Q \leftarrow$ valeurs aléatoires
$\hat{T} \leftarrow \{\}$
pour $\forall s \in S$ et $\forall a \in A$ **faire**
  $\hat{R}(s,a) \leftarrow 0$
  $D(s,a) \leftarrow 0$
pour chaque épisode **faire**
  $s \leftarrow$ état de départ
  répéter
    $a \leftarrow$ EXPLORATION(s)
    $x \leftarrow$ EXECUTER(a)
    $s' \leftarrow O(x)$
    $r \leftarrow R(s,a,s')$
    $\hat{R}(s,a) \leftarrow \hat{R}(s,a) + r$
    $D(s,a) \leftarrow D(s,a) + 1$
    $Q(s,a) \leftarrow Q(s,a) + \alpha\,[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $\hat{T} \leftarrow \hat{T} \cup \{< s,a,s' >\}$
    $s \leftarrow s'$
    pour N répétions **faire**
      $< s,a,s' > \leftarrow$ aléatoirement parmi $\hat{T}$
      $\hat{r} \leftarrow \frac{\hat{R}(s,a)}{D(s,a)}$
      $Q(s,a) \leftarrow Q(s,a) + \alpha\,[\hat{r} + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  jusqu'à s est terminal

Dans l'implémentation Monte Carlo suggérée à la section 3.2.1, l'algorithme est tel que les mises à jour se font à chaque instant t. Pour cette implémentation, le modèle se paufine donc à chaque itération. Ceci dit, chaque fois que l'algorithme a une nouvelle version du modèle, il serait théoriquement possible d'effectuer directement un infinité de mises à jour de planification afin obtenir la politique optimale pour cette nouvelle approximation du modèle réel de l'environnement. De même, lorsqu'on utilise un algorithme de TD, n'est-ce pas de l'exploration perdue que de ne pas réutiliser l'expérience

des transitions observées pour propager les récompenses d'un état à l'autre en respectant l'équation de mise à jour ?

Le Dyna-Q, ainsi qu'on peut le constater dans l'algorithme 3.5, permet de faire le compromis entre Monte Carlo et TD. Il utilise le complément de deux boucles de mises à jour simultanées dont l'une (ligne 8) met à jour les Q-valeurs et le modèle (lignes 13 à 16) directement à partir de la transition courante effectuée dans l'environnement, et une autre boucle (ligne 18) simule des transitions basées sur le passé pour effectuer des mises à jour sur des transitions simulées (ligne 21).

Les mises à jour sur les transitions simulées à partir de l'expérience antérieure forment la boucle de planification. Le nombre d'itérations à effectuer dans cette boucle est un paramètre de l'algorithme. Dans la plupart des cas, plus la planification est complète, plus l'algorithme s'améliore rapidement, mais le temps de réaction est délaissé au profit de la délibération.

3.2.8 Prioritized Sweeping

Dans la boucle de planification de l'approche Dyna-Q, les transitions simulées sont générées aléatoirement parmi les transitions observées préalablement. Le principal désavantage de cette sélection aléatoire est que tous les états n'ont pas une utilité aussi importante. Les mises à jour basées sur les états ayant une faible utilité sont souvent moins pertinentes que celles qui concernent des états plus susceptibles de mener à un gain de récompenses à long terme. Parmi les transitions observées, certaines sont donc plus concluantes que d'autres en ce qui a trait à la vitesse de convergence vers une politique optimale.

Dans leur algorithme de *Prioritized Sweeping*, Moore et Atkesson proposent donc de donner une priorité aux transitions observées afin de diriger l'exploration de la boucle de planification vers les transitions les plus importantes [Moore and Atkeson(1993)]. La clé du prioritized sweeping est l'utilisation d'une file de priorité. En effet, on place dans une file les états prédécesseurs susceptibles d'avoir favorisé une plus grande amélioration de l'utilité d'un état. Ce faisant, on prime la planification sur les états les plus prometteurs. Les plus prometteurs sont ceux qui réduisent le plus l'erreur de Bellman. La priorité donnée à chaque transition est directement proportionnelle à la variation Δ apportée à la Q-valeur associée à la transition courante. La liste des successeurs est établie afin de créer une propagation rétrospective lors des mises à jour, en commençant par les états ayant obtenu une meilleure récompense.

Algorithm 3.6 PrioritizedSweeping

$Q \leftarrow$ valeurs aléatoires
$\hat{q} \leftarrow$ zéros partout
pour $\forall s \in S$ **faire**
  pour $\forall a \in A$ **faire**
    $\hat{R}(s,a) \leftarrow 0$
    $D(s,a) \leftarrow 0$
    $\hat{T}(s,a) \leftarrow \{\}$
  $P(s) \leftarrow \{\}$
  $F \leftarrow$ file prioritaire vide
pour chaque épisode **faire**
  $s \leftarrow$ état de départ
  répéter
    $a \leftarrow$ EXPLORATION(s)
    $x \leftarrow$ EXECUTER(a)
    $s' \leftarrow O(x)$
    $r \leftarrow R(s,a,s')$
    $\hat{R}(s,a) \leftarrow \hat{R}(s,a) + r$
    $D(s,a) \leftarrow D(s,a) + 1$
    $\hat{q}(s,a,s') \leftarrow \hat{q}(s,a,s') + 1$
    PROMOUVOIR$(F, s,$TOP$)$
    tant que F non vide, **max** N répétitions **faire**
      $s \leftarrow$ PRIORITAIRE(F)
      $\hat{r} \leftarrow \frac{\hat{R}(s,a)}{D(s,a)}$
      $v \leftarrow \max_a[\hat{r} + \gamma \sum_{s' \in \hat{T}(s,a)} \frac{\hat{q}(s,a,s')}{D(s,a)} \max_{a'} Q(s',a')]$
      $\Delta \leftarrow |v - Q(s,a)|$
      $Q(s,a) \leftarrow Q(s,a) + \alpha v$
      $\hat{T}(s,a) \leftarrow \hat{T}(s,a) \cup \{s'\}$
      $P(s') \leftarrow P(s') \cup \{< s,a >\}$
      $s \leftarrow s'$
      pour chaque $< s', a' > \in P(s)$ **faire**
        $\rho \leftarrow \frac{\hat{q}(s',a',s)}{D(s',a')}\Delta$
        si $\rho > \epsilon$ **alors**
          PROMOUVOIR(F, s', ρ)
  jusqu'à s est terminal

La version originale de Moore et Atkesson utilise une boucle d'évaluation Monte Carlo pour estimer R et T. Les probabilités $\hat{q}$ estimées de T sont ensuite utilisées pour

majorer les priorités données aux transitions afin que celles-ci reflètent la vraisemblance que les dites transitions aient mené à la variation à laquelle la priorité est associée.

Avec le concept de TD, il est possible de diminuer la contribution du Monte Carlo dans la série des mises à jour lors de la planification. En effet, en calculant préalablement la différence qui sera issue des prochaines mises à jour, on peut utiliser directement cette valeur plutôt qu'une valeur représentant l'espérance d'une telle différence. De cette manière, nous pouvons mettre de coté $\hat{q}$.

L'algorithme 3.6 montre une version du Prioritized Sweeping appliquée au contrôle qui, comme le Dyna-Q, utilise deux boucles de mises à jour. La première boucle (ligne 12) permet de recueillir l'information de l'environnement et mettre à jour le modèle en y ajoutant (ligne 19) une transition. La deuxième boucle (ligne 21) effectue de la planification en rapport aux états visités en utilisant la file de priorité pour favoriser les états ayant un plus grand effet convergeant. Ce concept a été utilisé de façon similaire pour le Dyna-Q-queue [Peng and Williams(1992)] (ou Queue-Dyna [Peng and Williams(1993)]).

3.2.9 Exploration

Les algorithmes d'apprentissage en temps réel de ce chapitre construisent un estimé du modèle de transition par exporation. Dans ces algorithmes, cette exploration est effectuée par deux méthodes : EXPLORATION et EXECUTER. Premièrement, l'algorithme obtient une action par l'intermédiaire de EXPLORATION, qui constitue la stratégie de selection des actions. Ensuite, l'exécution de cette action dans l'environnement par la méthode EXECUTER permet d'observer la transition et ajuster l'estimation du modèle.

L'exploration d'un environnement peut se faire de façon totalement aléatoire, car le but de l'exploration est de découvrir de nouveaux états ou d'obtenir l'information manquante du modèle. Cependant, la statégie de selection influence la fréquence et l'ordre dans lesquels les états sont visités. Donc, il est souvent préférable que cette stratégie permette d'explorer principalement les avenues qui semblent prometteuses. Ces avenues sont identifiées par une forte utilité des états dans l'estimé du modèle. Lorsqu'on utilise l'estimé actuel du modèle pour selectionner les actions, il s'agit d'*exploitation.*

Il y a alors une dualité entre l'exploration de l'environnement et l'exploitation du modèle. La manière la plus simple de palier à ce dilemme est probabiliste. Le *ϵ-Greedy* est un algorithme d'exploration dans lequel ϵ est la probabilité qu'une action soit sélectionnée alétoirement, et $1 - \epsilon$ est la probabilité que l'algorithme sélectionne, selon le modèle, l'action la plus utile pour l'état courant.

Chapitre 4

Implémentation

Ce chapitre vise à expliqué comment développer et mettre en pratique des agents utilisant des algorithmes d'apprentissage par renforcement tels que présentés au chapitre précédent. Avant d'aborder les particularités de l'implémentation, il convient de présenter quelques environnements pour lesquels ces algorithmes sont destinés. Premièrement, un bref aperçu sera fait sur les domaines de test ayant été utilisés pour comparer les méthodes d'apprentissage. Puis, le sujet du combat maritime sera abordé dans la description du projet NEREUS et de l'instance du problème WTA que celui-ci suggère. Finalement, un présentation sera faite sur l'application technique de l'apprentissage par renforcement à un problème d'allocation de ressources issu d'un environnement de combat maritime.

4.1 Aperçu

Cette section couvre les domaines d'application utilisés principalement pour la compréhension de certaines notions de comparaison et d'analyse que nous verrons au prochain chapitre.

4.1.1 Labyrinthes

Les labyrinthes sont des environnements faciles à illustrer. Ils forment un groupe d'exemples de problèmes souvent simples à résoudre et qui sont également transposables dans un environnement naturel avec un agent robotisé.

La tâche consiste à atteindre un point fixe sur un terrain en deux dimensions ayant des obstacles, tels que des murs, limitant les déplacements de l'agent. Ce dernier débute son parcours à un endroit donné et doit franchir le labyrinthe jusqu'à la sortie en empruntant le chemin le plus court possible.

Labyrinthe déterministe

Le premier type de labyrinthe est celui dans lequel les actions effectuées sont déterministes. Pour ne pas compliquer les choses et, particulièrement, ne pas obtenir un espace d'états trop grand, ces labyrinthes sont souvent de petites dimensions. Ce genre de labyrinthe est idéal pour valider les algorithmes et les comparer. Bien que ces labyrinthes peuvent être étendus à des instances très grandes, les résultats dans de petites instances sont souvent aussi convainquants.

Pour ce type de labyrinthe, les actions de l'agent seront limitées à quatre déplacements : gauche, droite, haut et bas. L'exploration est ainsi plus facile que lorsque l'agent doit effectuer une rotation ou autre en plus des déplacements. L'agent a parfaitement "conscience" de sa position dans l'environnement. Cependant, il ne connaît pas le plan du labyrinthe, ni ne peut voir plus loin que le bout de son nez ; bref, il est comme un aveugle qui sait néanmoins où il se trouve. La figure 4.1 est un exemple dans lequel un agent doit traverser un labyrinthe déterministe de la case départ (**D**) à la case finale (**F**).

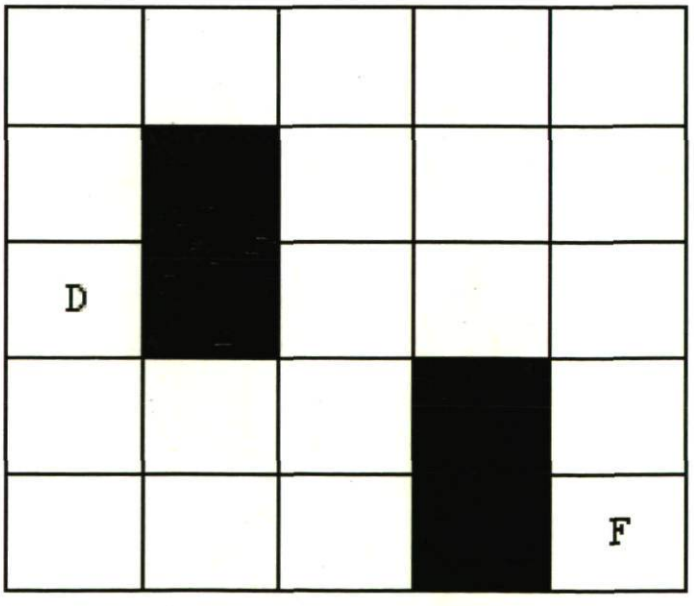

FIGURE 4.1 – Exemple de labyrinthe déterministe.

Labyrinthe stochastique

Un labyrinthe stochastique ressemble beaucoup à un labyrinthe déterministe, mais il en diffère en ce qui concerne les actions. Les actions effectuées dans un labyrinthe stochastique ne sont pas garanties. En effet, lorsque l'agent effectue une action, il est possible que cette action se produise avec une probabilité p, mais il se peut également que l'action produise un résultat autre que celui espéré. Par exemple, dans l'environnement proposé par Russell et Norvig [Russell and Norvig(2003)] illustré à la figure 4.2, l'action a 80% des chances de donner le résultat escompté, mais avec 20% de risque que cette action échoue et produise une déviation dans l'une ou l'autre des directions perpendiculaires.

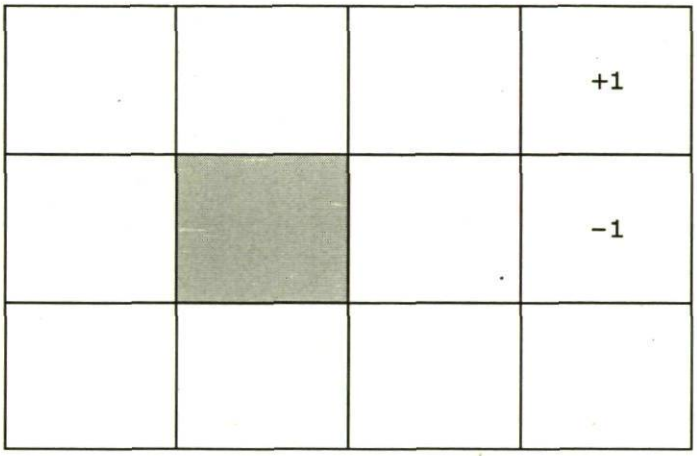

FIGURE 4.2 – Labyrinthe stochastique de Russell et Norvig.

Ce genre de labyrinthe est particulièrement utile pour vérifier si l'algorithme est capable de gérer l'incertitude.

4.1.2 Le feu de signalisation

Le feu de signalisation est un exemple d'environnement simple, dans lequel l'apprentissage par renforcement peut être utilisé pour effectuer une tâche de contrôle. Cet environnement est une route sur laquelle on place une voiture, l'agent, et au bout de laquelle on retrouve un feu de signalisation. Le feu de signalisation passe du vert au jaune, du jaune au rouge et du rouge au vert. Le feu rouge signifie l'interdiction de passer et le feu vert est le signal permettant de traverser la ligne d'arrêt. Le passage sur un feu jaune est permis, mais il est moins primé.

L'automobile a trois vitesses qu'elle peut conserver : 50km/h, 10km/h et 0km/h. La

route en question est une zone de 50km/h maximum, c'est pourquoi l'auto ne dépasse jamais cette vitesse. Pour s'arrêter, il serait trop brusque que la voiture passe de 50km/h à 0 en un instant. Celle-ci doit donc ralentir aux environs de 10km/h avant de s'arrêter complètement.

Lorsque la voiture traverse la ligne d'arrêt qui se trouve sous le feu de signalisation, l'épisode se termine et l'agent reçoit une récompense. La récompense est fortement positive si l'agent traverse sur un feu vert, faiblement positive s'il traverse sur un feu jaune et fortement négative si le feu est rouge. Également, lorsque la voiture progresse sur la route, elle est pénalisée si elle roule trop lentement et d'avantage si elle s'arrête au milieu du chemin. On s'attend à ce que l'agent apprenne à conduire jusqu'au feu d'intersection et qu'il ne franchisse la ligne d'arrêt que si le feu n'est pas rouge.

Il est intéressant d'utiliser ce domaine pour montrer l'évolution de la politique d'apprentissage, car les caractéristiques de l'espace d'états s'illustrent bien dans des graphiques en deux dimensions.

4.2 NEREUS

Le projet NEREUS est un projet initié par RDDC Valcartier, en collaboration avec Lockeed Martin Canada et l'Université Laval. Ce projet vise à faire une étude sur les différents algorithmes que pourrait utiliser le C2 pour effectuer les tâches décrites à la section 2.1. Le laboratoire DAMAS de l'Université Laval a été mandaté principalement sur ce qui concerne l'allocation des ressources en général. Plus spécialement, ce mémoire se concentre sur le processus de Plannification & Exécution en temps réel.

L'aspect temps réel est relativement important dans le projet NEREUS. Un agent temps-réel doit être fiable, adaptatif et particulièrement rapide. En effet, il est demandé d'une part de fournir des algorithmes efficaces capables d'agir sous une forte contraite de temps, mais d'autre part que ceux-ci soient capable d'intéragir directement avec un environnement d'évaluation. De plus, les algortihmes temps-réels sont souvent faciles à déployer dans un environnement réaliste, plus dynamique ou naturel. Dans ce chapitre, il sera expliqué comment une instance d'un problème WTA peut être implémentée, déployée et évaluée dans un environnement de simulation de combat maritime.

4.2.1 Frégate

Afin de définir une instance bien précise d'un problème d'allocation de ressources, il convient de choisir une plate-forme type et d'en énumérer les propriétés. Dans le cadre du projet NEREUS, il a été jugé approprié d'utiliser les propriétés typiques d'une frégate canadienne de type HALIFAX.

4.2.2 Armement

Une frégate canadienne de type HALIFAX dispose des ressources suivantes :
- SAM
- STIR
- Gun
- CIWS
- Chaff
- Jammer

Ces types de ressources sont regroupés en deux classes : *hardkill* et *softkill*. Un hardkill est un type d'armement qui a pour but de détruire physiquement une menace, tandis qu'un softkill tente de déjouer la menace en brouillant ses signaux ou en l'attirant vers un leurre.

Le *Surface to Air Missile* (SAM) est un hardkill de longue portée. C'est un missile anti-missile qui va directement à l'encontre de la menace pour provoquer une collision et la faire exploser. Lorsque qu'une menace aérienne se dirige vers le navire, le SAM est généralement le premier type d'armement utilisé pour détruire celle-ci, à une distance allant jusque 20km. Pour que le SAM puisse se diriger correctement vers sa cible, il doit être assisté d'un illuminateur, le *STIR*. Le STIR sert à pointer sur une menace et fait l'effet d'un marqueur.

Lorsque la menace s'approche encore, il est alors possible d'utiliser l'arme de moyenne portée qu'est le *Gun*. Le Gun est également un hardkill puisque qu'il s'agit encore de détruire la menace. Un Gun lance des salves de 5 balles à la fois, pour un maximum de 6 salves consécutives. Le Gun doit être combiné au STIR pour fonctionner. Cependant, un STIR peut suffire à la fois pour le SAM et le Gun si la menace illuminée est la même menace visée par les deux types de ressources.

Si la menace n'est pas détruite et qu'elle s'approche très près de la frégate, le *Close-In Weapon System* (CIWS) s'active et tente de détruire la menace en la mitraillant de

balles ; 55 balles à la seconde. Le CIWS dispose de son propre système de repérage donc n'a pas besoin qu'un STIR pointe la menace pour se mettre en oeuvre.

Dans la catégorie des softkills, le *Jammer* est une ressource peu coûteuse puisqu'il s'agit de brouiller le système de guidage de la menace. Pour se diriger, la menace émet des signaux qui sont réfléchis sur les parois du navire et ainsi retournés en direction de la menace. Cette dernière utilise ces signaux comme le ferait un radar pour s'orienter en direction de la plate-forme qu'elle veut atteindre. Le Jammer émet des signaux semblables à ceux réfléchis vers la menace et tente de fausser les lectures de ses capteurs. Le Jammer suggère à la menace, par des signaux biaisés, un emplacement fictif du navire. La menace ainsi bernée ne touchera pas sa cible et elle se retrouvera le bec à l'eau.

Le second type de softkill dont dispose une frégate canadienne est le *Chaff*. C'est un projectile lancé dans les airs, qui détonne au bout d'un moment et crée un nuage de pellicules métalliques qui refléteront fortement les signaux de guidage des menaces. La menace percevra alors un spectre qu'elle sera tentée de confondre avec sa cible. Ce type de ressources est souvent utilisé en conjonction avec une série de manoeuvres permettant de s'échapper du champ de vision de la menace. En effet, lorsqu'on déploie un Chaff comme un écran entre la menace et le navire, les spectres perçus par la menace seront pour le moins juxtaposés. En séparant tranquillement les deux spectres, la menace sera alors forcée de choisir lequel, parmi les deux, elle doit cibler. Selon le type de menace et la qualité de l'écran, la menace peut fort probablement se laisser dérouter. Afin d'améliorer les chances qu'elle change effectivement sa cible, le Jammer est souvent utilisé lors d'une pareille manoeuvre, suggérant ainsi la position du Chaff comme étant la position réelle du bateau.

4.2.3 Contraintes

Les ressources ne peuvent pas toutes être utilisées sans restrictions. Elles ont leurs limites et contraintes :

1. Chaque ressource a une portée à l'intérieur de laquelle la cible doit être engagée. Ces portées sont identifiées dans la table 4.1.
2. La plupart des menaces ont une zone aveugle, cela est fréquemment dû à la disposition des équipements sur la frégate. La table 4.2 montre les angles de début et de fin de la zone aveugle pour chaque type de ressource. La zone est définie en sens horaire. Il est à noter que les deux STIR présents sur la frégate ne sont pas positionnés identiquement. Il en résulte donc des zones aveugles différentes pour

Arme	Minimum	Maximum
SAM	2.2	20
Gun	0.9	5
CIWS	0	2.5
STIR	0	50

TABLE 4.1 – Portée des armes (km)

le premier et le deuxième STIR. De plus, le Jammer est contraint par deux zones aveugles, à l'avant et à l'arrière du navire.

Arme	Début	Fin
SAM	-	-
Gun	145	215
Jammer	350	10
	170	190
CIWS	245	15
STIR 1	240	120
STIR 2	60	300

TABLE 4.2 – Zones aveugles

3. Un SAM doit absolument être accompagné d'un STIR sinon il perd la trace de la menace et devient inefficace.
4. Un STIR peut difficilement, ou pas du tout, pointer une menace au travers d'un Chaff. Bien que cette contrainte ne soit pas aussi rigide que les précédentes, elle doit néanmoins être considérée puisqu'une telle assignation ne donne pratiquement que de mauvais résultats.

4.2.4 Secteurs

Considérant toutes les contraintes en relation avec l'azimut de la menace, on identifie 12 secteurs [Plamondon(2003)] autour de la frégate dans lesquels les assignations possibles risquent d'être les mêmes. Ce découpage en secteurs peut être très utile pour simplifier l'espace de caractéristiques de la menace. La figure 4.3 met en évidence par des traits foncés les azimuts utilisés pour discrétiser le domaine de la caractéristique

d'azimut. Puisque les actions possibles sont les mêmes pour chaque azimut d'un secteur donné, l'hypothèse est donc que la valeur d'une action est constante dans tout le secteur.

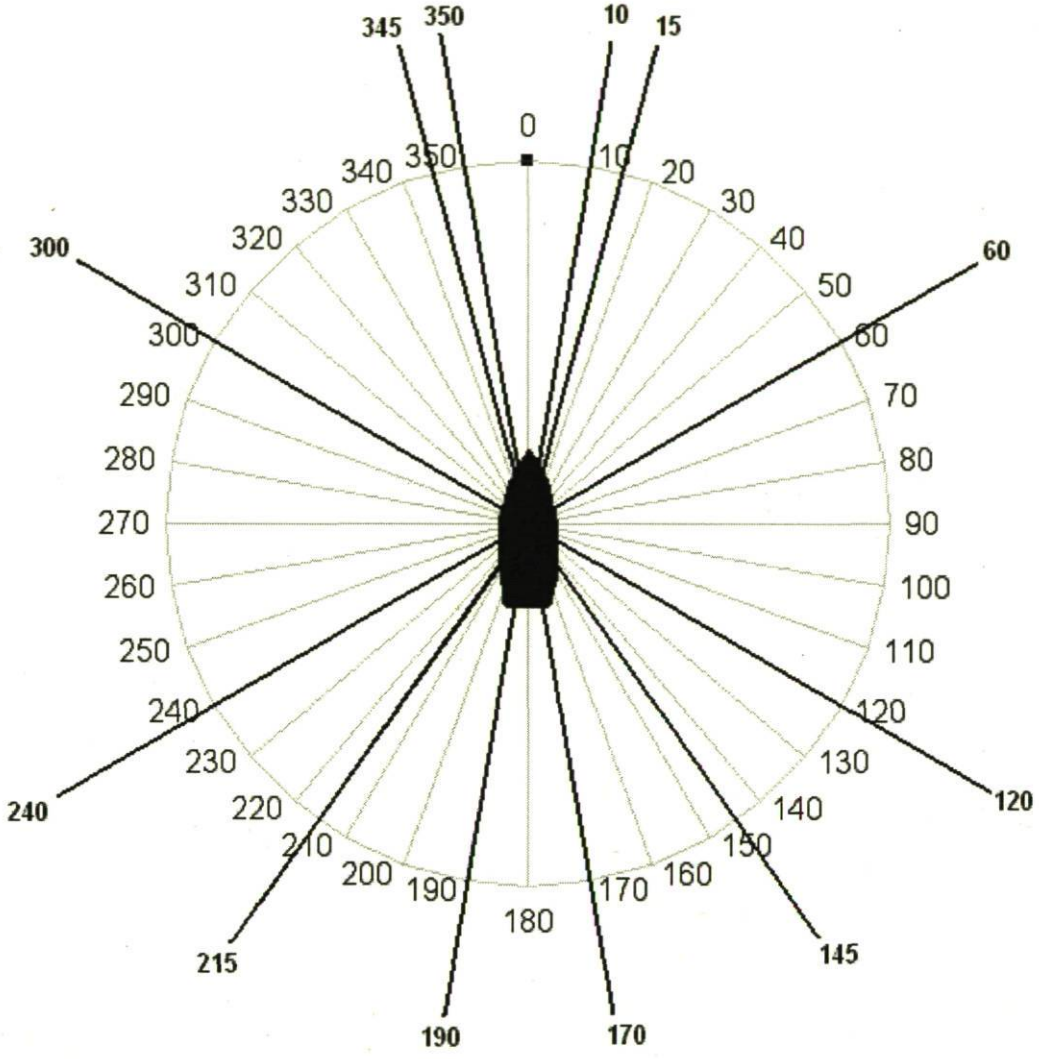

FIGURE 4.3 – Disposition des secteurs pertinents autour de la frégate.

4.2.5 Discrétisation

Afin de pouvoir utiliser le Q-Learning comme apprentissage par renforcement pour résoudre un MDP, il faut d'abord que l'espace d'état soit discret. Le nombre de menaces que peut recevoir une plate-forme navale est discret, cependant, les caractéristiques qui

permettent d'identifier et de décrire une menace peuvent être des valeurs dans des espaces continus. Dans ce cas, ces caractéristiques doivent être discrétisées de façon judiscieuse. En effet, le choix des valeurs discrètes peut influencer les résultats de l'apprentissage, car il est possible de créer des états cachés.

Les états cachés sont ceux qui ne peuvent être observés, car la valeur d'une de leurs caractéristiques se retrouve dans la même valeur discrète qu'un autre état qui, en temps normal, aurait des effets différents sur les résultats de la fonction de transition réelle dans l'environnement. Par exemple, si on discrétise un arc-en-ciel en rouge, jaune, vert et bleu, alors la couleur orange ne peut être observée. Si orange est un seuil dans la définition de la fonction de transition, alors cette discrétisation n'aura pas beaucoup d'impact. Cependant, si le fait de voir orange est completement différent que de voir rouge ou jaune, alors le fait d'observer l'une ou l'autre de ces deux dernières couleurs pourrait biaiser les résultats de l'apprentissage. Il faut donc discrétiser les caractéristiques de manière à ce que toutes les valeurs réelles rammenées à une valeur discrète soient susceptibles de représenter un même état.

Pour une frégate canadienne, le nombre de ressources applicables à une menace dans une position quelconque constitue un bon indice permettant de déterminer quelles sont les positions caractérisant des états semblables. En effet, puisque les probabilités sont assumées sensiblement constantes entre la distance maximale et la distance minimale de la portée d'une ressource, il est possible de dire que seules les possibilités d'assignation définissent l'impact du positionnement d'une menace. Ainsi, si on prend les portées de la table 4.1 combinées aux secteurs identifiés à la section précédente, il y a 96 zones significatives autour de la frégate. Ces zones constitueront les valeurs discrètes d'une caractéristique de position d'une menace dans l'espace d'état.

4.2.6 Fonction de récompenses

La fonction de récompenses, ou *Reward function*, doit être cohérente avec la tâche à accomplir. Il est espéré que l'agent apprenne à lancer les contre-attaques afin de détruire les menaces. Aussi, on ne sait pas exactement quelle(s) action(s) ont contribuées à la destruction d'une menace lorsque cela survient. Malgré que la récompense est à retardement, une récompense de 1000 est donnée pour chaque menace détruite. Cependant, si une menace atteint la plate-forme, une récompense négative, soit -1000, sera alors reçue par l'agent. De plus, puisque les actions sont contraintes et qu'aucun modèle d'engageabilité n'est utilisé, une valeur de 10 est soustraite de la récompense si l'action qui vient d'être tentée a généré une erreur lors de son exécution dans l'environnement.

4.2.7 Agrégation

Lorsque confronté à plusieurs menaces, chacune des menaces doit être observée de la même manière. En effet, les propriétés pertinentes pour une menace sont les mêmes que pour les autres. Le désavantage avec l'ajout de menaces supplémentaires est l'explosion de l'espace d'état de l'agent. Déjà exponentiel au nombre de propriétés d'une menace, ce nombre d'état devient alors exponentiel à son tour au nombre de menaces présentes dans un scénario donné.

Heureusement, chaque menace, bien identifiable uniquement, est semblable à une autre en ce sens qu'elle s'observe sur la base des mêmes propriétés. Par exemple, dans un scénario contenant la menace 1 et la menace 2, ces deux menaces ont toutes deux une distance et un azimut respectifs et donc on peut dire que *la menace 1 est dans l'état* X *et la menace 2 est dans l'état* Y. Bien qu'elles soient identifiées uniquement par les numéros 1 et 2, ces menaces sont définies par les mêmes propriétés. Que la menace dans l'état X soit la menace 1 ou la menace 2 ne change rien au fait que *il y a une menace dans l'état* X *et une autre dans l'état* Y.

Le principe est donc de se baser uniquement sur les caractéristiques communes des objets semblables à observer. Suivant cette idée, il est possible de faire une agrégation simple de l'espace d'état global de l'agent si on considère les états des menaces individuellement.

Supposons que l'espace d'état global soit de N dimensions, correspondant au nombre N de menaces à observer. Pour les deux menaces précédentes, l'état global sera alors $< X, Y >$; soit une valeur de X sur le premier axe et Y sur le deuxième. Tel qu'observé précédemment, $< X, Y >$ est équivalent à $< Y, X >$ si la liste des propriétés des objets composant les axes de l'espace d'états global est commune à tous les objets. Afin de réduire l'espace d'état global, il convient alors éliminer l'une ou l'autre de ces coordonnées dans l'espace et n'utiliser qu'une seule coordonnée pour représenter les deux états équivalents. Une méthode simple serait de quantifier chacun des états que peut avoir une menace et ensuite trier les coordonnées de l'espace d'état global. Pour X et Y des états que peut avoir une menace et $X \leq Y$, la coordonnée qui sera toujours utilisée serait alors $< X, Y >$. Bref, si la coordonnée est $< Y, X >$, le tri sur les coordonnées résulte en $< X, Y >$, car $X \leq Y$. Dans le cas où $X = Y$, il se peut que la coordonnée reste $< Y, X >$ après le tri, mais puisque $X = Y$, alors dans ce cas $< Y, X >=< X, Y >$.

Les tableaux 4.3 et 4.4 contiennent les données concernant l'évolution de la taille de l'espace d'états en fonction du nombre de menaces et du nombre de caractéristiques par

Menaces	Normal	Agrégé	Reduction (%)
1	10	10	0
2	100	55	45
3	1000	220	78
4	10,000	715	92.85
5	100,000	2002	98
6	1,000,000	5005	99.5
7	10,000,000	11440	99.89
8	100,000,000	24310	99.98

TABLE 4.3 – Aggrégation de 1 à 8 menaces ayant 10 caractéristiques

Menaces	Normal	Agrégé	Reduction (%)
1	10^2	100	0
2	10^4	5050	49.5
3	10^6	171,700	82.83
4	10^8	4,421,275	95.58
5	10^{10}	91,962,520	99.08
6	10^{12}	1,609,344,100	99.84
7	10^{14}	24,370,067,800	99.98
8	10^{16}	325,949,656,800	99.99

TABLE 4.4 – Aggrégation de 1 à 6 menaces ayant 100 caractéristiques

menaces. Avec une aggrégation telle que proposée précédemment, on constate un taux de réduction considérable qui s'améliore à mesure que le nombre de menaces augmente. La courbe illustrée à la figure 4.6 met en évidence la croissance rapide du taux de réduction qui tend vers une réduction de 100% avec un nombre infini de menaces.

Avec des espaces d'états devenant exponentiellement grands selon le nombre de menaces, un taux de réduction de plus de 99% ralentit fortement l'explosion de la taille de l'espace d'états. Il est à noter que la réduction croit proportionnellement avec le nombre de menaces, mais également avec le nombre de caractéristiques. En effet, similairement à la figure 4.4 montrant la réduction selon les menaces, la figure 4.5 révèle également une réduction lorsque le nombre de caractérisques augmente. Bien entendu, l'espace d'état demeure exponnentiel, mais il est clair que les limites réalistes d'utilisation de cet espace d'états sont repoussées.

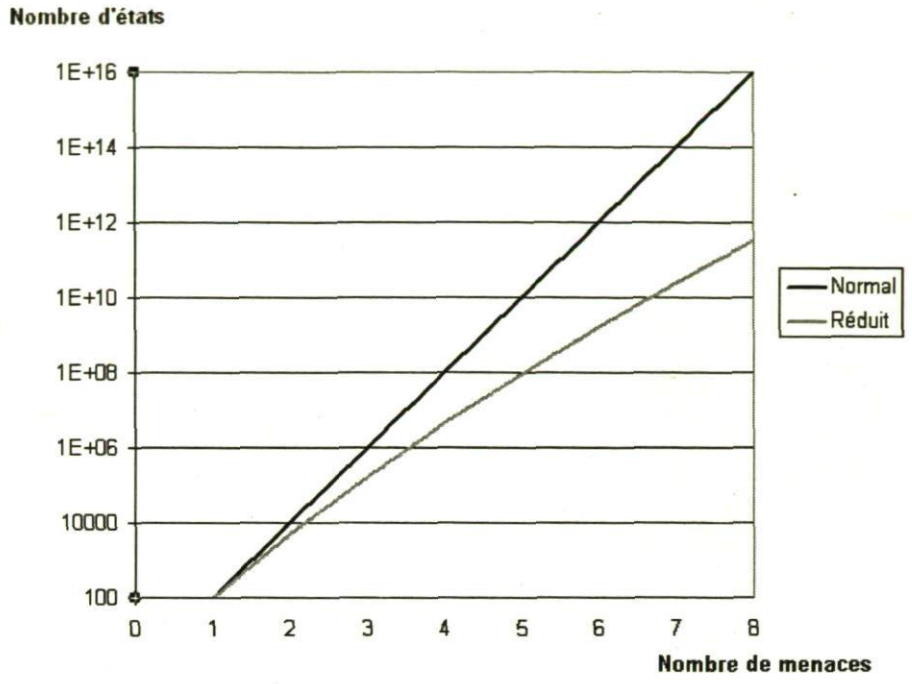

FIGURE 4.4 – Réduction de l'espace d'états selon les menaces.

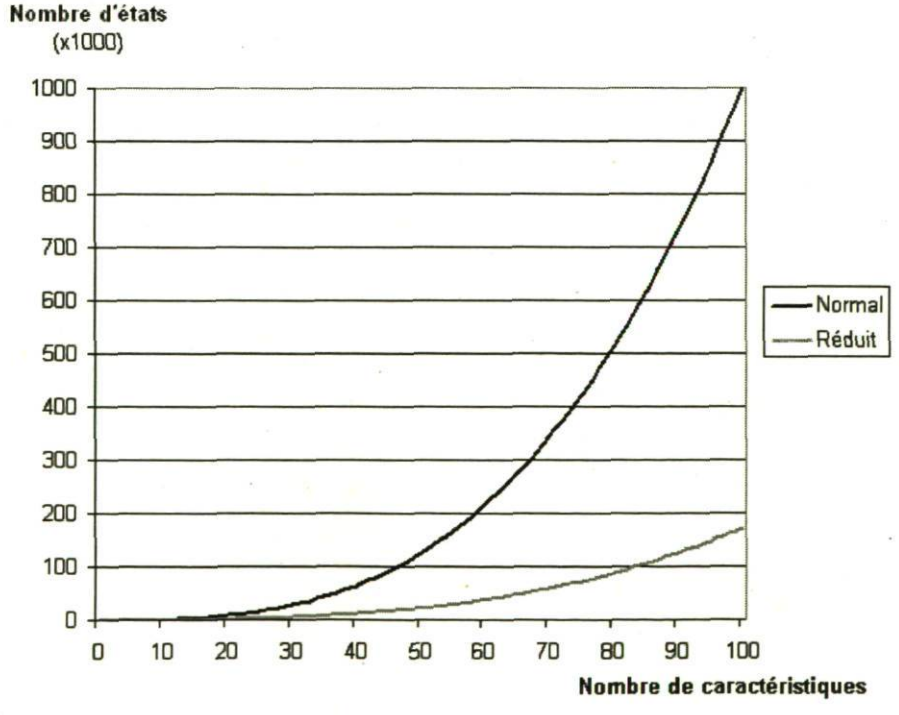

FIGURE 4.5 – Réduction de l'espace d'états selon les caractéristiques.

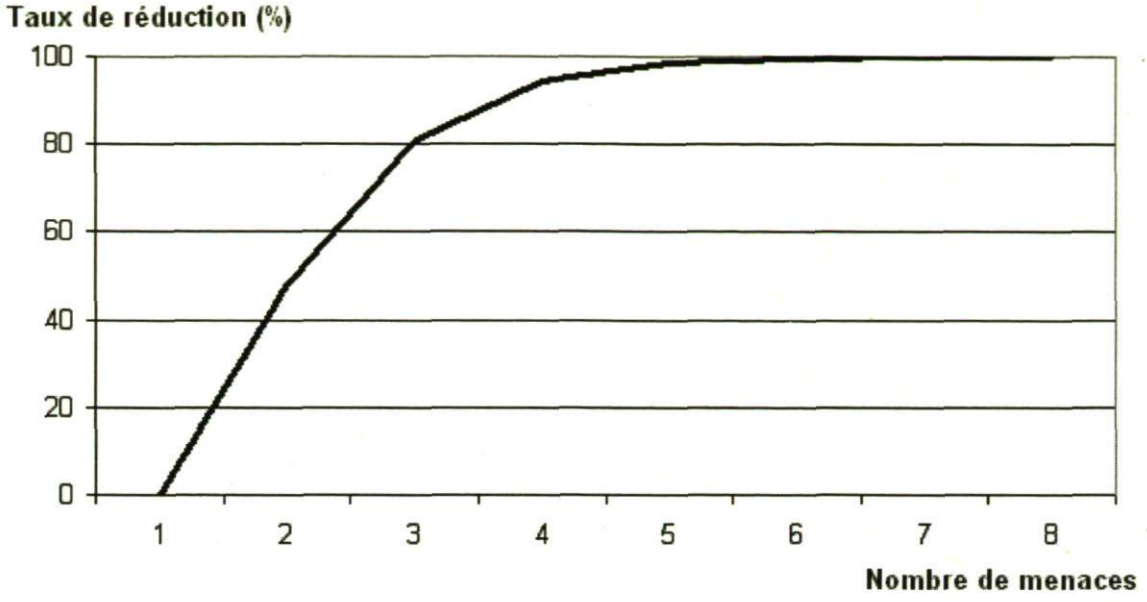

FIGURE 4.6 – Taux de réduction par rapport au nombre de menaces.

4.2.8 Résumé

L'instance du WTA définie dans les sections précédentes comporte cinq types de ressources et un nombre variable de menaces allant de 1 à 16 menaces (parfois même plus). L'objectif est de minimiser le coût total des ressources en assurant la survie de la frégate. Pour les fins du projet, il est supposé que les menaces détruisent assurément le navire si elles parviennent à établir le contact. Ainsi, la valeur donnée à une menace est équivalente à la valeur totale du navire. Puisque toutes les menaces ont la même valeur, ce sont les probabilités de détruire les menaces qui seront primées, et ainsi le coût des ressources n'affectera principalement que les choix entre des solutions plutôt équivalentes en terme de probabilité de survie. Le CIWS n'est pas inclus dans la planification puisqu'il s'agit d'une ressource de dernier recours.

Le résultat des assignations ressources-menaces est stochastique et le modèle est inconnu. Certes, des probabilités fictives et réalistes peuvent être fixées, mais celles-ci ne refléteront pas celles de l'environnement précisément. Elles peuvent être estimées par des simulations Monte Carlo dans l'environnement, mais cette pratique est computationnellement très coûteuse. Bien que les deux approches précédentes soient néanmoins envisageables, il n'en demeure pas moins que l'utilisation des probabilités ainsi établies empêchera d'atteindre l'optimalité dans l'environnement.

4.3 Simulateur

Pour évaluer les performances d'une solution proposée dans le cadre du projet NEREUS, un simulateur de combat maritime non classifié a été utilisé. Le simulateur Ship Air Defense Model (SADM) permet de modéliser une plate-forme maritime de combat dans un environnement virtuel. La plate-forme choisie est la frégate canadienne de type HALIFAX.

Le simulateur de combat maritime SADM doit être configuré sur certains volets avant de pouvoir exécuter une simulation. Les premiers volets permettent de spécifier le navire utilisé : type (une frégate HALIFAX), forme physique, écho radar qu'il propage, vitesse de déplacement, manoeuvres, etc. Ensuite, on peut configurer l'armement. La frégate peut être virtuellement équipée de tout l'armement détaillé au chapitre 4.2.2.

Finalement, les derniers volets permettent de spécifier quelles seront les menaces présentes lors de l'exécution. En spécifiant le nombre, le type, la vitesse, la distance initiale et autres attributs des menaces, un scénario est construit pour l'agent C2. On peut faire varier légèrement la plupart des paramètres afin que les scénarios semblables ne soient pas exactement les mêmes d'une fois à l'autre.

Voici trois scénarios testés dans le cadre du projet NEREUS :

Scénario menace isolée

C'est le plus simple des scénarios. Il consiste en une seule menace, qui peut arriver de n'importe quel point et se diriger vers la plate-forme ciblée à une vitesse variant entre 300 mètres secondes et 815 mètres secondes. Ce scénario de base a pour objectif de vérifier si l'algorithme de l'agent comprend bien la notion d'engagement. En effet, il peut donner à l'agent une expertise de base sur l'engagement lorsque simulé sur 360 degrés.

Scénario uniforme

Ce scénario est appelé uniforme, car la distribution d'une menace sur l'azimut est uniforme, c'est-à-dire que chaque menace a la même probabilité de se retrouver à un azimut ou un autre en début de scénario. Dans ce scénario, toutes les menaces sont identiques, à l'exception bien sur de leur azimut. Une vitesse de 815 mètres par seconde leur est donnée et leur trajectoire débute à une distance d'environ 40 kilomètres. Ce scénario stresse notamment l'agent sur les choix d'engagement. Selon la distribution

des menaces, des combinaisons d'engagements ou des séquences particulières peuvent s'avérer plus efficaces.

Scénario fixe

Celui-ci comporte trois menaces dont l'une est plus rapide que les autres, bien que plus loin lorsque le scénario débute. Parmi les moins rapides, la première menace est toujours au même azimut, tandis que l'autre varie de façon uniforme sur les 360 degrés. Le but de ce scénario est de tester si l'agent est capable de trouver une politique tenant compte de la dangerosité des menaces.

4.3.1 Architecture

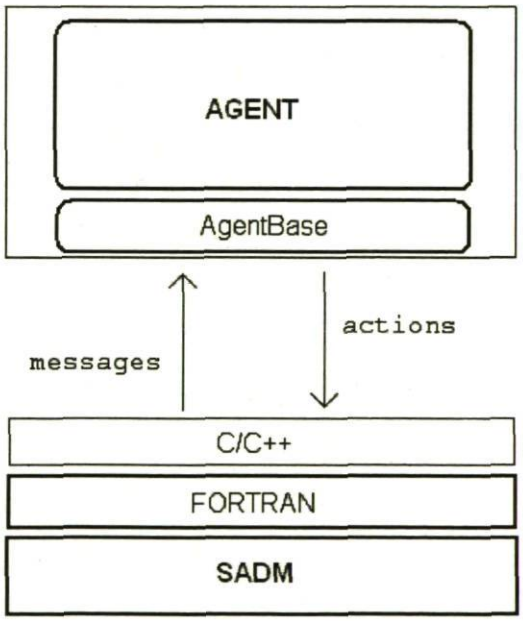

FIGURE 4.7 – Couches d'interaction dans l'architecture de l'agent.

Une architecture de test a été établie afin d'assurer la cohésion des différents éléments requis pour utiliser un agent en temps réel dans un environnement SADM. L'architecture de base a été utilisée dans la plupart des simulations du projet NEREUS. Elle offre une bonne abstraction des technicalités du simulateur SADM, tout en ne limitant pas les fonctionnalités.

La figure 4.7 montre une vue simplifiée de l'architecture de test utilisée. L'outil de simulation SADM est un logiciel indépendant, mais qui offre une interface de communication permettant de contrôler les décisions du C2 d'une simulation donnée. Cette interface a été introduite dans l'architecture de test pour effectuer un pont communicationnel entre l'agent et l'environnement SADM. L'interface fournie par SADM permet d'échanger deux types de messages :

- les messages contenant l'état de l'environnement (observations)
- les messages suggérant les actions à effectuer

Additionnellement, la figure 4.7 montre que les observations passent de l'environnement jusqu'à l'agent et les actions sont envoyées à l'environnement depuis l'agent. On remarque dans cette figure la présence de couches intermédiares. Effectivement, le langage utilisé par les différentes composantes n'est pas le même et par conscéquent, une couche de tranduction entre les langages Fortran et C/C++ est nécessaire. De plus, les données bruts reçues de SADM sont triées et organisées dans une couche de structuration de base pour l'agent. Cette couche facilite notamment la conception d'agent, qui peut alors être faite dans un langage plus convivial et orienté-objets.

4.4 Framework

Face au nombre d'algorithmes disponibles et de domaines d'application, il est tentant de chercher des moyens de réutiliser l'implémentation d'un algorithme d'une application à une autre. C'est pourquoi un framework a été mis sur pied. Celui-ci permet d'encadrer les algorithmes dans un moule prédéfini, qui est assez générique pour permettre la réutilisation des implémentations sans devoir les modifier.

Un avantage majeur de l'adoption d'un tel outil est sans doute la simplicité d'utilisation, sans compter que cette utilisation devient standardisée d'agent en agent. En effet, les algorithmes étant tous implémentés sur la même base, la création d'un nouvel agent ne nécessite que quelques étapes communes à tous les agents, c'est-à-dire :

1. implémentation des actions
2. définition de la perception
3. instanciation du modèle
4. choix des algorithmes
5. création des signaux d'observation

Un exemple d'implémentation pour le domaine du feu de signalisation de la section 4.1.2 est présenté en annexe A.

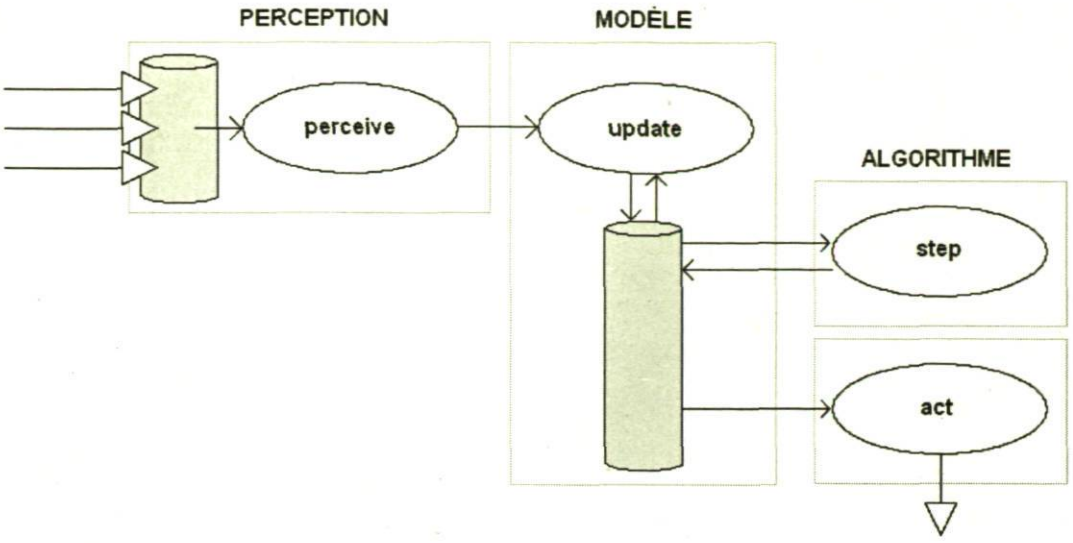

FIGURE 4.8 – Schéma de haut niveau du fonctionnement du framework.

Il est à noter que le framework n'est applicable qu'aux agents utilisant l'apprentissage par renforcement ou se conformant au schéma présenté à la section 1.1.2 du chapitre 1. La figure ci-dessous montre le paralèlle entre les notions d'agents et l'intégration du framework. On peut remarquer des similitudes dans cette figure lorsque comparée avec la figure 1.1.

Les signaux entrant dans le module de perception sont les caractéristiques qui composent le signal x décrivant l'environnement. La méthode *perceive* joue le rôle de la fonction d'observation $O()$ et transforme les signaux en une observation o que le module de modèle comprend. C'est par la méthode *update* que le modèle reçoit l'observation et le signal de renforcement r et crée l'instance de la transition obtenue au temps t. Cette transition est utilisée pour mettre à jour les paramères nécessaires à l'exécution de l'algorithme d'apprentissage. Le module d'algorithme est divisé en deux méthodes. La méthode *set* permet d'effectuer une étape de raisonnement de l'agent en modifiant les données enmagasinées dans le module de modèle. En effet, les valeurs permettant de générer la politique finale ainsi que la croyance sur l'état actuel de l'agent sont mémorisées dans le module de modèle. La méthode *set* utilise ces valeurs comme paramètres pour effectuer l'apprentissage. C'est l'équivalent de la fonction S présentée au chapire 1. La deuxième méthode constituant le module d'algorithme permet de générer une action a à effectuer dans l'environnement. Effectivement, la méthode *act* fourni une action a qui respecte les besoins de l'algorithme d'apprentissage en ce qui concerne l'exploration et l'exploitation des états de l'environnement.

Tous les algorithmes de la section 3.2 peuvent être implémentés suivant ce frame-

work. Une fois implémentés, la réutilisation d'un agent à l'autre des algorithmes codés se fait facilement, car les algorithmes dépendent du modèle et non de l'agent ou de l'environnement. Seuls les signaux et la liste des actions diffèrent d'un environnement à l'autre, mais l'algorithme les considère de manière abstraite. L'implémentation reste donc la même pour des domaines d'application différents.

Afin de rendre possibles le plus d'applications différentes au niveau de la perception, le framework permet le chaînage de plusieurs modules de perception. Lorsqu'un module de perception n'est pas directement rattaché au modèle, l'observation résultante peut être relié à un signal d'un autre module de perception à l'aide de la méthode de chaînage *connect*. La figure 4.9 illustre le principle de chaînage des perceptions.

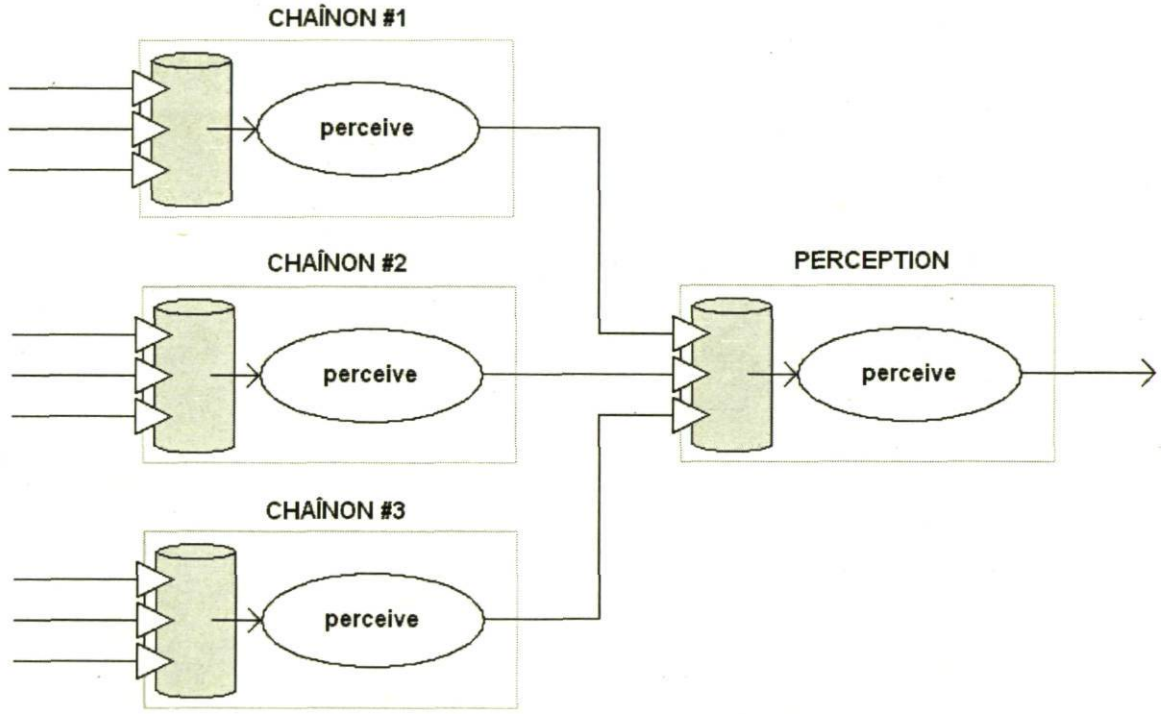

FIGURE 4.9 – Chaînage des modules de perception dans le framework.

4.5 Unité

Lorsqu'on se donne des outils de travail et que ceux-ci sont clairs, facilement utilisables et unis entre eux, il est alors possible de se focaliser uniquement sur les concepts fondamentaux de l'agent à développer. Avec des environnements d'essai tels que présentés dans ce chapitre et avec l'aide d'une architecture uniformisée comme le framework décrit

à la section 4.4, il est plus aisé de se concentrer sur l'implémentation et l'évaluation des algorithmes du chapitre 3.

Chapitre 5

Évaluation

Les différents environnements d'essai documentés au chapitre précédent ont été implémentés suivant les lignes directrices suggérées par le framework.

Afin de pouvoir les comparer, la plupart des algorithmes en-ligne du chapitre 4 ont été testés dans les environnements choisis.

5.1 Métriques

Avant toute chose, il convient de déterminer ce que l'on veut mesurer. Il faut des propriétés quantifiables pour pouvoir comparer objectivement les algorithmes. Il ne suffit pas de dire qu'une méthode est meilleure qu'une autre ; il faut spécifier le contexte pour lequel cette conclusion a été émise et quelles ont été les caractéristiques comparées. Par exemple, deux personnes promouvant respectivement les méthodes A et B peuvent toutes deux affirmer sans mentir que leur méthode est supérieure à l'autre. Ce qui n'est pas dit, c'est que l'une prétend que sa méthode surpasse l'autre, mais sur le plan de la vitesse, tandis que l'autre, elle, a une meilleure méthode, mais sur le plan de l'utilisation des ressources. Mais dans cet exemple, est-il mieux d'avoir de la vitesse ou moins de ressources utilisées ? C'est un compromis qui dépend largement du contexte dans lequel on est. Bref, il faut comprendre que la comparaison peut être très subjective, du moins quant au choix des caractéristiques à comparer. Néanmoins, il est possible de rester objectif lorsqu'on compare les caractéristiques choisies si celles-ci sont mesurables et quantifiables. Dans ce cas, nous pouvons établir la valeur les métriques logicielles basées sur les caractéristiques.

Les métriques pertinentes pour chacun des environnements sont les suivantes :
- Nombre de backups
- Nombre de cycles
- Nombre d'états
- Utilisation mémoire

Il faut garder à l'esprit que les équipements informatiques sont souvent différents quant à leurs performances. Un même algorithme exécuté sur deux machines différentes peut parvenir à une même solution, mais avec un temps différent en terme de secondes. C'est pourquoi le temps d'exécution est mesuré en nombre de cycles, en nombre de backups ou, à la limite, en nombre d'instructions.

5.2 Procédures d'évaluation

Les algorithmes d'apprentissage ont deux phases distinctes d'exécution que sont l'apprentissage et l'évaluation. Une question vient vite à l'esprit lorsqu'il s'agit de passer d'une phase à l'autre. Il n'est en effet pas toujours évident de savoir quand l'algorithme a terminé son apprentissage. Lorsque l'agent apprend une politique optimale qu'il ne connaît pas, il n'est pas clair pour celui-ci de déterminer le moment où il atteint son but.

Ainsi que mentionné lors de la description de l'algorithme d'Itération de Valeurs, il est parfois préférable d'arrêter l'apprentissage lorsque l'ensemble des mise à jour n'a généré qu'une minime variation des Q-valeurs. Cependant, la politique optimale peut bien souvent être obtenue avant ce moment, dépendamment de l'exploration. Dans des environnements déterministes par exemple, si les valeurs ne se trouvant pas dans la suite logique des états rencontrés lors de l'exécution de la politique optimale ne sont pas précis, les performances de la politique n'en seront pas affectées.

Pour des fins de comparaison, on peut fixer un nombre d'épisodes selon lequel chaque algorithme s'entraîne, après quoi ils sont tous soumis à un nombre fixe d'épisodes pour évaluation des performances. Il s'agit d'établir des résultats selon le principe général de Monte Carlo.

Parfois, il peut être envisageable d'analyser le taux de convergence selon le nombre d'itérations dans l'environnement, c'est-à-dire d'essayer de mesurer la distance en terme de qualité de la politique par rapport à la solution optimale. La mesure de cette distance est relativement subjective, car on peut la mesurer de plusieurs façons : la moyenne des différences entre les Q-valeurs et les Q^*-valeurs, le nombre de cycles pour atteindre le

but par rapport au chemin optimal, etc. Lorsqu'un rendement satisfaisant est atteint, on cesse l'apprentissage. Il faut faire attention, un algorithme approximatif peut ne jamais atteindre un résultat escompté.

Dans l'idée du Policy Iteration, il est possible d'utiliser le critère selon lequel la politique ne change pas comme critère d'arrêt. Dans les cas d'environnement dans lequel on cherche à optimiser une tâche de contrôle, i.e. la voiture et le feu de signalisation, il est souvent suffisant de constater que la politique est la même depuis un certain nombre d'itérations, et donc nous pouvons avoir une certaine confiance que celle-ci ne changera plus.

L'alternance entre les phases d'apprentissage et d'évaluation est également possible. Par exemple, pour le labyrinthe déterministe dont les résultats sont donnés en table 5.1, l'apprentissage prend fin lorsque la politique mène directement à la sortie. À la fin de chaque épisode, la politique est testée pour vérifier si celle-ci permet d'atteindre la sortie sans détour. Dans le cas de ce labyrinthe, si l'épisode d'évaluation se termine en 9 cycles, alors la politique trouvée a atteint l'optimum puisque le point de départ est fixe.

Algorithme	**Episodes**	**Cycles**
Q-Learning	14	6876
Dyna-Q	4.2	1698
PrioritizedSweeping	4.4	1788
Dyna-Q-Queue	4	1568

TABLE 5.1 – Resultats

Ces résultats ont été obtenus en vérifiant à chaque épisode si l'algorithme avait trouvé le chemin optimal vers la sortie. De cette façon, le nombre de cycles dans la table reflète, à l'épisode près, le temps de convergence de chacun des algorithmes.

On remarque que les méthodes avec boucle de planification convergent en moins de cycles. Cependant, on sait que ceux-ci font jusque 5 fois le nombre de mises à jour de l'algotihme de Q-Learning pour un même cycle. Le fait marquant est la réduction du nombre d'épisodes nécessaires pour compléter la convergeance. Avec 5 backups additionnels par cycle, les algorithmes sont capables de converger en 3 fois moins d'épisodes.

Dans le cas du labyrinthe stochastique, on ne peut pas cesser l'apprentissage lorsque l'agent réussit à faire le trajet optimal. En fait, c'est qu'on ne peut pas vraiment savoir si l'agent a obtenu une bonne politique, car même une politique optimale peut dévier

dû aux probabilités que les actions échouent. L'agent risque donc de se retrouver à un endroit non voulu dans le labyrinthe et il lui faut donc rattraper la déroute de façon optimale. La politique doit donc inclure tous les scénarios possibles d'exécution et doit toujours effectuer la meilleure action, peu importe dans quel état l'agent se trouve. Il est très difficile de vérifier si l'agent a atteint ce point seulement par l'observation de son cheminement. Il faut en effet vérifier si les actions concordent avec les probabilités de déroute. Ce genre de vérification revient à effectuer l'Itération de Valeurs et nécessite donc la connaissance de T. Heureusement, dans le cas de notre labyrinthe, les probabilités associées aux actions sont connues et nous pouvons donc comparer la politique obtenue avec celle calculée par l'Itération de Valeurs. Dans le cas où le modèle de transition est inconnu, il devient beaucoup plus ardu de vérifier si la politique a convergé.

Le domaine de test impliquant la voiture et le feu rouge ne révèle pas le même genre

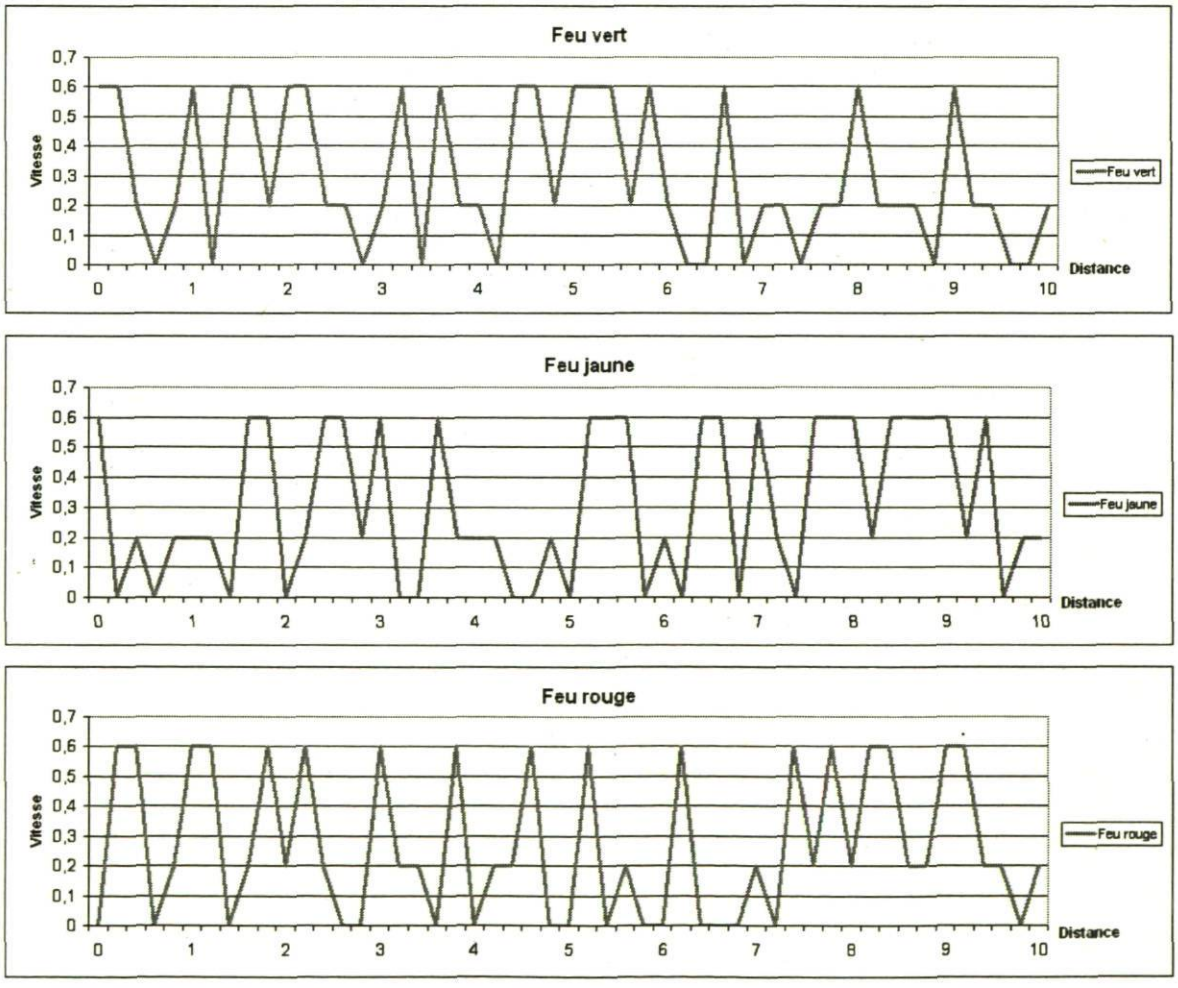

FIGURE 5.1 – Politique pour le Feu de signalisation après mille épisodes.

d'information. Dans ce domaine, on cherche à observer une optimisation du contrôle de la vitesse. On peut donc voir rapidement si l'algorithme est souhaitable dans une situation où le reward est attribué après une bonne utilisation des actions de contrôle pour la tâche à effectuer.

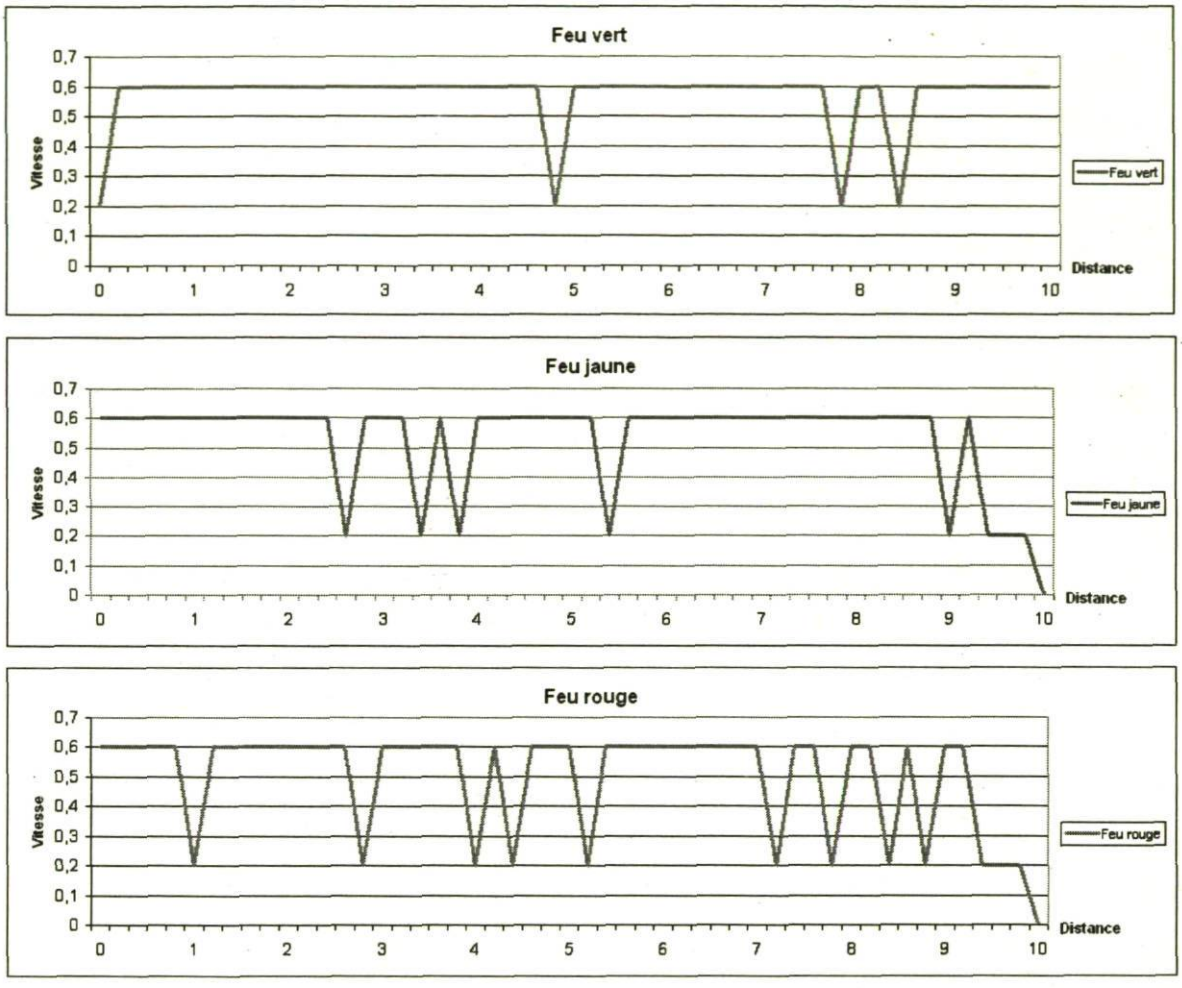

FIGURE 5.2 – Politique pour le Feu de signalisation après un million épisodes.

Néanmoins, la vitesse à laquelle un algorithme parvient à obtenir une politique de contrôle optimale est importante. En effet, dans des situations contraintes par le temps, il est préférable d'avoir un algorithme convergeant rapidement vers une politique acceptable.

L'exemple de l'agent automobiliste est tout à fait approprié pour vérifier visuellement l'évolution de la politique. Dans la plupart des cas, l'évolution peut être facilement testée par les résultats, mais il est rare que l'on s'attarde à regarder l'ensemble de la politique sur l'espace d'état en entier. La figure 5.1 de la politique d'un agent Q-Learning dans l'environnement de conduite après mille épidodes peut être comparée avec la po-

litique après un million d'épisodes illustrée à la figure 5.2.

Après un million d'épisodes, la politique de l'agent contrôlant l'automobile produit un comportement souhaité dans l'environnement. Sauf quelques exceptions probablement dues à l'exploration, l'agent a compris qu'il est préférable de mettre plein gaz jusqu'à l'approche de l'intersection. Lorsque la lumière est jaune ou rouge à l'approche de l'intersection, la voiture s'immobilise à la ligne d'arrêt pour ensuite repartir lorsque le feu devient au vert.

Ce genre de comportement est exactement ce que l'on désire obtenir. Des résultats semblables montrent qu'il est souhaitable d'appliquer des méthodes d'apprentissage à des problèmes de contrôle simples. La politique obtenue ici n'est pas optimale, mais elle laisse présager qu'elle le deviendra bientôt, ainsi que le démontrent les preuves de convergence du Q-Learning.

5.3 SADM

Il a été constaté jusqu'à maintenant que les méthodes mixant l'apprentissage des Q-valeurs et l'apprentissage du modèle convergent beaucoup plus rapidement que l'algorithme de Q-Learning standard. Cependant, la quantité d'espace mémoire utilisée est bien supérieure lorsqu'il s'agit de conserver les données permettant de faire l'approximation du modèle. Plus encore, cet espace est proportionnel au nombre d'états visités.

Dans certains cas, un algorithme mettant plus de temps à converger est préférable à un algorithme plus rapide, mais devant interrompre l'apprentissage parce que l'exploration est allée à un point tel que l'espace mémoire est complètement utilisé. Pour cette raison, l'algorithme de Q-Learning est recommandé pour un agent C2.

L'environnement temps-réel simulé par SADM est très complexe. Plusieurs facteurs entrent en ligne de compte, tant qu'il est presque impossible pour un programmeur de gérer toutes les possibilités de transition de cet environnement de simulation. En effet, certains paramètres de cet environnement sont des fonctions et plusieurs aspects du combat maritime y sont modélisés de manière plutôt complexe. De plus, plusieurs paramètres sont également probabilistes de sorte que l'environnement devient rapidement imprévisible.

Cet état d'incertitude de l'environnement permet de vérifier le comportement de

l'agent dans des conditions similaires à un environnement naturel, dans lequel la plupart des phénomènes sont difficilement prévisible, particulièrement lorsqu'il s'agit de les prévoir en temps-réel.

5.4 Agent Réflexe

Pour démontrer la praticabilité de la gestion du C2 par un agent entraîné par méthodes d'apprentissage par renforcement, cet agent a été confronté à un agent-réflexe. Cet agent-réflexe est un agent réactif, utilisant des règles d'engageabilité simples pour initier ses actions. Ce genre d'agent a fait ses preuves, notamment dans les études précédemment faite pour le compte du projet NEREUS .

Globalement, l'agent-réflexe engage les menaces selon une classification basée sur la dangerosité. La dangerosité d'une menace, dans le cas présent, est représentée par le produit de sa vitesse, son angle d'approche et sa distance.

L'agent assigne à la menace la plus dangereuse un SAM et un Gun, avec un premier STIR. Avec le second STIR, l'agent engage la menace se trouvant au deuxième rang dans la classification de dangerosité. Lorsqu'une menace est détruite, cette règle est appliquée de nouveau jusqu'à ce que toutes les menaces soient détruites.

L'agent-réflexe est un bon point de repère dans la comparaison des différentes méthodes de planification. Ce genre d'algorithme n'utilise pas de planification à long terme et on espère faire mieux que ce genre d'algorithme lorsqu'on emploie des méthodes de planification nécessitant beaucoup de calculs. L'agent-réflexe agent sera donc évalué dans des conditions semblables à l'agent Q-Learning afin de voir si ce dernier peut justifier la planification implicite d'un agent ayant bénéficié d'un apprentissage par renforcement.

Afin que la comparaison soit la plus juste possible, les deux agents auront exactement les mêmes actions disponibles pour interagir avec l'environnement. Il est à noter que ceci explique probablement les écarts entre les performances du même agent-réflexe présentées ici et ceux présentés dans les autres travaux connexes du projet NEREUS [S. Chouinard and Chaib-draa(2007)].

Puisque l'agent-réflexe ne gère que les moyens de défense du type hardkill, la comparaison sur le banc d'essais SADM se fera sur les actions propres aux SAM, Gun et STIR. Les déplacements de la frégate ne seront pas permis.

5.5 Agent Q-Learning

L'agent Q-Learning a été implémenté en suivant le principe événementiel présenté à la section 2.2.4. Les événements sont générés en fonction des états. Utilisant le framework, le module de perception servira de point de repère pour déceler le changement d'état. Ainsi, des mises à jours sur le modèle et des cycles sur les algorithmes seront effectués seulement lorsque l'observation courante sera différente de l'observation enregistrée précédemment dans le modèle.

Ce genre de procédé événementiel permet entre autres de recourir à une discrétisation des dimensions continues de l'environnement sans que la propriété de Markov soit compromise dans le modèle utilisé. La discrétisation des dimensions est ici souhaitable afin de restreindre la multitude d'états possibles en un nombre d'états plus raisonnable.

Afin d'obtenir des résultats comparatifs, l'exécution des deux agents a été faite dans le scénario uniforme et le scénario fixe présentés au chapitre précédent. Les deux scénarios ont été exécutés 100 fois chacun pour chaque agent. Pour avoir un aperçu général sur les performances dans ces scénarios, des paramètres ont volontairement été laissés au hasard. Ainsi, chaque instance d'un type de scénario respecte l'idée globale du scénario, mais est unique en quelque sorte car les variations des paramètres aléatoires ne sont pas susceptibles de donner exactement le même scénario deux fois.

Une fois l'exécution terminée, le simulateur SADM donne une foule de données concernant le scénario simulé. La métrique qui nous intéresse le plus est le taux de survie de la frégate. Le coût des ressources est important, mais l'objectif principal est de garder la plate-forme en service.

Le taux de survie est établi par le rapport du nombre d'exécutions où la frégate est restée intacte sur le nombre total d'exécutions du même scénarios. Dans le cas du scénario uniforme, un taux de survie est donné pour chaque quantité de menaces présentes dans le scénario. De plus, dans le cas de l'agent Q-Learning, l'évolution du taux de survie est analysée en fonction du temps d'apprentissage alloué à l'agent.

5.6 Résultats

La figure 5.3 montre la courbe d'apprentissage de l'algorithme Q-Learning. Premièrement, l'algorithme a été exécuté pendant quelques jours, avec des scénarios ayant

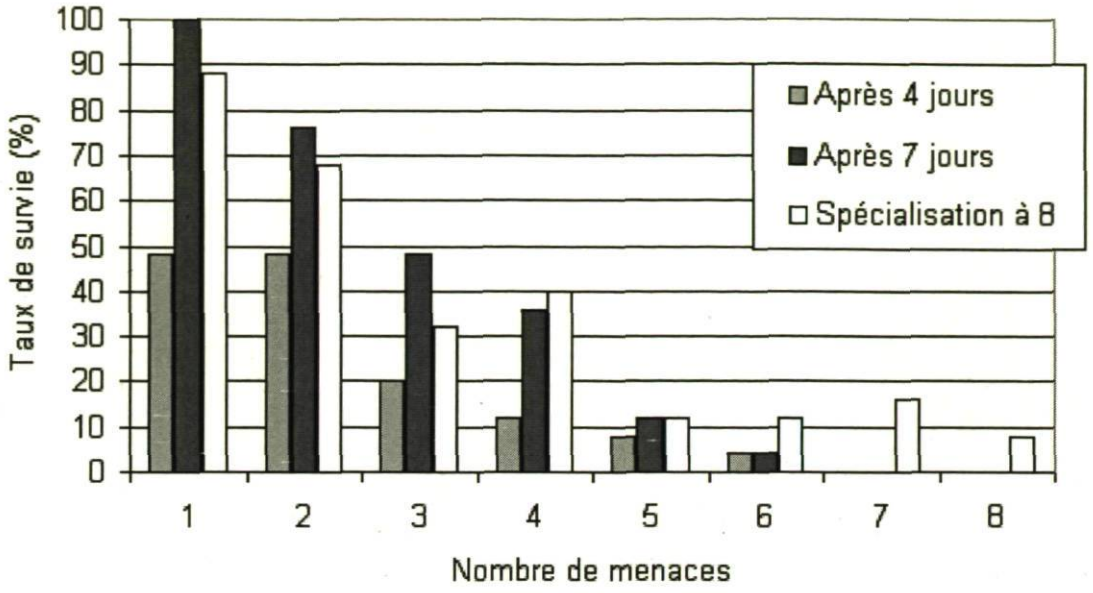

FIGURE 5.3 – Courbe d'apprentissage de l'agent C2 dans SADM.

1 à 8 menaces. La politique a ensuite été testée dans SADM pour obtenir les probabilités de survie. Suivant la même méthodologie, une politique plus acceptable a été obtenue après une semaine. Les résultats comparatifs de performance de la politique obtenue sont affichés dans la figure 5.4. Cependant, puisque l'algorithme Reflex a obtenu de meilleurs résultats pour les scénarios dans lesquels il y avait plus de 4 menaces, une séance d'un million d'épisodes avec scénarios de 8 menaces a été réalisée afin de spécialiser l'agent en présence d'autant de menaces. Le comportement de la politique ainsi obtenue (figure 5.5) est devenu généralement préférable à celui du Reflex pour des scénarios de 4 menaces ou plus. En contrepartie, les performances ont diminué pour les scénarios impliquant 3 menaces ou moins. Ceci est possiblement causé par le fait que la fonction de récompenses attribue une plus grande récompense pour avoir bien agit contre plus de menaces et moins de récompense lorsque le scénario semble plus facile, i.e. contre moins de menaces. L'agent a été récompensé plus généreusement pour s'être tiré d'une situation complexe (8 menaces), donc l'agent pourrait être tenté d'attendre que la situation se complique dans l'espoir d'obtenir une plus grande récompense, toujours dans un souci de maximiser la somme des récompenses obtenues.

L'agent Q-Learning a été testé dans un scénario orienté-softkill. Le scénario n'impliquait qu'une seule menace ; le but étant d'apprendre une politique utilisant les softkills exclusivement. La figure 5.6 montre comment l'agent s'est défendu pour chaque angle d'attaque de la menace. Les distances des points par rapport au centre du graphique

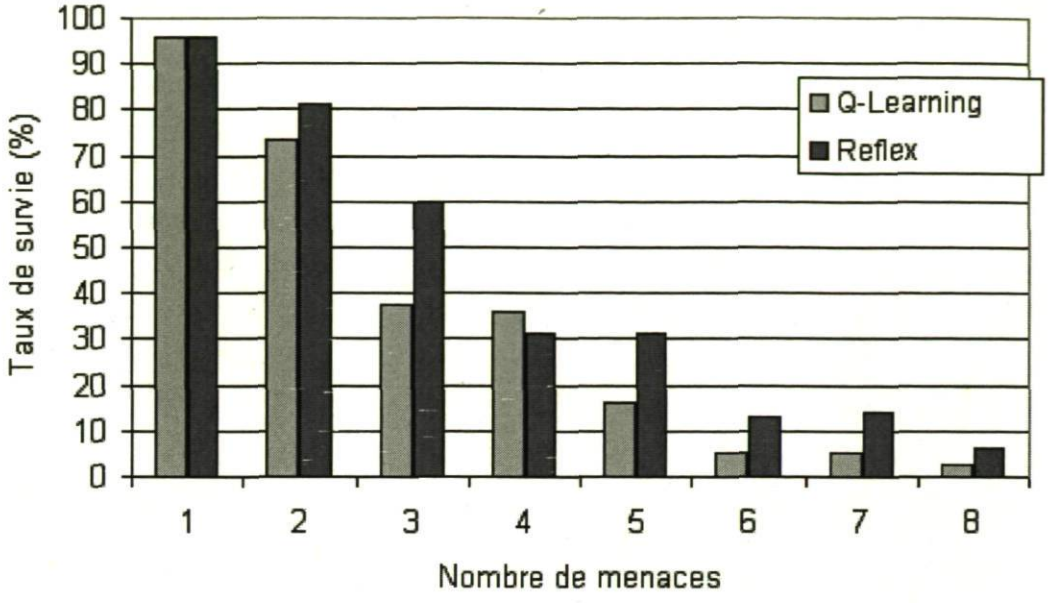

FIGURE 5.4 – Première comparaison de la politique Q-Learning à un agent Reflex.

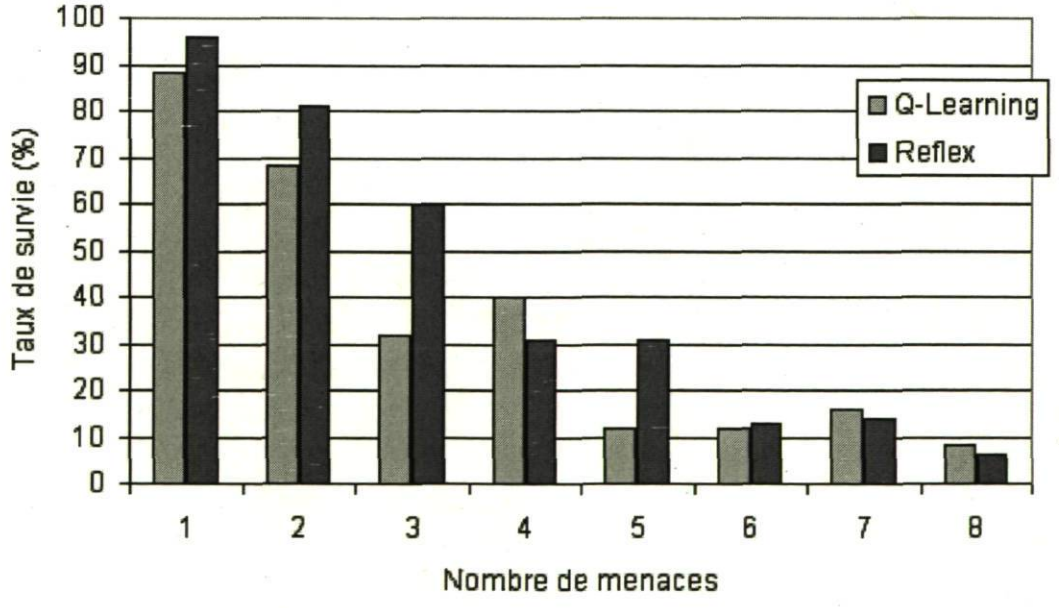

FIGURE 5.5 – Comparaison de la politique Q-Learning après spécialisation.

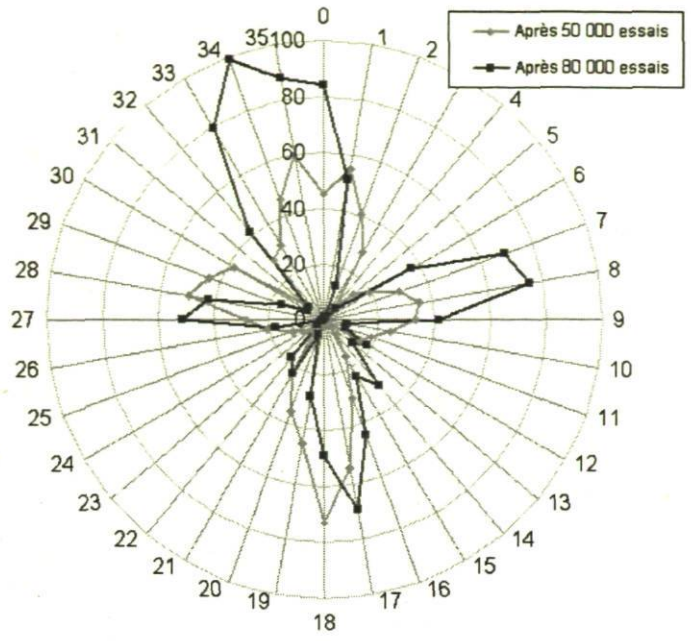

FIGURE 5.6 – Résultats de l'apprentissage pour un armement softkill.

donne la probabilité de survie pour chaque azimut. Il est à noter que les azimuts ayant les plus faibles taux de survie correspondent aux alentours des angles de lancement des Chaffs. On remarque une amélioration de la qualité de la politique entre 50k épisodes et 80k épisodes. Ces résultats auraient probablement pu être davantage concluants si un espace d'état plus complet, incluant Chaff et Jammer, avait été utilisé.

5.6.1 Analyse des résultats

À partir d'une expérience nulle (i.e. Q-valeurs initialisées à zéro), l'algorithme a été capable d'apprendre à agir dans un environnement nouveau et inconnu. Avec un minimum de connaissances donné à l'agent, celui-ci est parvenu à faire des assignations de ressources pour des menaces en utilisant les actions mises à sa disposition. De ce point de vue, l'algorithme Q-Learning a rempli les exigences d'un processus de décision fonctionnel.

De plus, l'algorithme a démontré une bonne performance dans le scénario uniforme, où les menaces pouvaient arriver de n'importe quelle direction avec une vitesse équivalente. Cela veut dire que l'agent a été capable de choisir parmi les menaces quelles étaient celles qui devaient être engagées en premier et ce, sans avoir à sa disposition de classement par dangerosité.

Un autre fait est que les actions utilisées n'étaient pas déterministes. La plupart des actions pouvaient générer des résultats très différents dans un scénario. De plus, plusieurs aspects du simulateur SADM sont stochastiques et ont pu affecter le résultat des actions. Même si les actions fournies à l'agent n'étaient pas nécessairement optimales pour l'environnement SADM en terme d'exécution, l'algorithme a pu généraliser sur le modèle de transition. Cette généralisation est un point positif en ce sens qu'elle permet de gérer l'incertitude dans l'environnement.

L'apprentissage prend beaucoup de temps et plusieurs cycles sont nécessaires pour explorer et apprendre suffisamment. L'espace mémoire alloué à l'agent devient vite comblé lorsque le scénario implique trop de menaces, car plus il y a de menaces à gérer, plus l'exploration est fastidieuse.

Pour ce qui est des scénarios, l'algorithme n'a pas été en mesure d'égaler l'agent-réflexe en performance pour le scénario fixe, où une menace était clairement plus rapide que les autres et provenait toujours du même azimut. L'algorithme n'a pas rejoint les attentes dans ce type de scénario, où on avait fait l'hypothèse que la différence entre les Q-valeurs aurait favoriser l'exploration. Le Q-Learning avait été soupçonné de pouvoir trouver facilement une politique acceptable pour ce genre de scénario plus complexe.

5.6.2 Améliorations

L'espace d'états est aux prises avec la malédiction de la dimentionnalité (*Curse of Dimensionality*). C'est donc dire que le nombre de dimensions augmente exponentiellement le nombre d'états. Afin de réduire l'effet de ce mauvais sort, il est suggéré que les dimensions soient discrètes et toujours plus petites. L'implémentation actuelle utilise une discrétisation simple, mais l'espace d'états est encore trop grand pour pouvoir s'attaquer à des scénarios de plusieurs menaces.

Limiter le nombre d'actions disponibles pourrait réduire l'utilisation de la mémoire, car ceci réduirait la taille de la table de Q-valeurs. Ce nombre d'actions peut être limité par un modèle d'engageabilité [Gagné(2007)]. En réduisant le nombre d'actions disponibles à chaque état avec un tel modèle d'engageabilité, le facteur de branchement de l'exploration sera grandement réduit également, ce qui mènera sans doute à une amélioration du temps de convergence.

Afin d'améliorer le concept, il serait possible d'utiliser des algorithmes pouvant créer une abstraction de l'espace d'états[L. Li and Littman(2006)] sensiblement équivalente à l'espace d'états original.

En effet, lorsqu'il devient impossible de tenir compte de tous les états possibles dans la perception de l'agent, on peut recourrir à des méthodes qui permettent de réduire le nombre d'états à considérer afin d'obtenir une perception sélective. Parmi ces méthodes, on retrouve celles construisant une abstraction sur l'espace d'état à l'aide d'un arbre de classification poursuivant le concept du *Utility Tree* (U-Tree) proposé par McCallum [McCallum(1996)].

Le concept du U-Tree repose sur le raffinement de l'abstraction par l'utilisation d'un critère de séparation dans un test statistique permettant de créer des distinctions utiles parmi les états abstraits. Les transitions dans l'environnement sont utilisées pour former l'échantillon du test statistique. Un critère de séparation sera alors appliqué afin de déterminer s'il est utile de diviser un état en deux et ainsi créer une distinction par rapport à l'utilité de chacun de ces nouveaux états [S. Paquet and Chaib-draa(2004)], [Quinlan(1993)], [Kolmogorov(1933)], [Smirnov(1939)].

Plusieurs investigations pourraient être réalisées afin de mesurer l'efficacité de méthodes similaires au U-Tree dans le problème de WTA. En effet, en modifiant le schéma d'abstraction et en choisissant des critères de séparation différents, il est possible d'adapter le concept pour divers environnements. Par exemple, la première variante est sans doute le *Continuous U-Tree* [Uther and Veloso(1998)]. Mise au point par William Uther et Manuella Veloso, cette variante du U-Tree permet l'utilisation du concept dans des environnements avec espace d'états continu. Plus encore, Uther propose dans sa thèse [Uther(2002)] de faire abstraction à la fois des etats, mais également des actions. Il développe ainsi son algorithme *Trajectory Tree* et en fourni les preuves de convergence.

5.7 Bilan

Une approche basée sur l'apprentissage par renforcement a le potentiel de résoudre efficacement des problèmes de contrôle, mais nécessite des efforts d'abstraction pour permettre une perception sélective ou une généralisation de l'espace de caractéristiques. Avec un peu d'entraînement, un agent peut trouver des politiques de gestion de ressources intéressantes dans un environnement temps réel.

Chapitre 6

Conclusion

Pour terminer ce mémoire, les éléments importants discutés tout au long des chapitres précédents seront passés en revue. Il est souvent intéressant de revoir globalement le cheminement des idées d'un ouvrage afin de garder en tête une vue d'ensemble unifiée de tout ce qui a été présenté.

Une fois le sujet de l'intelligence artificielle introduit par une définition et un historique, l'apprentissage par renforcement a été situé au sein de ce domaine de recherche. Les concepts de base devaient être expliqués, il a donc été expliqué qu'un agent est une entité autonome qui interagit avec son environnement, que les environnements ont des propriétés qui les caractérisent et que les agents peuvent être confrontés à des problèmes complexes. Une introduction à la théorie de la complexité a permis d'identifier que certains problèmes sont trop complexes pour être résolus de façon raisonnable par des ordinateurs tels qu'on les connaît.

Après avoir brièvement expliqué les notions de *Command and Control* dans le contexte d'un combat maritime, un des problèmes complexes avec lesquels un C2 doit composer dans son processus de décision a été identifié. Le WTA est un problème d'allocation de ressources étudié depuis longtemps, lequel a été étudié pour le compte du projet NEREUS. Une revue de littérature sur le WTA a été présentée, notamment sur le formalisme de ce problème ainsi que les méthodes utilisées jusqu'à ce jour pour le résoudre.

Une variante orientée-événement du WTA a été mise sur pied, ce qui permet d'ajouter une certaine flexibilité au problème, soit la possibilité d'appliquer des méthodes de résolution en temps réel. Cette version du WTA représente mieux la réalité du combat maritime. En effet, le déroulement d'une confrontation est généralement difficile à

prévoir et les nouvelles informations arrivent sur une base évènementielle plutôt que par intervalles fixes.

Une approche d'apprentissage par renforcement a été proposée pour trouver une politique capable de résoudre le problème de WTA. Les processus de décision markoviens permettent de modéliser le problème dans un cadre ou plusieurs méthodes de résolutions ont été développées. Un tour d'horizon des principales méthodes d'apprentissage par renforcement utilisant les MDP a été fait et les algorithmes qui leur sont propres ont été présentés. Dans la plupart des environnements réalistes, le modèle de transition est inconnu de l'agent et il lui faut l'apprendre ou en faire abstraction. Les deux orientations principales sont d'apprendre le modèle et de calculer la politique à partir de l'approximation obtenue ou d'apprendre les valeurs des actions en faisant des différences temporelles.

Des environnements simples de test ont permis d'affirmer que la méthode de Q-Learning est préférable aux autres méthodes dans une situation où la mémoire est limitée. Cependant, le temps d'apprentissage est plus long, car la convergence est moins rapide qu'avec des méthodes utilisant une boucle de planification pour propager l'expérience de l'agent sur les états non courrants.

Le framework utilisé pour implémenter l'agent est une bonne base pour permettre d'interchanger facilement les algorithmes d'apprentissage de l'agent. De plus, les agents deviennent facilement adaptables à de nouveaux environnements en modifiant seulement leurs signaux ou leurs actions.

Le simulateur SADM a été choisi comme environnement de test pour le projet NEREUS. Les métriques souhaitables pour évaluer un algorithme d'apprentissage dans cet environnement ont été identifiées et des scénarios d'exécution ont été définis, dans lesquels un agent-reflexe a été comparé à un agent Q-Learning. La liste des actions dont dispose une frégate canadienne de type HALIFAX a été détaillée ainsi que les contraintes sur leur utilisation. Finalement, une discrétisation des caractéristiques continues de l'espace d'état a été proposée ainsi qu'une fonction de récompense adaptée au problème.

L'agent utilisant la méthode Q-Learning a démontré une évolution des performances lors de l'apprentissage. De plus, il a su être compétitif contre l'agent-reflèxe en présentant un taux de survie semblable à ce dernier. Les résultats montrent que des méthodes d'apprentissage peuvent être efficace dans la résolution de problèmes tels que le WTA.

La principale complication qui survient lors de l'usage des MDP est le *Curse of Dimensionnality*. La mémoire devient vite saturée lorsque l'exploration découvre de

plus en plus de nouveaux états. Pour l'environnement et l'instance du problème de WTA, on a effectué une aggrégation des états dans le but de réduire l'espace mémoire utilisé. Bien que cette technique réduise grandement le nombre d'états par rapport à l'espace d'état initial, la grosseur de l'espace d'états demeure quand même exponentielle au nombre de menaces. Dans l'optique de réduction de l'utilisation de la mémoire, des algorithmes d'abstraction de l'espace d'états sont suggérés pour réduire davantage les effets de la malédiction de la dimentionnalité. Des méthodes de perception selective ont été identifiées et jugées intéressantes en tant que perspectives futures pour des approches d'apprentissage par renforcement.

Bibliographie

[Bellman(1957)] R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, New Jersey, 1957.

[Bertsekas and Tsitsiklis(2000)] D. P. Bertsekas and J. N. Tsitsiklis. Gradient convergence in gradient methods. *SIAM Journal on Optimization*, 10 :627–642, 2000.

[Cook(1971)] S. Cook. The complexity of theorem proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[G. G. DenBroeder and Emerling(1958)] R. E. Ellison G. G. DenBroeder and L. Emerling. On optimum target assignments. *Operation Research*, 7 :322–326, 1958.

[Gagné(2007)] Olivier Gagné. Ordonnancement de ressources en temps réel avec contraintes dynamiques dans un environnement non déterministe. Master's thesis, Univeristé Laval, 2007.

[Howard(1960)] R. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[J. M. Rosenberger and Brungardt(2005)] H. S. Hwang J. M. Rosenberger and E. G. Brungardt. The generalized weapon target assignment problem. In *10th International Command and Control Research and Technology Symposium, The Future of C2*, McLean, VA, 2005.

[Karp(1972)] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, Plenum, 1972.

[Kolmogorov(1933)] A. N. Kolmogorov. On the empirical determination of a distribution function. *Giornale dell'Istituto Italiano degli Attuari*, 4 :83–91, 1933.

[L. Li and Littman(2006)] T. J. Walsh L. Li and M. L. Littman. Towards a unified theory of state abstraction for MDPs. In *9th International Symposium on Artificial Intelligence and Mathematics*, 2006.

[Lloyd and Witsenhausen(1986)] S. P. Lloyd and H. S. Witsenhausen. Weapons allocation is NP-complete. In *Proceedings of the 1986 Summer Computer Simulation Conference*, Reno, Nevada, 1986.

[Manne(1958)] A. S. Manne. A target-assignment problem. *Operations Research*, 6 : 346–351, 1958.

[Matlin(1970)] S. M. Matlin. A review of the literature on the missile-allocation problem. *Operation Research*, 18 :334–373, 1970.

[McCallum(1996)] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State.* PhD thesis, University of Rochester, Rochester, New-York, 1996.

[Metler and Preston(1990)] W. A. Metler and F. L. Preston. A suite of weapon assignment algorithms for a SDI mid-course battle manager. Nrl memorandum report 671, Naval Research Laboratory, Washington, DC, 1990.

[Moore and Atkeson(1993)] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping : Reinforcement learning with less data and less time. *Machine Learning*, 13 :103–130, 1993.

[Murphey(1999a)] R. A. Murphey. Target-based weapon target assignment problems. In P. M. Pardalos and L. S. Pitsoulis, editors, *Nonlinear Assignment Problems : Algorithms and Applications*, volume 7, pages 39–53, Alexandra, Virginia, 1999a. Kluwer Academic Publishers.

[Murphey(1999b)] R. A. Murphey. An approximate algorithm for a weapon target assignment stochastic program. In *Approximation and Comlexity in Numerical Optimization : Continuous and Discrete Problems.* Kluwer Academic Publishers, 1999b.

[P. A. Hosein and Athans(1988)] J. Walton P. A. Hosein and M. Athans. Dynamic weapon-target assignment problems with vulnerable C2 nodes. In *Proceedings of the 1988 Command and Control symposium*, 1988.

[Peng and Williams(1992)] J. Peng and R. J. Williams. Efficient learning and planning within the dyna framework. *In Proceedings of the Second International Conference on Simulation of Adaptive Behavior : From Animals to Animats*, pages 281–290, 1992.

[Peng and Williams(1993)] J. Peng and R. J. Williams. Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 1(4) :437–454, 1993.

[Plamondon(2003)] P. Plamondon. A frigate survival approach based on real-time multiagent planning. Master's thesis, Computer Science & Software Engineering Department, Laval University, February 2003.

[Quinlan(1993)] J. R. Quinlan. Combining instance-based and model-based learning. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 236–243, Amherst, Massachusetts, 1993. Morgan Kaufmann.

[R. K. Ahuja and Jha(2003)] A. Kumar R. K. Ahuja and K. Jha. Exact and heuristic methods for the weapon target assignment problem. *Social Science Research Network Electronic Library*, 2003.

[Russell and Norvig(2003)] S. J. Russell and P. Norvig. *Artificial Intelligence : A Modern Approach.* Prentice-Hall, Englewood Cliffs, 2003.

[S. Chouinard and Chaib-draa(2007)] P. Cinq-Mars O. Gagné P. Plamondon S. Chouinard, C. Besse and B. Chaib-draa. Resource management algorithms implementation and evaluation. Technical task report, DAMAS Laboratory, Laval University, QC, 2007.

[S. Paquet and Chaib-draa(2004)] N. Bernier S. Paquet and B. Chaib-draa. From global selective perception to local selective perception. *Third International Joint Conference on Autonomous Agents and Multiagent Systems*, 3 :1352–1353, 2004.

[Smirnov(1939)] N. V. Smirnov. On the estimation of the discrepency between empirical curves of distribution for two independent samples. *Bulletin Math. Univ. Moscou*, 2 :3–16, 1939.

[Sutton and Barto(1998)] R. S. Sutton and A. G. Barto. *Reinforcement Learning. An Introduction.* MIT Press, Cambridge, MA, 1998.

[Toet and de Waard(1995)] Alexander Toet and Huub de Waard. The weapon-target assignment problem, 1995.

[Tsitsiklis(1994)] J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine learning*, 16 :185–202, 1994.

[Uther(2002)] W. T. B. Uther. *Tree based hierarchical reinforcement learning.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2002.

[Uther and Veloso(1998)] W. T. B. Uther and M. M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *AAAI-Press/MIT-Press*, pages 769–774, Menlo Park, CA, 1998.

[Wacholder(1989)] E. Wacholder. A neural network-based optimization algorithm for the static weapon-target assignment problem. *ORSA Journal on Computing*, 4 : 232–246, 1989.

[Watkins(1989)] C. J. Watkins. *Learning from Delayed Rewards.* PhD thesis, King's College, Cambridge, UK, 1989.

[Watkins and Dayan(1992)] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8 :279–292, 1992.

Annexe A

Utilisation du Framework

```
// Exemple d'utilisation du framework avec le domaine du feu
// de signalisation.

#include "stdafx.h"
#include <windows.h>
#include <time.h>

#include "../basicmdp/BasicMDP.h"
#include "../basicmdp/GeneralMDP.h"
#include "../basicmdp/Moore.h"
#include "../basicmdp/Watkins.h"

#include <iostream>
using namespace std;

int g_iColor = 0; //0=vert 1=jaune 2=rouge
double g_dDistance = 0;
BasicAction* g_pLastAction = 0;

class Move : public BasicAction
{
double m_dMove;

public:
Move(double d) : m_dMove(d) {};

operator double() { return m_dMove; };

void execute() { g_dDistance += m_dMove; g_pLastAction = this; };
};

class CarPerceiver : public BasicPerceiver
{
public:
CarPerceiver(BasicAbstractModel* m, BasicObservation* p) : BasicPerceiver(m, p)
{
m_pModel = m;
}
void perceive(BasicAction* a, double reward)
{
```

```
BasicSignal<double,0>* distance;
BasicSignal<int,0>* color;

distance = (BasicSignal<double,0>*)(m_pInputObservation->getSignal(0));
color = (BasicSignal<int,0>*)(m_pInputObservation->getSignal(1));

int state = int(distance->getSignalValue()*100)+color->getSignalValue();

m_pModel->update(a, state, reward);
}
};

int _tmain()
{
srand ( time(NULL) );

BasicModel model;
model.addAction(new Move(0.6));
model.addAction(new Move(0.2));
model.addAction(new Move(0.0));
QLearning algo(&model, 0.95, 0.05);

EpsilonGreedy explore(&model, 0.5);

BasicObservation vision;
BasicSignal<double,0>& distance = vision.addSignal<double>(0);
BasicSignal<int,0>& color = vision.addSignal<int>(0);
CarPerceiver perceiver(&model, &vision);

int nbEpisode = 1000000;

int count = 1;
double time = 0;
double reward = 0;

for (int i=nbEpisode; i>=0; i--)
{
//initialize to zero
g_dDistance = 0;
g_iColor = 0;
g_pLastAction = 0;

//loop until end of episode
while (count++)
{
//update time
time++;
if (time > 19)
time = 0;

//change signalisation
if (time <= 9)
g_iColor = 0;
else if (time <= 11)
g_iColor = 1;
else
g_iColor = 2;

//calculate reward
if (g_dDistance >= 10.0)
{
if (g_iColor == 2)
```

```
reward = -100;
else
reward = 500;
}
else
{
reward = -1;
}

//update perception
color = g_iColor;
distance = g_dDistance;

//cout << "D = " << g_dDistance << " \t C = " << g_iColor << endl;

perceiver.perceive(g_pLastAction, reward);
algo.step();

//check if end of episode
if (g_dDistance >= 10.0)
break;

//act
g_pLastAction = (Move*) explore.act();
}
}
return 0;
}
```