UNIVERSITÉ DU QUÉBEC À MONTRÉAL

APPROCHES CENTRÉES SUR LA PLANIFICATION POUR RÉSOUDRE DES PROBLÈMES D'APPRENTISSAGE PAR RENFORCEMENT À RÉCOMPENSES ÉPARSES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SIMON OUELLETTE

SEPTEMBRE 2024

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui ont contribué à mon projet de recherche.

Je voudrais dans un premier temps remercier mes directeurs de recherche, M. Beaudry, professeur et directeur du département d'informatique à l'UQAM, et M. Bouguessa, également professeur du département d'informatique à L'UQAM. Leur patience, disponibilité, et judicieux conseils ont contribué au bon développement de mon projet et de mon mémoire.

Je désire aussi remercier les professeurs de l'UQAM, qui m'ont fourni les outils nécessaires à la réussite de mes études universitaires.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

ACRONYMES

IA Intelligence artificielle.

MCTS *Monte-Carlo Tree Search*.

IDA* *Iterative Deepening A**.

RNN *Recurrent Neural Networks*.

DQfD *Deep Q-Learning from Demonstrations*.

DQN *Deep Q-Network*.

DDQN *Double Deep Q-Network*.

SQIL *Soft Q Imitation Learning*.

SARSA *State-action-reward-state-action*.

REINFORCE *REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility*.

WMG *Working Memory Graphs*.

NOTATION

Nombres

a : un scalaire.

A : une matrice.

Fonctions

$P(X|a)$: la distribution de probabilité de la variable aléatoire X, conditionnelle à a.

v_π : la fonction v associée à π (ex.: valeur v selon la politique π).

$F : a \rightarrow b$: on définit une fonction F qui prend en entrée a et donne en sortie b.

Opérations

$\mathbb{E}[X]$: valeur attendue de X.

$\nabla_\theta \pi$: gradient de π sous la politique paramétrisée par θ.

$max_a F(x)$: on trouve la valeur de a qui maximise F(x).

$a\tilde{X}$ on échantillonne a depuis la variable aléatoire X.

RÉSUMÉ

Les avancées en apprentissage par renforcement profond ont mené à plusieurs succès, notamment avec les échecs, le jeu de Go et le jeu de stratégie Starcraft II. Ces problèmes sont constitués d'environnements virtuels dans lesquels il est possible de générer une quantité illimitée d'interactions en relativement peu de temps.

Or, pour donner des résultats intéressants, l'apprentissage par renforcement profond requiert un nombre particulièrement élevé d'interactions avec l'environnement. Ceci n'est pas un problème pour les environnements virtuels ou simulés, comme pour les exemples de succès décrits précédemment. Toutefois, pour les environnements réels comme en robotique, où le nombre d'interactions est plus limité, il est difficile de bien appliquer l'apprentissage par renforcement.

Lorsque le problème contient, en plus, des récompenses éparses (*sparse rewards*), cette complexité échantillonnale devient souvent inacceptable. En effet, lorsque les récompenses sont rares, il est difficile pour une exploration aléatoire de les atteindre. L'apprentissage par démonstrations atténue ce problème, en présentant à l'algorithme un bon nombre d'exemples complets de solutions. La vaste majorité des approches d'apprentissage par démonstrations sont basées sur un paradigme **sans modèle**, qui présentent souvent des difficultés significatives d'apprentissage lorsque la solution du problème requiert de la planification (actions devant être soigneusement séquencées).

Dans ce mémoire, nous proposons deux nouvelles approches basées sur le paradigme **avec modèle**, afin d'atténuer le problème des récompenses éparses sans sacrifier la capacité de planifier. Les performances de ces deux nouvelles approches sont évaluées sur des problèmes nécessitant de la planification et pour lesquels les récompenses sont éparses. Plus spécifiquement, ces algorithmes sont évalués dans les environnements virtuels Minigrid et Sokoban. Nous démontrons que nos approches performent significativement mieux que les approches existantes.

Mots-clé : apprentissage par renforcement, apprentissage par démonstrations, récompenses éparses, planification.

CHAPITRE 1
INTRODUCTION

1.1 Mise en contexte

L'apprentissage par renforcement est une branche de l'intelligence artificielle qui se concentre sur la manière dont les agents peuvent apprendre à prendre de meilleures décisions en interagissant avec un environnement. Les racines de l'apprentissage par renforcement peuvent être retracées jusqu'aux travaux de Minsky en 1954 et au modèle de programmation dynamique de Bellman en 1957, qui ont jeté les bases théoriques pour la résolution de problèmes de décision séquentielle.

La principale motivation derrière le développement de l'apprentissage par renforcement est de créer des systèmes capables d'améliorer leur performance en apprenant de leurs propres expériences, un peu comme les êtres humains et les animaux apprennent de leurs interactions avec l'environnement. Cela permet aux agents d'adapter leurs stratégies pour maximiser la récompense cumulée au fil du temps, même dans des environnements complexes et changeants où des fonctions de transition explicites seraient difficiles à spécifier. Avec l'apprentissage par renforcement, il suffit de fournir à l'agent un signal de récompense représentant l'objectif à atteindre, lui permettant d'apprendre par lui-même comment résoudre le problème.

Au fil des ans, l'apprentissage par renforcement a réalisé plusieurs accomplissements notables, démontrant son potentiel à résoudre une gamme de problèmes complexes. Peut-être l'exemple le plus célèbre est *AlphaGo* (Silver *et al.*, 2017; Schrittwieser *et al.*, 2020), qui en 2016 a battu le champion du monde de Go, un jeu connu pour sa complexité combinatoire et sa profondeur stratégique. Cet événement marquant a montré que l'apprentissage par renforcement pouvait maîtriser des jeux considérés auparavant comme trop complexes pour l'intelligence artificielle. Par la suite, il y a eu *AlphaStar* (Vinyals *et al.*, 2019), qui a battu en 2018 un champion au jeu vidéo *StarCraft II*, et l'algorithme *Agent57* (Badia *et al.*, 2020) qui est capable d'atteindre une performance surhumaine dans 57 jeux *Atari* différents.

Dans ces jeux, la machine peut apprendre d'un très grand nombre d'expériences en simulant un nombre illimité de parties, à très faible coût et sans risque. Toutefois, dans le domaine de la robotique, les enjeux sont différents. D'une part, l'interaction d'un robot avec son environnement se produit en temps réel, plutôt que dans un environnement simulé qui peut être accéléré. Autrement dit, cette interaction est très lente, en

comparaison avec les environnements simulés, sans compter toute la préparation qui peut être associée à déplacer un robot, le calibrer, configurer l'environnement pour un certain type de problème, etc. De plus, il y a parfois des enjeux de sécurité associés à l'opération d'un robot : des risques de dommages matériaux ou même de blessures. Pour ces raisons associées à un grand nombre d'interactions d'essai et erreur, l'apprentissage par renforcement profond est souvent considéré comme difficile à utiliser en robotique (Deisenroth *et al.*, 2013; Pignat et Calinon, 2019; Deisenroth et Rasmussen, 2011).

Il existe une catégorie de problèmes qui exacerbent le défi associé à la complexité échantillonnale de l'apprentissage par renforcement : les problèmes à **récompenses éparses (ou rares)**. Dans ce type de problème, l'agent doit exécuter une séquence bien spécifique d'actions afin d'obtenir un signal de renforcement (positif ou négatif). Par conséquent, l'exploration aléatoire a très peu de chances de tomber sur la séquence d'actions qui émet un signal de renforcement : il faut donc compenser en augmentant significativement le nombre d'interactions. Par exemple, si un robot assistant a comme tâche de faire un thé, mais que ses actions de base sont des déplacements, des mouvements de bras et des mouvements de main, la séquence pour faire un thé avec succès (et donc obtenir la récompense de renforcement) est très précise et complexe. Si on le voit comme un problème d'apprentissage par renforcement, on peut donc parler de récompenses éparses.

Différentes catégories de méthodes ont été proposées pour mitiger le problème des récompenses éparses. Par exemple, l'idée du ***reward shaping*** (Grześ, 2017) consiste à ajouter un signal de renforcement intermédiaire agissant comme heuristique pour guider l'apprentissage. En fournissant un signal plus fréquent, celui-ci permet donc à l'agent d'apprendre de façon plus progressive. Ainsi, si on revient à l'exemple du robot qui doit faire un thé, on peut y incorporer certaines connaissances préalables des étapes de préparation d'un thé. On peut donc modifier la fonction de récompense pour retourner des valeurs positives lorsque le robot réussit à faire bouillir de l'eau, à sortir une tasse, à placer un sachet de thé dans la tasse et à placer l'eau bouillante dans la tasse contenant le sachet.

Une autre catégorie de méthodes consiste à proposer des **stratégies d'explorations différentes** (Ladosz *et al.*, 2022). Un exemple d'une telle stratégie consiste à utiliser l'apprentissage auto-supervisé pour déterminer si l'agent est capable de prédire les conséquences d'une certaine action dans un certain état. Après avoir exécuté l'action, si la transition ne correspond pas à la prédiction faite, l'agent se retrouve dans un état qu'on pourrait qualifier de surprise. Par conséquent, celui-ci mettra une certaine emphase sur ce type de

transitions, afin d'en apprendre plus sur celles-ci. C'est une approche basée sur la curiosité (*curiosity-driven* en anglais) (Ladosz *et al.*, 2022). Dans notre exemple, au lieu d'aléatoirement essayer de nouvelles actions (qui seront souvent redondantes), le robot va éventuellement tenter de prédire ce qui va arriver s'il exerce certaines actions de manipulation envers le sachet de thé. En réalisant qu'il prédit mal les résultats de ses actions, il va orienter ses tentatives futures dans la même direction jusqu'à ce qu'il maîtrise la manipulation et le déplacement du sachet de thé.

Un autre exemple de stratégie d'exploration alternative est **l'apprentissage par démonstrations**. C'est la stratégie qui nous intéressera principalement dans ce mémoire. Comme le nom l'indique, dans cette stratégie, des démonstrations faites par des experts humains (qui contrôlent typiquement le robot ou l'agent virtuel en mode manuel) sont fournies au modèle d'apprentissage. Il est possible de combiner ces démonstrations avec d'autres formes d'exploration, comme l'exploration aléatoire. Pour notre robot qui doit apprendre à faire du thé, il s'agirait donc de fournir plusieurs démonstrations (en contrôlant manuellement le robot) de préparation de thé, sous différentes conditions, avec différentes sortes de thé, différents types de bouilloire, etc. Ainsi, il serait capable éventuellement d'apprendre une forme générale de la procédure de préparation de thé.

1.2 Motivation

Le problème des récompenses éparses demeure un problème ouvert. Tel qu'il sera démontré dans ce mémoire, les techniques actuelles sont loin d'être suffisantes pour les problèmes plus difficiles. En particulier, bien que l'apprentissage par démonstrations et l'approche d'apprentissage par renforcement avec modèle apportent tous les deux des améliorations à ce niveau, il n'y a eu que peu, ou pas du tout, de recherche combinant les avantages des deux.

En effet, au meilleur de notre connaissance, dans la littérature actuelle les méthodes d'apprentissage par démonstration sont uniquement basées sur des approches sans modèle (*model-free reinforcement learning*). Bien que l'approche sans modèle soit efficace pour une variété de problèmes, **elle l'est beaucoup moins lorsqu'une forme de planification est requise** pour résoudre ceux-ci (Racanière *et al.*, 2017; Schrittwieser *et al.*, 2020; Moerland *et al.*, 2020). Ce phénomène a été confirmé par nos résultats empiriques.

L'apprentissage par renforcement avec modèle, en contraste à l'approche sans modèle, permet d'apprendre explicitement un modèle de l'environnement. Ainsi, en apprenant les règles fondamentales de celui-ci, il

peut par la suite faire appel à un algorithme de recherche ou un module neuronal dont le but est de planifier les étapes à suivre afin d'atteindre l'objectif. Bien que cette approche n'échappe pas à la difficulté liée à l'exploration aléatoire (la faible probabilité d'atteindre la récompense éparse), elle permet toutefois de diminuer grandement le nombre d'échantillons nécessaires pendant l'apprentissage. Cette efficacité échantillonnale provient du fait que l'apprentissage par renforcement avec modèle ne dépend pas d'un modèle de l'espace combinatoire de toutes les trajectoires possibles (contrairement à l'approche sans modèle) : il suffit simplement d'apprendre les règles fondamentales de l'environnement, ce qui représente généralement un espace d'apprentissage beaucoup plus restreint.

Combiner naïvement l'approche avec modèle et l'apprentissage par démonstrations ne fonctionne pas bien (tel que démontré par nos études d'ablation *NaivePlanner-MCTS* et *NaivePlanner-IDA**). En effet, les démonstrations expertes ne couvrent pas toutes les transitions possibles. Par exemple, dans *Minigrid*[1], le démonstrateur n'essaiera pas volontairement d'avancer l'agent dans un mur, ou dans une porte fermée. Ainsi, ces combinaisons particulières d'action et d'état ne seront pas présentes dans les données d'entraînement. Par conséquent, le modèle ne sera pas capable de faire des prédictions valides lorsque l'algorithme de planification interrogera ces possibilités : déraillant potentiellement la trajectoire planifiée.

1.3 Contributions

Dans ce mémoire, nous présentons deux stratégies différentes pour combiner l'approche avec modèle et l'apprentissage à partir de démonstrations. Nous comparons ces approches entre elles, ainsi qu'avec les approches existantes connexes. Notre travail a été présenté au *workshop Bridging the Gap Between AI Planning and Reinforcement Learning* de la conférence *International Conference on Automated Planning and Scheduling (ICAPS)* 2024 (Ouellette *et al.*, 2024).

La première stratégie consiste à obtenir une notion d'incertitude pour les prédictions faites. À partir de cette incertitude, une version modifiée de l'algorithme de planification utilisé fait l'élagage des sous-arbres trop incertains. De plus, l'algorithme optimise une nouvelle notion que nous appelons la **conviction**, soit le ratio de la valeur attendue sur l'incertitude cumulative de la trajectoire d'actions proposée.

1. Minigrid est un environnement de simulation pour l'apprentissage par renforcement, basé sur une grille comportant des obstacles comme des murs, des portes ou autres objets. Un agent s'y déplace afin d'atteindre une case finale. Voir section 4.1.1 pour plus de détails.

La seconde stratégie explorée consiste à combiner l'apprentissage à partir de démonstrations avec l'exploration aléatoire. Ainsi, les démonstrations permettront d'obtenir un bon nombre de signaux de renforcement, alors que les explorations aléatoires permettront de couvrir toutes les transitions possibles, même celles qu'un démonstrateur n'utiliserait pas normalement.

Nos deux stratégies ont été évaluées empiriquement avec les environnements virtuels *Minigrid* (Chevalier-Boisvert *et al.*, 2018) et Sokoban (Schrader, 2018). Les résultats obtenus démontrent que notre approche basée sur la conviction performe mieux que toutes les autres approches pertinentes, incluant notre seconde approche qui consiste à combiner l'apprentissage par démonstrations et l'exploration aléatoire. En particulier, nous dépassons l'état de l'art sur Sokoban (Loynd *et al.*, 2020) (parmi les techniques basées sur l'apprentissage automatique) en termes d'efficacité échantillonnale, et ce, d'un facteur d'approximativement 100 fois.

En résumé, nos contributions sont les suivantes :

1. Nous proposons deux nouvelles stratégies pour s'attaquer au problème de récompenses éparses en apprentissage par renforcement qui dépasse l'état de l'art en apprentissage automatique sur Sokoban, en termes d'efficacité échantillonnale.
2. Nous introduisons le concept de conviction, qui est utilisé dans une des deux stratégies proposées.
3. Nous concevons des variantes des algorithmes *Iterative Deepening A* (IDA*)* et *Monte-Carlo Tree Search* qui exploitent les notions de conviction et d'incertitude pour planifier à partir d'un modèle incomplet de l'environnement. [2]

1.4 Structure et organisation du mémoire

Dans le chapitre 2, nous faisons une revue de la littérature pertinente qui couvre l'usage de l'apprentissage par renforcement. En particulier, nous introduisons l'apprentissage par renforcement en général, puis les approches avec et sans modèle, incluant les algorithmes avec lesquels sont comparés nos approches proposées. Finalement, nous expliquons plus en détail le problème des récompenses éparses.

Dans le chapitre 3, nous continuons la revue de la littérature pertinente, mais cette fois-ci pour ce qui

2. Dans ce mémoire, nous utiliserons des termes anglophones en italique pour représenter des techniques ou algorithmes bien établis pour lesquels aucune traduction couramment utilisée n'existe.

concerne l'aspect planification. En particulier, nous présentons les algorithmes *Monte-Carlo Tree Search (MCTS)* et *Iterative Deepening A* (IDA*)*, car ils font partie de nos solutions proposées.

Dans le chapitre 4, nous expliquons nos deux solutions proposées : les concepts sous-jacents, leur architecture, et les détails des algorithmes de planification pour l'approche par conviction. Dans le chapitre 5, nous présentons le protocole de nos expériences ainsi que leurs résultats. Finalement, dans le chapitre 6, nous discutons de certaines conséquences et interprétations des résultats empiriques.

CHAPITRE 2
L'APPRENTISSAGE PAR RENFORCEMENT

Ce chapitre présente une revue de la littérature pour tout ce qui touche à l'aspect apprentissage par renforcement. L'apprentissage par renforcement est un élément central des types de problèmes et de solutions que nous considérons dans ce mémoire. Nous introduisons d'abord le concept de l'apprentissage par renforcement, établissons les bases théoriques, puis expliquons les concepts d'apprentissage par renforcement sans modèle et avec modèle.

Dans la section sur l'apprentissage par renforcement sans modèle, nous expliquons en détail les algorithmes *Deep Q-learning from Demonstrations* et *Working Memory Graphs*, puisqu'ils sont des algorithmes de comparaison utilisés dans nos expériences. Dans la section sur l'apprentissage par renforcement avec modèle, nous décrivons d'abord le concept au sens général, puis nous expliquons l'algorithme MuZero, puisqu'il sert d'inspiration principale pour nos solutions proposées. Finalement, nous présentons une section sur les récompenses éparses, puisqu'il s'agit du problème central de ce mémoire.

2.1 Mise en contexte

L'apprentissage par renforcement est une branche de l'apprentissage automatique dans laquelle on entraîne un agent à résoudre un problème par des interactions avec son environnement qui incluent un *feedback* sous forme de récompenses. Contrairement à l'apprentissage supervisé, où le modèle est entraîné à partir de données dont les sorties désirées sont spécifiées explicitement (par exemple, par annotation humaine), ou l'apprentissage non supervisé, qui identifie des motifs ou des structures cachées dans des données non annotées, l'apprentissage par renforcement se base sur l'interaction avec l'environnement pour obtenir des récompenses ou éviter des pénalités.

Dans ce paradigme, l'agent est un modèle ou un algorithme qui prend des décisions. L'environnement est le cadre ou le contexte dans lequel l'agent opère. L'objectif de l'agent est d'apprendre à naviguer dans cet environnement de manière à maximiser une récompense cumulative. La récompense est une valeur émise par l'environnement suite à une action effectuée par l'agent dans un état donné.

La fonction de valeur estime la quantité totale de récompense qu'un agent peut espérer accumuler à partir

d'un état donné, en suivant une politique particulière. L'équilibre entre exploration (essayer de nouvelles actions pour découvrir leur efficacité) et exploitation (utiliser des actions connues pour obtenir de bonnes récompenses) est crucial pour s'approcher d'une performance optimale.

Le gabarit théorique de l'apprentissage par renforcement se fonde sur les processus décisionnels de Markov (Sutton et Barto, 2018). Un processus décisionnel de Markov se définit par un tuple (S, A, R, P, γ), dans lequel :

- S est un ensemble d'états possibles ;
- A est un ensemble d'actions possibles ;
- $R(s, a)$ est une fonction de récompense, associant une paire (état, action) à une valeur numérique ;
- $P(s\prime|s, a)$ est une fonction de transition indiquant la distribution des probabilités d'atteindre $s\prime$ à partir de l'état s en appliquant l'action a ;
- γ est un facteur d'escompte (*discount factor*), $0 \leq \gamma \leq 1$.

À chaque état observé $s \in S$, l'agent applique une action $a \in A$ qui le place dans un nouvel état $s\prime \in S$ dont la distribution de probabilités est basée sur $P(s\prime|s, a)$. De plus, l'agent reçoit une récompense de l'environnement selon $R(s, a)$. Le facteur d'escompte γ permet de pondérer l'importance relative des récompenses futures par rapport à la récompense immédiate. En quelque sorte, il permet de déterminer le niveau de patience ou d'impulsivité de l'agent. Lorsque γ est proche de 1, le poids des récompenses futures est plus élevé. Cela incite l'agent à prendre des décisions qui peuvent ne pas avoir de grands avantages immédiats, mais qui sont bénéfiques à long terme. À l'inverse, si γ est proche de 0, l'agent privilégie les récompenses immédiates, négligeant les conséquences futures.

Une politique π est une fonction qui prend en entrée l'état de l'environnement $s \in S$ et qui donne en sortie l'action $a \in A$ que l'agent doit exécuter. Le but de cette politique est de maximiser la somme escomptée (via le facteur d'escompte γ) des récompenses que l'agent reçoit au cours du temps. Dans l'apprentissage par renforcement, l'agent apprend progressivement à améliorer sa politique en interagissant avec l'environnement et en recevant des récompenses associées à ses actions. La politique peut être déterministe, où un état particulier mène toujours à la même action, ou stochastique, où un état donne une distribution de probabilités sur un ensemble d'actions.

Il existe deux grandes classes d'algorithmes d'apprentissage par renforcement : les algorithmes sans modèle, qui seront présentés à la section 2.2, et les algorithmes avec modèle, qui seront présentés à la section 2.3. Le

terme modèle, dans ce contexte, fait spécifiquement référence à la fonction de transition P. Autrement dit, les approches sans modèle ne tentent pas d'apprendre explicitement un modèle de la fonction de transition P, alors que les approches avec modèle tentent de le faire.

Les applications de l'apprentissage par renforcement incluent le domaine de la robotique (Kober *et al.*, 2013), les systèmes de recommandation (Afsar *et al.*, 2022) et les véhicules autonomes (Shalev-Shwartz *et al.*, 2016; Campbell *et al.*, 2010). Toutefois, ce domaine présente plusieurs défis, notamment la nécessité de grandes quantités de données d'interaction, la difficulté de trouver un équilibre entre exploration et exploitation, et la complexité de concevoir des récompenses qui guident efficacement l'apprentissage.

2.2 Apprentissage par renforcement sans modèle

L'apprentissage par renforcement sans modèle (ou *model-free reinforcement learning* en anglais) est une approche qui se concentre sur l'apprentissage de stratégies optimales pour prendre des décisions dans un environnement sans avoir une connaissance explicite de la dynamique de cet environnement. Autrement dit, on ne connaît pas la fonction de transition P du processus décisionnel de Markov. Il permet à un agent d'apprendre directement à partir de l'interaction avec l'environnement, sans utiliser de modèle explicite pour prédire les transitions entre les états. Dans ce contexte, l'agent explore son environnement en effectuant des actions provenant de l'ensemble A et en observant les réponses de l'environnement, puis ajuste sa politique de décision afin de maximiser la récompense obtenue au fil du temps.

Cette exploration peut être réalisée de différentes manières, telles que l'exploration aléatoire, l'exploration guidée par des politiques probabilistes ou même des méthodes d'apprentissage par curiosité pour découvrir de nouvelles informations utiles. Cela se réalise typiquement par l'utilisation d'algorithmes tels que les méthodes basées sur la valeur, comme le *Q-learning* (voir section 2.2.3) et SARSA (Rummery et Niranjan, 1994), ou les méthodes basées sur la politique (comme les méthodes de gradients de politique).

2.2.1 Méthodes basées sur la valeur

Les méthodes basées sur la valeur visent à estimer les valeurs d'état (V) ou d'action-état (Q) du problème. Ces méthodes utilisent les concepts clés de l'équation de Bellman (2.1) pour estimer les valeurs espérées des états ou des actions en se basant sur les récompenses observées et les transitions d'état. L'équation de Bellman décrit la relation entre la valeur d'un état ou d'une action et les récompenses futures espérées

dans un processus de décision séquentiel. Elle s'exprime généralement pour les fonctions de valeur.

Équation de Bellman pour la valeur d'état (V) :

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\
&= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\
&= \sum_a \pi(a \mid s) \sum_{s', r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right], \quad \text{pour tout } s \in \mathcal{S}
\end{aligned}
\tag{2.1}
$$

L'équation (2.1) (Sutton et Barto, 2018) énonce l'expression pour la valeur espérée lors du suivi de la politique π à partir de l'état s. Les variables a proviennent de l'ensemble A, s et s' de l'ensemble S, et r de la fonction $R(s, a)$. La première ligne définit la fonction de valeur comme la récompense totale espérée G_t étant donné que l'agent est dans l'état s au temps t, en suivant la politique π. G_t est la récompense totale accumulée à partir du temps t. L'horizon peut être fini ou infini. Dans ce dernier cas, un facteur d'escompte inférieur à 1 nous assurera une somme finie.

Dans la deuxième ligne, le retour G_t est décomposé en la récompense immédiate R_{t+1} plus le retour escompté γG_{t+1} auquel on peut s'attendre à partir du temps $t + 1$. La troisième ligne développe l'espérance en une somme sur toutes les actions a, les prochains états s', et les récompenses r. Le terme $\pi(a|s)$ est la probabilité de prendre l'action a dans l'état s sous la politique π, et $p(s', r|s, a)$ est la probabilité de transiter vers l'état s' et de recevoir la récompense r après avoir pris l'action a dans l'état s. En contraste avec la modélisation précédente du processus décisionnel de Markov selon laquelle on avait deux fonctions distinctes $P(s'|s, a)$ et $R(s, a)$, ici on combine les deux en la distribution de probabilités $p(s', r|s, a)$ afin de simplifier la formule.

Enfin, à la dernière ligne, l'espérance $\mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s']$ est remplacée par la fonction de valeur $v_\pi(s')$, démontrant la nature récursive de l'équation de Bellman. De plus, on continue la simplification effectuée à la troisième ligne, en combinant la somme pour s' et r. Cette équation est valable pour tous les états s dans l'espace d'états S.

Dans le contexte de l'apprentissage par renforcement sans modèle, ces équations de Bellman sont utilisées pour mettre à jour itérativement les estimations des valeurs d'état (V) ou d'action (Q) à mesure que l'agent

interagit avec l'environnement, collecte des données d'expérience et affine ses stratégies de décision. Les algorithmes tels que le *Q-learning* ou les méthodes de politique de gradient exploitent les équations de Bellman pour estimer et mettre à jour les valeurs d'état ou d'action, permettant à l'agent d'apprendre des politiques efficaces sans avoir une connaissance explicite du modèle de l'environnement.

2.2.2 Méthodes basées sur la politique

Les méthodes basées sur la politique sont une classe d'algorithmes d'apprentissage par renforcement sans modèle qui visent à apprendre une politique directement. Contrairement aux méthodes basées sur les valeurs, ces approches mettent à jour directement les paramètres d'une politique, souvent représentée par un réseau de neurones. Les méthode du gradient de politique (*Policy Gradient Methods* en anglais) est un exemple d'algorithme dans cette classe. Elle utilise le calcul des gradients pour ajuster les paramètres d'une politique afin de maximiser les récompenses cumulées. L'objectif est d'apprendre les poids du réseau de neurones (ou tout autre représentation de la politique) pour qu'ils génèrent une meilleure politique.

L'approche de base consiste à maximiser la fonction objectif, souvent appelée la fonction de récompense $J(\Theta)$, où Θ correspond aux paramètres de la politique. $J(\Theta)$ correspond à la valeur espérée de la somme des récompenses pondérées par la probabilité des trajectoires générées par cette politique. La mise à jour des paramètres Θ de la politique s'effectue en utilisant la montée de gradient stochastique (*Stochastic Gradient Ascent* en anglais) ou les méthodes basées sur des estimateurs de gradient comme l'algorithme REINFORCE (un acronyme pour : "REward Increment = Nonnegative Factor × Offset Reinforcement × Characteristic Eligibility").

REINFORCE (Williams, 1992) est un algorithme d'apprentissage par renforcement basé sur les gradients de politique. Il utilise la méthode de la montée de gradient pour mettre à jour les paramètres de la politique. Pour une trajectoire donnée (une séquence d'états, d'actions et de récompenses), l'algorithme REINFORCE met à jour les poids de la politique en utilisant la formule :

$$\Delta\Theta = \alpha \cdot \gamma \cdot G_t \cdot \nabla_\Theta \log \pi_\Theta(a_t|s_t) \tag{2.2}$$

où :

— $\Delta\Theta$ est le changement des paramètres de la politique ;

— α est le taux d'apprentissage ;
— γ est le facteur d'escompte ;
— G_t est le rendement total de la trajectoire après l'étape t ;
— $\nabla_\Theta \log \pi_\Theta(a_t|s_t)$ est le gradient du logarithme de la probabilité de choisir l'action a_t dans l'état s_t sous la politique paramétrisée par Θ.

En utilisant cette mise à jour, REINFORCE ajuste progressivement les paramètres de la politique pour augmenter la probabilité des actions qui ont conduit à de bonnes récompenses et diminuer la probabilité des actions menant à des récompenses moindres.

2.2.3 *Q-learning*

Le *Q-learning* est l'un des algorithmes les plus connus dans l'apprentissage par renforcement sans modèle basé sur la valeur. Cet algorithme est utilisé pour estimer la fonction d'action optimale (Q) qui évalue chaque paire état-action en fonction des récompenses observées et des états suivants. Autrement dit, l'agent apprend cette fonction Q qui estime la valeur de prendre une action particulière dans un état donné.

L'algorithme du *Q-learning* utilise une mise à jour itérative des estimations de la valeur Q selon la formule de mise à jour basée sur l'équation de Bellman pour la valeur d'état-action présentée à l'équation (2.3). En utilisant cette mise à jour, le *Q-learning* permet à l'agent d'apprendre progressivement les valeurs optimales des actions dans chaque état sans nécessiter de modèle explicite de l'environnement.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [r + \gamma \cdot max_{a'} Q(s',a') - Q(s,a)] \tag{2.3}$$

où :

— $\leftarrow$ représente l'opération de mise à jour ;
— $Q(s,a)$ est la valeur estimée de l'action a dans l'état s ;
— $Q(s',a')$ est la valeur estimée de l'action suivante a' dans l'état suivant s' ;
— α est le taux d'apprentissage, $0 < \alpha < 1$;
— r est la récompense immédiate observée après avoir pris l'action a depuis l'état s ;
— γ est le facteur d'escompte ;
— s' est l'état suivant résultant de l'action a.

Cette fonction Q est traditionnellement représentée dans un tableau, mais le *Deep Q-learning* (DQN) utilise un réseau de neurones profonds pour approximer cette fonction, lui permettant de gérer des environnements avec des espaces d'états de haute dimension, comme les jeux vidéo.

Le taux d'apprentissage α permet de pondérer la vitesse à laquelle on ajuste les valeurs de Q. Plus α s'approche de 1, plus la valeur de $Q(s, a)$ sera ajustée à une valeur proche du dernier exemple observé. À l'inverse, plus la valeur α s'approche de 0, plus $Q(s, a)$ sera une moyenne couvrant un grand nombre d'observations. Une valeur trop élevée peut donc apporter une instabilité d'apprentissage empêchant la convergence vers une valeur stable. Une valeur trop faible peut causer un lenteur excessive au niveau du processus de convergence.

2.2.4 *Double Deep Q-learning*

Le *Double Deep Q-learning* (DDQN) est une amélioration de l'algorithme original de *Deep Q-learning* (DQN), qui est lui-même une combinaison du *Q-learning* et des réseaux de neurones profonds. Les réseaux de neurones dans le *Deep Q-learning* permettent de généraliser à partir de données pour traiter des états jamais vus auparavant, ce qui est crucial dans des environnements aux états très nombreux ou infinis. Ils gèrent mieux les environnements de grande dimension et réduisent le besoin de vastes tableaux de valeurs Q, qui deviennent rapidement difficiles à gérer. De plus, les réseaux peuvent efficacement apprendre à partir d'entrées complexes comme des images, offrant une flexibilité accrue par rapport aux méthodes classiques. Le DDQN aborde un problème spécifique du DQN, connu sous le nom de surévaluation des valeurs-Q.

Problème de surévaluation dans le DQN : Le DQN a tendance à surévaluer les valeurs-Q à cause de l'opération max utilisée dans la mise à jour du *Q-learning*. Lors de l'estimation de la valeur de la meilleure action dans l'état suivant, il utilise le même réseau à la fois pour sélectionner la meilleure action et pour évaluer sa valeur. Cela peut conduire à une surévaluation systématique des valeurs-Q et à un apprentissage sous-optimal. Autrement dit, la valeur d'un état dépend en partie de la valeur attendue de la meilleure action (valeur-Q maximale) qui peut être faite à partir de cet état. Or, si l'estimation des valeurs-Q contient une forme de bruit, l'utilisation du maximum favorisera les cas où le bruit est positif et maximal. Statistiquement, donc, on aura tendance à surévaluer la valeur-Q d'un état s.

***Double Deep Q-learning* (DDQN)** : Le DDQN introduit une modification simple mais efficace pour combattre le problème de surévaluation. Tel que montré à la figure 2.1, deux réseaux de neurones sont utilisés. Le ré-

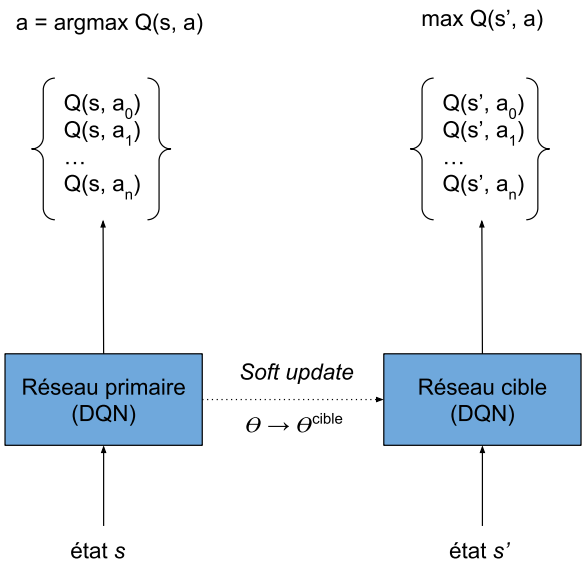

Figure 2.1 Architecture du *Double Deep Q-learning*.

seau principal (ou primaire), qui est utilisé pour sélectionner la meilleure action dans l'état suivant (sélection d'action) et le réseau cible, qui est une copie du réseau principal, mais dont les poids sont gelés pendant un certain nombre d'étapes. Ce dernier est utilisé pour évaluer la valeur de l'action choisie par le réseau principal (évaluation d'action). Ceci atténuera le problème de surévaluation, car nous utilisons en réalité deux échantillons distincts de valeur. D'abord, avec le réseau primaire, nous déterminons l'action optimale qui peut être prise à partir de l'état dont on essaie d'estimer la valeur. Ce choix peut inclure un biais dû au bruit présent dans les récompenses. Par contre, nous prenons ensuite l'estimation de valeur de cette action à partir du réseau cible, qui correspond à l'équivalent d'un deuxième échantillon statistiquement indépendant. Ici, le bruit contenu dans la valeur-Q pour cette action sélectionnée peut autant être positif, négatif, ou même absent.

Pour récapituler, étant donné qu'une valeur-Q est mise à jour en considérant la valeur-Q maximale qui peut être atteinte dans l'état suivant s' (soit $max_{a'}Q(s', a')$), et considérant un environnement dans lequel il y a un aspect aléatoire aux récompenses, il y a un biais statistique causé par l'usage du maximum. Si, toutefois, on applique le maximum sur un estimateur (le réseau primaire) pour sélectionner l'action, et on utilise un autre estimateur (le réseau cible) pour obtenir sa valeur-Q espérée, on affaiblit ce biais statistique. Puisque le réseau cible est mis à jour plus lentement, et avec délai, lorsqu'il y a une récompense aléatoirement très

élevée, elle pourra introduire un biais de sélection d'action, mais elle ne résultera pas en un biais d'estimation de la valeur-Q, puisque le réseau cible n'inclura pas cette récompense élevée. Ainsi, les mises à jour de la valeur-Q du réseau primaire seront atténuées par le réseau cible.

Le DDQN obtient souvent de meilleures performances que le DQN, en particulier dans les tâches où la surévaluation des valeurs-Q est un problème important. Ceci est souvent le cas lorsque la fonction de récompense R est stochastique et présente une variance relativement élevée dans les valeurs retournées. On peut imaginer, comme exemple, l'application du DQN à la négociation boursière automatisée. L'aspect hautement aléatoire des récompenses dans ce genre d'environnements peut causer une surévaluation de certaines combinaisons état-action. Toutefois, il faut noter que l'usage du DDQN apporte une complexité additionnelle au niveau de l'implémentation et ajoute un coût de calcul causé par la présence d'un deuxième réseau de neurones. Il peut donc être préférable d'utiliser le DQN au lieu du DDQN lorsque le problème est simple ou présente peu de variance dans les récompenses.

2.2.5 *Deep Q-learning from Demonstrations* (DQfD)

L'algorithme *Deep Q-learning from Demonstrations* (DQfD) est une technique qui combine le *Deep Q-learning* avec l'utilisation de démonstrations d'experts humains pour accélérer et améliorer l'apprentissage des agents. Voici les étapes clés de l'algorithme DQfD :

1. **Collecte des démonstrations** : Au lieu de commencer l'apprentissage à partir de zéro, des démonstrations d'experts humains sont collectées. Ces démonstrations représentent des séquences d'actions prises par des experts dans l'environnement.
2. **Pré-entraînement du réseau** : Les démonstrations d'experts sont utilisées pour pré-entraîner le réseau de neurones qui représente la fonction Q. Ce pré-entraînement initial vise à initier le réseau avec des connaissances préalables sur des actions potentiellement efficaces dans l'environnement.
3. **Apprentissage par renforcement profond** : Après le pré-entraînement initial, l'agent commence à interagir avec l'environnement en utilisant le *Deep Q-learning*. Il exploite l'algorithme *Q-learning* ou ses variantes, comme le DDQN, pour apprendre à prendre des décisions en fonction des récompenses reçues dans l'environnement.
4. **Tampon de relecture** : Pendant l'apprentissage, l'agent accumule ses expériences (séquences d'états, d'actions, de récompenses, etc.) dans un tampon de relecture (*replay buffer*). Ce tampon est utilisé

pour échantillonner aléatoirement des expériences passées lors de la mise à jour du réseau de neurones. Cela aide à stabiliser l'apprentissage en brisant la corrélation entre les expériences successives.

5. **Correction des erreurs par les démonstrations** : Une composante clé de DQfD est l'utilisation des démonstrations d'experts pour guider l'apprentissage. Lorsque l'agent prend des actions qui diffèrent des actions démontrées par les experts dans des situations similaires, une pénalité est ajoutée à la fonction de perte pour encourager l'agent à apprendre des actions plus proches de celles des experts. Cela permet de corriger les erreurs d'apprentissage potentielles et d'accélérer l'apprentissage vers des politiques plus performantes.

En combinant l'apprentissage par renforcement profond avec l'utilisation des démonstrations d'experts, DQfD vise à accélérer le processus d'apprentissage en utilisant les connaissances préalables des experts tout en permettant à l'agent de continuer à explorer et à améliorer ses politiques à travers l'interaction directe avec l'environnement. Cela aide à surmonter les problèmes de lenteur et de non-convergence parfois associés à l'apprentissage par renforcement seul dans des environnements complexes.

L'algorithme 1 présente le DQfD. L'entrée D^{replay} est un tampon de données initialisé avec les démonstrations. Il s'agit d'une liste de tuples (s, a, r, s') représentant l'état actuel $s \in S$, l'action $a \in A$ qui a été faite dans la démonstration à cette étape, la récompense r qui a été obtenue, et l'état suivant $s' \in S$ qui a été atteint. La politique $\pi^{\epsilon Q_\theta}$ est une politique avare epsilon (*epsilon-greedy* en anglais). Il s'agit d'une stratégie de sélection d'actions où, avec une probabilité principale (1 - epsilon), l'agent choisit l'action qui a la plus haute estimation de valeur, et avec une petite probabilité (epsilon), il choisit une action au hasard. Cette approche permet à l'agent de privilégier l'exploitation des connaissances actuelles tout en explorant occasionnellement de nouvelles actions pour améliorer son apprentissage.

De plus, ce pseudo-code fait référence une fonction de coût L_{DQfD}, qui combine deux composantes principales :

1. **Coût de l'apprentissage par renforcement** : Il s'agit de la composante traditionnelle utilisée dans le *Q-learning*. Cette composante de la fonction de coût est basée sur la différence entre les valeurs prédites par le réseau de neurones et les cibles de *Q-learning*.
2. **Coût de l'apprentissage à partir des démonstrations** : Cette composante de la fonction de coût vise

Algorithme 1 : *Deep Q-learning from Demonstrations*

Entrée : D^{replay} : tampon initialisé à partir des démonstrations

θ : paramètres initiaux du *Deep Q-network*

θ' : paramètres initiaux du *Deep Q-network* cible

τ : fréquence à laquelle mettre à jour le *Deep Q-network* cible

k : nombre d'itérations de pré-entraînement

T : nombre d'itérations d'entraînement en ligne

Résultat : Les paramètres θ optimaux

Début

`// Phase de pré-entraînement à partir de démonstrations`

pour $i \in \{1, 2, \dots k\}$ **faire**

Échantillonner un mini-lot de n transitions dans D^{replay}

Calculer la fonction de coût L_{DQfD} en utilisant le *Deep Q-Network* cible

Effectuer une itération de descente de gradients pour mettre à jour θ

si $i \bmod \tau = 0$ **alors**

$\theta' \leftarrow \theta$

fin

fin

`// Phase d'apprentissage par renforcement`

pour $t \in \{1, 2, \dots T\}$ **faire**

Échantillonner une action à partir de la politique, $a \sim \pi^{\epsilon Q_\theta}$

Exécuter l'action a et observer (s', r)

Ajouter (s, a, r, s') dans D^{replay} (on écrase la plus vieille transition si à capacité maximale)

Échantillonner un mini-lot de n transitions dans D^{replay}

Calculer la fonction de coût L_{DQfD} en utilisant le *Deep Q-Network* cible

Effectuer une itération de descente de gradients pour mettre à jour θ

si $t \bmod \tau = 0$ **alors**

$\theta' \leftarrow \theta$

fin

$s \leftarrow s'$

fin

Fin

à minimiser l'écart entre les actions prises par les experts humains dans des situations données et les actions prédites par le réseau de neurones. Cela consiste à encourager le réseau de neurones à reproduire les comportements des experts dans des situations similaires.

Ainsi, la fonction de coût totale dans DQfD est une combinaison pondérée de ces composantes (ainsi qu'un terme de régularisation $L2$) :

$$L_{DQfD} = L_{1-step} + \lambda_1 L_{10-step} + \lambda_2 L_E + \lambda_3 L_{L2} \tag{2.4}$$

où :

- $L_{1-step} + \lambda_1 L_{10-step}$: le coût d'apprentissage par renforcement. Il y a deux termes, car l'implémentation de DQfD originale utilise deux horizons de temps pour calculer ce coût : $t+1$, et $t+10$. C'est la notion du *n-step loss*, expliquée plus en détail ci-dessous. Voir aussi l'équation (2.7).
- $\lambda_2 L_E$: le terme d'apprentissage supervisé qui permet d'apprendre à partir de démonstrations. Voir l'équation (2.8), dans laquelle a_E est une action prise dans la démonstration, et $l(a_E, a)$ est une fonction qui vaut 0 quand $a = a_E$ et est positive dans les autres cas.
- $\lambda_3 L_{L2}$: le terme de régularisation, connu sous le nom $L2$ *loss* en anglais. Permet de réduire le risque de sur-apprentissage. Voir l'équation (2.9), dans laquelle w_j fait référence à la valeur du poids j du modèle neuronal.

La notion de *n-step loss* est une extension de l'algorithme *Q-learning* traditionnel qui vise à améliorer la convergence et la stabilité de l'apprentissage en considérant des séquences d'actions et de récompenses sur plusieurs pas temporels au lieu d'une seule transition à la fois. Dans le *Q-learning* classique, la mise à jour de la valeur d'action $Q(s, a)$ se fait en considérant une seule transition d'état à l'état suivant, basée sur la récompense immédiate et la meilleure estimation de la valeur d'action future (voir équation (2.3)).

Cependant, cette approche ne prend en compte qu'une seule transition à la fois et peut être limitée dans la manière dont elle utilise l'information temporelle disponible. L'idée du *n-step loss* est d'étendre cette mise à jour en considérant une séquence de transitions sur plusieurs pas temporels pour ajuster les valeurs d'action de manière plus globale. La mise à jour du *n-step loss* consiste à calculer la somme pondérée des

récompenses sur n pas temporels suivie d'une estimation de la valeur d'action à l'étape n (voir équation (2.5)).

$$G_t^{(n)} = \gamma^0 \cdot r_{t+1} + \gamma^1 \cdot r_{t+2} + ... + \gamma^{n-1} \cdot r_{t+n} + max_a Q(s_{t+n}, a) \tag{2.5}$$

Puis, la valeur d'action $Q(s_t, a_t)$ est mise à jour en utilisant cette estimation de récompense à n pas temporels (voir équation (2.6)). Cela permet à l'algorithme d'apprentissage de prendre en compte des séquences plus longues d'actions et de récompenses, offrant une vision plus étendue de l'environnement et potentiellement une meilleure estimation des valeurs d'action.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot [G_t^{(n)} - Q(s_t, a_t)] \tag{2.6}$$

Le choix de la valeur de n dans le *n-step loss* est un compromis entre la précision de l'estimation (plus n est grand, plus l'information temporelle est étendue) et la variance de l'estimation (plus n est grand, plus les estimations peuvent être bruitées). En pratique, différentes valeurs de n peuvent être testées pour trouver un équilibre approprié entre précision et stabilité dans l'apprentissage.

$$L_{1-step} + L_{10-step} = G_t^{(1)} + G_t^{(10)} \tag{2.7}$$

$$L_E = max_{a \in A}[Q(s, a) + l(a_E, a)] - Q(s, a_E) \tag{2.8}$$

$$L_{L2} = \sum_j^M w_j^2 \tag{2.9}$$

2.2.6 *Working Memory Graphs*

Les *Working Memory Graphs* (WMG) (Loynd *et al.*, 2020) sont une approche d'apprentissage par renforcement sans modèle, basée sur l'architecture du Transformeur (*Transformer* en anglais) (Vaswani *et al.*,

2017). Ce dernier est une architecture de réseaux de neurones qui permet de traiter des séquences via le mécanisme d'attention plutôt que la récurrence. Le mécanisme d'attention permet au modèle de focaliser différemment sur différentes parties d'une séquence d'entrée lors du traitement de l'information. Il fonctionne en calculant des scores d'attention qui déterminent l'importance relative de chaque élément de la séquence pour une tâche spécifique. Ces scores aident le modèle à agréger les informations de manière sélective, améliorant ainsi sa capacité à comprendre et générer des réponses pertinentes en fonction du contexte donné. L'avantage du mécanisme d'attention par rapport à la récurrence est, d'une part, qu'il permet d'apprendre des relations plus lointaines dans la séquence que la récurrence, et d'autre part, qu'il permet le traitement parallèle de l'exécution du réseau de neurones.

Le développement des WMG est motivé par le succès des Transformeurs par rapport aux *Gated Recurrent Neural Networks* (Chung *et al.*, 2014) dans le traitement de séquences de texte, et par le besoin d'améliorer les capacités des agents d'apprentissage par renforcement dans les processus décisionnels markoviens partiellement observables. Les WMG représentent l'état de l'art pour une catégorie d'approches connues sont le nom de planification implicite en apprentissage par renforcement. En effet, bien que ces approches ne comportent pas d'algorithme classique qui effectue explicitement de la planification, leur architecture est conçue pour permettre d'apprendre une forme implicite de planification.

Par exemple, un prédécesseur des WMG, le *Deep Repeated ConvLSTM* (Guez *et al.*, 2019), est un algorithme récurrent qui contient une boucle d'itérations sur un état latent. Par état latent, nous faisons référence à une représentation d'état qui est interne au modèle, apprise par celui-ci, au lieu d'être explicitement spécifiée dans les données. Autrement dit, cette architecture permet d'apprendre à prévoir plusieurs coups en avance, puisque chaque itération de la boucle peut modéliser une interaction dans son état latent. C'est pourquoi on peut parler de planification implicite malgré l'absence d'algorithme explicite de planification.

Les caractéristiques clés des WMG (figure 2.2) incluent :

Mécanisme d'auto-attention multi-têtes : Les WMG utilisent un mécanisme d'auto-attention multi-têtes pour traiter un ensemble dynamique de vecteurs représentant des états observés et récurrents. Ce mécanisme fait partie de l'architecture des Transformeurs, et permet un raisonnement complexe sur les observations passées et la planification future, remédiant aux limitations des modèles précédents qui dépendaient d'une chaîne linéaire d'états cachés de réseaux de neurones récurrents pour le flux d'informations.

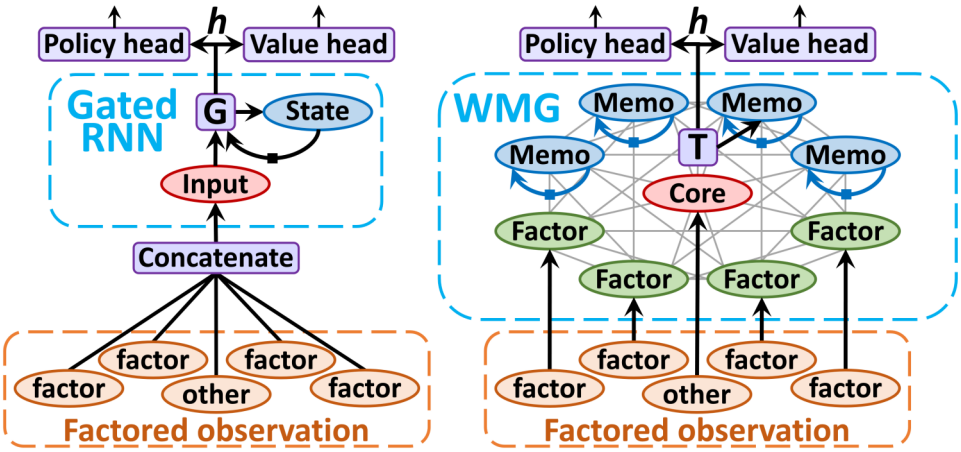

Figure 2.2 Architecture des *Working Memory Graphs* (à droite). Comparaison avec les Gated RNNs (à gauche). Source : figure 1 dans (Loynd *et al.*, 2020).

Chemins raccourcis pour le flux d'information : Le modèle utilise une forme novatrice de récurrence en raccourci, permettant à l'information de circuler à travers de multiples chemins depuis les observations passées jusqu'à l'action actuelle. Cela est réalisé grâce à un ensemble dynamique de vecteurs d'état cachés appelés "Memos".

Memos : Dans le contexte des WMGs, les vecteurs Memo sont un concept clé. Ils sont essentiellement des plongements (*embeddings* en anglais) de sommets dans le graphe qui encapsulent les informations sur l'état de chaque sommet (entité ou caractéristique) et son contexte au sein du graphe. Ces vecteurs sont mis à jour à mesure que le graphe évolue, garantissant qu'ils représentent toujours l'état le plus actuel et les relations des sommets auxquels ils correspondent.

Observations factorisées : Les WMG tirent efficacement parti des observations factorisées, où les observations sont décomposées en composants ou facteurs séparés. Cette capacité à gérer les observations factorisées permet au modèle d'apprendre des politiques plus performantes de manière plus efficace par rapport aux architectures traditionnelles.

Les graphes en tant que métaphore : Il est important de comprendre que la notion de graphe dans les WMGs est plutôt conceptuelle que littérale. Le graphe représente conceptuellement les relations et interactions entre différentes entités (sommets) dans l'environnement. Cela ne signifie pas nécessairement qu'une structure de données de graphe est explicitement utilisée ; plutôt, les relations et la dynamique d'un

graphe sont imitées.

Les WMG diffèrent de l'usage typique d'un Transformeur en partie à cause du prétraitement des observations qui sont faites. Au lieu de fournir à ce dernier directement les observations brutes, une représentation factorisée est faite. Une observation provenant d'un environnement simulé sous forme de grille, par exemple, peut être efficacement représentée par un ensemble de facteurs décrivant les types, les couleurs et les coordonnées relatives à l'agent de tous les objets actuellement visibles sur cette grille. Cette observation factorisée est potentiellement plus compacte que l'observation native, mais variera en taille en fonction du nombre d'objets en vue. Chaque facteur d'observation (ou objet) est intégré dans un vecteur Factor (figure 2.2 à droite) qui sert d'entrée au Transformeur des WMG, avec d'autres Facteurs et les Memos.

Ces WMG prédisent à chaque temps t un état latent h_t qui est ensuite fourni à une tête neuronale pour apprendre la politique et à une tête neuronale pour apprendre les valeurs espérées. Dans un cadre d'apprentissage par renforcement avec la méthode *Actor-Critic* (Konda et Tsitsiklis, 1999), la tête de politique prend en entrée cet état h_t et estime l'action optimale a_t à prendre. La tête de valeur espérée prend en entrée cet état h_t et prédit la valeur estimée à long-terme de l'action.

Donc, comme dans l'approche *Actor-Critic*, l'estimation faite par le modèle de valeur espérée est utilisée pour mettre à jour les gradients du modèle de politique dans le but de maximiser la valeur espérée. Les WMG ont été évalués dans trois environnements différents, dont Sokoban. Ces évaluations ont démontré la supériorité du WMG en termes de performance d'apprentissage et de raisonnement par rapport aux modèles concurrents dans des environnements avec des observations factorisées.

2.3 Apprentissage par renforcement avec modèle

L'apprentissage par renforcement avec modèle est une méthode d'apprentissage machine qui combine les concepts de l'apprentissage par renforcement et l'utilisation d'un modèle de l'environnement. Le concept de modèle de transition fait référence à une représentation abstraite ou mathématique utilisée dans le contexte de l'apprentissage par renforcement.

Ce modèle permet de prédire comment l'environnement va évoluer lorsque l'agent prendra une action donnée à partir d'un état donné. Il est utilisé pour estimer les probabilités de transition entre différents états de l'environnement suite à l'exécution d'une action spécifique par l'agent. Ces transitions décrivent com-

ment l'état de l'environnement évolue en réponse aux actions de l'agent. Mathématiquement, le modèle de transition est souvent représenté sous forme de distribution de probabilités. L' équation (2.10) prend en compte l'état actuel de l'environnement et l'action choisie par l'agent, et retourne la probabilité de passer à chaque état possible. Comme dans l'équation (2.1), on combine parfois s' et r dans la même fonction, plutôt que d'utiliser deux fonctions séparées P et R comme dans notre notation.

$$P(s'|s, a) \tag{2.10}$$

où :

- s est l'état actuel de l'environnement ;
- a est l'action choisie par l'agent ;
- s' est l'état suivant prédit de l'environnement ;
- P représente la probabilité de la transition.

Ce modèle de transition peut être appris à partir de données d'entraînement (par exemple, des observations de l'environnement ou des simulations) ou peut être construit à partir de connaissances préalables sur l'environnement. L'utilisation d'un modèle de transition permet à l'agent d'effectuer des prédictions sur les conséquences potentielles de ses actions sans avoir à les expérimenter réellement dans l'environnement. Cela peut aider l'agent à planifier ses actions de manière plus efficace, à explorer différentes stratégies et à prendre des décisions plus informées pour maximiser ses récompenses dans l'apprentissage par renforcement.

Une fois qu'un modèle de transition P et un modèle de la valeur $Q(s, a)$ (équation 2.3) sont appris à partir d'interactions avec l'environnement, l'algorithme d'apprentissage par renforcement avec modèle est en mesure d'approximer une politique optimale π en générant des simulations. Cette politique peut être apprise directement via un réseau de neurones (typiquement un Transformeur ou un réseau de neurones récurrent) (Racanière *et al.*, 2017; Ritter *et al.*, 2020). Elle peut également être apprise via un algorithme de recherche comme *Monte-Carlo Tree Search* (Coulom, 2006) ou *IDA** (Korf, 1985), parfois lui-même augmenté d'un réseau de neurones comme dans *MuZero* (Schrittwieser *et al.*, 2020) ou dans les algorithmes proposés ici.

Initialement, l'agent apprend les modèles de transition et de valeur en utilisant des données d'entraînement. Ces données peuvent provenir d'observations réelles de l'environnement ou de simulations générées par des connaissances préalables sur l'environnement. Une fois que l'agent a construit ses modèles, il peut simuler des scénarios futurs en utilisant ceux-ci. Il peut alors utiliser des algorithmes de recherche pour décider des actions à prendre afin de maximiser la récompense espérée dans ces scénarios simulés. Ensuite, l'agent peut utiliser les résultats de ses simulations pour agir dans l'environnement réel. Il continue à collecter des données et à mettre à jour son modèle pour l'améliorer au fil du temps.

L'utilisation d'un modèle dans l'apprentissage par renforcement peut accélérer l'apprentissage en permettant à l'agent d'explorer différentes stratégies de manière virtuelle, sans avoir besoin d'expérimenter directement dans l'environnement réel, ce qui peut être coûteux ou prendre beaucoup de temps. Cependant, la précision et la qualité du modèle sont cruciales, car des erreurs de prédiction peuvent conduire à des actions sous-optimales dans le monde réel.

2.3.1 MuZero

L'architecture des deux solutions que nous proposons dans ce mémoire au chapitre 4 (soit l'approche par conviction et l'approche hybride) est basée sur l'algorithme MuZero (Schrittwieser *et al.*, 2020), mais le processus d'apprentissage est différent. Par exemple, nous n'utilisons pas de *self-play* et nous entraînons les modèles à partir de démonstrations. Par contre, l'architecture en soi, l'interaction entre les modules, et l'usage d'un algorithme de planification classique comme *Monte-Carlo Tree Search* en font l'inspiration principale. En particulier, l'architecture de MuZero se compose de 3 fonctions :

1. **La fonction de représentation** : équivalente à ce que nous appelons l'encodeur, ce module se charge de transformer les observations brutes en une représentation interne optimale.
2. **La fonction de la dynamique** : équivalente à ce que nous appelons le modèle de transition. Ce module prédit l'état suivant à partir de l'état actuel et d'une action proposée.
3. **La fonction de prédiction** : ce module apprend une politique, c'est-à-dire une distribution de probabilités sur les actions possibles, ainsi qu'un prédiction de valeur espérée, en fonction d'un état donné. Ce module est utilisé pour guider la recherche MCTS au moment de la planification.

MuZero a été conçu principalement pour apprendre à jouer à des jeux sans avoir une connaissance préalable

des règles spécifiques de ces jeux. Il s'agit d'une extension améliorée de l'approche utilisée par AlphaZero, capable de maîtriser des jeux sans avoir d'informations explicites sur les règles du jeu. Il se distingue d'autres approches d'apprentissage par renforcement avec modèle de plusieurs façons :

Modélisation du monde sans nécessiter la connaissance des règles du jeu : Contrairement à AlphaZero, son prédécesseur, MuZero apprend un modèle de son environnement directement à partir des données d'expérience. Il crée un modèle interne qui prédit les récompenses futures, les états suivants, et les valeurs des états sans avoir besoin de connaître les règles spécifiques ou la physique de l'environnement.

***Self-play* :** il s'agit d'une méthode où un agent apprend en jouant contre lui-même. L'agent applique ses stratégies actuelles pour prendre des décisions tout en s'adaptant continuellement à ses propres tactiques et styles de jeu. Le *self-play* permet à l'agent de découvrir et d'apprendre de nouvelles stratégies et subtilités du jeu, améliorant progressivement ses performances sans nécessiter de données ou d'adversaires humains.

Planification par recherche Monte Carlo : MuZero utilise l'algorithme *Monte Carlo Tree Search* (Coulom, 2006) pour simuler différentes séquences possibles d'actions et d'états, combinant ces simulations avec les estimations fournies par ses modèles neuronaux. Cette méthode permet à MuZero de planifier ses actions de manière efficace pour maximiser ses récompenses potentielles. La fonction de prédiction est utilisée pour guider cette recherche en fournissant les valeurs espérées requises à différentes étapes de l'algorithme (sélection de noeuds d'expansion, rétro-propagation, etc.). De plus, les probabilités d'actions optimales fournies par la fonction de prédiction sont utilisées dans la formule qui priorise les noeuds à explorer afin de favoriser les actions plus probables.

Monte Carlo Tree Search est un algorithme de recherche utilisé principalement pour la prise de décision dans des environnements où les choix sont nombreux et les conséquences difficiles à prédire, comme les jeux de stratégie. Il fonctionne en construisant de manière progressive un arbre de recherche pour explorer les différentes possibilités d'un jeu ou d'une décision. Itérativement, à chaque étape, il sélectionne un noeud, fait l'expansion de celui-ci, génère des simulations à partir de l'un des noeuds enfant nouvellement évalués, et fait une rétropropagation de la valeur espérée. Plus de détails seront fournis à la section 3.2.

2.4 Les récompenses éparses

Dans le cas des récompenses éparses qui nous intéresse, la fonction de récompense $R(s, a)$ qui retourne une récompense r en fonction de l'action a dans l'état s aura très fréquemment une valeur de zéro (ou négative, comme par exemple lorsque chaque déplacement a un coût fixe) et très rarement une valeur positive. Il est donc très difficile, par exploration purement aléatoire, d'effectuer une séquence d'actions qui mène à une récompense positive. Par conséquent, les données d'entraînement comportent un signal de renforcement très faible. Il faut donc beaucoup plus de données pour compenser pour la rareté du signal de renforcement.

Il existe plusieurs stratégies de résolution de problèmes d'apprentissage par renforcement à récompenses éparses. Par exemple, il est possible de restructurer les récompenses de façon à donner à l'algorithme un *feedback* plus constant et progressif. Les implémentations pour les expériences sur Minigrid (introduit à la section 4.1.1), dans ce mémoire, feront appel à cette idée. Une autre stratégie consiste à améliorer l'exploration. Ceci peut se faire de plusieurs façons, comme en favorisant la visite d'états préalablement non explorés (Ladosz *et al.*, 2022). Il est aussi possible de structurer l'entraînement sous forme de curriculum, c'est-à-dire avec une progression planifiée dans la complexité des instances de problème présentées à l'algorithme (Narvekar *et al.*, 2020).

Toutefois, la stratégie qui sera explorée dans cet ouvrage est celle de l'apprentissage par démonstrations. Cette stratégie atténue le problème de l'exploration aléatoire en redéfinissant, en quelque sorte, le problème. En effet, l'idée consiste à entraîner le modèle à partir de démonstrations d'experts. Dans le contexte de l'apprentissage par démonstrations, la politique est apprise, au moins partiellement, à partir de ces démonstrations d'experts, plutôt que seulement par exploration aléatoire.

Nous nous concentrons sur cette stratégie plutôt qu'une autre (comme par exemple, le *reward shaping*), parce qu'elle est plus directe. On fournit directement des solutions, alors qu'avec le *reward shaping* il faut quand même explorer (essai et erreur). En effet, DQfD démontre en pratique que l'état de l'art est généralement basé sur des démonstrations, lorsqu'on considère que l'efficacité de l'utilisation des données est cruciale.

Il faut noter que l'apprentissage par démonstrations ne devrait pas être confondu avec le concept d'apprentissage par renforcement inverse (*Inverse Reinforcement Learning*). Ils ont des similitudes, puisque les deux

dépendent de démonstrations, mais leurs buts sont distincts. Dans l'apprentissage par renforcement inverse (Ng *et al.*, 2000), le but est d'inférer la fonction de récompenses à partir des exemples d'interactions. En d'autres mots, la littérature de l'apprentissage par renforcement inverse ne s'intéresse pas aux récompenses éparses ou à la notion d'efficacité échantillonnale. La recherche dans le domaine de l'apprentissage par renforcement inverse est plutôt motivée par le fait qu'il soit difficile, pour certains problèmes, de définir explicitement la fonction de récompense, car celle-ci est trop complexe ou ambigüe (Abbeel et Ng, 2004).

Deep Q-learning from Demonstrations (DQfD) est un algorithme hybride entre le *Deep Q-learning* (exploration aléatoire) et l'apprentissage par démonstrations (Hester *et al.*, 2018). Voir la section 2.2.5 pour une explication détaillée de l'algorithme. Les auteurs ont comparé DQfD à une variété d'algorithmes d'apprentissage par renforcement dont le *Prioritized Dueling Double DQN*. Ils ont montré qu'il atteint la meilleure performance parmi ces algorithmes sur 11 jeux du *Arcade Learning Environment* (Bellemare *et al.*, 2013) à partir de beaucoup moins d'échantillons d'interactions. De plus, grâce à sa capacité d'apprendre à partir d'exploration aléatoire, sa performance peut excéder celle des démonstrations.

Ceci est possible grâce à un nouveau terme dans la fonction de coût qui vise à pénaliser les valeurs espérées (*Q-values*) associées aux actions non observées pour un certain état. Le *Soft Q Imitation Learning* (SQIL)(Reddy *et al.*, 2019), *Normalized Actor-Critic* (Gao *et al.*, 2018), *Monte Carlo augmented Actor-Critic* (Wilcox *et al.*, 2022), *Cycle-of-Learning* (Goecks *et al.*, 2020) et *Deep Deterministic Policy Gradient from Demonstrations* (Vecerik *et al.*, 2017) utilisent tous essentiellement la même idée, combinée avec l'apprentissage à partir de démonstrations.

La différence principale entre SQIL and DQfD est que SQIL ne requiert pas de récompense explicite de l'environnement. La différence entre *Monte-Carlo augmented Actor-Critic* et DQfD est que ce premier utilise l'approche *actor-critic*, alors que ce dernier fait du *Q-learning*. *Deep Deterministic Policy Gradient from Demonstrations*, quant à lui, est une extension de DQfD pour les espaces d'action continus. En résumé, les approches par démonstrations précédentes sont toutes basées sur la même idée : un algorithme sans modèle qui, pour chaque état, pénalise les actions non observées dans les démonstrations, tout en récompensant les actions observées dans celles-ci.

CHAPITRE 3
LES ALGORITHMES DE RECHERCHE

Ce chapitre présente une revue de la littérature pour les techniques de recherche et de planification utilisées par nos solutions. En effet, nos solutions proposées au chapitre 4 étant des approches d'apprentissage par renforcement avec modèle, la planification à partir d'un modèle prédictif de l'environnement devient cruciale. Sans chercher à faire un survol exhaustif du domaine très vaste qu'est la planification, nous présentons en détail les deux algorithmes de recherche utilisés dans ce mémoire, soit le *Monte-Carlo Tree Search* et l'*Iterative Deepening A**.

Nous débutons ce chapitre avec, dans la section 3.1, une introduction conceptuelle aux algorithmes de recherche. Nous y présentons nos motivations pour la sélection des deux algorithmes considérés. Dans la section 3.2, nous expliquons en détail, incluant du pseudo-code, l'algorithme de *Monte-Carlo Tree Search*. Ce pseudo-code sera pertinent, car, plus tard, nous présenterons seulement les modifications nécessaires à l'algorithme en question pour l'adapter à notre approche. Puis, dans la section 3.3, nous traitons du *Iterative Deepening A** avec le même niveau de détail.

3.1 Mise en contexte

Les algorithmes de recherche, lorsqu'ils sont utilisés dans un contexte de planification, visent à trouver un ou des plans permettant d'atteindre des objectifs spécifiques dans des environnements contrôlés et prévisibles. Ceci implique de définir un état initial, un ensemble d'actions possibles, des contraintes, et un ou plusieurs états cibles ou objectifs. L'objectif est de trouver une séquence d'actions qui mène de l'état initial à l'état cible tout en respectant les contraintes. Cette planification est souvent représentée sous forme d'un espace d'états, où chaque état possible du monde est un sommet dans un graphe, et chaque action possible représente une transition entre ces états (c.-à-d. un arête dans le graphe). Les algorithmes de recherche peuvent être catégorisés en deux grands types : les recherches non informées et les recherches informées.

Recherches non informées : Ces méthodes (Russell et Norvig, 2010), telles que la recherche en profondeur (*Depth-First Search*) et la recherche en largeur (*Breadth-First Search*), explorent l'espace des états sans information spécifique sur la direction dans laquelle l'objectif peut être trouvé. Elles sont simples, mais peuvent être inefficaces dans de grands espaces d'états.

Recherches informées : Ces algorithmes (Russell et Norvig, 2010), comme A* et ses variantes, utilisent des heuristiques pour guider la recherche vers les états les plus prometteurs. Ces méthodes sont souvent plus rapides et efficaces que les recherches non informées, surtout lorsqu'il est possible de concevoir une heuristique permettant de guider la recherche vers le but.

L'usage des algorithmes de recherche en IA est confrontée à plusieurs défis, notamment la gestion des grands espaces d'états et la nécessité d'adapter les plans en réponse à des changements dans l'environnement. Malgré ces défis, ces techniques trouvent des applications dans de nombreux domaines, tels que la robotique, la gestion de projet, les jeux vidéo et les systèmes de recommandation.

3.2 *Monte-Carlo Tree Search*

L'algorithme 2 présente *Monte-Carlo Tree Search* (MCTS), qui est une technique de recherche qui vise à approximer des décisions optimales. Il est particulièrement connu pour son utilisation dans des jeux comme le Go, mais son application s'étend à divers problèmes de décision. Il est particulièrement efficace dans les situations où l'espace de recherche est trop grand pour être entièrement exploré, et il ne nécessite pas de connaissances spécifiques sur le domaine, à part comment simuler des mouvements et reconnaître un état terminal. Il faut noter que, toutefois, il est possible d'ajouter une connaissance du domaine pour guider la recherche. En effet, comme dans MuZero (voir section 2.3.1), il est possible d'utiliser un modèle de probabilités pour prioriser l'ordre d'expansion des noeuds en fonction de l'état observé.

MCTS est unique, car il combine la généralité des méthodes de recherche aléatoire avec la précision des techniques de recherche arborescente. Voici une description détaillée de son fonctionnement :

1. **Sélection** (algorithme 3) : On commence à la racine de l'arbre et on sélectionne successivement des nœuds enfants jusqu'à atteindre un nœud qui n'est pas entièrement développé (toutes les actions possibles n'ont pas encore été explorées) ou un nœud terminal (fin de partie ou fin de scénario). La sélection est guidée par une stratégie qui équilibre l'exploration de nouveaux nœuds avec l'exploitation des nœuds dont les simulations précédentes ont indiqué qu'ils étaient prometteurs. Une formule courante utilisée pour cette équilibre est l'UCT (*Upper Confidence bounds applied to Trees*). Ici, $Q(v)$ représente la valeur $valeur(v)$ du noeud divisée par le nombre de visites $nombre\_visites(v)$.
2. **Expansion** (algorithme 4) : Si le nœud sélectionné n'est pas un nœud terminal, on crée un ou plu-

sieurs nœuds enfants en explorant les actions non évaluées à partir de cet état.

3. **Simulation** (algorithme 5) : À partir du nouveau nœud, on effectue une simulation aléatoire du jeu ou du problème. Cela implique de faire des mouvements aléatoires (ou selon une heuristique légère) jusqu'à ce qu'un état terminal soit atteint.
4. **Rétropropagation** (algorithme 6) : Une fois la simulation terminée, l'issue (victoire, défaite, score) est rétropropagée à travers l'arbre, en remontant jusqu'à la racine. Chaque nœud visité lors de la sélection et de l'expansion est mis à jour avec les nouvelles informations. Spécifiquement, cela implique de mettre à jour le nombre de visites du nœud et l'estimation de la valeur (par exemple, le taux de victoire) de celui-ci.

Algorithme 2 : Monte-Carlo Tree Search

Entrée : $noeud\_racine$: le nœud racine de l'arbre de recherche

N, le nombre d'itérations de la recherche

Résultat : $action$, la meilleure action à partir du nœud racine

```
1 Début
2     pour i ← 0..N faire
3         noeud_choisi ← Sélection de noeud MCTS(noeud_racine)
4         noeud_étendu ← Expansion MCTS(noeud_choisi)
5         valeur ← Simulation MCTS(noeud_étendu)
6         Rétropropagation MCTS(noeud_étendu, valeur)
7     fin
      // Basé sur les statistiques des nœuds enfants
8     action ← choisir la meilleure action à partir de noeud_racine
9 Fin
```

Algorithme 3 : Sélection de noeud MCTS

Entrée : v_0 : le nœud initial

C : paramètre de compromis exploration/exploitation

Résultat : v, le nœud sélectionné

Début

$v \leftarrow v_0$

`// On parcourt les noeuds de l'arbre jusqu'à une feuille non explorée`

Tant que v *est entièrement exploré et n'est pas terminal* **faire**

$meilleurs\_enfants \leftarrow \{\}$

$meilleur\_UCT \leftarrow -\infty$

`// À chaque étape on choisit le ou les enfants qui maximisent l'UCT`

pour $enfant \in enfants(v)$ **faire**

si $nombre\_visites(enfant) = 0$ **alors**

$UCT \leftarrow \infty$

fin

sinon

$UCT \leftarrow Q(v) + 2C \cdot \frac{\sqrt{(2\log(nombre\_visites(v)))}}{nombre\_visites(enfant)}$ `// Calcul de la valeur UCT`

fin

si $UCT \geq meilleur\_UCT$ **alors**

si $UCT > meilleur\_UCT$ **alors**

$meilleurs\_enfants \leftarrow \{\}$

$meilleur\_UCT \leftarrow UCT$

fin

$meilleurs\_enfants \leftarrow meilleurs\_enfants \cup \{enfant\}$

fin

fin

`// Sélection aléatoire en cas de plusieurs meilleurs enfants`

$v \leftarrow choix(meilleurs\_enfants)$

fin

Fin

Algorithme 4 : Expansion MCTS

Entrée : v_t : le nœud actuel

Résultat : $enfant\_t$, un enfant aléatoire de v_t (ou v_t si terminal)

Début
  si v_t *est terminal* **alors**
    retourner v_t
  fin
  si v_t *n'est pas entièrement exploré* **alors**
    `// On crée un enfant pour chaque action possible`
    pour *chaque action* a *non explorée à partir de* v_t **faire**
      $s' \leftarrow$ Appliquer a sur l'état de v_t
      $enfant \leftarrow$ Créer un nouveau nœud pour s'
      Ajouter $enfant$ comme enfant de v_t
      Marquer a comme explorée pour v_t
    fin
    Marquer v_t comme entièrement exploré
  fin
  $enfant\_t \leftarrow$ un enfant aléatoire de v_t
Fin

Algorithme 5 : Simulation MCTS

Entrée : v_t : le nœud initial

P : la profondeur maximale de simulation

γ : le facteur d'escompte

Résultat : $valeur\_noeud$, estimation de la valeur du nœud

Début

  $valeur\_noeud \leftarrow 0$

  $v \leftarrow v_t$

  $profondeur \leftarrow 0$

  $escompte \leftarrow 1$

  `// On simule une séquence d'actions et la récompense cumulative`

  Tant que $profondeur < P$ *et* v *n'est pas terminal* **faire**

    $a \leftarrow$ choisir une action aléatoire pour v

    $noeud\_suivant, r \leftarrow$ appliquer l'action a sur v

    $valeur\_noeud \leftarrow valeur\_noeud + escompte \cdot r$

    $escompte \leftarrow escompte \cdot \gamma$

    $v \leftarrow noeud\_suivant$

    $profondeur \leftarrow profondeur + 1$

  fin

Fin

Algorithme 6 : Rétropropagation MCTS

Entrée : v_t : le nœud de départ de la rétropropagation

r : la récompense à rétropropager

Résultat : Valeurs des noeuds antérieurs mises à jour

```
1 Début
2     v ← v_t
3     Tant que v n'est pas la racine faire
4         incrémenter nombre_visites(v)
5         valeur(v) ← valeur(v) + r
6         v ← parent(v)                          // Remonter vers la racine
7     fin
8 Fin
```

Dans l'algorithme 2, qui représente la boucle principale de *Monte-Carlo Tree Search*, ces quatre étapes sont répétées un grand nombre de fois, ce qui permet à l'algorithme de construire progressivement une estimation de plus en plus précise des actions les plus prometteuses. Une fois le processus de recherche terminé (souvent après un certain temps ou un nombre de simulations donné), MCTS sélectionne l'action à entreprendre à partir de la racine de l'arbre, généralement l'action correspondant au nœud le plus visité ou ayant la plus haute valeur estimée.

La performance de MCTS dépend de la qualité des simulations. Dans certains cas, l'algorithme peut nécessiter un grand nombre de simulations pour atteindre la performance désirée, et la sélection d'actions durant les simulations peut affecter l'efficacité de l'algorithme.

En conclusion, MCTS est une méthode puissante et flexible pour aborder des problèmes de décision complexes, surtout dans les domaines où la conception d'heuristiques précises est difficile ou impossible. Son utilisation dans des jeux comme le Go a démontré sa capacité à résoudre des problèmes avec un haut degré de complexité et d'incertitude.

3.3 *Iterative Deepening A**

Puisque *Iterative Deepening A** (IDA*) est basé sur l'algorithme A*, il est d'abord nécessaire de se familiariser avec ce dernier. A* est un algorithme de recherche qui sert à trouver le chemin le plus court entre

un sommet de départ et un sommet satisfaisant le but dans un espace de recherche. Cet algorithme est connu pour sa capacité à trouver le chemin le plus court de manière efficace en utilisant une heuristique. A* évalue les chemins à travers le graphe de recherche en combinant le coût réel pour atteindre un sommet (noté g) et une estimation heuristique du coût pour atteindre l'objectif à partir de ce sommet (noté h). La fonction d'évaluation est donc $f = g + h$.

L'efficacité de A* dépend essentiellement de la qualité de l'heuristique utilisée. Une bonne heuristique vise à s'approcher autant que possible du coût réel pour atteindre l'objectif sans toutefois le surestimer. Ceci garantit que A* trouve toujours le chemin le plus court si un tel chemin existe. Bien que A* soit efficace en termes de recherche du chemin optimal, il peut consommer beaucoup de mémoire, car il doit mémoriser tous les sommets explorés et leurs alternatives.

Pour mieux comprendre *Iterative Deepening A* (IDA*)*, nous pouvons décomposer son fonctionnement en trois composantes principales : la boucle principale, la boucle de recherche, et la fonction d'expansion des nœuds enfants. Cette structure reflète la manière dont IDA* intègre les caractéristiques de l'algorithme A* dans une stratégie de recherche en profondeur.

Algorithme 7 : *Iterative Deepening A**

Entrée : $sommetInitial$: sommet correspondant à l'état de départ

N : nombre maximal d'expansions de sommets

Résultat : *cheminTrouvé*, Séquence d'actions pour atteindre un sommet but ou indication d'échec

```
1  Début
2      trouvé ← faux
3      cheminInitial ← {sommetInitial}
4      g ← 0
5      limiteCout ← heuristique(sommetInitial)
6      totalExpansionSommets ← 0
7      Tant que non trouvé et totalExpansionSommets < N faire
8          trouvé, cheminTrouvé ← Recherche IDA*(cheminInitial, limiteCout, g)
9          si trouvé alors
10             renvoyer cheminTrouvé
11         fin
12         limiteCout ← limiteCout + 1
13         totalExpansionSommets ← totalExpansionSommets + taille (cheminTrouvé)
14     fin
15     renvoyer «Echec»
16 Fin
```

La Boucle Principale (algorithme 7) : La boucle principale contrôle l'augmentation itérative de la limite de coût pour l'exploration des chemins. Elle démarre avec une limite fixée à la valeur heuristique du sommet de départ. Après chaque exploration exhaustive jusqu'à cette limite, elle augmente la limite pour la prochaine itération. L'heuristique joue un rôle crucial en déterminant la limite initiale et en guidant les augmentations successives.

Algorithme 8 : Recherche IDA*

Entrée : $chemin$: liste de sommets parcourus (contient seulement le sommet de départ au premier appel)

$limiteCout$: limite de coût heuristique pour cette itération

g : coût réel du sommet de départ jusqu'au sommet actuel

Résultat : (Booléen indiquant si un sommet but a été trouvé ; $chemin$, le chemin trouvé)

```
Début
    sommet ← dernier élément de chemin
    h ← heuristique(sommet)
    f ← g + h
    si f > limiteCout alors
        renvoyer faux, chemin
    fin
    si sommet est un sommet but alors
        renvoyer vrai, chemin
    fin
    sommetsAExplorer ← Expansion IDA*(sommet)
    // Récursion pour trouver un chemin de profondeur limiteCout ou moins
    pour sommetVoisin dans sommetsAExplorer faire
        si sommetVoisin n'est pas dans chemin alors
            chemin ← chemin ∪ {sommetVoisin}
            g_tmp ← g+ coût(sommet, sommetVoisin)
            resultat, tmp_chemin ← Recherche IDA*(chemin, limiteCout, g_tmp)
            si resultat alors
                renvoyer vrai, tmp_chemin
            fin
            chemin ← chemin\{sommetVoisin}
        fin
    fin
    renvoyer faux, chemin
Fin
```

Algorithme 9 : Expansion IDA*

Entrée : $sommetInitial$: le sommet initial

Résultat : $voisinsValides$, la liste de voisins valides

```
1 Début
2     voisinsValides ← voisins(sommetInitial)
      // Si nous n'avons jamais fait l'expansion de ce sommet:
3     si voisinsValides est vide alors
4         pour action parmi les actions possibles depuis sommetInitial faire
5             voisin ← appliquer action à sommetInitial
6             ajouter voisin à voisinsValides
7         fin
8     fin
9 Fin
```

La boucle de recherche (algorithme 8) : Cette boucle réalise la recherche en profondeur dans les limites de coût définies par la boucle principale. Elle parcourt les sommets, évaluant leur coût total ($f = g + h$) et les comparant à la limite actuelle. Si le coût d'un sommet dépasse la limite actuelle, la boucle de recherche s'arrête et remonte, permettant à la boucle principale d'augmenter la limite pour de futures itérations.

La fonction d'expansion des sommets voisins (algorithme 9) : Cette fonction gère l'expansion des voisins du sommet courant. Pour chaque sommet, elle génère les sommets voisins et les ajoutent à la liste de voisins valides.

En résumé, IDA* utilise la boucle principale pour gérer de manière itérative les limites de coût, la boucle de recherche pour explorer l'espace de recherche dans ces limites, et la fonction d'expansion des sommets voisins pour générer de manière sélective de nouveaux chemins à explorer. Cette structure permet à IDA* d'être efficace en termes de mémoire tout en conservant la capacité de l'algorithme A* à trouver le chemin le plus court.

CHAPITRE 4
MÉTHODOLOGIE

Dans ce chapitre, nous présentons d'abord à la section 4.1 les deux environnements de simulation que nous utilisons pour nos expériences, car nous y faisons référence tout au long de notre explication des solutions proposées. À la section 4.2, nous expliquons le raisonnement et les considérations que nous avons explorées afin d'arriver à nos solutions proposées. À la section 4.3, nous détaillons l'architecture sous-jacente aux deux approches. À ce stade-ci nous ne distinguons pas encore ces deux approches, car elles possèdent une grande base commune. Aux sections 4.4 et 4.5, toutefois, nous introduisons les différences entre les deux approches proposées, soit l'approche par conviction et l'approche hybride respectivement.

4.1 Environnements de simulation

Dans cette section, nous présentons les deux environnements de simulation qui seront utilisés dans nos expériences. Ces environnements virtuels se veulent comme des exemples d'utilisation permettant de comparer la performance de nos solutions proposées, sans toutefois limiter les algorithmes à ceux-ci. En effet, les solutions proposées dans ce mémoire sont plus générales et peuvent s'appliquer à divers domaines de problèmes qui requièrent une planification dans l'espace.

Nous avons donc choisi ces environnements car, d'une part, ils sont simples d'utilisation et d'intégration, permettant ainsi de nous concentrer sur le développement de l'algorithme plutôt que des détails d'interaction ou de perception. De plus, ils sont tous les deux des environnements considérés comme difficiles pour l'apprentissage par renforcement typique, à cause de leurs récompenses éparses. Finalement, ils requièrent tous les deux un bon degré de planification, à cause de leur complexité combinatoire relativement élevée.

Il faut également noter que notre approche présume la nécessité de l'apprentissage. C'est-à-dire qu'en principe, il serait possible de simplement programmer un modèle de ces environnements et un modèle de valeur (ou une description de l'état à atteindre) et d'utiliser des techniques classiques de recherche ou de planification pour les résoudre. Or, notre objectif s'inscrit dans la lignée de la recherche en apprentissage automatique, en visant la généralité. L'objectif est de découvrir des approches plus générales pour résoudre des problèmes. Donc, ici, nos algorithmes utilisés devront être capables d'apprendre les buts et les règles de l'environnement par eux-mêmes.

4.1.1 Minigrid

Minigrid (Chevalier-Boisvert *et al.*, 2018) est un environnement virtuel spécialement conçu pour évaluer des méthodes d'apprentissage par renforcement. Il est basé sur des grilles en deux dimensions. Un agent contrôlé par l'algorithme doit naviguer cette grille en utilisant seulement les 4 actions suivantes :

— Avancer (dans la direction pointée) ;
— Tourner à gauche (de 90 degrés) ;
— Tourner à droite (de 90 degrés) ;
— *Toggle* (ce qui ouvre ou ferme une porte).

Minigrid contient différents niveaux avec différents objectifs, comme atteindre certaines cases ou ramasser certains objets. Dans nos expériences, nous n'avons utilisé que des niveaux dont l'objectif est d'atteindre une case spéciale dans la grille. Cette case est visuellement représentée par une case verte. De plus, cette case doit être atteinte en moins de 250 interactions, sinon la tentative de résolution est un échec.

Une particularité de Minigrid est qu'il s'agit d'un environnement partiellement observable. L'agent n'observe qu'une fenêtre de 7×7 devant lui, en tout temps. La figure 4.1 présente un exemple de grille Minigrid. Dans cet exemple, la séquence d'actions (gauche, avancer, avancer, avancer, avancer, avancer, gauche, avancer) permettrait de résoudre le problème. L'agent reçoit une récompense positive lorsque la cellule verte, correspondant au but final, est atteinte. Sinon, pour chaque action, il reçoit une récompense de zéro.

La section devant le pointeur qui est colorée de façon plus pâle correspond à la fenêtre d'observation de l'agent. Dans certains types de grille (que nous n'utilisons pas dans nos expériences), on peut également ramasser des items. Le but est typiquement d'atteindre une cellule spécifique, bien que Minigrid contienne plusieurs types de niveaux ayant différents buts (comme ramasser certains objets). Les niveaux utilisés dans nos expériences auront tous le même objectif : atteindre une cellule spécifique représentée visuellement par un carré vert. Nous expérimentons avec deux niveaux différents :

1. `MiniGrid-FourRooms-v0` : dans ce niveau, chaque grille a une dimension de 19×19. La figure 4.1 montre un exemple. Il y a 3 actions pertinentes : avancer, tourner à droite et tourner à gauche.
2. `MiniGrid-MultiRoom-N6-v0` : dans ce niveau, chaque grille a une dimension de 25×25. La figure 4.2 montre un exemple. Les 4 actions sont requises pour ce niveau.

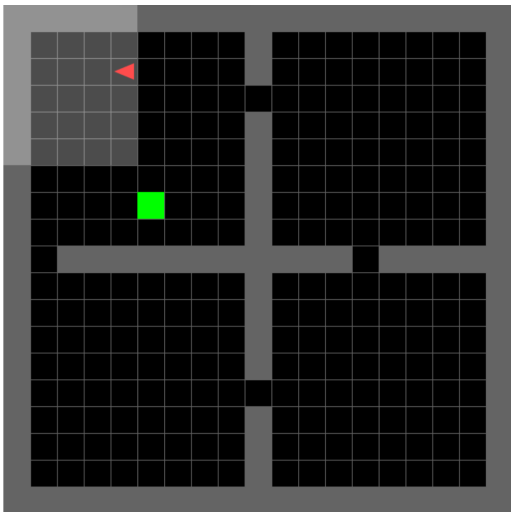

Figure 4.1 Exemple d'une grille dans `MiniGrid-FourRooms-v0`. Le triangle rouge indique la position et l'orientation de l'agent, et la case verte est le but à atteindre.

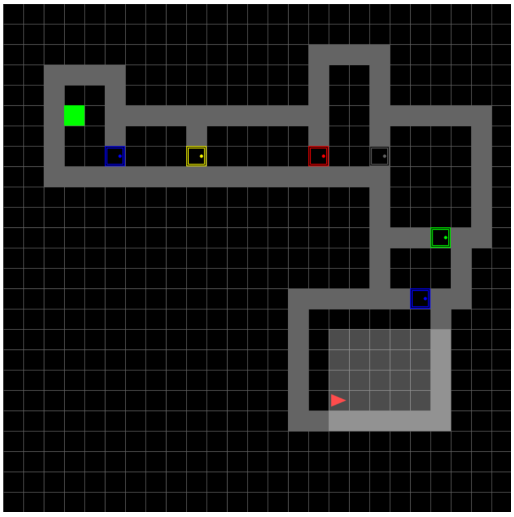

Figure 4.2 Exemple d'une grille dans `MiniGrid-MultiRoom-N6-v0`. Le triangle rouge indique la position et l'orientation de l'agent. La case verte est le but à atteindre. Les cellules encadrées de couleur avec un point à l'intérieur représentent des portes à ouvrir.

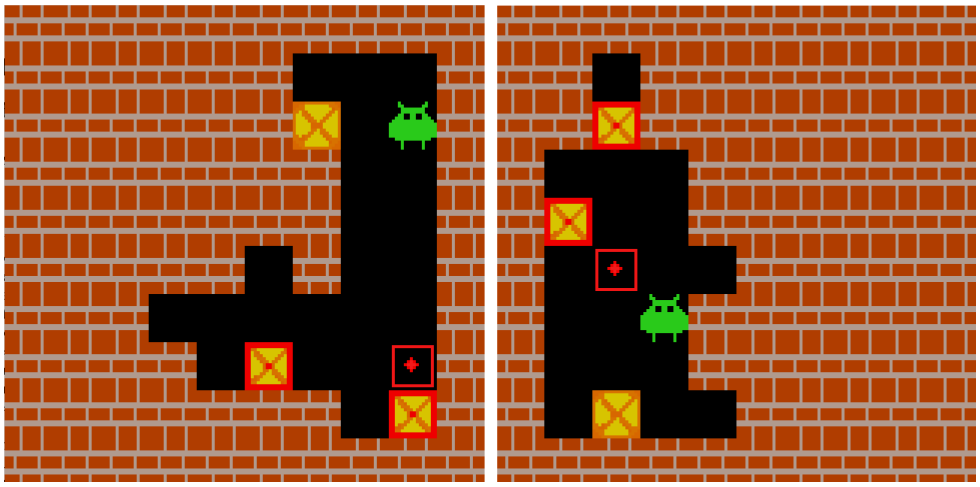

Figure 4.3 Exemples de grilles Sokoban. Les cases oranges contenant un 'X' sont des boîtes. Lorsque leur encadré est rouge, plutôt qu'orange, la boîte est sur une case cible. Les carrés rouges avec un point au centre sont des cases cibles, la créature verte est l'agent. **Gauche** : *deadlock* dû à une boîte prise dans un coin. **Droite** : *soft deadlock* sur le mur du bas, car la boîte ne pourra jamais atteindre la cible non occupée.

Notre choix de Minigrid comme domaine de problèmes est justifié par le fait que Minigrid est normalement considéré comme une problème difficile pour les techniques d'apprentissage par renforcement standard. Leur documentation mentionne également ce fait. Cette difficulté est principalement dûe au fait que l'exploration aléatoire peut difficilement résoudre un niveau.

4.1.2 Sokoban

Sokoban est également un environnement virtuel en deux dimensions qui utilise des grilles. Le but est de pousser des boîtes vers des cellules spécialement identifiées comme étant les cibles. Chaque grille contient un nombre égal de boîtes et de cibles, puisque chaque boîte devra être associée à une cible respective. Lorsque chaque boîte se retrouve sur une case cible, le niveau est complété.

L'implémentation de Sokoban que nous utilisons comporte 4 actions possibles : haut, bas, droite, gauche. Si l'agent se déplace dans une direction contenant une boîte, il va pousser la boîte d'une cellule dans cette même direction. Si, toutefois, la cellule dans la direction visée contient un mur, ou une boîte derrière laquelle se trouve un obstacle (soit un mur ou une autre boîte), l'agent ne se déplacera pas. Autrement dit, il n'est pas possible de pousser deux boîtes en même temps.

L'agent reçoit une récompense de 1 lorsqu'il pousse une boîte sur une cellule cible, -1 lorsqu'il pousse une

boîte d'une cellule cible vers une cellule vide régulière. Il obtient une récompense de 10 lorsqu'il complète le niveau. Toutes les instances de problème générées ont une solution possible, donc pour maximiser la récompense l'agent doit trouver cette solution.

Bien que l'environnement Minigrid soit un bon point de départ pour analyser nos algorithmes proposés, son niveau de difficulté n'est pas aussi élevé que Sokoban en ce qui concerne l'aspect planification. De plus, Minigrid est plus facile à résoudre avec l'approche par apprentissage par démonstrations que Sokoban. Il y a plusieurs raisons pour cette différence de difficulté, qui seront explorées au cours de ce mémoire. Entre autres, l'existence de *deadlocks*, l'existence d'heuristiques simples pour résoudre certains niveaux de Minigrid (se diriger vers les portes fermées, par exemple), et la simplicité conceptuelle de Minigrid en font tous partie. En effet, dans Minigrid il ne suffit que de se déplacer vers la case verte. Dans Sokoban, il y a la complexité additionnelle de devoir déplacer des objets vers certaines cases spécifiques, et ce seulement en les poussant. Il y a ainsi plusieurs niveaux de planification : où les boîtes doivent aller, comment se rendre à chaque boîte, comment se positionner pour les pousser. Donc, l'environnement Sokoban nous permet de mieux repousser les limites de nos algorithmes comparatifs, et ainsi de mieux mettre en valeur nos approches proposées.

De plus, l'objectif de ce mémoire est de développer un algorithme capable de résoudre une variété de problèmes de planification avec peu de modifications spécifiques au problème. On présume un contexte où la fonction de transition et de récompense du problème n'est pas spécifié explicitement à l'agent. Autrement dit, les règles et les mécanismes du jeu ne sont pas explicitement spécifiés. Il faut donc utiliser l'apprentissage automatique pour les découvrir. Ceci motive l'usage d'un deuxième domaine de problème, permettant de montrer qu'avec seulement quelques modifications (en particulier, l'encodage des observations) on peut ré-utiliser l'algorithme dans différents contextes.

Un aspect de Sokoban qui le rend plus difficile à résoudre que Minigrid est le problème des *deadlocks*. Les *deadlocks* sont des situations à éviter, car elles rendent le niveau impossible à résoudre. Plus précisément, un *deadlock* se produit lorsqu'une boîte n'est pas sur une cellule cible et il n'existe aucune séquence de mouvements possible pour l'apporter vers une cellule cible non occupée. Ces situations sont possibles, car il n'existe pas d'action permettant de tirer une boîte : on peut seulement les pousser.

On parle donc de *deadlock* lorsqu'une boîte (n'occupant pas une case cible) est prise dans un coin. Il est

alors impossible pour l'agent de se placer de l'autre côté de la boite afin de la pousser de cette case (figure 4.3, image de gauche). On peut également parler de *soft deadlock* lorsque la boîte est collée à un mur, qu'il n'est pas possible de la décoller de ce mur et que cette colonne ou cette rangée ne contient pas de cible non occupée (figure 4.3, image de droite). On peut donc anticiper que les méthodes basées sur l'exploration aléatoire, comme l'apprentissage par renforcement avec ou sans modèle standard, auront beaucoup de difficulté à produire des séquences d'actions qui permettent de compléter un niveau. Par conséquent, on peut s'attendre à une complexité échantillonnale très élevée pour ces méthodes.

Certaines expériences (voir les expériences avec WMG, à la section 5.1.2) utilisent les niveaux Boxoban comme dans (Loynd *et al.*, 2020). Ces niveaux sont des niveaux pré-générés pour le jeu Sokoban, disponibles à partir du répertoire GitHub : `https://github.com/google-deepmind/boxoban-levels`. Les règles ne sont pas différentes de ce dernier. Ils correspondent simplement à des instances spécifiques de niveaux Sokoban, avec quatre boîtes et quatre cases cible, pré-classifiées en niveaux de difficulté et en groupements d'entraînement et de test. Il a seulement fallu faire quelques modifications mineures au code de l'environnement Sokoban que nous utilisons pour pouvoir charger et évaluer ces niveaux.

4.2 Considérations

Malgré l'avantage de la planification (implicite ou explicite) pour les environnements hautement combinatoires, il n'en reste pas moins que la rareté des récompenses demeure un problème pour l'apprentissage par renforcement avec modèle ou l'apprentissage par renforcement sans modèle avec planification implicite. En effet, ces paradigmes sont normalement fondés sur l'exploration aléatoire. Or, dans un contexte de récompenses éparses, l'exploration aléatoire rend difficile l'observation de récompenses positives. Puisque la probabilité d'observer une récompense positive est très faible, il faut un nombre d'interactions beaucoup plus élevé. C'est pourquoi l'apprentissage par démonstrations est fortement utile dans ce contexte, et c'est pourquoi les deux solutions proposées dans ce mémoire consistent à combiner l'apprentissage par démonstrations et le paradigme d'apprentissage par renforcement avec modèle.

Il est vrai que le *reward shaping* pourrait améliorer la situation en rendant ces récompenses plus fréquentes. Il n'en demeure pas moins qu'il faudrait plus d'interactions d'exploration que si nous obtenons directement un exemple de solution. De plus, la complexité du *reward shaping* requis augmente avec la complexité du problème. Dans Sokoban, on pourrait donner une récompense progressive lorsque l'agent s'approche d'une boîte, et lorsqu'une boîte s'approche d'un but. Mais cette récompense intermédiaire est potentiellement

confondante car elle peut encourager des situations qui placent une boîte en *deadlock*, par exemple. Il faudrait alors ajouter une notion de détection de *deadlock* à la récompense intermédiaire. Aussi, dans plusieurs problèmes, l'ordre de placement des boîtes et leur assignements aux cases but sont cruciaux pour l'obtention d'une solution. La récompense intermédiaire mentionnée précédemment ne tient pas compte de ce concept, et pourrait encourager l'agent à déplacer une boîte qui devrait être déplacée en dernier, ou à la placer sur une case but qui n'est pas la bonne (empêchant ainsi la résolution du reste du problème). Ainsi, nous nous concentrons sur l'apprentissage par démonstrations.

Chaque étape d'une démonstration consiste en un tuple $\{s, a, s', r\}$ selon lequel on applique l'action a dans l'état s afin d'atteindre un nouvel état s' et une récompense r. Dans certaines solutions, comme DQfD, la politique est pré-entraînée sur ces séquences de démonstrations, et une phase subséquente est ajoutée dans laquelle une exploration aléatoire acquiert des données d'interactions additionnelles. Puisque les démonstrations expertes incluent normalement ces récompenses rares, car celles-ci sont des solutions complètes pour chaque problème présenté, ceci permet au modèle de valeur d'apprendre à partir de peu d'interactions. En quelque sorte, on peut considérer les démonstrations comme une forme plus dense (en termes de récompenses) d'interactions que l'exploration aléatoire.

Les approches d'apprentissage par démonstration dans la littérature sont toutes basées sur des paradigmes d'apprentissage par renforcement sans modèle. Nous proposons dans ce mémoire deux façons différentes d'utiliser l'apprentissage par démonstration avec un paradigme d'apprentissage par renforcement avec modèle (voir section 2.3) afin de bénéficier de ses avantages. On cherche donc à combiner l'approche avec modèle et l'apprentissage à partir de démonstrations afin de profiter des avantages des deux en termes d'efficacité échantillonnale (voir tableau 4.1).

Table 4.1 Comparaison qualitative entres les catégories d'approches : leur capacité de gérer les récompenses éparses et de planifier

Algorithme	**Récompenses éparses**	**Capacité de planifier**
Approches de planification implicite	Non	Oui
Apprentissage par renforcement avec modèle	Non	Oui
Apprentissage par démonstrations sans modèle	Oui	Non
Planificateurs par conviction et hybride	Oui	Oui

Or, la combinaison naïve des deux paradigmes ne fonctionne pas bien, tel que démontré empiriquement

dans ce mémoire par les résultats de *NaivePlanner-MCTS/IDA** à la section 5.3. En effet, la combinaison de l'apprentissage par renforcement avec modèle et de l'apprentissage par démonstrations implique que le modèle de transition est appris à partir de démonstrations d'experts, plutôt que d'exploration aléatoire. Les démonstrations d'experts sont, de par leur nature, des séquences (quasi-)optimales d'interactions qui mènent à une solution pour chaque problème. Ainsi, les paires état-action présentes dans les démonstrations ne couvrent pas la totalité des possibilités : les transitions absurdes, invalides, ou contre-productives ne sont typiquement pas effectuées. Par exemple, dans Minigrid, un expert ne déplacera pas volontairement l'agent dans un mur, ou dans une porte fermée. Dans Sokoban, le démonstrateur ne tentera pas de pousser un bloc dans un mur.

Par conséquent, le modèle de transition entraîné uniquement sur des démonstrations est incomplet. Si on l'interroge sur des paires état-action qui n'ont jamais (ou très rarement) été observées pendant l'entraînement, les prédictions faites ne seront pas fiables. Dans Minigrid, par exemple, nous avons interrogé le modèle afin de prédire le résultat d'avancer dans une porte fermée, et celui-ci a prédit que l'agent allait passer au travers de celle-ci.

Évidemment, ceci est un problème pour l'apprentissage par renforcement avec modèle puisque l'algorithme de planification, qu'il s'agisse de *Monte-Carlo Tree Search*, *d'IDA**, ou autre, interroge le modèle de transition systématiquement sur tout un intervalle d'actions pour chaque état qui lui est présenté. Inévitablement, donc, l'algorithme de recherche interrogera le modèle de transition pour une paire état-action qui n'a jamais été observée et celui-ci retournera une prédiction incorrecte. Ceci causera un écart entre la trajectoire prévue et la réalité, pouvant faire en sorte qu'une opportunité de résoudre le problème soit ignorée, ou qu'une fausse solution soit proposée.

Si, toutefois, la planification est un module appris (comme un réseau de neurones), celui-ci peut se trouver éventuellement dans un état jamais observé. En effet, rien ne garantit, dans un tel module de planification, que l'état résultant fasse partie de la distribution des données d'entraînement. Donc, à ce moment, la prédiction sera incorrecte, et on se retrouve alors dans la même situation d'une trajectoire planifiée déconnectée de la réalité. Nos deux solutions proposées (Ouellette *et al.*, 2024), expliquées subséquemment dans ce chapitre, visent à contourner ce problème.

4.3 Architecture de modèle

L'architecture du modèle suit le barème typique des algorithme d'apprentissage par renforcement avec modèle. Elle comporte 3 composantes principales :

- un encodeur : $E : (o_t, a_t) \rightarrow s_t$;
- un modèle de valeur : $V : s_t \rightarrow \hat{V}_t$;
- un modèle de transition : $T : (s_t, a) \rightarrow \delta s_t$.

Pour Minigrid, le modèle de valeur est entraîné à estimer la récompense associée à l'état s_t (0 partout sauf lors de l'atteinte de l'objectif). Pour Sokoban, les détails varient (voir section 4.4.2). Dans le modèle de transition, δs_t correspond à une prédiction du différentiel entre le vecteur d'entrée s_t et le vecteur d'état s_{t+1} résultant de l'application de l'action a.

L'encodeur est optionnel : Il est utile dans le cas où l'environnement ne produit pas une observation symbolique, et produit plutôt une observation brute o_t. Dans un tel cas, il serait nécessaire d'avoir, par exemple, un auto-encodeur convolutif qui condense les observations brutes en un espace latent plus accessible à l'algorithme proposé. Ce serait alors dans cet espace latent que les itérations de planification se produisent. Toutefois, dans nos expériences avec Sokoban et Minigrid, nous avons déjà des observations symboliques. Autrement dit, nous recevons directement s_t, sous forme d'un vecteur *one-hot encoded*.

Pendant l'entraînement, l'algorithme minimise la fonction de coût représentée par l'équation (4.1).

$$L = L_s + L_V \tag{4.1}$$

où :

$$L_s = \sum (\delta \hat{s}_t | a'_t - \delta s_t | a'_t)^2 \tag{4.2}$$

$$L_V = \sum (\hat{V}_t - V_t)^2 \tag{4.3}$$

Dans la fonction de coût L_s, $\delta \hat{s}_t | a'_t$ est le différentiel d'état prédit pour l'action a'_t, et $\delta s_t | a'_t$ est le vrai dif-

férentiel qui a été observé pour la même action. L'action a'_t correspond à l'action qui a été exécutée dans la démonstration ou l'interaction aléatoire (dans le cas de l'approche hybride). $\hat{V}_t$ correspond à la valeur estimée par le modèle de valeur pour s_t, et V_t correspond à vraie valeur de s_t.

À chaque itération de la procédure d'entraînement, on échantillonne un lot à partir des données de démonstrations. L'observation o_t est ensuite facultativement fournie à un encodeur E, qui produit s_t. Dans notre cas, les données sont déjà symboliques, donc nous recevons directement s_t. Ce vecteur est ensuite passé au modèle de transition T, qui prédit le différentiel d'état δs_t. En parallèle à ce flux, s_t est également fourni directement au modèle de valeur, qui émet une prédiction $\hat{V}_t$. Nous avons ensuite toutes les informations nécessaires pour calculer les coûts et appliquer les *gradients*.

4.4 Approche par conviction

La stratégie que nous proposons dans l'approche par conviction est d'obtenir une estimation de l'incertitude pour chaque prédiction faite par le modèle de transition. Celle-ci est ensuite utilisée dans l'algorithme de planification afin d'ignorer les trajectoires trop incertaines et choisir la trajectoire proposée qui maximise la conviction (équation (4.4)), une nouvelle métrique que nous avons introduite. Par trajectoire, nous faisons référence à une séquence d'actions proposées par l'algorithme de planification et des transitions d'état prédites pour chaque action. Ainsi, l'incertitude d'une trajectoire correspond à la somme des incertitudes associées à chaque prédiction de transition dans sa séquence.

La figure 4.4 présente l'architecture de l'approche par conviction. Elle contient un encodeur interne et un tableau d'incertitude. L'encodeur externe, qui est optionnel, correspond à l'encodeur permettant de convertir une observation non-symbolique en un vecteur interne, tel que mentionné à la section 4.3. L'encodeur interne est un nouveau module, obligatoire pour l'approche par conviction.

Notre implémentation de l'estimation d'incertitude fonctionne en sauvegardant en mémoire les encodages d'espace latent z_t provenant de l'ensemble de données d'entraînement. Au moment de l'inférence, lorsque l'encodage z_t est produit afin d'effectuer une nouvelle prédiction de transition, nous calculons la distance entre z_t et le vecteur le plus similaire dans les vecteurs mémorisés. Cette distance σ_t sert d'approximation de l'incertitude dans notre modèle. Une distance de zéro signifie que cette transition a été observée pendant l'entraînement, ce qui indique que la prédiction faite est fiable.

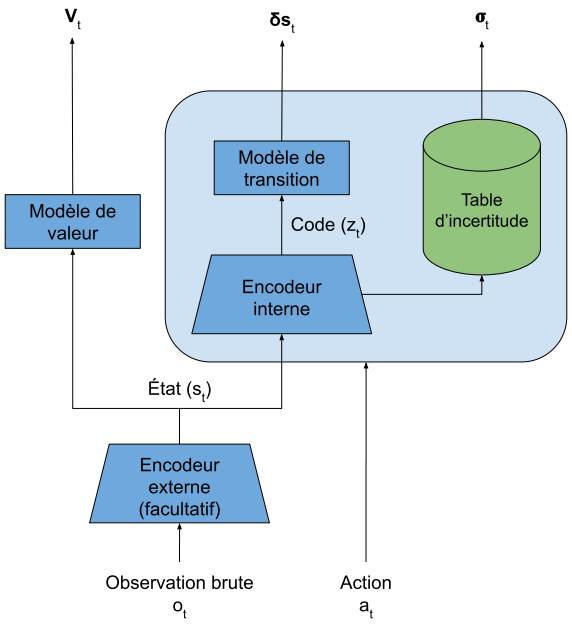

Figure 4.4 Architecture du *ConvictionPlanner*.

Il faut noter que cette implémentation est un cas particulier, spécifique aux environnements déterministes. En effet, dans un environnement non déterministe, il y a une valeur ajoutée au fait d'utiliser plusieurs observations et de considérer leur variance. Ainsi, dans de tels environnements, il est plus efficace d'utiliser un modèle stochastique, comme par exemple un processus gaussien. Nos expériences sont faites avec des environnements déterministes, donc notre implémentation suffit.

$$\psi = \frac{\hat{V}_{t+K}}{\sum_{\tau=t}^{t+K} \sigma_\tau + \epsilon} \tag{4.4}$$

Dans l'équation (4.4), $\hat{V}_{t+K}$ correspond à la valeur estimée pour l'état prédit à la fin de la trajectoire planifiée de longueur K. Le dénominateur consiste en l'incertitude (σ_τ) cumulative des transitions prédites dans la séquence planifiée. ϵ est une constante très proche de zéro qui sert à prévenir une division par zéro. Cette constante n'a pas d'impact significatif sur les performances de l'algorithme, à l'intérieur d'un intervalle raisonnable (par exemple, < 0.001).

Les éléments clé de l'approche par conviction sont :

— une fonction d'encodage qui transforme l'état s_t en un état interne z_t qui est sémantiquement pertinent ;
— un mécanisme qui estime l'incertitude associée à une prédiction du modèle de transition ;
— une modification de l'algorithme de planification qui élague les trajectoires trop incertaines et qui priorise les trajectoires à haute conviction.

Le mécanisme qui retourne l'incertitude de prédiction dans le modèle de transition peut être implémenté de diverses façons, comme des approches Bayésiennes ou des ensembles de réseaux de neurones (Lakshminarayanan *et al.*, 2017). La méthode que nous avons utilisée n'entre techniquement dans aucune des deux catégories, mais pourrait être interprétée comme une implémentation simplifiée d'un processus gaussien. Le succès d'une solution basée sur l'approche par conviction dépend beaucoup de la pertinence sémantique du mécanisme d'encodage. Plus précisément, la pertinence sémantique se résume par deux concepts :

— il ne devrait pas y avoir de distinctions pas nécessaires, dans l'espace d'état interne (z_t), entre des observations sémantiquement équivalentes ;
— il devrait y avoir suffisamment de distinctions, dans l'espace d'état interne (z_t), entre des observations sémantiquement distinctes.

Des observations sont considérées comme sémantiquement équivalentes si leur distinction n'est pas nécessaire afin de résoudre le problème. Des observations sont considérées comme sémantiquement distinctes si leur différentiation est nécessaire afin de résoudre le problème. Concrètement, dans Minigrid, par exemple, il est important de distinguer les portes ouvertes des portes fermées. Ainsi, l'encodeur devrait transformer l'état s_t en un état z_t permettant de les distinguer. À l'inverse, il n'est pas important de distinguer les différentes couleurs des portes, puisqu'elles n'ont aucun impact sur la résolution du problème. Ainsi, l'espace d'état interne devrait représenter les portes ouvertes avec le même vecteur (ou code) indépendamment de leur couleur.

Si on introduit des distinctions non nécessaires entre des observations sémantiquement équivalentes, par exemple en ayant une représentation distincte de z_t pour chaque couleur de porte, on gonfle artificiellement l'espace d'état interne. Ceci augmentera la complexité échantillonnale de la solution, puisque le modèle devra avoir observé chacune de ces possibilités distinctes (par exemple, couleurs distinctes) afin de pouvoir faire des prédictions exactes dans chacun de ces cas. À l'inverse, si on ignore des distinctions né-

cessaires entre des observations sémantiquement distinctes, l'algorithme ne sera pas capable de résoudre le problème de façon fiable.

L'implémentation de la fonction d'encodage varie donc d'un domaine de problème à l'autre, puisque la pertinence sémantique des observations varie d'un domaine de problème à l'autre. Voir les sections 4.4.1 et 4.4.2 pour des détails sur la fonction d'encodage interne utilisée pour Minigrid et Sokoban respectivement. Ceci représente le désavantage principal de l'approche par conviction : elle nécessite un certain effort d'ingénierie afin de bien définir la fonction d'encodage.

Il faut noter que ce module d'encodage ne peut pas être appris par apprentissage supervisé à partir d'un ensemble de données incomplet (ce qu'on présume être le cas dans l'apprentissage par démonstrations). Si on effectue une annotation qui associe les transitions (s_t, a_t) à leurs représentations internes désirées (z_t), l'encodeur souffrira du même problème d'incertitude que le modèle de transition. En effet, si on suppose une nouvelle transition (s^*, a^*) suffisamment différente des données d'entraînement (par exemple, une interaction invalide qu'un démonstrateur n'effectuerait pas), et qu'on la fournit à l'encodeur, nous ne pouvons avoir aucune garantie que l'état interne z^* prédit par celui-ci soit valide (pour la même raison que rien ne garantit la prédiction du modèle de transition). La transformation faite par l'encodeur sera arbitraire et possiblement incorrecte. Par exemple, le vecteur z^* pourrait être égal au vecteur z_t d'un état qui a été observé pendant l'entraînement, résultant en une fausse notion de certitude.

D'autre part, l'apprentissage non supervisé ne mène pas nécessairement à une représentation interne qui soit sémantiquement pertinente. Il est très difficile de s'assurer que cette apprentissage non supervisé soit donc utile pour dériver une bonne notion d'incertitude. Il serait peut-être possible d'avoir une boucle extérieure d'optimisation qui guide cet apprentissage non supervisé de façon à aligner la représentation interne et la pertinence sémantique de celles-ci. Cette avenue de recherche est laissée à de futurs travaux : dans ce mémoire, nous nous contentons d'une fonction d'encodage construite manuellement, équivalente à une forme de pré-traitement de données.

Une fois les modèles de transition et de valeur entraînés, il est possible de résoudre des problèmes avec l'aide d'un algorithme de planification. Dans nos expériences, nous utiliserons deux algorithmes : *IDA** (avec une fonction heuristique apprise) (Korf, 1985) et *Monte-Carlo Tree Search* (Coulom, 2006). Dans les deux cas, nous partons d'un état s_t de l'environnement, et l'algorithme cherche itérativement la séquence d'actions

qui mène à une valeur prédite finale s_{t+K} qui optimise notre métrique, qu'on appelle la *conviction*.

En effet, un élément clé de notre solution proposée est que nous utilisons l'estimation d'incertitude lorsque nous planifions. D'une part, au lieu de seulement optimiser la valeur attendue de l'état final prédit, nous optimisons plutôt la valeur de *conviction* (ψ) prédite pour cet état (voir équation (4.4)). De plus, nous élaguons une partie de l'espace de recherche lorsqu'une paire état-action particulière est associée à une incertitude qui surpasse un seuil prédéterminé. Ce seuil est un hyperparamètre identifié empiriquement, et peut varier d'un problème à l'autre.

La raison initiale pour élaguer un sous-ensemble du graphe lorsqu'on s'aperçoit qu'il est trop incertain était seulement à des fins d'optimisation de coût de calcul. Toutefois, les expériences avec l'approche hybride ont démontré un avantage additionnel pour ce mécanisme : il réduit l'espace de recherche. Ceci permet donc de trouver une solution avec moins d'expansions de noeuds, ce qui a un impact sur le taux de succès.

4.4.1 MCTS avec conviction

IDA* n'est pas approprié pour un environnement partiellement observable, car ce dernier nécessite en tout moment un but vers lequel optimiser la trajectoire. Puisque ce but est inobservable dans la majorité des interactions, IDA* est difficilement applicable. Pour cette raison, nous utilisons plutôt l'algorithme *Monte Carlo Tree Search* pour les expériences avec Minigrid. En effet, *Monte Carlo Tree Search* cherche plutôt à maximiser une valeur attendue, et l'utilisation de la récompense intermédiaire qui favorise l'exploration s'aligne bien avec cet algorithme.

Afin de gérer l'observabilité partielle de Minigrid, nous utilisons une grille en mémoire interne qui permet au modèle de se rappeler des cellules qui ont déjà été observées. Nous structurons cette mémoire de sorte que les cellules non observées ont une valeur spéciale, dans le 7^e canal de leur *one-hot encoding*, les identifiant comme ayant un contenu présentement inconnu. Ceci nous permet, lors de la planification, de déterminer si les trajectoires proposées révèlent de nouvelles cellules, chose qui est encouragée via une récompense intermédiaire. De plus, ceci permet au modèle de connaître en tout temps sa position actuelle dans la grille globale (par opposition à la grille de 7×7 reçue comme observation), ce qui est également nécessaire pour le processus de planification.

Quelques adaptations ont été réalisées à l'algorithme MCTS. Dans la fonction `Expansion MCTS` (algorithme

4, ligne 7) et la fonction `Simulation MCTS` (algorithme 5, ligne 8), l'algorithme original doit prédire l'impact d'appliquer l'action a dans l'état s, pour obtenir s' et une récompense attendue r. Notre fonction de prédiction fait appel à la notion d'incertitude, tel que décrit dans le pseudo-code de l'algorithme 12.

Algorithme 10 : Expansion MCTS (modifié)

Entrée : v_t : le nœud actuel

ϕ : la valeur de la récompense pour atteindre l'objectif

τ : le seuil d'incertitude

Résultat : $enfant\_t$, un enfant aléatoire de v_t (ou v_t si terminal)

```
1  Début
2      si v_t est terminal alors
3          retourner v_t
4      fin
5      si v_t n'est pas entièrement exploré alors
           // On crée un enfant pour chaque action possible
6          pour chaque action a non explorée à partir de v_t faire
7              s', r, σ ← Prédiction MCTS(v_t, a, φ, τ)
8              si σ <= τ alors
9                  enfant ← Créer un nouveau nœud pour s'
10                 Ajouter enfant comme enfant de v_t
11             fin
12             Marquer a comme explorée pour v_t
13         fin
14         Marquer v_t comme entièrement exploré
15     fin
16     enfant_t ← un enfant aléatoire de v_t
17 Fin
```

Algorithme 11 : Simulation MCTS (modifié)

Entrée : v_t : le nœud initial

P : la profondeur maximale de simulation

ϕ : la valeur de la récompense pour atteindre l'objectif

τ : le seuil d'incertitude

Résultat : $conviction$, estimation de la conviction du nœud

Début

  $v \leftarrow v_t$

  $profondeur \leftarrow 0$

  $incertitudes \leftarrow \{\}$

  `// On simule une séquence d'actions et la récompense cumulative`

  Tant que $profondeur < P$ *et* v *n'est pas terminal* **faire**

    $a \leftarrow$ choisir une action aléatoire pour v

    $noeud\_suivant, r, \sigma_i \leftarrow$ Prédiction MCTS(v, a, ϕ, τ)

    si $\sigma_i <= \tau$ **alors**

      $incertitudes \leftarrow incertitudes \cup \sigma_i$

      $v \leftarrow noeud\_suivant$

      $profondeur \leftarrow profondeur + 1$

    fin

  fin

  $conviction \leftarrow$ Valeur($noeud\_suivant$) / ($\sum$ incertitudes$+\epsilon$)

Fin

L'algorithme 12 utilise l'encodeur (*encodeur()*), le modèle de transition (*transition()*) et le modèle de valeur (*valeur()*) pour faire des prédictions de transition d'état et de récompense comme dans l'algorithme MCTS régulier, mais elle renvoie également une notion d'incertitude associée à cette prédiction. Pour la récompense prédite, si le modèle de valeur détecte que le but a été atteint, on fixe la valeur à ϕ, un paramètre prédéterminé de valeur grandement supérieure à 1 (dans nos expériences, nous avons déterminé empiriquement la valeur 25).

L'algorithme 12 fait aussi référence aux fonctions *position_courante()* et *incertitude()*. *Incertitude()* se réfère au processus de calcul de la distance entre z_t et le vecteur le plus proche dans le tableau d'incertitude. *Position_courante()* se réfère à une simple fonction utilitaire qui extrait la position actuelle de l'agent dans la grille (à partir du vecteur s_t). De plus, la fonction `Prédiction MCTS` inclut une logique relative à la récompense intermédiaire qui encourage l'exploration. Chaque fois que la transition prédite amène l'agent

Algorithme 12 : Prédiction MCTS

Entrée : v_t : le nœud initial

a : l'action proposée

ϕ : la valeur de la récompense pour atteindre l'objectif

τ : le seuil d'incertitude

Résultat : (s' l'état prédit ; r la récompense prédite ; σ l'incertitude de la prédiction)

Début

$z \leftarrow encodeur(v_t)$

$prev\_pos \leftarrow position\_courante(v_t)$ `// Position de l'agent dans la grille`

$\delta s' \leftarrow transition(z, a)$

$\sigma \leftarrow incertitude(z, a)$

$\hat{v} \leftarrow sigmoid(valeur(v_t))$ `// Prédiction binaire: but atteint ou non`

$s' \leftarrow$ application de la transition prédite $\delta s'$ à l'état du noeud v_t

$exploration \leftarrow Faux$

si $\hat{v} > 0.5$ **alors**

  $r \leftarrow \phi$

fin

sinon

  $cellule \leftarrow cellule\_courante(s')$ `// État prédit de la case où se trouvera l'agent`

  `// Si on explore une nouvelle cellule`

  si $cellule = \emptyset$ *et* $position\_courante(s') \neq prev\_pos$ **alors**

    $r \leftarrow 1$

    $exploration \leftarrow Vrai$

    $\sigma \leftarrow 0$

  fin

fin

si $\sigma \geq \tau$ *et exploration est Faux* **alors**

  retourner $etat(v_t), 0, \sigma$

fin

retourner s', r, σ

Fin

sur une cellule auparavant inexplorée, nous fixons la récompense à 1, plutôt que zéro. Il est à noter que cette partie du code est spécifique aux environnements partiellement observables.

Il ne faut pas confondre UCT (*Upper Confidence bound for Trees*) avec ce *reward shaping* que nous effectuons. UCT est typiquement utilisé avec MCTS. En réalité, ici, nous n'utilisons pas UCT, mais remplaçons essentiellement cette formule par notre notion de conviction. Il s'agit d'une des modifications principales que nous effectuons dans MCTS avec conviction. L'UCT favorise les actions qui maximisent la valeur estimée (exploitation) tout en priorisant les états moins visités (exploration). Dans l'approche par conviction, nous ne voulons pas explorer le plus d'états non-visités possibles au sens général, puisque ceci impliquerait visiter des états incertains (et donc pour lesquels on ne peut faire de prédictions fiables).

Toutefois, puisqu'il s'agit d'un environnement partiellement observable, et que pour se diriger vers le but il faut d'abord le voir, nous encourageons le fait de révéler des cellules cachées tant et aussi longtemps que le but n'est pas visible. Il faut noter la distinction entre visiter un nouvel état et révéler une cellule cachée. Chaque position et orientation différente de l'agent est un état distinct. Ce n'est pas que nous encourageons l'agent à visiter des états préalablement non-visités. C'est que nous encourageons le fait d'amener à sa fenêtre de visiblité le plus de cellules préalablement non-observées. Aussi, il faut noter que ce *reward shaping* n'est pertinent que parce que l'environnement est partiellement observable. Ainsi, le *reward shaping* que nous faisons n'a pas le même but qu'UCT.

Dans la fonction `Expansion MCTS (modifié)` (algorithme 10), l'incertitude retournée par `Prédiction MCTS` est utilisée pour déterminer si nous pouvons continuer à étendre l'arbre dans cette direction ou non. Donc, la différence entre la fonction originale et la nôtre est que les lignes 8 et 9 qui consistent à créer et ajouter un nouvel enfant ne sont pas exécutées si $\sigma > \tau$ (σ étant la valeur d'incertitude, τ étant le paramètre de seuil d'incertitude). Autrement dit, si l'incertitude est trop élevée pour l'action proposée à partir du nœud donné, nous élaguons l'ensemble du sous-arbre à partir de cette arête. Il est inutile de continuer à explorer ce sous-arbre si nous avons peu de confiance dans les prédictions faites sur sa racine (l'erreur de prédiction étant cumulative).

Dans la fonction `Simulation MCTS (modifié)` (algorithme 11), nous utilisons cette incertitude de deux manières. Premièrement, nous ignorons une prédiction de transition dans la boucle de simulation si son incertitude est trop élevée. C'est-à-dire qu'elle ne compte pas comme partie de l'estimation finale de la

valeur ou de la séquence de transition d'état. Donc, les lignes 9 à 12 ne sont pas exécutées si $\sigma > \tau$. Deuxièmement, $valeur\_noeud$ dans le pseudo-code devient $conviction$ et correspond à ce que nous appelons la conviction dans ce mémoire : le rapport de la valeur attendue de l'état final sur l'incertitude cumulative des prédictions qui nous ont conduit à cette action. Valeur($noeud\_suivant$) fait référence à la valeur estimée pour l'état du $noeud\_suivant$ via notre modèle de valeur.

Dans Minigrid, notre fonction d'encodage extrait le contenu de la cellule immédiatement devant l'agent. Chaque cellule dans la grille est un vecteur encodé en *one-hot* à six dimensions. Cette information est concaténée à l'encodage en *one-hot* à quatre dimensions de la direction actuelle de l'agent. Le vecteur final z a donc une dimensionnalité totale de dix. Ce sont les seules informations nécessaires pour faire une prédiction de transition correcte pour toutes les actions disponibles dans Minigrid.

Il faut également noter que dans le prétraitement qui convertit l'observation brute de Minigrid en l'état encodé en *one-hot*, les informations sur la couleur de la porte sont écartées. L'observation brute de Minigrid contient, pour chaque cellule de la grille, trois canaux : le type d'objet, sa couleur, et son état. Nous ignorons donc simplement le deuxième canal de par son inutilité dans ce problème. Cela devrait être considéré comme faisant partie de l'alignement sémantique codé en dur requis pour que l'encodeur fonctionne efficacement pour ce problème.

Pendant la planification, il y a une récompense intermédiaire qui est utilisée pour encourager l'exploration. Ceci est nécessaire, car l'environnement n'est que partiellement observable, et donc il n'est pas possible de planifier son mouvement jusqu'à la case cible tant que celle-ci n'est pas visible. Cette récompense intermédiaire, qui est implémentée dans la formule de calcul de la récompense dans l'algorithme de planification, ajoute 1 à la récompense lorsqu'une nouvelle cellule est explorée. En contraste, atteindre la case cible ajoute 25 à la récompense retournée. Ainsi, l'algorithme de planification va sélectionner et exécuter les séquences d'actions qui maximisent le nombre de cellules révélées, tant et aussi longtemps qu'aucune case cible n'est visible. Lorsque la case cible est visible, toutefois, la récompense associée aux trajectoires qui atteignent cette cible sera grandement supérieure à celles qui ne font qu'explorer de nouvelles cellules.

Il faut noter qu'en théorie, ceci implique qu'une trajectoire qui explore d'abord quelques nouvelles cellules avant d'aller atteindre la case cible sera préférée à une trajectoire qui va directement à la cible. En pratique, toutefois, ce phénomène est négligeable pour les niveaux que nous utilisons. Aucune situation n'a été ob-

servée dans laquelle le fait d'allonger la trajectoire finale en ajoutant de l'exploration mène à un échec plutôt qu'un succès. La raison semble être que dans la plupart des cas, une fois que la cible est visible, nous avons observé également l'entièreté de la pièce.

Il y a rarement des cellules non-observées derrière le but (c'est presque tout le temps vrai pour MiniGrid-MultiRoom-N6-v0, un peu moins pour MiniGrid-FourRooms-v0). Même s'il y en a, les pièces ne sont simplement pas assez grandes pour que l'agent ait besoin de se déplacer un grand nombre de pas derrière le but pour terminer l'exploration. À la limite, il peut arriver dans `MiniGrid-FourRooms-v0` que l'agent dépasse le but d'un ou deux pas, puis fait demi-tour et revient vers le but, n'ayant aucun impact significatif sur la performance totale. Si, toutefois, on désire remédier à cet effet, il suffit de modifier la formule de la conviction en ajoutant un nouveau terme qui favorise les trajectoires plus courtes.

4.4.2 IDA* avec conviction

À partir de l'algorithme IDA* classique (algorithme 7), nous développons la variante pour supporter la notion de conviction. La différence se trouve dans l'algorithme 9 qui gère l'expansion des sommets voisins. Afin d'explorer les sommets voisins à partir d'un certain sommet, il faut effectuer une prédiction de transition à partir de l'état du sommet actuel, et de l'action proposée (à la ligne 5 de l'algorithme 9). Dans l'algorithme avec conviction, cette prédiction retourne une valeur d'incertitude, qui est ensuite utilisée de deux manières. D'abord, la ligne 6 de l'algorithme `Expansion IDA*` (algorithme 9) n'est pas exécutée si $\sigma > \tau$ (σ étant la valeur d'incertitude, τ étant le paramètre de seuil d'incertitude). Cette logique correspond à enlever l'arête qui correspond à la transition prédite dont l'incertitude est trop élevée. Autrement dit, on élague le sous-arbre correspondant aux trajectoires possibles passant par cette transition. Voir l'algorithme 13 pour la version modifiée.

Algorithme 13 : Expansion IDA* (modifié)

Entrée : $sommetInitial$: le sommet initial

Résultat : $voisinsValides$, la liste de voisins valides

Début
  $voisinsValides \leftarrow voisins(sommetInitial)$
  `// Si nous n'avons jamais fait l'expansion de ce sommet:`
  si *voisinsValides est vide* **alors**
    $tmpVoisinsValides \leftarrow \{\}$
    pour $action$ *parmi les actions possibles depuis* $sommetInitial$ **faire**
      $voisin, \hat{v}, \sigma \leftarrow$Prédiction IDA*$(action, sommetInitial)$
      si $\sigma <= \tau$ **alors**
        $conviction = (100 - \hat{v})/(incertitude(sommetInitial) + \sigma + \epsilon)$
        ajouter $(voisin, conviction)$ à $tmpVoisinsValides$
      fin
    fin
    $voisinsValides \leftarrow$ Trier en order décroissant de conviction $tmpVoisinsValides$
  fin
Fin

Algorithme 14 : Prédiction IDA*

Entrée : a : l'action proposée

v_t : le sommet initial

Résultat : (s' l'état prédit ; $\hat{v}$ l'estimation heuristique ; σ l'incertitude de la prédiction)

Début
  $z \leftarrow encodeur(v_t)$
  $\delta s' \leftarrow transition(z, a)$
  $\sigma \leftarrow incertitude(z, a)$
  $\hat{v} \leftarrow valeur(v_t)$ `// Prédiction: nombre de pas restants`
  $s' \leftarrow$ application de la transition prédite $\delta s'$ à l'état du noeud v_t
  retourner $s', \hat{v}, \sigma$
Fin

De plus, dans notre implémentation, la fonction heuristique d'IDA* est apprise à partir d'un modèle entraîné

sur les démonstrations. Nous avons un réseau de neurones *feed-forward* de 5 couches ayant 1 024 neurones chacune sauf pour la dernière, qui est de 512 neurones, entraîné de façon supervisée sur les démonstrations. Celui-ci est entraîné sur la même portion de données d'entraînement que le modèle de transition. On lui fournit en entrée un état sélectionné aléatoirement dans une démonstration, et pour celui-ci il doit prédire le nombre de pas qui restent dans cette démonstration à partir de cet état. Il s'agit donc d'un modèle de régression, entraîné en minimisant l'erreur quadratique moyenne entre les prédictions et les vrais nombres de pas restants.

Puisque la valeur d'heuristique dans IDA* correspond à une estimation du nombres de pas restants pour résoudre le problème, plutôt qu'une récompense attendue, il faut favoriser les valeurs plus basses, à l'inverse des récompenses. C'est pourquoi, dans notre calcul de conviction pour IDA*, nous utilisons une valeur intermédiaire $V = 100 - h$, pour convertir ce nombre de pas restants en une valeur qui augmente au fur et à mesure qu'on s'approche du but. La constante 100 a été déterminée empiriquement en observant la distribution des valeurs du modèle d'heuristique IDA*. La valeur retournée par le modèle ne dépassait pas 100 dans les données d'entraînement, justifiant ainsi ce choix. La valeur de conviction est ensuite utilisée pour prioriser l'ordre d'exploration des sommets voisins. Nous commençons par le sommet ayant la conviction la plus élevée.

Pour Sokoban, la fonction d'encodage utilisée examine d'abord le contenu des deux cellules adjacentes devant, à droite, à gauche et derrière l'agent. Nous avons remarqué qu'il était possible de compresser davantage les informations contenues dans chaque groupe de deux cellules, et donc d'améliorer l'efficacité de l'échantillonnage. Spécifiquement, deux cellules peuvent être compressées en dix cas spécifiques :

1. vide-quelque chose : la cellule immédiatement adjacente est une cellule vide (régulière ou cible), donc le contenu de la cellule secondairement adjacente n'est pas important.
2. mur-quelque chose : la cellule immédiatement adjacente est un mur, donc le contenu de la cellule secondairement adjacente n'est pas important.
3. boîte-boîte/mur : la cellule immédiatement adjacente est une boîte, et la cellule secondairement adjacente contient un obstacle qui nous empêche de bouger cette boîte, comme une autre boîte ou un mur.
4. boîte-vide : la cellule immédiatement adjacente est une boîte, et la cellule secondairement adjacente est une cellule vide.

5. boîte-coin : la cellule immédiatement adjacente est une boîte, et la cellule secondairement adjacente correspond à un coin. Un coin est défini comme une cellule vide non-cible entourée d'un obstacle (boîte ou mur) derrière elle, et d'au moins un obstacle de chaque côté. Cette configuration est importante, car pousser une boîte dedans mène instantanément à une situation de blocage.
6. boîte-cible : la cellule immédiatement adjacente est une boîte, et la cellule secondairement adjacente est une cellule cible.
7. cible+boîte-boîte/mur : la cellule immédiatement adjacente est une boîte sur une cellule cible, et la cellule secondairement adjacente est un obstacle.
8. cible+boîte-vide : la cellule immédiatement adjacente est une boîte sur une cellule cible, et la cellule secondairement adjacente est une cellule vide non-cible.
9. cible+boîte-cible : la cellule immédiatement adjacente est une boîte sur une cellule cible, et la cellule secondairement adjacente est une cellule cible vide.
10. cible+boîte-coin : la cellule immédiatement adjacente est une boîte sur une cellule cible, et la cellule secondairement adjacente est un coin.

4.5 Approche hybride

L'approche hybride, dont l'architecture est présente à la figure 4.5, vise à régler le problème d'incertitude non pas en la quantifiant, mais plutôt en l'éliminant. Ceci est fait en incluant des explorations aléatoires (ou même, des démonstrations délibérément incorrectes) dans le but d'obtenir une couverture suffisamment complète des transitions possibles. Ainsi, l'algorithme de planification n'aura pas à se soucier de la possibilité de prédictions potentiellement extrapolatives au niveau du modèle de transition. Dans l'approche hybride, l'algorithme de planification n'est aucunement modifié. De plus, l'architecture neuronale ne contient pas de module permettant de mesurer l'incertitude.

Ceci comporte un avantage majeur sur l'approche par conviction. D'abord, l'algorithme est plus simple que ce dernier. De plus, il n'est pas nécessaire d'implémenter un prétraitement des données spécifique au problème, dans l'encodeur, rendant cette approche beaucoup plus flexible et générale. En effet, cette approche a été proposée principalement dans le but d'alléger la contrainte d'effort d'ingénierie associée au mécanisme d'encodage présent dans l'approche par conviction. L'approche hybride ne nécessite plus d'encodage sémantiquement pertinent, elle nécessite seulement un modèle de transition capable de faire ses prédictions, soit en prenant en entrée directement les observations, ou apprenant par soi-même son espace latent

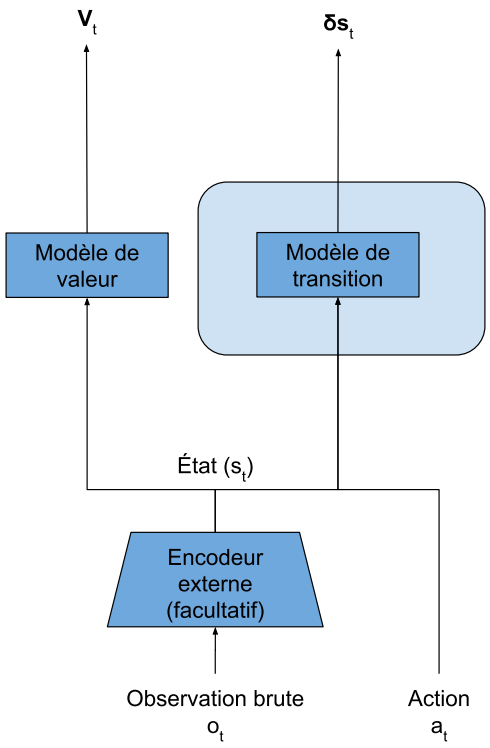

Figure 4.5 Architecture du *HybridPlanner*.

d'encodage (sans supervision requise).

Toutefois, il faut noter une conséquence importante de l'ablation du mécanisme d'incertitude dans l'approche hybride. En effet, considérons l'espace total de trajectoires possibles dans lequel la recherche s'effectue. Il s'agit d'un arbre dans lequel chaque noeud est un état, duquel découlent toutes les actions possibles (les arêtes). Lorsque l'approche par conviction apprend qu'une certaine action a_i n'a jamais été observée dans un état s_j, il s'en suit automatiquement que l'arête a_i du noeud s_j peut être enlevé, réduisant ainsi l'espace de recherche (parfois considérablement).

Par conséquent, l'espace de recherche de l'approche par conviction est plus petit que l'espace de recherche de l'approche hybride (qui considère, au contraire, toutes les actions possibles puisque son modèle est capable de les prédire). On peut donc comprendre pourquoi la recherche de l'approche hybride est moins efficace que celle de l'approche par conviction, résultant en une moins bonne performance que celle-ci (tout autres choses étant égales).

L'astuce principale de l'approche hybride consiste seulement en la composition particulière de ses données d'entraînement. Plutôt que d'entraîner l'architecture uniquement sur des données de démonstrations (comme dans l'approche naïve, voir section 5.1), ou que d'entraîner l'architecture uniquement sur des interactions aléatoires (comme dans l'apprentissage par renforcement avec modèle traditionnelle), on entraîne sur un mélange de ces deux sources de données. Ceci devrait suffire, en principe, pour faire en sorte que le modèle de transition puisse prédire le résultat de toutes les interactions requises.

En résumé, nous proposons deux stratégies pour gérer le problème des récompenses éparses. Les deux stratégies se basent sur l'idée de combiner l'apprentissage par renforcement avec modèle et l'apprentissage à partir de démonstrations. La combinaison naïve de ces deux techniques ne fonctionne pas bien, car les démonstrations ne couvrent normalement qu'un sous-ensemble des interactions possibles, menant à un modèle de transition incomplet.

Les deux stratégies proposées sont des façons différentes de résoudre ce problème du modèle de transition incomplet. L'approche par conviction mesure l'incertitude liée à chaque prédiction de transition, et utilise cette information lors de la planification. L'approche hybride tente plutôt d'atténuer ce problème en mélangeant aux données d'entraînement des interactions aléatoires.

CHAPITRE 5
EXPÉRIENCES ET RÉSULTATS

Ce chapitre présente le protocole de nos expériences ainsi que les résultats empiriques. Dans la section 5.1, nous expliquons les choix d'algorithmes utilisés dans les expériences ainsi que la procédure d'évaluation de ceux-ci. Dans la section 5.2, nous détaillons les différences entre les algorithmes, ainsi que certains choix d'hyperparamètres. Finalement, dans la section 5.3, nous présentons les résultats de ces expériences.

5.1 Expériences

Nous proposons deux séries d'expériences. D'abord, celles sur l'environnement virtuel Minigrid (voir section 4.1.1) nous ont permis d'obtenir une estimation initiale des performances relatives de différentes approches, et de mieux en comprendre les fonctionnements. Ensuite, en se basant sur ces résultats, nous avons fait le choix de l'environnement Sokoban (voir section 4.1.2) pour une deuxième série d'expériences, en y ajoutant deux nouveaux algorithmes comparatifs (WMG et l'approche hybride). Cette deuxième série d'expériences nous a permis d'obtenir des résultats comparatifs plus clairs et prononcés entre les différentes approches.

La procédure d'évaluation consiste à entraîner plusieurs instances différentes des modèles de transition, de valeur et de politique associés à chaque algorithme de comparaison, sur différentes quantités d'interactions avec l'environnement. Lorsque nous parlons d'interaction, nous faisons référence à une étape dans laquelle l'agent (ou le démonstrateur) applique une action et reçoit une récompense, effectuant ainsi une transition d'état. Nous évaluons ensuite chacune de ces instances de modèle sur 100 instances de problèmes provenant des interactions de test. Les instances de test sont des instances de problèmes qui ont été générées postérieurement à la phase d'entraînement, avec des valeurs distinctes de *seed* aléatoire. Ces instances de problème n'ont donc jamais été observées par les agents pendant leur phase d'entraînement. Ceci nous permet de dresser un portrait de la performance (en termes de taux de solution d'instances de problèmes) en fonction du nombre d'interactions d'entraînement utilisées. Nous obtenons donc ainsi un portrait de l'efficacité échantillonnale de chaque méthode.

5.1.1 Expériences sur Minigrid

Puisque nous combinons les approches d'apprentissage par renforcement avec modèle et l'apprentissage par démonstrations, il est naturel de comparer à ces méthodes. La première série d'expériences a été faite sur Minigrid, visant à comparer notre approche par conviction (voir section 4.4) avec les approches traditionnelles d'apprentissage par renforcement avec modèle et d'apprentissage par démonstrations individuellement. Ainsi, l'algorithme *ModelBasedRL-MCTS* met en œuvre l'apprentissage par renforcement avec modèle, avec exploration aléatoire et utilisant *Monte-Carlo Tree Search* classique.

Pour notre référence en apprentissage par démonstrations, nous utilisons le *Deep Q-learning from Demonstrations* (DQfD), car c'est l'état de l'art dans les environnements discrets. Nous réalisons trois expériences différentes à partir de cet algorithme : *DQfD-PreTraining*, *DQfD-Hybrid* et *DQfD-Model-Free*. *DQfD-PreTraining* consiste uniquement en la phase de pré-entraînement à partir de démonstrations. Nous ne réalisons aucune phase d'exploration aléatoire, et en tant que tel, la politique est apprise strictement à partir de l'entraînement supervisé sur des démonstrations d'experts. *DQfD-Model-Free* utilise l'algorithme DQfD sans aucun pré-entraînement sur les démonstrations, il est donc essentiellement le même que l'apprentissage profond *Q-learning* standard. *DQfD-Hybrid* implémente les deux aspects de DQfD.

Nous effectuons également une étude d'ablation pour montrer que la combinaison naïve de l'apprentissage par renforcement avec modèle et l'apprentissage par démonstrations est inefficace. *NaivePlanner-MCTS* a exactement la même architecture que *ConvictionPlanner-MCTS*, entraîné sur des démonstrations comme ce dernier. Toutefois, contrairement à ce dernier, l'approche naïve optimise uniquement la valeur estimée (plutôt que la conviction) et elle n'élague pas les sous-arbres en fonction de l'incertitude.

Cette première série d'expériences nous a permis de mieux comprendre les impacts de différentes procédures d'entraînement et de choix architecturaux (voir chapitre suivant). Toutefois, nous avons réalisé que l'environnement Minigrid était un peu trop facile à résoudre pour vraiment mettre en valeur notre approche par conviction. C'est pourquoi nous avons ensuite fait une deuxième série d'expériences sur Sokoban. De plus, dans cette deuxième série d'expériences, nous avons ajouté deux nouveaux candidats. D'abord, les *Working Memory Graphs* obtiennent de bons résultats sur Sokoban, donc nous avons cru bon de les utiliser dans nos comparaisons. De plus, nous y avons ajouté une nouvelle stratégie, notre approche hybride.

5.1.2 Expériences sur Sokoban

Sokoban s'est révélé être un meilleur environnement que Minigrid pour bien mettre de l'avant les avantages de nos solutions proposées, ainsi que leurs différences en performance. D'abord, nous ré-évaluons tous les algorithmes utilisés dans la première série d'expériences sur Minigrid, mais en utilisant l'algorithme de planification *Iterative Deepening A** plutôt que *Monte-Carlo Tree Search*, lorsque applicable.

Ensuite, en ajoutant notre approche hybride parmi les algorithmes de comparaison, nous évaluons ce qui se passe si nous entraînons une approche d'apprentissage par renforcement avec modèle sur un mélange de démonstrations et d'exploration aléatoire. On peut supposer que cela éliminerait le besoin de gérer l'incertitude dans le modèle de transition, puisque les explorations aléatoires couvriraient l'espace de transition. C'est l'expérience utilisant l'algorithme *HybridPlanner-IDA**, que nous réalisons sur l'environnement Sokoban.

Enfin, nous comparons avec les *Working Memory Graphs* (WMG), en utilisant les niveaux Boxoban de la catégorie *unfiltered* comme dans (Loynd *et al.*, 2020), au lieu de nos niveaux aléatoires habituels à trois boîtes. Voir le tableau 5.4 pour les résultats. En raison de l'efficacité d'échantillonnage inférieure de WMG, nous devons comparer notre nombre le plus élevé d'interactions sur le *ConvictionPlanner-IDA** et l'*HybridPlanner-IDA** avec les résultats de WMG à des nombres beaucoup plus élevés d'interactions. Le nombre de démonstrations nécessaires pour évaluer à ces régimes de données plus élevés est prohibitif, c'est pourquoi nous nous arrêtons à 50 000 pour nos deux algorithmes.

5.2 Détails des algorithmes comparatifs

DQfD-PreTraining : Nous utilisons le code source provenant du répertoire GitHub : `https://github.com/DPS0340/DQNDemo`. L'algorithme *DQfD-PreTraining* consiste à utiliser seulement la phase de pré-entraînement de DQfD. Cette phase utilise seulement des démonstrations, et entraîne la politique de façon supervisée. Elle correspond essentiellement à ce qu'on appelle parfois, dans la littérature, le *behavioral cloning* (Torabi *et al.*, 2018). Il faut noter que dans l'implémentation originale de DQfD, la phase de pré-entraînement fait usage des quatre termes de la fonction de coût (équation (2.4)). Toutefois, le terme L_E est le terme d'apprentissage supervisé (équation (2.8)) qui permet au modèle de produire une distribution de probabilités sur les actions. Dans ce contexte, les termes L_{1-step} et $L_{10-step}$ (équation (2.7)) servent uniquement à s'assurer que le modèle de *Q-learning* satisfasse l'équation de Bellman, le permettant d'être

utilisé comme point de départ pour la phase suivante : l'apprentissage à partir d'interactions aléatoires. Ainsi, puisque pour cet algorithme nous n'utilisons que la phase de pré-entraînement, nous ignorons les deux termes L_{1-step} et $L_{10-step}$ de la fonction de coût. Ceci est d'autant plus justifié que par le fait que nous observons une dégradation mineure de performance si on les inclut.

DQfD-Model-Free : Il utilise l'algorithme DQfD sans aucun pré-entraînement sur des démonstrations. Seulement la phase d'entraînement à partir d'exploration aléatoire est utilisée. Ainsi, il s'agit donc d'une approche typique d'apprentissage par renforcement sans modèle (le *Deep Q-learning*, pour être plus précis). Dans cette version du modèle, le terme L_E (équation (2.8)) n'est pas utilisé. Dans notre implémentation, comme dans l'implémentation originale, la valeur de n du *n-step loss* correspond à 10, ce qu'on appelle donc un *10-step Q-loss* (équation (2.7)).

DQfD-Hybrid : Il implémente les deux phases de DQfD, soit le pré-entraînement sur les démonstrations, suivi de l'exploration aléatoire en mode sans modèle. C'est donc l'utilisation normale de DQfD. Tous les termes de la fonction de coût sont utilisés.

ModelBasedRL-MCTS/IDA* : Nous avons implémenté cet algorithme d'apprentissage par renforcement avec modèle, suivant une architecture très similaire à nos solutions. En fait, l'architecture est identique à *HybridPlanner-MCTS* et *HybridPlanner-IDA**, la différence étant que les données d'entraînement sont obtenues uniquement par exploration aléatoire. Dans Sokoban, l'algorithme de planification que nous utilisons est IDA*, alors que dans Minigrid, nous utilisons MCTS. Pour IDA*, nous permettons l'expansion jusqu'à 1.5 millions de sommets. Si aucune solution n'est trouvée après ce nombre d'expansions limite, nous considérons la tentative comme un échec et passons au prochain niveau. Pour MCTS, pour permettons 2 000 simulations par feuille, et le paramètre C qui contrôle le ratio d'exploration vs exploitation est de 0.5. La profondeur maximale des séquences simulées est de 20.

NaivePlanner-MCTS/IDA* : Nous avons implémenté cet algorithme d'apprentissage par renforcement avec modèle, suivant une architecture identique à *HybridPlanner-MCTS* et *HybridPlanner-IDA**. La différence, ici, est que l'entraînement se fait uniquement sur les démonstrations. L'algorithme de planification IDA* est utilisé pour Sokoban alors que pour Minigrid, l'algorithme de planification utilisé est MCTS. Les paramètres de MCTS et IDA* sont les mêmes que pour *ModelBasedRL-MCTS/IDA**.

ConvictionPlanner-MCTS/IDA* : Ceci est notre nouvelle approche basée sur la conviction, telle que dé-

crite à la section 4.4. Dans Sokoban, l'algorithme de planification utilisé est IDA*, alors que dans Minigrid, l'algorithme de planification utilisé est MCTS. Les paramètres de MCTS et IDA* sont les mêmes que pour *ModelBasedRL-MCTS/IDA**.

HybridPlanner-IDA* : Ceci est notre approche hybride, telle que décrite à la section 4.5. L'algorithme de planification utilisé est IDA*, et les paramètres d'IDA* sont les mêmes que pour *ModelBasedRL-IDA**. Le nombre total d'interactions est divisé en interactions aléatoires et en démonstrations. On utilise un ratio de 30% interactions aléatoires, 70% étapes de démonstrations. Donc, pour 50 000 interactions, on a 15 000 interactions aléatoires et 35 000 étapes de démonstrations.

WMG : Nous avons obtenu le code directement du répertoire *GitHub* des auteurs : `https://github.com/microsoft/wmg_agent`. Avec un peu de pré-traitement des données, nous avons pu utiliser les niveaux Boxoban pour les algorithmes *WMG*, *HybridPlanner-IDA** et *ConvictionPlanner-IDA**, permettant ainsi une comparaison directe entre les trois, ainsi qu'avec les résultats présentés dans (Loynd *et al.*, 2020).

5.3 Résultats

L'algorithme par conviction obtient une performance égale ou supérieure à toutes les autres approches, sur presque toutes les quantités d'interactions d'entraînement. Une exception à cette sur-performance est en comparaison à *ModelBasedRL-MCTS* sur `MiniGrid-FourRooms-v0` pour 750, 1 500, et 6 000 interactions d'entraînement. Sur ces expériences, *ModelBasedRL-MCTS* obtient 100% alors que *ConvictionPlanner-MCTS* obtient 99%. Voir le tableau 5.1 et figure 5.1.

Dans l'environnement `MiniGrid-MultiRoom-N6-v0` (tableau 5.2, figure 5.2), *ConvictionPlanner-MCTS* performe mieux que toutes les autres approches, à l'exception de *DQfD-PreTraining*, qui égalise sa performance à 100% lorsque présenté avec 6 000 données d'entraînement ou plus. Dans l'environnement Sokoban (tableau 5.3, figure 5.3), *ConvictionPlanner-IDA** performe significativement mieux que les approches comparatives avec un taux de succès de 65%, qui a été obtenu avec 30 000 interactions pendant l'entraînement. La deuxième meilleure approche est *HybridPlanner-IDA**, qui atteint un taux de succès maximal de 38% pour le même nombre de données d'entraînement.

Sur les niveaux Boxoban (voir le tableau 5.4), on remarque que *ConvictionPlanner-IDA** obtient la performance de 43% à partir de seulement 30 000 interactions d'entraînement, alors que WMG en nécessite 4

Table 5.1 Taux de succès (%) sur `MiniGrid-FourRooms-v0`

	Nombre d'interactions d'entraînement							
Algorithme	**30**	**750**	**1.5k**	**6K**	**9K**	**15K**	**22.5K**	**30K**
DQfD-PreTraining	1	26	35	55	55	55	57	61
DQfD-Hybrid	0	16	29	65	65	67	67	68
DQfD-Model-Free	0	0	0	0	0	0	0	0
ModelBasedRL-MCTS	5	**100**	**100**	**100**	**100**	**100**	**100**	**100**
NaivePlanner-MCTS	17	20	30	32	32	34	34	34
ConvictionPlanner-MCTS	**60**	99	99	99	**100**	**100**	**100**	**100**

Table 5.2 Taux de succès (%) sur `MiniGrid-MultiRoom-N6-v0`

	Nombre d'interactions d'entraînement					
Algorithme	**30**	**750**	**1.5k**	**6K**	**9K**	**15K**
DQfD-PreTraining	0	89	96	**100**	**100**	**100**
DQfD-Hybrid	0	35	72	95	87	94
DQfD-Model-Free	0	0	0	0	0	0
ModelBasedRL-MCTS	28	67	64	64	64	64
NaivePlanner-MCTS	17	20	30	32	32	34
ConvictionPlanner-MCTS	**99**	**99**	**99**	**100**	**100**	**100**

millions pour atteindre la même performance. De plus, la performance de 49% est atteinte avec 50 000 interactions pour *ConvictionPlanner-IDA**, alors que WMG en nécessite 5 millions. On constante donc une efficacité échantillonnale environ 100 fois plus élevée avec l'algorithme *ConvictionPlanner-IDA** qu'avec WMG.

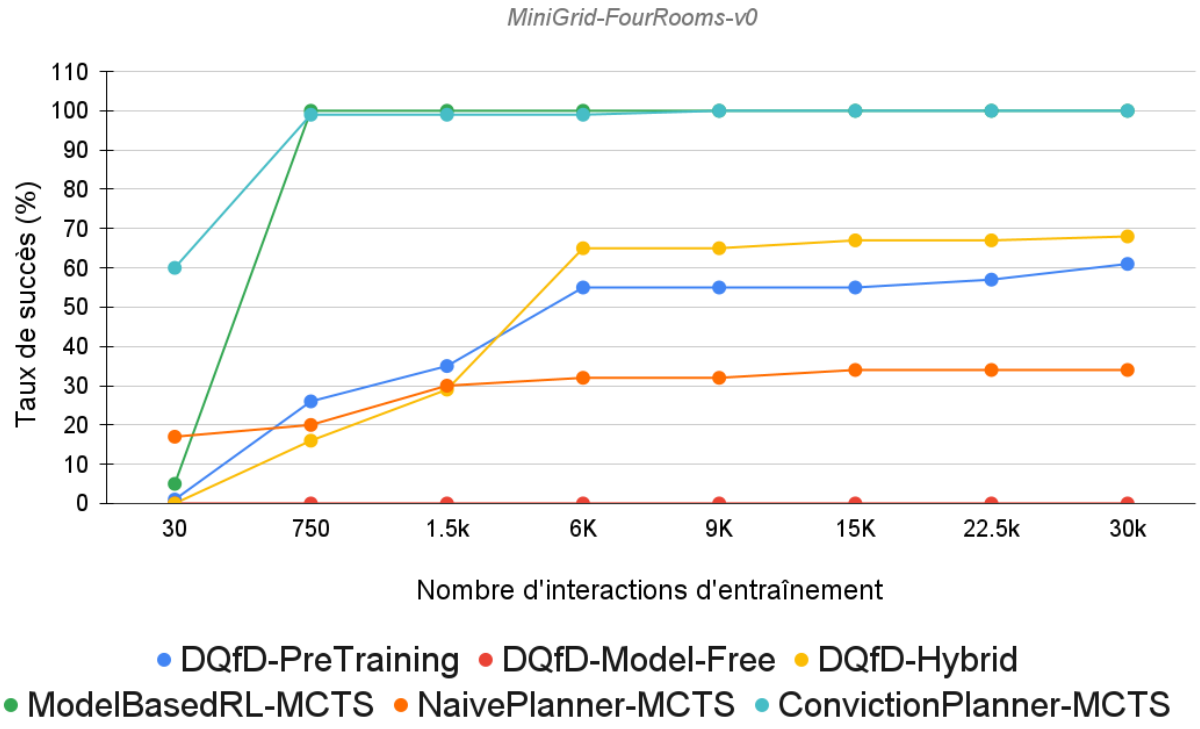

Figure 5.1 Taux de succès (en %) sur 100 instances de test du problème `MiniGrid-FourRooms-v0`, en fonction du nombre d'interactions avec l'environnement pendant l'entraînement.

Table 5.3 Taux de succès (%) sur *Sokoban*

	Nombre d'interactions d'entraînement						
Algorithme	**1K**	**1.5K**	**5K**	**10K**	**15K**	**25K**	**30K**
DQfD-PreTraining	0	0	0	0	0	0	0
DQfD-Hybrid	0	0	0	0	0	0	0
DQfD-Model-Free	0	0	0	0	0	0	0
WMG	0	0	0	0	0	0	0
ModelBasedRL-IDA*	0	0	0	0	5	5	6
NaivePlanner-IDA*	0	0	0	1	1	2	2
HybridPlanner-IDA*	1	7	26	28	31	34	38
ConvictionPlanner-IDA*	**9**	**12**	**40**	**58**	**61**	**64**	**65**

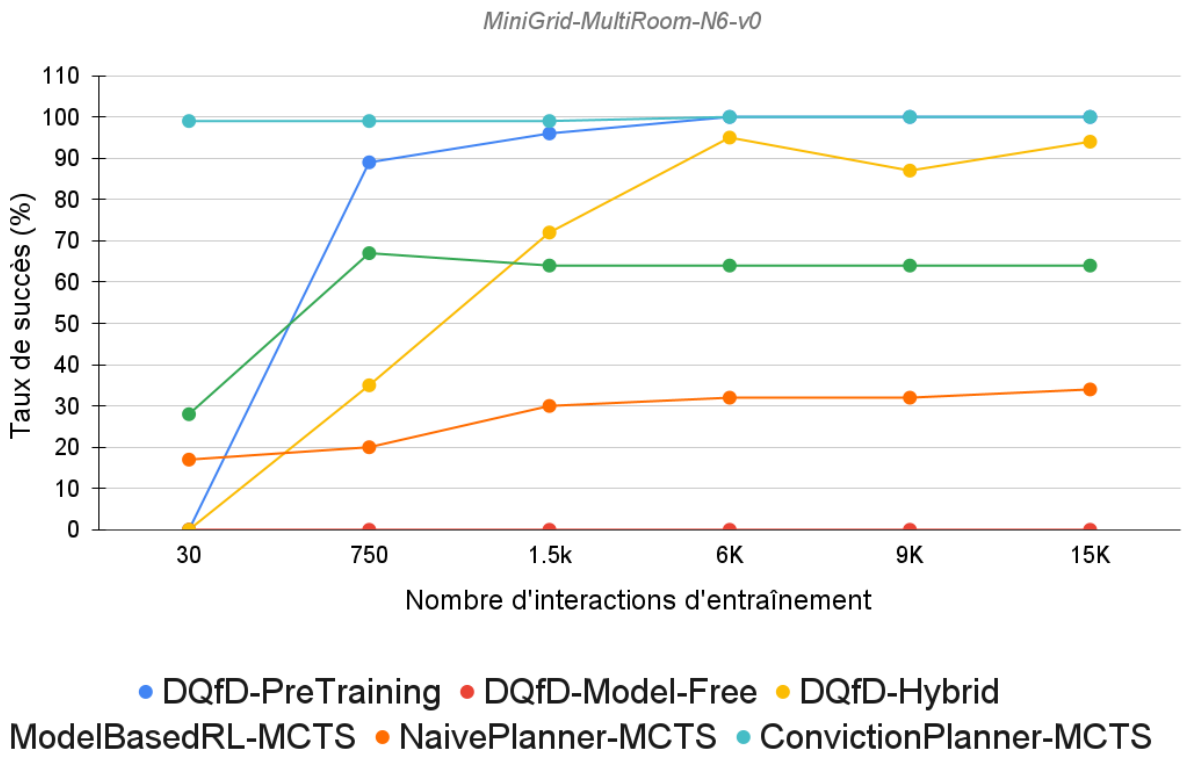

Figure 5.2 Taux de succès (en %) sur 100 instances de test du problème `MiniGrid-MultiRoom-N6-v0`, en fonction du nombre d'interactions avec l'environnement pendant l'entraînement.

Table 5.4 Taux de succès (%) sur les niveaux *Boxoban*

	Nombre d'interactions d'entraînement					
Algorithme	**30k**	**50k**	**1M**	**2.5M**	**4M**	**5M**
WMG	0	<1	10.4	33	43	49
HybridPlanner-IDA*	21	26	N/A	N/A	N/A	N/A
ConvictionPlanner-IDA*	**43**	**49**	N/A	N/A	N/A	N/A

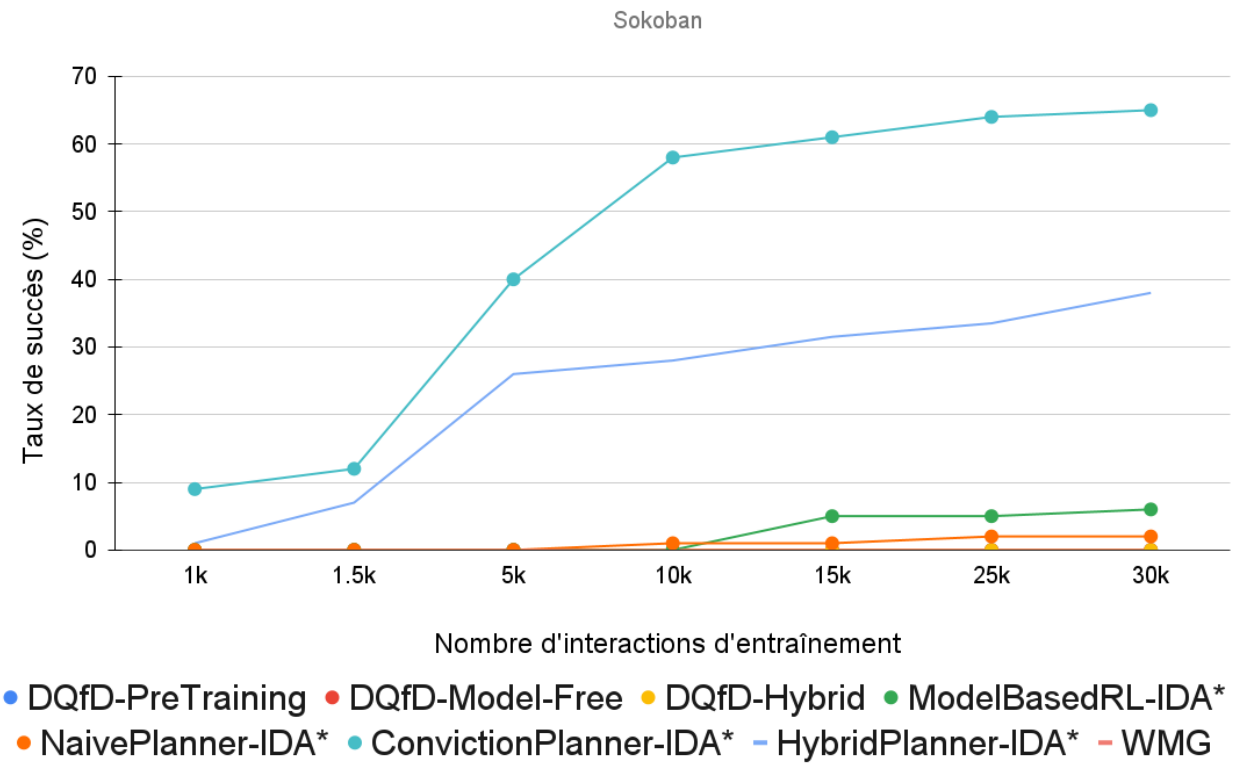

Figure 5.3 Taux de succès (en %) sur 100 instances de test du problème Sokoban, en fonction du nombre d'interactions avec l'environnement pendant l'entraînement.

CHAPITRE 6
DISCUSSION

Dans ce chapitre, nous discutons de nos interprétations des résultats empiriques présentés dans le chapitre 5. Dans la section 6.1, nous présentons un argument sur l'impact de l'utilisation de l'apprentissage par démonstrations versus l'exploration aléatoire. Dans la section 6.2, nous discutons de l'impact de la planification sur les problèmes qui dépendent plus fortement d'une séquence complexe d'interactions pour atteindre une solution.

Par la suite, dans la section 6.3, nous constatons un avantage de l'approche par conviction sur l'approche hybride, lié à l'élagage de l'arbre de recherche fait par ce premier. Dans la section 6.4, nous expliquons certaines contraintes et hypothèses faites par nos approches proposées. La section 6.5 discute de la performance relative des *Working Memory Graphs* et de notre approche par conviction. Finalement, dans la section 6.6 nous résumons les observations et présentons nos recommandations en ce qui concerne l'usage de l'approche par conviction et de l'approche hybride : pour quels types de problèmes et pour quels contextes sont-ils préférables aux autres approches considérées.

6.1 L'importance des démonstrations

DQfD-Model-Free est incapable de résoudre les problèmes que nous avons explorés, du moins pour les quantités de données qui nous intéressent. Sur toutes les tailles d'échantillons testées, l'exploration aléatoire a rarement permis de sortir de la 2ème salle dans `MiniGrid-MultiRoom-N6-v0`. Elle n'a jamais atteint la fin, donc elle n'a jamais vu de récompense non nulle. Nous observons le même problème avec l'apprentissage par renforcement avec modèle sur Sokoban. La seule raison pour laquelle l'apprentissage par renforcement avec modèle fonctionne si bien sur Minigrid est grâce aux récompenses intermédiaires qui encouragent l'exploration dans l'algorithme de planification. Ceci confirme donc l'importance d'utiliser des techniques permettant de gérer le problème des récompenses éparses, soit l'apprentissage par démonstrations et les récompenses intermédiaires dans notre cas.

6.2 L'importance de la planification

Nous croyons que, sur Sokoban, il est difficile à partir d'un état donné de prédire la meilleure action suivante sans raisonner à travers les étapes. En d'autres termes, étant donné deux grilles autrement identiques, une seule différence dans une cellule peut suffire à changer complètement la solution. C'est probablement pourquoi, sur Sokoban, les trois meilleurs algorithmes utilisent tous une planification explicite. Nous n'excluons pas, toutefois, la possibilité de l'usage de planification implicite comme avec les WMG.

Sur `MiniGrid-MultiRoom-N6-v0`, notre hypothèse est qu'une politique supervisée basée sur des démonstrations (*DQfD-PreTraining*) fonctionne relativement bien, car tout ce que l'agent doit apprendre est qu'il doit se diriger vers la porte fermée visible dans sa vue partielle puis l'ouvrir. Ainsi, il y a peu de planification requise, car cette astuce peut facilement être utilisée à partir de la plupart des observations partielles.

Il est possible de faire quelque chose d'équivalent avec `MiniGrid-MultiRoom-N6-v0`, comme par exemple de se diriger vers le trou dans le mur le plus proche pour passer à l'autre pièce, mais l'espace de chaque pièce est plus grand. En effet, dans `MiniGrid-FourRooms-v0`, beaucoup d'états observés seront des états vides qui ne contiennent aucun repère évident vers lequel se diriger. Par conséquent, nous pensons que cet environnement nécessite un peu plus de planification que `MiniGrid-MultiRoom-N6-v0`, bien que pas autant que Sokoban. Cela peut expliquer pourquoi *ModelBasedRL-MCTS* dépasse *DQfD-PreTraining* comme la deuxième meilleure approche dans `MiniGrid-FourRooms-v0`.

6.3 Réduction de l'espace de recherche

La performance inférieure de *HybridPlanner-IDA** par rapport à *ConvictionPlanner-IDA** motive davantage notre approche basée sur la conviction. Cette différence de performance est expliquée par le fait que les approches basées sur la conviction incluent l'élagage de sous-arbres de trajectoires possibles en fonction de l'incertitude, ce qui réduit considérablement l'espace de recherche. Avec IDA*, ceci se traduit par le fait d'enlever les arêtes du graphe correspondant à des transitions trop incertaines, alors que pour MCTS, il s'agit plutôt d'élaguer le sous-arbre qui suit la transition incertaine.

L'approche de planification hybride ne fait pas cela, résultant en un espace de recherche plus grand et donc une moins bonne performance. Ainsi, une fonctionnalité qui était initialement destinée à prévenir uniquement les prédictions invalides dans un modèle incomplet est également devenue un moyen d'éviter

d'explorer les trajectoires contenant une transition qui est *a priori* discutable, puisqu'elle n'a jamais été vue pendant l'entraînement.

6.4 Limitations et hypothèses

Nos solutions proposées sont plus appropriées pour les problèmes où l'acquisition de démonstrations est considérée comme moins coûteuse que l'exécution d'un nombre indéfini d'interactions d'exploration. Ce n'est presque jamais le cas pour les environnements simulés ou les données synthétiques. Cependant, cela peut souvent être le cas en robotique et pour des ensembles de données coûteux ou autrement limités.

L'approche par conviction présente une contrainte additionnelle : il faut effectuer un prétraitement judicieux (typiquement connu sous le nom de *feature engineering* en anglais) des observations brutes. Avant de l'utiliser, il faut donc se poser la question si ce travail de prétraitement est avantageux, en comparaison à une approche classique dans laquelle on définit explicitement une modélisation de l'espace d'états.

Dans le chapitre sur la méthodologie, nous mentionnons également trois hyperparamètres qui ont été déterminés empiriquement : la récompense de l'atteinte du but (25), la constante pour transformer le nombre de pas restants (100) en valeur pour la conviction, et le seuil d'incertitude pour l'élagage. Comme pour toute méthode, ces hyperparamètres requièrent un peu d'essai et erreur, mais ne sont pas très difficiles à déterminer. La constante pour transformer le nombre de pas restants en valeur a été déterminée instantanément en observant la distribution de valeurs retournées par le modèle d'heuristique appris. La récompense de l'atteinte du but a été facile à déterminer, puisqu'il suffit d'avoir une valeur supérieure à la somme des récompenses d'exploration possibles en une seule trajectoire (pour éviter que l'agent favorise une séquence d'explorations à l'atteinte du but). Le seuil d'incertitude se détermine en quelques essais. On peut aussi observer la distribution des incertitudes lorsqu'on fournit des transitions non-observées dans l'entraînement et les comparer aux incertitudes obtenues pour les transitions observées pendant l'entraînement.

Il faut également noter que notre implémentation actuelle ne traite que des espaces d'états et d'actions discrets, et les expériences effectuées se limitent présentement à des environnements déterministes. Cependant, nous pensons qu'avec des modifications mineures, notre approche peut être généralisée à des espaces d'actions et d'états continus, ainsi qu'à des environnements stochastiques (nous abordons déjà un peu cette possibilité à la section 4.4).

En principe, l'approche par conviction telle que formulée dans ce mémoire devrait pouvoir supporter les états continus. En effet, notre formulation de la conviction (équation (4.4)) supporte les états continus en incluant une notion continue d'incertitude σ_τ. La majorité des modifications requises seraient plutôt nécessaires pour intégrer des actions continues. D'abord, dans notre implémentation de la fonction d'encodage qui transforme (s, a) en état latent ou code z_t, présentement nous avons un dictionnaire distinct par action. Ceci ne serait évidemment pas possible pour des actions continues. Il faudrait plutôt utiliser un autre mécanisme pour modéliser l'incertitude, comme des processus gaussiens. De plus, IDA* et MCTS ne supportent pas automatiquement la notion d'actions continues. Une possibilité serait d'utiliser un autre algorithme de recherche, comme par exemple les Rapidly Exploring Random Trees (Xu, 2024), qui sont spécifiquement conçus pour un espace d'actions continues. On pourrait alors y intégrer la formule de conviction un peu de la même façon que nous avons fait pour IDA* et MCTS. On pourrait également utiliser IDA* et MCTS et discrétiser les actions.

Finalement, il faut noter que nos deux approches proposées, comme toutes les approches qui font usage de planification explicite, sont plus lentes d'exécution que les approches purement neuronales. Ceci peut donc être un facteur important de décision, lorsque le temps de calcul est critique.

6.5 Comparaison avec WMG

WMG est destiné à fonctionner dans un régime de données plus élevé que notre algorithme. Il est environ 100 fois moins efficace en termes d'échantillonnage que *ConvictionPlanner-IDA** sur Sokoban, comme le montre le fait que le taux de succès de 49% est atteint après 5 millions d'interactions pour WMG contre 50 000 pour *ConvictionPlanner-IDA** (voir le tableau 5.4). Aussi, tout comme notre approche par conviction, WMG nécessite un *feature engineering* spécifique au problème.

Ceci étant dit, il est possible que dans des régimes de données plus élevés, il surpasse *ConvictionPlanner-IDA**, puisque le réseau neuronal *feed-forward* utilisé comme fonction heuristique dans notre implémentation actuelle d'IDA* n'est pas aussi puissant qu'un Transformeur. Ce dernier pourrait être capable d'apprendre des heuristiques de recherche plus efficaces que ce premier.

6.6 Résumé

De ces observations, nous concluons que nos deux approches proposées ont tendance à se démarquer particulièrement, par rapport aux approches alternatives, lorsqu'il y a une rareté des récompenses et une rareté des motifs exploitables dans les observations (voir discussion à la section 6.2 sur la présence de porte vers laquelle se diriger dans `MiniGrid-MultiRoom-N6-v0`). Une autre façon d'exprimer cela est que le *ConvictionPlanner* et le *HybridPlanner* sont mieux adaptés aux problèmes où la récompense est rare et leur solution nécessite une proportion non négligeable de planification.

L'approche par conviction nécessite toutefois un *feature engineering* qui peut, dans certains problèmes, nécessiter beaucoup d'effort. Dans ces cas, il est possible d'utiliser notre solution hybride, qui est plus générale, car elle ne nécessite pas d'encodeur codé en dur. Bien que ce choix implique une performance moins bonne (dû à un espace de recherche plus vaste que dans l'approche par conviction), il n'en reste pas moins que sa performance dépasse les approches alternatives connues.

CONCLUSION

Nous avons proposé deux stratégies appartenant au paradigme d'apprentissage par démonstrations. Ces deux stratégies s'attaquent aux difficultés d'apprentissage liées aux problèmes avec récompenses éparses. Nous démontrons que ces approches ont un avantage plus prononcé lorsque les problèmes nécessitent un niveau de planification plus avancé. Les deux stratégies que nous avons proposées consistent à combiner les bénéfices de l'apprentissage par renforcement avec modèle et les bénéfices de l'apprentissage par démonstrations. Pour faire fonctionner cet amalgame, nous avons conçus deux mécanismes différents.

Le premier, *ConvictionPlanner-MCTS/ConvictionPlanner-IDA**, utilise une estimation de l'incertitude faite par le modèle de transition, qui sert à faire l'élagage de l'arbre de recherche et à calculer la métrique de conviction qu'on optimise lorsqu'on choisit la trajectoire à exécuter. Le second, *HybridPlanner-IDA**, contourne le problème du modèle de transition incomplet en combinant les démonstrations avec des données d'exploration aléatoire. Nous avons démontré, avec des expériences sur les environnements virtuels Minigrid et Sokoban, que l'approche par conviction est supérieure aux autres en termes d'efficacité échantillonnale. De plus, nous avons démontré que l'approche par conviction est approximativement 100 fois plus efficace que les *Working Memory Graphs* pour les quantités de données sur lesquelles nous avons expérimenté, surpassant ainsi l'état de l'art sur Sokoban en termes d'efficacité échantillonnale pour les méthodes d'apprentissage automatique.

Toutefois, nous avons aussi mis en évidence les limites de cet algorithme, qui nécessite un espace d'encodage conçu avec justesse. En contraste, l'approche hybride ne nécessite pas un tel effort d'ingénierie et surpasse tout de même les autres algorithmes comparatifs. Certaines avenues de recherche futures s'ouvrent à nous suite à ce travail. D'une part, il faudrait montrer que les algorithmes présentés ici peuvent se généraliser aux environnements continus (actions et états). De plus, utiliser l'approche par conviction dans une application pratique de robotique, jointe à un module de perception, permettrait de mieux la motiver. Finalement, il serait avantageux d'améliorer les algorithmes de recherche utilisés dans nos approches proposées en guidant leur priorités d'exploration via un modèle neuronal plus sophistiqué.

BIBLIOGRAPHIE

Abbeel, P. et Ng, A. Y. (2004). Apprenticeship learning via inverse reinforcement learning. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, volume 69.

Afsar, M. M., Crump, T. et Far, B. (2022). Reinforcement learning based recommender systems : A survey. *ACM Computing Surveys*, *55*, 1–38.

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, Z. D. et Blundell, C. (2020). Agent57 : Outperforming the atari human benchmark. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, 507–517.

Bellemare, M. G., Naddaf, Y., Veness, J. et Bowling, M. (2013). The arcade learning environment : An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, *47*, 253–279.

Campbell, M., Egerstedt, M., How, J. P. et Murray, R. M. (2010). Autonomous driving in urban environments : approaches, lessons and challenges. *Philosophical Transactions of the Royal Society A : Mathematical, Physical and Engineering Sciences*, *368*(1928), 4649–4672.

Chevalier-Boisvert, M., Willems, L. et Pal, S. (2018). Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`.

Chung, J., Gulcehre, C., Cho, K. et Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. Dans *NIPS Workshop on Deep Learning*.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo Tree Search. Dans *Proceedings of the International Conference on Computers and Games (ICCG)*, 72–83. Springer.

Deisenroth, M. et Rasmussen, C. E. (2011). PILCO : A model-based and data-efficient approach to policy search. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, 465–472.

Deisenroth, M. P., Fox, D. et Rasmussen, C. E. (2013). Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *37*, 408–423.

Gao, Y., Xu, H., Lin, J., Yu, F., Levine, S. et Darrell, T. (2018). Reinforcement learning from imperfect demonstrations. *arXiv preprint arXiv :1802.05313*.

Goecks, V. G., Gremillion, G. M., Lawhern, V. J., Valasek, J. et Waytowich, N. R. (2020). Integrating behavior cloning and reinforcement learning for improved performance in dense and sparse reward environments. Dans *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 465–473.

Grześ, M. (2017). Reward shaping in episodic reinforcement learning. Dans *Proceedings of the International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, 565–573.

Guez, A., Mirza, M., Gregor, K., Kabra, R., Racanière, S., Weber, T., Raposo, D., Santoro, A., Orseau, L., Eccles, T., Wayne, G., Silver, D. et Lillicrap, T. (2019). An investigation of model-free planning. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, 2464–2473.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I. *et al.* (2018). Deep Q-learning from Demonstrations. Dans *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Kober, J., Bagnell, J. A. et Peters, J. (2013). Reinforcement learning in robotics : A survey. *The International Journal of Robotics Research*, *32*, 1238–1274.

Konda, V. et Tsitsiklis, J. (1999). Actor-critic algorithms. *Advances in Neural Information Processing Systems*, *12*.

Korf, R. E. (1985). Depth-first iterative-deepening : An optimal admissible tree search. *Artificial intelligence*, *27*, 97–109.

Ladosz, P., Weng, L., Kim, M. et Oh, H. (2022). Exploration in deep reinforcement learning : A survey. *Information Fusion*, *85*, 1–22.

Lakshminarayanan, B., Pritzel, A. et Blundell, C. (2017). Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in Neural Information Processing Systems*, *30*.

Loynd, R., Fernandez, R., Celikyilmaz, A., Swaminathan, A. et Hausknecht, M. (2020). Working memory graphs. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, 6404–6414.

Moerland, T. M., Broekens, J. et Jonker, C. M. (2020). Model-based reinforcement learning : A survey. *arXiv preprint arXiv :2006.16712*.

Narvekar, S., Peng, B., Leonetti, M., Sinapov, J., Taylor, M. E. et Stone, P. (2020). Curriculum learning for reinforcement learning domains : A framework and survey. *The Journal of Machine Learning Research*, *21*, 7382–7431.

Ng, A. Y., Russell, S. *et al.* (2000). Algorithms for inverse reinforcement learning. Dans *Proceedings of the International Conference on Machine Learning (ICML)*, volume 1, 663–670.

Ouellette, S., Beaudry, E. et Bouguessa, M. (2024). Conviction-based planning for sparse reward reinforcement learning problems. Dans *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning*.

Pignat, E. et Calinon, S. (2019). Bayesian gaussian mixture model for robotic policy imitation. *IEEE Robotics and Automation Letters*, *4*, 4452–4458.

Racanière, S., Weber, T., Reichert, D., Buesing, L., Guez, A., Jimenez Rezende, D., Puigdomènech Badia, A., Vinyals, O., Heess, N., Li, Y. *et al.* (2017). Imagination-augmented agents for deep reinforcement learning. *Advances in Neural Information Processing Systems*, *30*.

Reddy, S., Dragan, A. D. et Levine, S. (2019). SQIL : Imitation learning via reinforcement learning with sparse rewards. Dans *International Conference on Learning Representations (ICLR)*.

Ritter, S., Faulkner, R., Sartran, L., Santoro, A., Botvinick, M. et Raposo, D. (2020). Rapid task-solving in novel environments. *arXiv preprint arXiv :2006.03662*.

Rummery, G. A. et Niranjan, M. (1994). *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, Cambridge, UK.

Russell, S. et Norvig, P. (2010). Artificial intelligence : A modern approach. 3rd edition. *Prentice Hall*, *Upper Saddle River*.

Schrader, M.-P. B. (2018). gym-sokoban. `https://github.com/mpSchrader/gym-sokoban`.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T. et Silver, D. (2020). Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*, *588*, 604–609.

Shalev-Shwartz, S., Shammah, S. et Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *arXiv preprint arXiv :1610.03295*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A. *et al.* (2017). Mastering the game of go without human knowledge. *Nature*, *550*, 354–359.

Sutton, R. S. et Barto, A. G. (2018). *Reinforcement learning : An introduction, 2nd Edition*. MIT press.

Torabi, F., Warnell, G. et Stone, P. (2018). Behavioral cloning from observation. *arXiv preprint arXiv :1805.01954*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. et Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Vecerik, M., Hester, T., Scholz, J., Wang, F., Pietquin, O., Piot, B., Heess, N., Rothörl, T., Lampe, T. et Riedmiller, M. (2017). Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards. *arXiv preprint arXiv :1707.08817*.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P. *et al.* (2019). Grandmaster level in Starcraft II using multi-agent reinforcement learning. *Nature*, *575*, 350–354.

Wilcox, A., Balakrishna, A., Dedieu, J., Benslimane, W., Brown, D. et Goldberg, K. (2022). Monte Carlo augmented actor-critic for sparse reward deep reinforcement learning from suboptimal demonstrations. *Advances in Neural Information Processing Systems*, *35*, 2254–2267.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Xu, T. (2024). Recent advances in rapidly-exploring random tree : A review. *Heliyon*.