Titre: Title:	Combinaison de la programmation par contraintes et de l'apprentissage par renforcement profond pour résoudre les tâches de planification IA classique
Auteur: Author:	Dana Chaillard
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Chaillard, D. (2022). Combinaison de la programmation par contraintes et de l'apprentissage par renforcement profond pour résoudre les tâches de planification IA classique [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10476/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Combinaison de la programmation par contraintes et de l'apprentissage par renforcement profond pour résoudre les tâches de planification IA classique

DANA CHAILLARD
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Combinaison de la programmation par contraintes et de l'apprentissage par renforcement profond pour résoudre les tâches de planification IA classique

présenté par **Dana CHAILLARD**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Michel GAGNON, président
Gilles PESANT, membre et directeur de recherche
Quentin CAPPART, membre

REMERCIEMENTS

Je voudrais remercier d'abord Gilles, mon directeur de recherche pour son soutien pendant ces presque 2 ans malgré les hauts et les bas. Je veux remercier également mes colocataires Grégory, Katia et Clémence de m'avoir nourri dans les temps difficiles.

RÉSUMÉ

Bien qu'il y ait eu des progrès significatifs dans le domaine de la planification IA en général, certains domaines demeurent hors de portée des systèmes de planification IA actuels. Récemment, l'utilisation d'apprentissage par renforcement avec des réseaux de neurones profonds a permis de repousser cette frontière. Nous essayons de pousser cette frontière encore plus loin en utilisant la programmation par contraintes avec de la propagation de croyance. La programmation par contraintes est une très bonne méthode pour découvrir la structure d'un problème. On utilise cette information pour guider un agent d'apprentissage par renforcement sur un tâche de planification IA classique, Floortile. De nos agents avec de l'information de la CP on peut tirer trois types. Ceux où l'information est injectée dans les observations du monde. Ceux où l'information est ajoutée dans le choix de l'agent lorsqu'il utilise sa politique. Et enfin, ceux où l'information est ajoutée dans la récompense.

De ces trois types seul le second obtient des performances supérieures à l'agent témoin. Cependant, ces agents ne peuvent pas se passer de programmation par contraintes pour continuer à fonctionner et ne généralisent pas très bien à d'autres exemplaires du même problème. Le premier type d'agent obtient des performances bien en-dessous de celles de l'agent témoin et le troisième agent les égale.

ABSTRACT

While there has been significant progress in general AI planning, certain domains remain out of reach of current typical AI planning systems. Recently, the use of deep reinforcement learning has allowed to push that frontier further. We try to help further these RL agents with the use of constraint programming with belief propagation. Constraint programming is a very good method to uncover the structure of a problem and we use that information to guide a reinforcement learning agent on a classical AI planning task, Floortile. Our agents using constraint programming can be separated into three types : those where the information is used in the observations, those where the information is used in the decision process and those where the information is used in the reward.

Of these three types, only the second one achieves performances better than the control agent. However, these agents cannot work without constraint programming and don't generalize very well to other instances of the same problem. The first type achieves performances way below the control and the third one achieves about the same as the control.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

Acronymes

AI/IA Artificial Intelligence

$CB-BP$ Programmation par contraintes avec belief propagation

CP Constraint Programming / Programmation par Contraintes

CSP Constraint Satisfaction Problem

DQN Deep Q Network

RL Reinforcement Learning / Apprentissage par Renforcement

HRL Hierarchical Reinforcement Learning/ Apprentissage par renforcement hierarchique

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

En Intelligence Artificielle (IA), la planification est une tâche qui vise à développer des algorithmes capables de produire des plans. Ces plans dirigent typiquement un ou plusieurs robots ou tout autre agent pour résoudre un problème. Pour résoudre ces problèmes il existe toutes sortes d'algorithmes. On pourra noter par exemple l'algorithme A* qui est un algorithme de recherche de chemin guidé par une heuristique.

Comme exemple de l'une de ces tâches, on a la tâche Floortile. Dans cette tâche, on veut peindre un sol dallé en suivant un motif. Pour cela, on utilise un ou plusieurs robots qui portent une ou plusieurs couleurs. Ils doivent faire attention à ne pas endommager le motif une fois qu'il est peint et à ne pas s'endommager les uns les autres. Ils commencent dans une position de départ choisie à l'avance et doivent peindre le motif choisi par l'utilisateur. Ils cherchent à le faire en utilisant le moins d'actions possible. On a un départ et une arrivée connue ; on cherche juste le meilleur chemin.

Dans ce domaine de la planification IA, il y a beaucoup de types de planificateurs [Ghallab et al., 2016]. Beaucoup font usage d'informations données par l'utilisateur liées au domaine. On s'intéresse ici à deux types de planificateurs qui ont tous les deux fait leurs preuves dans le milieu. D'abord, les planificateurs basés sur l'apprentissage par renforcement (RL). Ces agents ont connu de très grands succès dans les dernières années (par exemple Feng et al., 2020 obtiennent des performances supérieures à l'état de l'art sur leur tâche). Ce sont des agents souvent difficiles à concevoir et qui demandent beaucoup d'itérations de conception avant d'atteindre leur meilleure performance.

D'autre part, on a les planificateurs qui font usage de la programmation par contraintes (CP). La CP est un paradigme de programmation déclaratif. L'utilisateur définit son problème en tant que variables et contraintes et n'a aucune prise sur la résolution. C'est donc une méthode assez accessible et facile à employer. La force de la CP est sa capacité à extraire très rapidement la structure centrale d'un problème et l'utiliser pour résoudre rapidement. On utilise souvent des contraintes très générales qui recouvrent une grande partie du problème ce qui les rend très puissantes. En revanche, ces planificateurs demandent une connaissance parfaite de l'environnement et aucune stochasticité, ce qui n'est pas le cas des agents RL. Ils doivent construire un plan complet à partir de la position de départ en connaissant donc le monde entier et toutes les interactions possibles.

Les agents de RL opèrent par rapport à un état du monde à un instant t. Par rapport à cet état, l'agent décide de la prochaine action à prendre. Au contraire, la CP s'intéresse au problème tout entier à la fois et à la structure globale du monde. On s'intéresse donc à la capacité de CP à donner de l'information sur la structure générale du problème à un agent guidé par la RL. L'information portée par la CP peut-elle aider à entraîner un agent plus rapidement ou simplement le rendre meilleur que l'état de l'art ?

Pour revenir à notre exemple, la CP pourrait diriger vers les directions les plus prometteuses à prendre pour notre robot. Elle pourrait aussi empêcher le robot de se mettre dans une situation qui rend le reste de la tâche impossible. Ces deux informations sont faciles à calculer dans un contexte de CP et pourraient faciliter la tâche de l'agent RL.

Notre recherche s'intéresse d'abord à la capacité de l'ajout d'information issue de la CP à aider les performances de l'agent. On injectera cette information à plusieurs endroits dans la boucle d'entraînement ou d'inférence de l'agent pour les comparer. On créera des exemples du problème de difficulté croissante et on essaiera les différents agents pour voir si les performances sont changées par l'ajout de la CP. On s'intéressera aussi au temps d'entraînement et au nombre d'itérations d'entraînement nécessaire. Ensuite, on s'intéressera à la capacité de ces agents à généraliser ce qu'ils ont appris à un autre problème similaire à celui sur lequel ils ont été entraînés. Un agent très performant sur un unique exemplaire n'est pas très intéressant pour faire une tâche au niveau industriel.

Dans ce mémoire, on commencera par passer en revue les notions de base et par faire une revue de littérature. Ensuite, on détaillera la méthodologie utilisée. Enfin, on observera les résultats obtenus et ce qu'ils impliquent et on fera une conclusion.

CHAPITRE 2 NOTIONS DE BASE ET REVUE DE LITTÉRATURE

Dans cette section, nous présenterons l'état de l'art et les différents domaines auxquels la recherche se relie. Nous présenterons en particulier les quelques articles qui sont utilisés comme base pour nos expériences.

2.1 Planification IA

La planification IA est un domaine de l'intelligence artificielle où l'on demande à un agent autonome de construire un plan pour qu'un opérateur suivant le plan puisse compléter une tâche. Ces problèmes existent dans notre vie quotidienne.

Les problèmes de planification IA se présentent comme suit : on a un état initial et un état final désiré, il faut créer un plan qui permette de relier ces deux états. On ne connaît pas à l'avance le nombre d'actions qu'il faut effectuer. On connaît en revanche l'ensemble des actions possibles. Les actions sont généralement accompagnées de prérequis qui sont eux aussi connus. En planification classique on connaît en plus l'état du monde au complet à tout instant et l'effet des actions sur le monde. Les actions ont un effet déterministe, ainsi, le plan est simplement une suite d'actions de longueur inconnue au départ qu'il faut trouver. Les problèmes de planification classique peuvent être représentés par un tuple $\{S, A, T, s_0, G\}$. S est l'ensemble des états, A l'ensemble des actions, T la fonction de transition qui associe à une paire état-action l'état qui suit, $s_0 \in S$ l'état initial et $G \subset S$ l'ensemble des états finaux.

La tâche à laquelle nous nous attelons ici, Floortile, est une tâche de planification classique qui sera décrite plus en détail dans la partie 2.2.1. Classiquement, pour résoudre les problèmes de planification, on utilise des algorithmes de recherche simple mais efficaces qui reposent sur une heuristique, comme A* [Hart et al., 1968]. A* est un exemple typique d'algorithme de planification car il est très simple mais obtient de très bonnes performances sur ces problèmes. Dans A*, on cherche un chemin qui mène de l'état de départ à l'état d'arrivée. On choisit une heuristique de mesure d'une distance à l'arrivée et on explore localement à partir de cette heuristique jusqu'à trouver le meilleur chemin.

2.2 Programmation par contraintes

La programmation par contraintes (CP) est un paradigme de programmation qui s'est avéré utile pour la résolution de problèmes combinatoires entre autres dans les domaines de la planification. À l'instar des autres formes de programmation mathématique comme la pro-

grammation linéaire ou la programmation en nombres entiers, elle peut résoudre des problèmes $\mathcal{NP}$ difficiles. Là où la CP se démarque des autres méthodes est la représentation des problèmes. La formulation de problèmes se fait simplement grâce à des concepts haut niveau qui sont proches de la formulation naturelle du problème [Pesant, 2014]. L'utilisateur décrit son problème et sa structure sous la forme d'un modèle mathématique sans avoir à préciser comment le résoudre ; le solveur se charge du reste.

Plus précisément, la programmation par contraintes représente les problèmes sous la forme d'un *Constraint Satisfaction Problem* (CSP). C'est-à-dire : un ensemble fini de variables $X = \{x_1, x_2, ..., x_n\}$ prenant leur valeur dans des domaines finis $x_i \in D_i \subset \mathbb{Z},\ 1 \leqslant i \leqslant n$ et un ensemble fini de contraintes $C = \{c_1, ..., c_m\}$ qui concernent chacune un sous-ensemble de variables $c_j(x_{j1}, x_{j2}, ..., x_{jk}) \subset \mathbb{Z}^k,\ 1 \leqslant j \leqslant m$. Cette manière de représenter le problème permet d'en exposer la structure haut niveau très facilement ce qui en fait sa force.

Le catalogue de contraintes pour la plupart des solveurs est assez large et propose des contraintes complexes. On trouve des contraintes globales qui au-delà d'exposer la structure du problème rendent la résolution plus rapide [Rossi et al., 2006].

On pourra prendre pour exemple la contrainte REGULAR qui vérifie qu'un mot appartienne à un langage. La contrainte `regular(`X, Π`)` est satisfaite si la suite finie de variables $X = \langle x_1, x_2, ..., x_k \rangle$ épelle un mot qui correspond au langage définit par l'automate $\Pi = (Q, \Sigma, \delta, q_0, F)$ où Q est un ensemble fini d'états, Σ un alphabet, $\delta : Q \times \Sigma \to Q$ est une fonction partielle de transition, $q_0 \in Q$ un état de départ et $F \subseteq Q$ est l'ensemble des états finaux. Elle est souvent utilisée pour vérifier si une suite de variables suit un motif recherché ou quand un problème est facilement représenté par un ou plusieurs automates.

L'objectif est d'attribuer à chaque variable une valeur de son domaine pour satisfaire toutes les contraintes. Pour cela, la CP utilise de puissants algorithmes de filtrage qui réduisent la taille des domaines en enlevant les valeurs impossibles. Ainsi, l'espace à explorer devient exponentiellement plus petit. L'exploration de cet espace s'effectue typiquement au moyen d'un arbre de recherche où à chaque noeud l'algorithme fixe la valeur d'une variable ou retire une valeur d'un domaine. La décision prise au noeud est ensuite répercutée sur les autres domaines au travers des contraintes et ainsi de suite. Si on arrive à une situation impossible, on fait demi-tour d'un cran et on continue («backtracking»).

Lors de la résolution d'un problème, la CP fait ce qu'on appelle de la *propagation de contraintes*. Pendant cette propagation, l'algorithme trouve les valeurs dans les domaines des variables qui sont incompatibles avec les contraintes. Par exemple, pour les domaines $D_1 = \{1, 2, 3\}$ et $D_2 = \{2, 3\}$ et la contrainte $x_1 \geqslant x_2$, la propagation de contraintes peut retirer la valeur 1 de D_1 car elle est incompatible. On retire des valeurs dans les domaines

des différentes variables et ces filtrages entraînent d'autres filtrages au travers du réseau de contraintes. Cette propagation converge nécessairement puisque le cardinal du produit cartésien des domaines est strictement décroissant.

Durant la recherche de solution, l'algorithme doit faire des choix de branchement dans l'arbre. Typiquement, il faut décider quelle valeur fixer pour quelle variable. Il existe plusieurs types d'heuristiques pour prendre cette décision. Les heuristiques les plus générales se basent sur des informations faciles à obtenir sur les variables comme la taille de leur domaine ou le nombre de contraintes dans lesquelles elles sont impliquées [Pesant et al., 2012]. On pourra prendre pour exemple l'heuristique `ddeg`, introduite par [Brélaz, 1979], qui choisit la variable avec le plus petit domaine et en cas d'égalité choisit la variable avec le plus grand *dynamic degree*. Pour calculer le degré dynamique, on regarde toutes les contraintes liées à la variable et on vérifie si elle est sont liées à des variables non fixées. Le degré dynamique est alors le nombre de contraintes liées à la variable qui contraignent encore d'autres variables non fixées.

On trouve également des heuristiques basées sur le dénombrement de solutions ou une approximation de la densité de solutions. Ces approches ne sont pas nouvelles. [Kask et al., 2004] partent d'une solution partielle pour un CSP et pour une variable choisie comptent le nombre total de solutions à partir de la solution partielle pour choisir la valeur.

On s'intéressera ici à l'approche de [Pesant et al., 2012] d'une telle heuristique. La première étape est de construire des algorithmes permettant d'obtenir un décompte des solutions de la part des contraintes classiques. Pour une paire variable/valeur, on parle de densité de solutions par rapport à une contrainte. Il s'agit de la proportion des solutions à cette contrainte qui contiennent cette valeur pour cette variable. Une fois que l'on est capables de récupérer ces décomptes de solutions, on peut aussi calculer les densités de solution pour toutes les paires variables/valeurs par rapport à toutes les contraintes. Ainsi, pendant la recherche, on peut utiliser ces valeurs de densité de solutions pour prendre les décision de branchement.

Algorithm 1 Heuristique maxSD

$max \leftarrow 0$
for c in C **do**
  for x in $scp(c)$ **do**
    for d in $D(x)$ **do**
      if $\mathbf{SD}(c, x, d) > max$ **then**
        $max \leftarrow SD(c, x, d)$
Branchement sur (x,d) lié au max

On prendra pour exemple une heuristique proposée par l'article, **maxSD** représentée dans

l'Algorithme 1. On itère sur tous les triplets contrainte/variable/valeur. Pour chacun de ces triplets, on calcule la densité de solutions. On récupère la valeur maximale de toutes ces densités, et on choisit la variable liée avec la valeur liée pour le prochain branchement.

Pour calculer ces densités de solutions, il faut des algorithmes de dénombrement puissants. Voyons par exemple celui pour la contrainte REGULAR décrite plus haut [Zanarini et Pesant, 2009]. Pour commencer, on déplie l'automate (exemple en figure 2.1) en un graphe en couches acyclique et orienté $G = (V, A)$ où les sommets d'une couche correspondent aux états de l'automate et les arcs sont des paires variable-valeur (*i.e.* des transitions de l'automate). Un exemple d'un tel graphe est donné en Figure 2.2. Puisque les arcs représentent une variable prenant une certaine valeur, cette valeur est représentée par la couleur de l'arc.

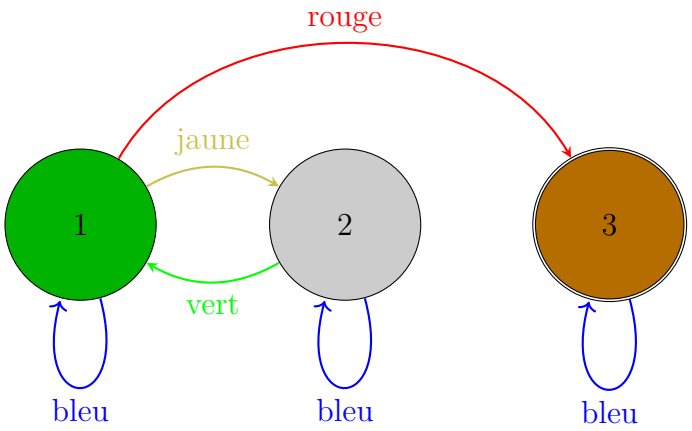

Figure 2.1 Automate lié au graphe de la figure 2.2. 1 est l'état de départ, 3 l'état final

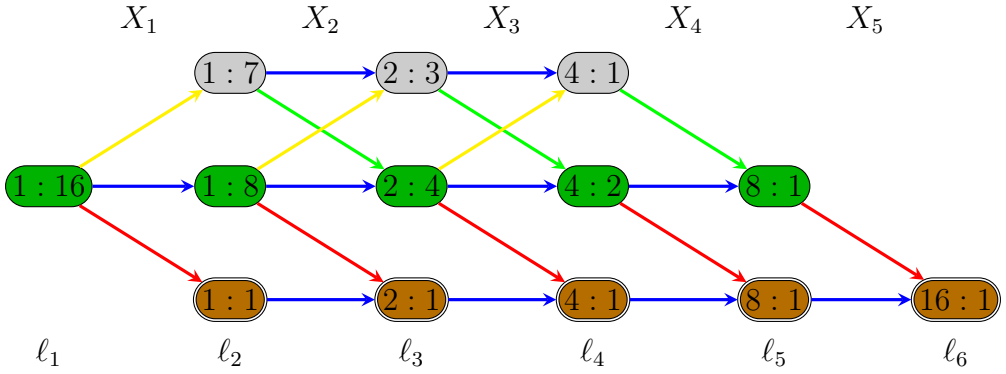

Figure 2.2 Le graphe en couches acyclique et orienté construit pour une contrainte Regular sur 5 variables. Les étiquettes sur les sommets sont le nombre de chemins entrants et sortants du sommet.

On note $v_{\ell,q}$ le sommet qui correspond à l'état q dans la couche ℓ. La première couche ne

contient que l'état de départ q_0 ; la dernière couche les états finaux. Un arc entre la couche L_i et L_{i+1} est lié à la variable x_i et représente une transition autorisée par l'automate. Tout chemin partant de l'état de départ et allant jusqu'à la dernière couche représente une solution à la contrainte et toutes les solutions sont représentées par un chemin dans le graphe. Pour faire du dénombrement de solutions et donc de densité de solutions, on doit calculer le nombre de chemins qui sortent d'un sommet pour aller à la dernière couche et le nombre de chemins qui arrivent jusqu'à un sommet depuis la première couche. On note $\#op(\ell, q)$ le nombre de chemins sortants d'un sommet et $\#ip(\ell, q)$ le nombre de chemins entrants comme représenté dans la Figure 2.2. On obtient ces valeurs par récurrence :

$$\begin{aligned}
\#op(n+1, q) &= 1 \\
\#op(\ell, q) &= \sum_{(v_{\ell,q}, v_{\ell+1,q'}) \in A} \#op(\ell+1, q'), \quad 1 \leqslant \ell \leqslant n \\
\#ip(1, q_0) &= 1 \\
\#ip(\ell+1, q') &= \sum_{(v_{\ell,q}, v_{\ell+1,q'}) \in A} \#ip(\ell, q), \qquad 1 \leqslant \ell \leqslant n
\end{aligned}$$

Le nombre total de solutions est donc :

$$\#\texttt{regular}(X, \Pi) = \#op(1, q_0)$$

Puisque les paires variable-valeur (x_i, d) sont représentées par les arcs entre la couche i et la couche $i+1$ qui correspondent à la valeur d, le nombre de solutions comprenant cette paire est le nombre de chemins empruntant un de ces arcs. Pour chacun de ces arcs $(v_{i,q}, v_{i+1,q'})$ le nombre de chemins passant par là est $\#op(i+1, q') \times \#ip(i, q)$. Puisque les deux sont facilement calculables, la densité de solutions par paire est elle aussi simple à calculer puisque c'est simplement :

$$\sigma(x_i, d, \texttt{regular}) = \frac{\sum_{(v_{i,q}, v_{i+1,q'}) \in A(i,d)} \#op(i+1, q') \times \#ip(i, q)}{\#op(1.q_0)}$$

En notant, $A(i, d) \subset A$ l'ensemble des arcs qui représentent la paire variable valeur (x_i, d)

Ces densités de solutions sont exactes et tractables en un temps raisonnable. On peut donc les utiliser dans l'heuristique décrite plus haut. Maintenant que nous avons vu comment la CP fonctionne et construit les solutions, nous allons nous intéresser à comment appliquer ces algorithmes très puissants pour des problèmes de planification IA en particulier.

2.2.1 CP pour la Planification IA

Historiquement cela fait plus de 20 ans que la CP est utilisée pour la planification IA. Les premiers travaux se concentraient principalement sur la reformulation des problèmes de planification en CSP. Par exemple, [Do et Kambhampati, 2001] proposent un système qui prend des problèmes de planification encodés en STRIPS et les transforme en CSP puis les résout à l'aide d'un solveur de programmation par contraintes. Ils montrent que cette manière d'encoder est moins coûteuse et lourde en mémoire que les encodages de problèmes SAT. [Lopez et Bacchus, 2003] proposent une manière d'encoder les problèmes de planification comme des CSP qui révèle plus de la structure du problème qu'un encodage en STRIPS. D'autres s'intéressent à utiliser la capacité de CP à résoudre les problèmes de scheduling et essayer de transférer cette force sur les tâches de planification, comme [Beek et Chen, 2000] puis [Barták et Toropila, 2008].

La CP s'intéresse généralement à des problèmes $\mathcal{NP}$ difficiles alors que la planification IA s'intéresse plutôt à des problèmes encore plus difficiles ($\mathcal{PSPACE}$ difficile). L'approche naturelle, qui permet de vérifier la possibilité de résoudre le problème et rend le problème $\mathcal{NP}$ difficile, consiste à essayer de résoudre une succession d'exemplaires de longueur croissante jusqu'à trouver la solution. Autrement dit, dans le cas de la planification, on essaie de régler le problèmes en une action, puis deux puis trois etc. On finit par trouver un nombre d'action suffisant pour régler le problème.

Pour résoudre ces problèmes de planification, la CP doit les encoder comme des CSP. Un moyen intéressant de faire cette représentation est l'usage d'automates (c.f. [Zanarini et al., 2006]). On représente le problème avec un ensemble d'automates, un par entité ou objet participant à la dynamique du problème. C'est-à-dire tous les objets dont l'état influence la résolution du problème. Les états de l'automate sont les états de l'objet qu'il représente et les transitions entre ces états sont les actions. L'état de tous les automates à un instant t donne l'état du monde complet. Les objectifs sont représentés par les états finaux des automates. Un plan est valide quand il amène tous les automates à un de leurs états finaux et qu'il ne contient que des actions valides par rapport aux automates.

On prendra pour exemple la tâche Floortile[1]. Dans cette tâche, un ensemble de robots doivent peindre un sol dallé en suivant un motif prédéterminé. L'automate en Figure 2.3 est celui d'une dalle, il vient de [Babaki et al., 2020].

Les robots ont accès à sept actions : ils peuvent bouger d'une case dans les quatre directions, peindre la case qui est au-dessus ou en dessous d'eux ou changer la couleur de la peinture

1. `http://www.plg.inf.uc3m.es/ipc2011-deterministic/DomainsSequential.html`

qu'ils tiennent. On a donc trois types d'actions :

- Pour chaque robot r et chaque couleur c, une action *change-colour(r,c)*.
- Pour chaque robot r et dalle à peindre t, une action *paint-up(r,t)* et *paint-down(r,t)*.
- Pour chaque robot r et dalle courants t, les actions *move-up(r,t), move-down(r,t), move-right(r,t), move-left(r,t)*.

Cela nous donne en tout pour n_r robots, n_t dalles et n_c couleurs $n_r \times (n_c + 6n_t)$ actions possibles. Si l'on observe plus en détail notre automate en Figure 2.3, on voit que l'on a défini un ensemble d'actions différent de nos actions générales qui se rapportent à cette dalle en particulier. Cet ensemble d'actions {*paint, move_to, move_from, other*}, est relié à nos actions générales par une matrice de correspondance qui fait partie de notre automate. Ainsi une action *move-up(r,t)* qui fait bouger le robot r depuis la dalle t donne une action *move_from* dans le nouvel ensemble pour la dalle *t*. Cette même action *move-up(r,t)* donnera *move_to* pour l'automate de la dalle *t'* qui est au-dessus de *t*. Pour toutes les autres dalles c'est une action *other*.

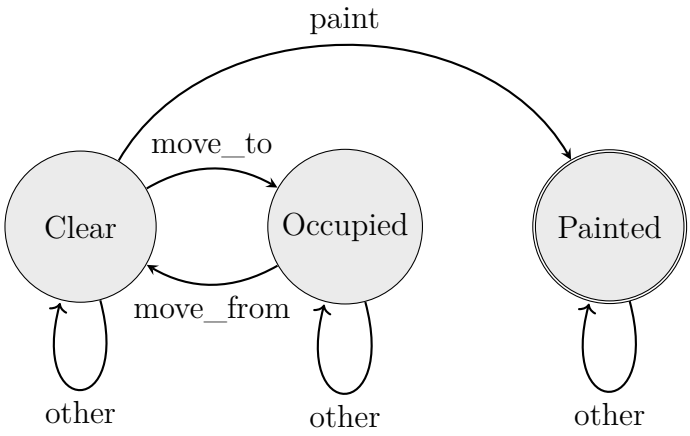

Figure 2.3 Automate d'une dalle à peindre pour la tâche Floortile

Il y a un automate différent pour le robot. On a un automate par robot et un automate par dalle. Le nombre d'états pour les robots est le nombre de dalles multiplié par le nombre de couleurs disponibles.

En effet, on peut entièrement décrire l'état du robot si on connaît sa position et la couleur qu'il a en main.

Pour définir le problème comme un CSP, il faut fixer la longueur du plan recherché (ℓ). On définit le problème avec une suite de variables X_i, la ième variable représente la ième action du plan, et un domaine d'actions disponibles D tel que $x_i \in D$, pour $1 \leqslant i \leqslant \ell$.

Les seules contraintes sont des contraintes REGULAR, une par automate. La contrainte

Algorithm 2 Résoudre un problème de planification en augmentant la longueur

$c^\star \leftarrow \infty$
for $\ell \leftarrow \ell_{min}$ **to** ∞ **do**
  if $c_{min}(\ell) \geqslant c^\star$ **then**
    return $c^\star$
  $c \leftarrow solve(model(\ell), T_{max})$
  if $c < c^\star$ **then**
    $c^\star \leftarrow c$

REGULAR vérifie que la suite d'actions dans le plan est compatible avec l'automate à qui la contrainte est liée. On résout ensuite le problème comme montré dans l'algorithme 2 qui vient de [Babaki et al., 2020].

On essaie de résoudre le modèle pour des longueurs croissantes de plan. On cherche le plan avec le coût minimal. $c_{min}()$ est une fonction non décroissante de la longueur de plan ℓ, elle donne une borne inférieure du coût du plan de longueur ℓ. Si cette fonction donne une valeur supérieure au coût minimal que l'on a trouvé jusqu'ici c'est qu'un plan de cette longueur ou plus long ne sera jamais meilleur que celui qu'on a déjà trouvé. On peut donc arrêter la recherche et renvoyer notre plan qui est le meilleur. On essaie de résoudre chaque modèle pour une durée maximale T_{max} pour éviter de perdre notre temps à démontrer qu'il n'y a pas de plan possible dans les longueurs trop courtes (comme vu chez [Barták et Toropila, 2008]). On garde dans $c^\star$ le coût minimal trouvé jusque là.

L'approche de Babaki et al., 2020 que nous allons reprendre dans nos expériences utilise l'heuristique de branchement **maxSD** vu à la section 2.2. On va ensuite présenter une approche qui part de la capacité à calculer ces densités de solutions et qui la pousse plus loin.

2.2.2 CP avec BP

On va ici présenter l'approche présentée par [Pesant, 2019]. Il s'intéresse à changer le processus de propagation de contraintes pour transmettre une information plus riche. Lors d'une propagation de contraintes classique, on peut considérer que l'on fait un passage de messages où le message passé est simplement un booléen, `false` si la valeur est retirée du domaine `true` sinon. C'est finalement peu d'informations. Si à la place on considère le domaine filtré d'une variable par rapport à une contrainte comme un ensemble de paires variable-valeur qui ont une fréquence non nulle dans l'ensemble de solutions ([Dechter et al., 2010]), on a une information plus intéressante.

[Pesant, 2019] ajoute là-dessus et transmet la distribution de fréquences sur toutes les

variables. Ces valeurs sont les mêmes que celles utilisées par l'heuristique de branchement vue à la section 2.2. On avait grâce à [Pesant et al., 2012] accès à un décompte du nombre de solutions par paire variable-valeur pour certains types de contraintes. Il est trivial d'en tirer des fréquences de présence dans les solutions. On peut choisir de considérer ces fréquences comme des probabilités de présence dans la solution (pour cette contrainte). On s'approche alors du belief propagation et des autres algorithmes de passage de messages pour l'inférence probabiliste.

Pour illustrer l'approche et montrer l'utilisation des algorithmes de dénombrement, nous allons prendre un exemple tiré de [Pesant, 2019] :

i. $\texttt{alldifferent}(a, c, b)$

ii. $a + b + c + d = 7$

iii. $c \leqslant d$

Avec ces contraintes, on prend le même domaine $\{1, 2, 3, 4\}$ pour les quatre variables $\langle a, b, c, d\rangle$. Il y a deux solutions possibles $\langle 2, 3, 1, 1\rangle$ et $\langle 3, 2, 1, 1\rangle$. Cependant, la propagation de contraintes ne fait aucun filtrage, on est donc obligé de faire un branchement alors que toutes les variables ont un domaine identique. On connaît la probabilité exacte pour une valeur d'être dans la solution puisqu'on connaît l'ensemble des solutions. On cherche à voir si le processus de CP-BP de [Pesant, 2019] arrive à approximer correctement ces vraies marginales. On a un passage de messages entre variables et contraintes. À chaque itération, on passe un message et on améliore les marginales. Si on note x les variables, c les contraintes, $\mu_{x\to c}$ le message envoyé de x à c et $\mu_{c\to x}$ le message de c à x, on a :

$$\begin{aligned} \mu_{x\to c}(v) &= \prod_{c'\in N(x)\setminus\{c\}} \mu_{c'\to x}(v) && \forall v \in D(x) \\ \mu_{c\to x}(v) &= \sum_{\mathbf{v}:\mathbf{v}[x]=v} f_c(\mathbf{v}) \prod_{x'\in N(c)\setminus\{x\}} \mu_{x'\to c}(\mathbf{v}[x']) && \forall v \in D(x) \end{aligned} \tag{2.1}$$

Avec $D(x)$ le domaine de la variable x, $N(x)$ le voisinage de x (*i.e.* les contraintes dans lesquelles x apparaît), $N(c)$ le voisinage de c (*i.e.* les variables qu'elle affecte), $\mathbf{v}$ est un tuple du produit cartésien des domaines de toutes les variables de $N(c)$. $\mathbf{v}$[x] est la valeur prise par la variable x dans $\mathbf{v}$.

f_c est la fonction associée à c : $f_c(\mathbf{v})$ vaut 1 si $\mathbf{v}$ satisfait à la contrainte et 0 sinon. On voit que le message passé des contraintes aux variables est en fait le décompte dont on parlait plus tôt pondéré par le belief tiré des messages reçus par la contrainte *i.e.* le produit dans la deuxième équation des équations 2.1.

On utilise donc ces messages pour calculer nos marginales θ_x pour chaque variable x :

$$\theta_x(v) = \prod_{c \in N(x)} \mu_{c \to x}(v) \qquad \forall v \in D(x)$$

On choisit le nombre d'itérations pour calculer les marginales que l'on fait. Pour en revenir à notre exemple, on regarde la variable a. On a comme marginales initiales $\theta_a^0 = \langle .25,\ .25,\ .25,\ .25 \rangle$. On calcule les marginales locales pour les deux contraintes qui touchent a, $\theta_a^i = \langle .25,\ .25,\ .25,\ .25 \rangle$ et $\theta_a^{ii} = \langle 10/20,\ 6/20,\ 3/20,\ 1/20 \rangle$. Après une itération on obtient donc $\theta_a^1 = \langle .50,\ .30,\ .15,\ .05 \rangle$.

On peut décider de s'arrêter là si on veut des marginales très brouillonnes. Mais si on continue, puisque notre calcul de fréquence de solutions est exact, nous obtiendrons de meilleures marginales. Pour notre exemple, après 10 itérations, on obtient $\theta_a^{10} = \langle .01,\ .52,\ .46,\ .01 \rangle$ ce qui est bien meilleur et très proche des vraies marginales.

Ces marginales sont des informations importantes si on veut choisir par exemple une prochaine étape dans un plan. On peut, dans nos cas de planification, partir d'un plan partiel qui fixe les i premières variables. On donne ce plan partiel à la CP ce qui nous amène à un point précis dans l'arbre de recherche où on ne peut plus faire de «backtracking». Ensuite, on laisse l'algorithme sortir les marginales uniquement sur la prochaine variable dans le plan. On a donc de l'information sur quelle action est la plus prometteuse pour la suite du plan. Expliquons d'abord le fonctionnement d'un agent d'apprentissage par renforcement avant d'expliquer comment cette information pourrait être utilisée.

2.3 Apprentissage par renforcement

L'apprentissage automatique ou «machine learning» consiste en des algorithmes qui apprennent à résoudre des problèmes grâce à de l'expérience. La manière d'obtenir cette expérience varie, mais l'idée est toujours d'utiliser cette expérience pour déduire des généralités sur cet environnement. On veut que l'agent intelligent soit capable à partir de cette expérience de résoudre des problèmes jusqu'ici inconnus [Michalski et al., 2013].

Il existe trois types d'apprentissages automatiques :

— Apprentissage non supervisé : il s'agit d'algorithmes d'analyse ou de regroupement («clustering») de données non étiquetées. Par exemple le k-means [Gentleman et Carey, 2008].

— Apprentissage supervisé : ici les données sont étiquetées par un humain en amont. L'agent apprend de ces données une fonction qui à partir d'un nouveau point de donnée donnera une nouvelle étiquette. Par exemple les k plus proches voisins [Hastie et al., 2009].

— Apprentissage par Renforcement (RL) : ici l'agent n'a pas de jeu de données, mais il interagit avec un environnement. L'agent a pour objectif de maximiser une fonction de récompense en interagissant avec son environnement [Sutton et Barto, 2018].

On s'intéressera ici uniquement à l'apprentissage par renforcement. Un environnement de RL est représenté la plupart du temps comme un processus de décision markovien (MDP) [BELLMAN, 1957]. Un MDP se définit par un tuple $\{S, A, P, r, T, \rho\}$ avec S un ensemble d'états, A un ensemble d'actions, P une probabilité de transition telle que $p(s'|s, a) = P(s', s, a)$, définissant l'effet des actions de l'agent sur l'environnement, r une fonction de récompense qui pour chaque transition donne une récompense scalaire, T un horizon de tâche et ρ la distribution des états initiaux.

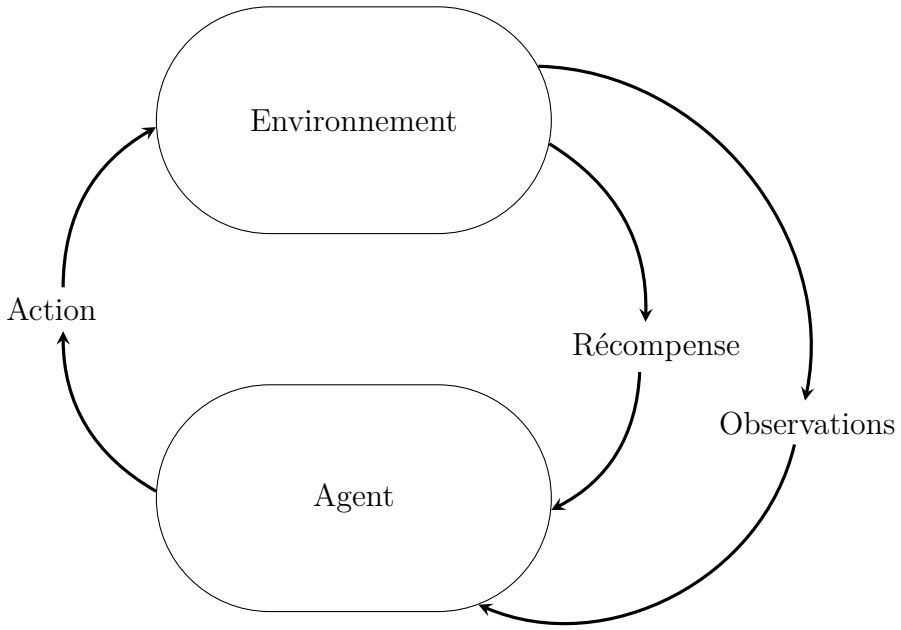

Figure 2.4 Schéma de fonctionnement d'un agent RL

En RL l'agent cherche à apprendre une politique optimale, ou quasi optimale, qui maximise la somme des récompenses obtenues [Sutton et Barto, 2018 3.3]. Une politique relie chaque état à une distribution de probabilité sur la prochaine action, $\pi(a|s)$ est la probabilité de prendre l'action a depuis l'état s. Une politique est optimale lorsqu'en la suivant on obtient le meilleur retour attendu possible à tous les pas de temps (retour attendu : $G_t = \sum_{i=t+1}^{T} r_i$).

Pour ce faire, l'agent RL interagit avec son environnement comme décrit dans la Figure 2.4. Il a trois voies de connexions avec son environnement, deux sont des informations qui lui viennent de l'environnement. D'abord les observations sont tout simplement les observations que l'agent reçoit qui décrivent l'état du monde à l'instant t. La récompense est reçue après avoir accompli une action et est le retour qui permet à l'agent d'apprendre. Enfin, l'agent interagit avec son environnement en agissant dedans et en le changeant.

La politique est ce qui permet à l'agent, à partir d'observations, de choisir l'action à accomplir. Elle peut prendre multiples formes. Dans beaucoup d'agents, elle se présente sous la forme d'une fonction `predict` qui avec en entrée les observations renvoie l'action à prendre. C'est une partie clé du fonctionnement de l'agent avec les fonctions qui servent à entraîner cette politique (souvent `train` ou `learn`) qui elles font usage de la récompense.

On peut séparer les tâches de RL en deux catégories [Sutton et Barto, 2018 3.3] :

- les tâches épisodiques qui ont un début et une fin. L'agent a un objectif à accomplir, quand il y arrive l'épisode s'arrête.
- les tâches continues qui n'ont pas de fin. L'agent a une tâche à accomplir, mais elle n'a pas de point d'arrêt facile à définir.

Les tâches de planification, celles qui nous intéressent ici, sont toujours des tâches épisodiques. Elles ont un début et une fin précis et prédéterminés.

2.3.1 DQN

Le Deep Q-Network (DQN) est un algorithme de Deep RL basé sur le Q-learning introduit par [Mnih et al., 2015]. Dans le Q-learning, on essaie d'apprendre une fonction de valeur notée Q qui associe à toute paire d'état-action une valeur numérique. Cette valeur $Q(s, a)$ représente le gain potentiel lié à faire l'action a depuis l'état s. Ensuite, la politique tirée de cette Q-valeur est simplement de choisir pour chaque état s l'action a telle que $Q(s, a)$ est maximale [Watkins et Dayan, 1992].

On utilise le Deep Q-learning quand on rencontre un problème supplémentaire : la représentation de l'espace des états. On ne peut en général pas apprendre une valeur pour chaque paire d'état-action car l'espace des états est beaucoup trop grand. On entraîne donc un réseau de

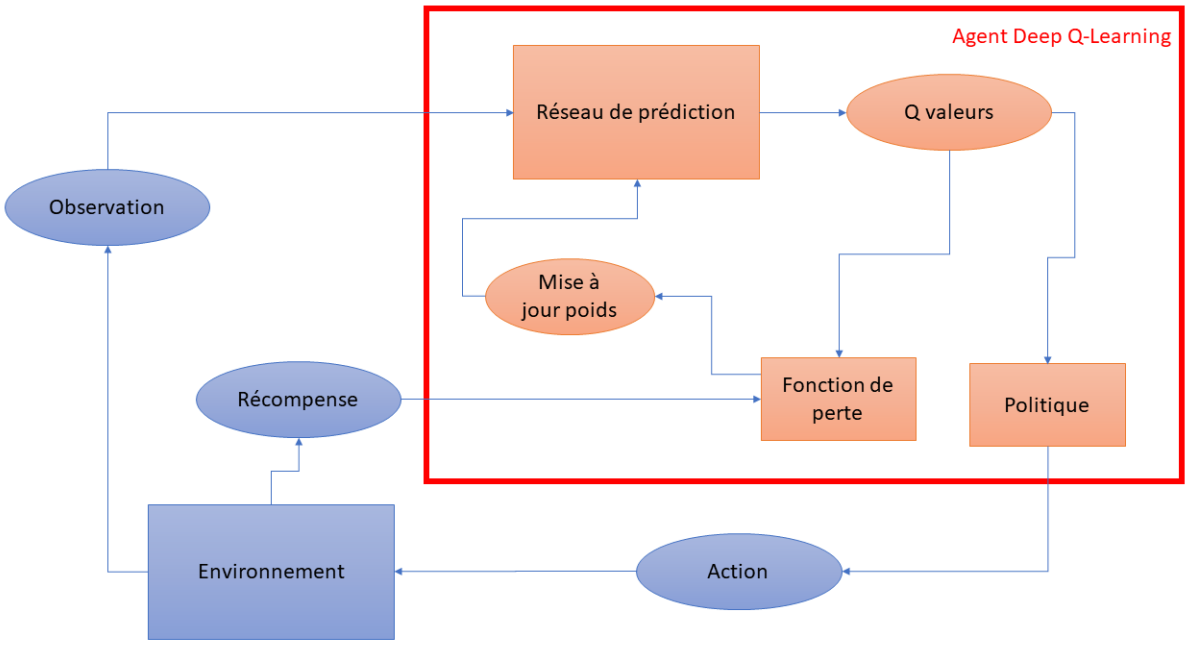

Figure 2.5 Schéma de fonctionnement d'un algorithme de Deep Q-learning

neurones profond qui pour tout état renvoie une Q valeur pour chaque action de l'espace des actions. On a donc un réseau à qui on donne un élément de l'espace des états et qui renvoie une liste de valeurs numériques, une par action [Mnih et al., 2013]. Ces valeurs, une fois le réseau entraîné, sont les Q-valeurs et donnent le gain potentiel lié à l'action. On utilise ces Q valeurs pour construire la politique de l'agent avec laquelle il choisit ses actions.

Tout ce processus se passe dans la méthode `predict()` à qui on donne une observation et qui, à l'aide du réseau de neurones, renvoie l'action choisie. Ces Q valeurs et la récompense obtenue en prenant l'action prédite sont utilisées dans une fonction de perte pour mettre à jour les poids du réseau de neurones. Le fonctionnement est décrit dans le schéma en figure 2.5.

Grâce aux réseaux de neurones profonds, on peut donc représenter des espaces des états trop complexes ou étendus pour être entièrement explorés par l'agent. On peut associer une valeur à une paire état-action même si elle n'a jamais été visitée, ce que l'on ne pouvait pas faire avec du Q-learning classique. Cependant, l'usage d'un réseau de neurones avec le Q-learning peut entraîner des instabilités [Hasselt et al., 2016]. Pour pallier ces problèmes, le DQN utilise deux techniques.

D'abord la relecture d'expérience introduite par [Schaul et al., 2015]. Chaque nouveau point d'expérience $e_t = (s_t, a_t, r_t, s_{t+1})$ est enregistré dans un tampon qui garde de la mémoire sur

de nombreux épisodes appelé le "replay buffer". On calcule les mises à jour du réseau en échantillonnant des transitions dans ce tampon. La mémoire tampon est limitée et les points arrivés les premiers sont remplacés en premier. Cela permet de réutiliser les expériences plusieurs fois, ce qui est plus efficace. Mais surtout l'intérêt est de briser les corrélations trop importantes entre nos expériences récentes. Lorsque les transitions utilisées pour l'entraînement sont corrélées, on risque de faire diverger ou osciller notre fonction qui donne les Q valeurs. Donc si on échantillonne dans un grand ensemble de transitions qui ne vient pas du même moment, on peut briser cette corrélation.

La deuxième technique est l'usage d'un réseau cible introduit par [Hasselt et al., 2016], on parle de double Q-learning. On a en fait deux réseaux de neurones dans l'algorithme, ils sont identiques et l'on recopie les paramètres du réseau de prédiction dans le réseau cible tous les τ pas de temps. Il s'agit ici aussi d'améliorer la stabilité de l'entraînement. On utilise les Q valeurs du réseau cible pour entraîner le réseau de prédiction. Ainsi les valeurs qu'on utilise pour l'entraînement ne sont mises à jour que tous les τ pas de temps et restent plus stables, une mauvaise mise à jour pourra se perdre dans la masse.

2.3.2 RL pour la planification IA

L'usage d'apprentissage pour la planification n'est pas une nouvelle chose [Fern et al., 2011]. Il s'agit souvent de réduire la taille de l'arbre de recherche pour accélérer le processus. On utilise des algorithmes classiques de planification que l'on renforce à l'aide d'apprentissage pour qu'ils puissent prendre des décisions éclairées par leur expérience.

Dans les dernières années, le deep RL, qui utilise des réseaux de neurones profonds, a obtenu des résultats prometteurs dans de nombreux domaines. On citera notamment Alpha Zero [Silver et al., 2017] de DeepMind qui découvre automatiquement la structure dans le domaine des jeux à deux joueurs comme les échecs ou le jeu de Go. La clé de sa puissance est de jouer contre soi-même. Au départ, l'agent est mauvais, mais au fur et à mesure de son entraînement contre lui-même il apprend et devient meilleur.

Cette méthode a été adaptée récemment pour les domaines de planning par [Feng et al., 2020] sous le nom de curriculum driven learning. Dans leur article, [Feng et al., 2020] travaillent en particulier sur Sokoban, un jeu vidéo de réflexion sorti en 82 où il faut ranger des boîtes dans un entrepôt où il est difficile de manoeuvrer. C'est un problème PSPACE complet de planification qui reste, pour les niveaux difficiles, hors de portée des planificateurs classiques même spécialisés. Les jeux vidéo à concept simples, en général, se prêtent bien à la RL. En effet, le score est une récompense naturelle et l'ensemble des actions disponibles est souvent réduit.

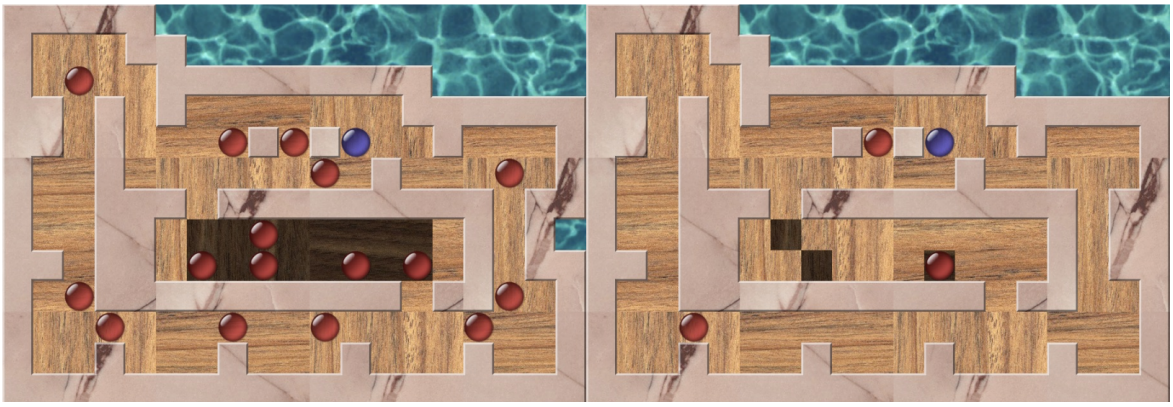

Figure 2.6 Exemple d'une instance de Sokoban (à gauche) et une version fortement simplifiée à trois boîtes (à droite). Les boules rouges sont des boîtes, la boule bleue l'agent. Les cases les plus claires sont des murs et les plus foncées des cases objectifs pour les boîtes. L'agent doit pousser les boîtes sur les cases objectifs

Ils expliquent qu'on ne peut pas lancer un agent incompétent directement sur les problèmes difficiles car il ne trouverait jamais l'état final en tâtonnant. Il y a trop de boîtes à ranger et trop de possibilités de se coincer pour y arriver au hasard. On prépare donc une suite de versions du problème semblables et de difficultés croissantes en commençant par des versions très faciles (c.f. Figure 2.6 pour un exemple d'instance fortement simplifiée). Il faut aussi s'assurer que ces problèmes simplifiés sont toujours faisables. Sur les problèmes les plus faciles, un agent non entraîné peut réussir. Une fois qu'il devient bon sur les versions faciles, il peut trouver les solutions pour les problèmes un peu plus difficiles et ainsi de suite. Cela implique que l'apprentissage sur les versions faciles est utile quand le problème se complexifie. La similarité des versions est donc très importante. Cependant, au fur et à mesure que leur agent s'améliore sur les versions difficiles il devient incapable de résoudre les versions faciles qui sortent de son expertise.

Cette méthode permet d'atteindre les performances de l'état de l'art dans de nombreux domaines sans utiliser de connaissances spécifiques au domaine ni de solutions données manuellement par l'utilisateur durant l'entraînement. Dans ce cas précis, elle obtient même des performances largement au-dessus de l'état de l'art sur les niveaux les plus difficiles.

Récemment, [Lee et al., 2022] ont eux proposé une méthode utilisant le AI planning et la RL ensemble pour résoudre des problèmes de RL hiérarchique (HRL). En RL, on souffre du fléau de la dimension. On essaie donc de trouver des relaxations de la dimension de l'espace des états ou des actions, c'est la HRL [Barto et Mahadevan, 2003]. On peut par exemple faire usage d'abstraction temporelle. Au lieu de prendre une décision à chaque pas, on appelle des sous-tâches étendues dans le temps qui utilisent leurs propres stratégies.

Ces sous tâches devraient être plus simples à résoudre que la tâche complète dans un espace des états local. C'est de là que vient le mot hiérarchique de la HRL. Il est naturel pour utiliser ce genre d'abstraction d'avoir une hiérarchie d'agents avec des sous-agents pour les sous-tâches.

Dans leur article, [Lee et al., 2022] proposent d'utiliser le AI planning pour déterminer ces sous-tâches. On définit côte à côte le problème en MDP et en problème de planification et on crée une fonction de transition capable de relier les états de ces deux représentations. La définition en problème de planification induit des contraintes qui ne sont pas représentées dans le MDP. C'est donc pour introduire ces contraintes dans le problèmes qu'on définit les sous tâches et donc les sous-MDP reliés. Grâce à cette technique, [Lee et al., 2022] obtiennent de bonnes performances et un échantillonage bien plus efficace que celui de l'état de l'art.

Maintenant que nous avons établi les bases nécessaires, nous allons parler du contenu de la recherche en elle-même. Nous détaillerons les expériences menées et les métriques utilisées.

CHAPITRE 3 MÉTHODOLOGIE

On cherche à savoir si l'utilisation de la programmation par contraintes et en particulier d'une version qui permet d'obtenir des marginales sur l'action à venir permet d'améliorer les performances d'un agent de RL classique, le DQN. On décrit ici exactement les expériences qui ont été menées. On explicite également à quel endroit dans l'architecture exactement les marginales ont été injectées.

3.1 Environnement RL

L'environnement utilisé est dérivé de la tâche de planification IA Floortile[1] . On construit cet environnement sur le OpenAI Gym [Brockman et al., 2016], ce qui nous permet de facilement créer un environnement capable d'interagir avec un agent de RL. Le Gym est une API standard pour l'apprentissage par renforcement, et une collection variée d'environnements de référence. Il permet notamment de créer son propre environnement à partir d'un modèle pour qu'il soit facilement utilisable avec toutes sortes d'agents classiques de RL.

Par rapport à la description de cette tâche donnée à la section 2.2.1, on s'intéresse à un seul robot ne portant qu'une seule couleur. Notre robot a donc accès à un ensemble de six actions : il peut bouger dans les quatre directions et peindre au-dessus ou au-dessous de lui. On a l'ensemble d'actions $\{paint\_up, paint\_down, move\_up, move\_down, move\_right, move\_left\}$. L'espace des observations est différent pour les différents agents.

Le robot ne peut pas quitter le sol dallé ni rouler sur une dalle qui a déjà été peinte. Si l'agent essaie de quitter la grille, l'action ne fait rien. Il ne peut peindre que des dalles qui ne sont pas déjà peintes. Puisqu'il s'agit d'une tâche de planification classique, on connaît l'état initial du monde en entier. En particulier la position initiale du robot et les dalles à peindre.

On construit la fonction de récompense pour que la tâche soit épisodique et qu'elle se termine quoiqu'il arrive. On s'inspire de la récompense d'une autre tâche impliquant un agent se déplaçant sur une grille, soit FrozenLake-v0[2] . Les actions qui mettent fin à l'épisode par une «erreur» donnent une récompense de -100 : cela comprend peindre une dalle qui ne devait pas être peinte et rouler sur une dalle déjà peinte. Toutes les actions de mouvement sauf celles qui terminent l'épisode ont une récompense de -5, qu'elles aboutissent à un mouvement ou pas (les mouvements qui mettent fin sont les erreurs qui obtiennent -100).

1. `http://www.plg.inf.uc3m.es/ipc2011-deterministic/DomainsSequential.html`
2. `https://gym.openai.com/envs/FrozenLake-v0/`

Les actions de peinture sur une case à peindre donnent une récompense de 10. On veut s'assurer que la tâche se termine, donc on ajoute une condition d'arrêt que l'on est certain d'atteindre. Après 50 actions, si l'épisode n'est pas terminé, on met fin à l'épisode avec une récompense de -50.

Pour évaluer nos agents, on les entraîne sur plusieurs exemplaires du problème de complexité croissante puis on teste leur capacité à les résoudre. La mesure de complexité de ces exemplaires est la longueur du plan optimal. Autrement dit, quel est le nombre minimal d'actions qu'il faut prendre pour résoudre le problème ? Par exemple, notre exemplaire le plus simple (vu en figure 3.1) a une complexité de 2.

Son plan optimal est *move_down* → *paint_down*.

2

R		
P		

Figure 3.1 Exemplaire le plus simple avec une complexité de 2. R est la position de départ du robot et P la dalle à peindre.

Les exemplaires sont tous relativement similaires. Pour augmenter la complexité d'une tâche, on ajoute des cases à peindre ou on les éloigne de la position de départ du robot ou on agrandit la grille. Tous les autres exemplaires sont représentés dans la Figure 3.2.

3

R		
	P	

4

R		
P	P	

5

R		
P		P

6

R		
P	P	P

7

R			
P			P

8

R			P
		P	

9

R			P
P		P	

10

R			P
P		P	P

Figure 3.2 Exemplaires de complexité 3 à 10 (de gauche à droite) utilisés pour évaluer nos agents. R est la position de départ du robot et P la dalle à peindre.

3.2 Environnement CP

Pour le modèle CP du problème, on utilise la représentation décrite dans [Babaki et al., 2020]. On a un automate par dalle, un automate par robot et une contrainte REGULAR par automate. Ces éléments suffisent pour avoir une représentation complète du problème à tout instant. L'automate de dalle a été donné plus tôt dans la partie 2.2.1. L'automate de robot pour la grille la plus simple de taille 3x3 est donné en figure 3.3.

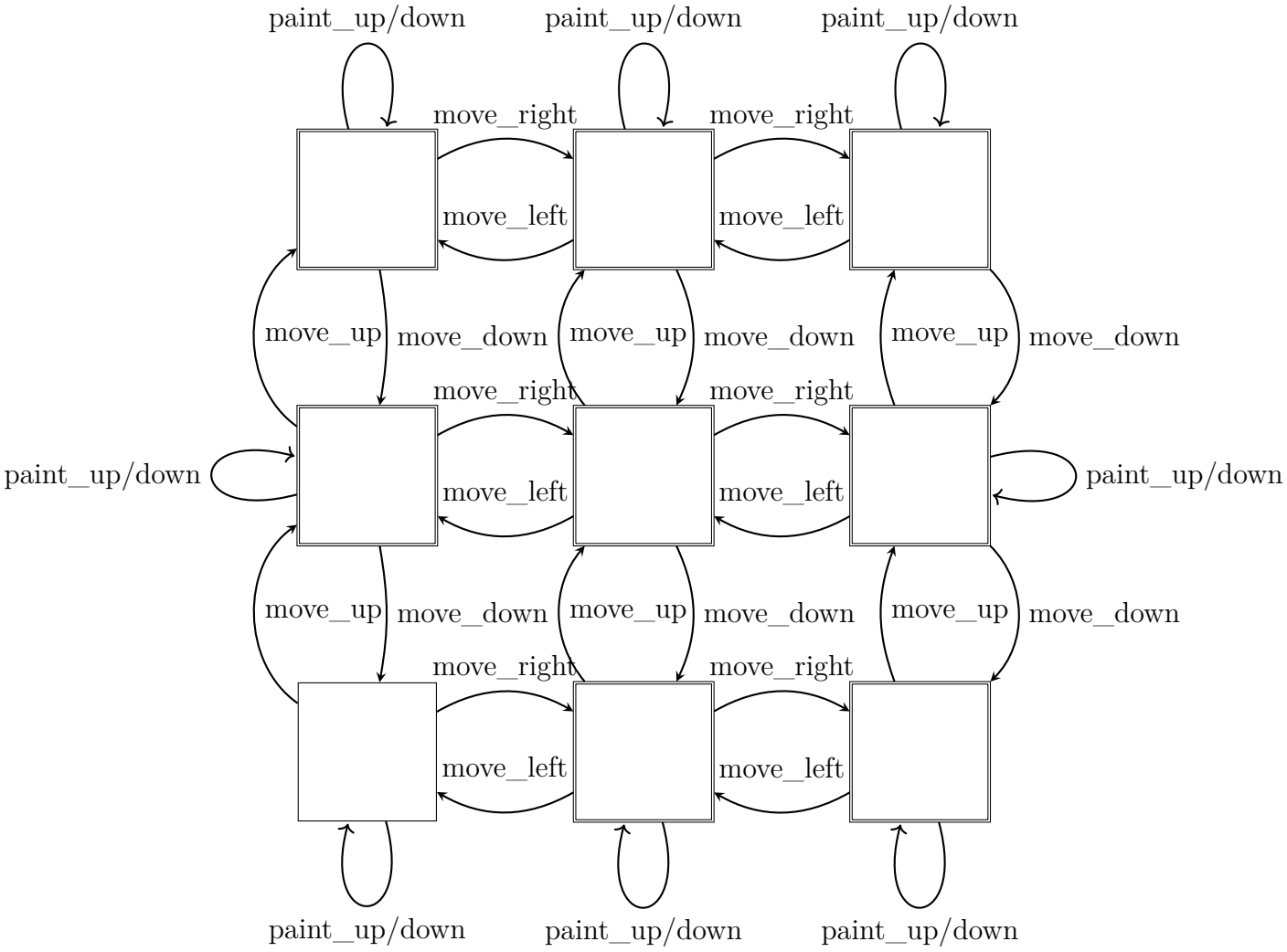

Figure 3.3 Automate d'un robot pour la grille de complexité 2 de la tâche Floortile, les actions de peinture ont été omises pour la dalle centrale

On a donc un nombre fixe de variables lié à la longueur du plan recherché, une variable par action dans le plan. Ces variables ont toutes le même domaine, toutes les actions possibles dans l'environnement. Ce nombre d'actions dépend de la grille en elle-même. Pour les grilles

de taille 3x3, cela donne 36 actions possibles pour le robot. On a donc un nombre n de variables qui doivent prendre une valeur entre 1 et 36. Ce plan doit satisfaire simultanément toutes les contraintes REGULAR liées aux automates.

Comme solveur de Programmation par contraintes on utilise MiniCPBP[3] . MiniCPBP est un solveur de programmation par contraintes basé sur minicp, un solveur open source et tourné vers la lisibilité. Le fonctionnement de minicp est altéré pour ajouter le belief propagation expliqué plus tôt. Le solveur permet entre autres de donner un nombre fixe de variables dont on connaît la valeur et d'obtenir en retour les marginales sur le prochain choix de valeur.

L'algorithme a besoin pour fonctionner de trois informations. D'abord, la disposition de l'environnement qui lui est donné par un fichier contenant tous les automates. Le fichier est produit automatiquement à partir d'un fichier d'entrée simple où l'utilisateur décrit la forme de la grille, les cases à peindre et le point de départ du robot.

1	2	3
4	5	6
7	8	9

pair	1/4	2/5	3/6	4/7	5/8	6/9
paint up	1	5	9	13	17	21
paint down	2	6	10	14	18	22
move up	3	7	11	15	19	23
move down	4	8	12	16	20	24

pair	1/2	2/3	4/5	5/6	7/8	8/9
move right	25	27	29	31	33	35
move left	26	28	30	32	34	36

Tableau 3.1 Numéro des actions pour la grille de taille 3x3. Les numéros des cases sont indiqués dans la grille à gauche

Ensuite, il faut un plan partiel. Il s'agit d'un autre fichier qui contient une suite de nombres qui représentent les actions déjà prises par le robot. Les actions sont numérotées de façon unique selon la paire de cases où elles sont prises et leur nature en elle-même. L'ordre dans lequel les paires de cases sont considérées dépend entièrement du fichier d'entrée de l'utilisateur. Si on prend pour exemple la grille en Figure 3.1, les actions sont numérotées comme vu dans le Tableau 3.1 : on parcourt toutes les actions liées à une paire de cases avant de passer à la suivante.

La dernière information dont le modèle CP a besoin est la longueur de plan. Lors d'une résolution normale, le modèle part d'une longueur de plan minimale puis itère vers des longueurs

3. `https://github.com/PesantGilles/MiniCPBP`

de plus en plus grandes jusqu'à trouver le plan optimal. Ici on ne vérifie la faisabilité qu'avec une longueur à la fois.

Pour résumer, après chaque action, l'agent donne au CP-BP un plan partiel (les i premières actions) et une longueur de plan complet et reçoit en retour des probabilités. Ce sont les probabilités pour chaque action que cette action mène à un plan complet de la longueur donnée. Si par exemple on donne comme plan partiel $\{25, 27, 28, 8\}$ ce qui correspond à {move_right, move_right, move_left, move_down} pour la grille de complexité 4, on n'obtient pas de solution pour toutes les longueurs inférieures à 7. Pour une longueur de 7, on obtient :

no d'action	17	18	7	20	31	30
action	paint_up	paint_down	move_up	move_down	move_right	move_left
marginales	0	0.714	0.106	0.004	0.083	0.092

Tableau 3.2 Marginales obtenues à partir du plan partiel {move_right, move_right, move_left, move_down} pour une longueur de 7

La seule action interdite est donc paint_up. On voit sur la figure 3.4 l'état actuel de la grille ; l'unique plan optimal restant est {paint_down, move_left, paint_down}. La marginale la plus haute est de loin celle de l'action paint_down ce qui paraît logique. Les marginales poussent donc dans la bonne direction, mais autorisent des actions sous-optimales comme move_up qui a la deuxième plus haute probabilité et n'a aucun intérêt.

	R	
P	P	

Figure 3.4 État de la grille de complexité 4 après le plan partiel {move_right, move_right, move_left, move_down}

Lors du fonctionnement avec l'agent RL, on démarre un nouvel épisode en donnant la longueur optimale à la CP. À chaque fois que la CP renvoie qu'il n'y a pas de solution pour cette longueur, on augmente de 1 et on garde cette nouvelle longueur en mémoire.

3.3 Agents

3.3.1 Agent sans CP

Notre premier agent est un agent de RL pur qui n'utilise pas la CP. Il utilise l'algorithme DQN (voir la partie 2.3.1 pour l'explication détaillée) dont l'implémentation est tirée de stable-

baselines3 [Raffin et al., 2021] (version 1.3.0). Son espace des observations est un tenseur de même format que la grille avec une profondeur de 3. La première couche est une grille de 0 avec un 1 à la position du robot. La seconde couche est remplie de 0 sauf à l'emplacement des cases à peindre où il y a des 1. La troisième couche représente les cases qui ont déjà été peintes en y mettant des 1 et garde des 0 dans les autres cases. Le reste de l'environnement (*i.e.* la récompense et l'espace des actions) est tel que décrit plus tôt.

Pour les hyperparamètres, une recherche complètement exhaustive a été impossible car elle prenait trop de temps. Nous avons testé les paramètres qui régissent l'exploration pendant l'entraînement (par tranches de 0.1), la valeur target_update_interval qui est le nombre de pas de temps avant de recopier dans le réseau cible (testé 1000, 5000, 10 000, 20 000, 50 000), la valeur tau qui régit la force des mises à jour sur le réseau (1 est une mise à jour normale, sinon on ne fait qu'une fraction et on a testé par tranches de 0.25) et leaning_starts qui donne le nombre d'actions aléatoires au début de l'entraînement avant de commencer à utiliser la fonction `predict`. Learning_starts est particulier, pour l'agent sans CP on gardé la valeur par défaut. On a seulement voulu tester des valeurs par incréments de 1000 entre 0 et 10 000 car certains de nos agents ont besoin de moins de 10 000 pas d'entraînement et on voulait savoir si l'usage d'actions aléatoires uniquement était mauvais pour les performances. Ce n'est pas le cas donc on garde la valeur par défaut partout.

Ainsi, on trouve que la combinaison d'hyperparamètres qui fonctionne le mieux pour notre agent sans CP est {exploration_final_eps=0.5, exploration_initial_eps=1.0, exploration_fraction=0.8, target_update_interval= 10 000, tau=1, learning_starts=50 000}. Les autres hyperparamètres sont laissés à leur valeur par défaut. Ces hyperparamètres sont gardés pour tous les agents pour s'assurer que la variation de performances ne vient pas de là.

L'architecture des deux réseaux de neurones est laissée à la valeur par défaut. Il s'agit d'un perceptron à 5 couches, on alterne des couches entièrement connectées et des couches avec la fonction d'activation ReLU comme représenté en figure 3.5. On récupère les observations et on les aplatit en un seul vecteur de longueur qui dépend de la taille de la grille. Ce vecteur est l'entrée de la première couche du perceptron. La dernière couche donne le vecteur de Q valeurs qui a la même taille que l'espace des actions.

Pour entraîner les poids, on suit l'algorithme 3 tiré de [Mnih et al., 2015].

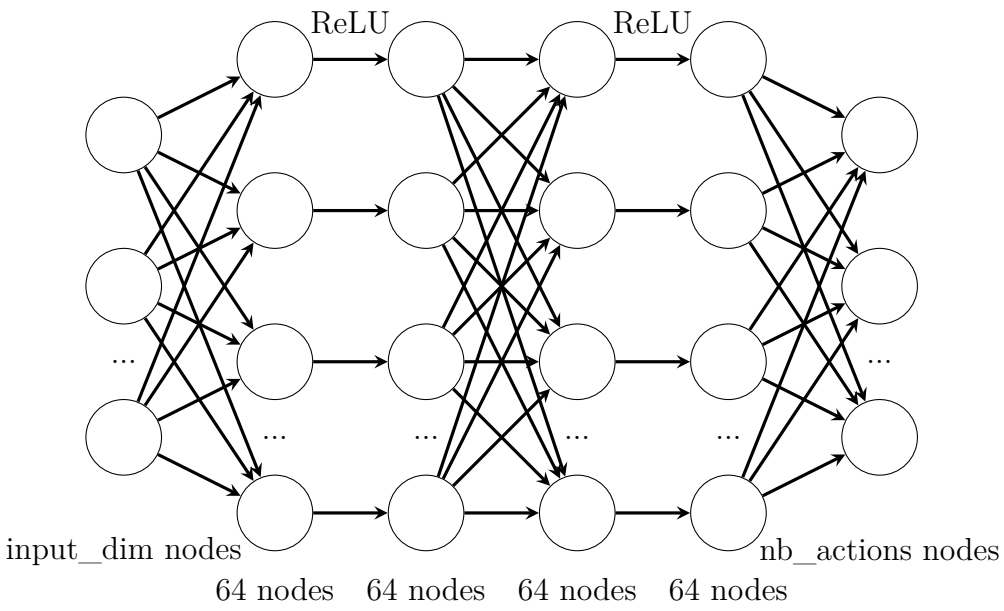

Figure 3.5 Architecture des réseaux de neurones du DQN

3.3.2 Agents avec CP

Nous avons identifié trois endroits où il pourrait être intéressant d'ajouter l'information obtenue à l'aide de la CP. Si on regarde la Figure 2.4, on voit qu'il y a trois points d'interaction entre l'agent et son environnement :

— les observations ;
— la récompense ;
— les actions/les Q valeurs.

C'est à ces trois points de contact que l'on essaie d'introduire de l'information venue de la CP sous plusieurs formes. Cela nous donne quatre types d'agents dont il faut évaluer la performance, un pour les observations, un pour la récompense et deux pour les Q valeurs. Ces trois types d'agents utilisent la CP pendant la phase d'entraînement de l'agent et ensuite pendant l'inférence. Les agents avec l'information dans les observations et la récompense utilisent toujours la version CP-BP avec les marginales alors que l'agent avec la CP qui intervient dans le choix d'action a une version avec et sans marginales. On utilisera comme témoin l'agent sans CP.

Algorithm 3 Algorithme d'entraînement du DQN

Initialisation de la mémoire de relecture D à capacité N
Initialisation de la fonction de valeur-action Q avec des poids aléatoires θ
for épisode= 1, M **do**
  Prendre l'état de départ $s_1 = \{x_1\}$ et la séquence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    Avec une probabilité ϵ, prendre l'action aléatoire a_t
    Sinon, prendre $a_t = max_a Q * (s_t, a; \theta)$
    Faire action a_t et obtenir récompense r_t et observation x_{t+1}
    $s_{t+1} = s_t, a_t, x_{t+1}$ et $\phi_{t+1} = \phi(s_{t+1})$ et enregistrer $(\phi_t, a_t, r_t, \phi_{t+1})$ dans D
    Échantillonner un minibatch aléatoire de transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ dans D
    Définir $y_j = \begin{cases} r_j & \text{si } \phi_{j+1} \text{ terminal} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{sinon} \end{cases}$
    Effectuer un pas de descente de gradient sur $(y_j - Q(\phi_j, a_j; \theta))^2$ pour améliorer θ

Agent avec marginales dans les observations

Pour cet agent, ce sont les observations qui sont changées (cf. Figure 3.6). On doit mettre dans les observations un toutes les informations sur la grille ainsi que les marginales obtenues par la CP. Il est impossible de mettre toutes ces informations dans un format qui leur permet de rentrer toutes dans un même tableau. Ces observations se présentent donc sous la forme d'un dictionnaire qui contient quatre éléments : un tableau `position`, un tableau `to_be_painted`, un tableau `painted` et une liste `marginals`. Le premier tableau est de même format que la grille et ne contient que des 0 sauf un 1 à la position du robot. De la même façon, le second tableau a des 1 sur les cases à peindre et le troisième tableau des sur les cases qui ont été peintes. Le dernier élément est obtenu grâce à la CP avec belief propagation (CP-BP). L'opération du DQN en lui-même n'est pas du tout altérée.

Agents avec marginales dans la récompense

Cet agent est très proche du précédent. Au lieu de transmettre l'information donnée par les marginales dans les observations, on les donne a posteriori dans la récompense (cf. Figure 3.7). Les observations sont donc uniquement le tenseur décrit plus tôt. Après chaque action, l'agent donne au CP-BP les informations nécessaires pour obtenir la probabilité de l'action qui vient d'être prise de mener à un plan complet. On n'utilise donc qu'une des marginales produites par le CP-BP.

Selon cette probabilité, on modifie la récompense comme ceci : $R'_t = R_t + \alpha \times \ln(marginal(a_t|s_t))$. $\alpha \in [0, 10]$ est un coefficient à déterminer qui donne le poids des marginales sur la récompense.

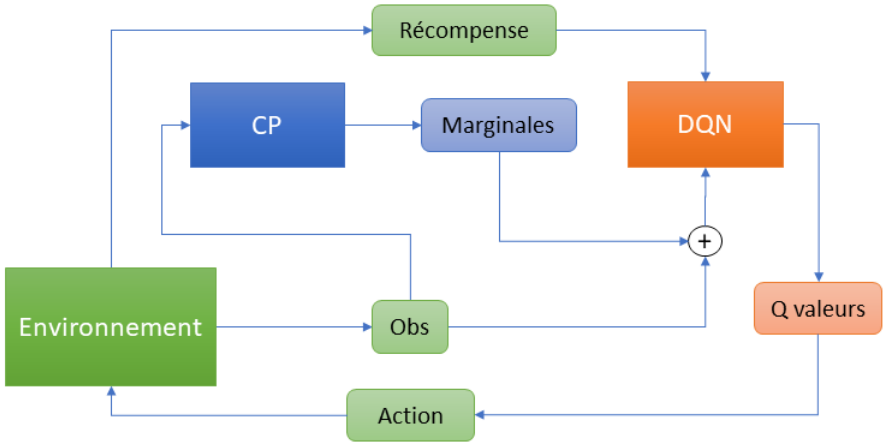

Figure 3.6 Schéma de fonctionnement de l'agent avec marginales dans les observations

Pour éviter de devoir calculer le logarithme d'une marginale nulle, on choisit une valeur ϵ en dessous de laquelle on fixe la valeur de $\ln(marginal(a_t|s_t)$ à $\ln(\epsilon)$. On fixe ϵ à 0.001 ce qui donne $\ln(\epsilon) \approx -7$. Le fonctionnement du DQN n'est pas modifié.

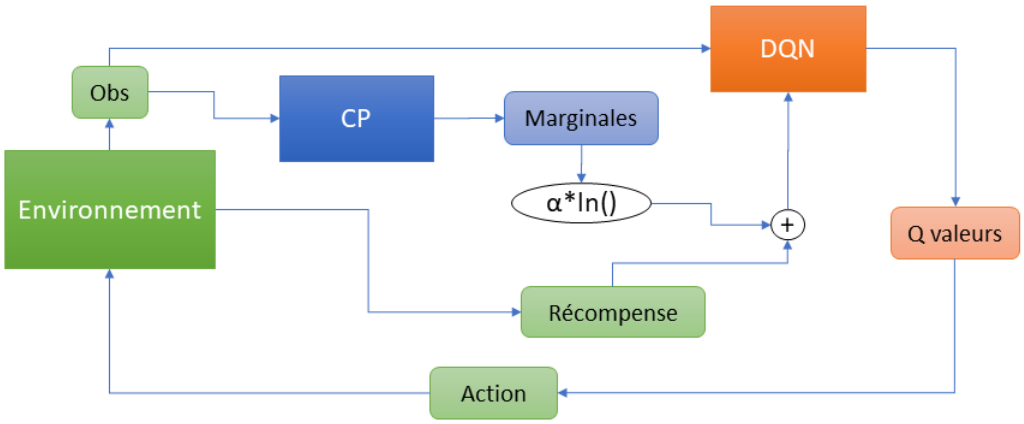

Figure 3.7 Schéma de fonctionnement de l'agent avec marginales dans la récompense

Agent avec CP dans le processus de prédiction

Cet agent utilise comme observation le même tenseur que décrit plus haut. Le reste de l'environnement est le même que décrit au début de la partie.

On a deux versions de cet agent. La première n'utilise que le modèle CP sans belief propagation (noté agent CP plus loin). On utilise la CP pour dégager des actions «interdites». Ces actions sont la plupart du temps les actions qui mettraient fin à l'épisode avec une mauvaise

récompense. On procède à cette vérification dans la méthode `predict()` du DQN. Dès que l'agent doit choisir une action à partir d'une observation, c'est la méthode `predict()` qui est appelée. Si les Q valeurs de l'agent pointent vers une action interdite par la CP, on prend à la place la première action (dans l'ordre de l'espace des actions) qui est autorisée.

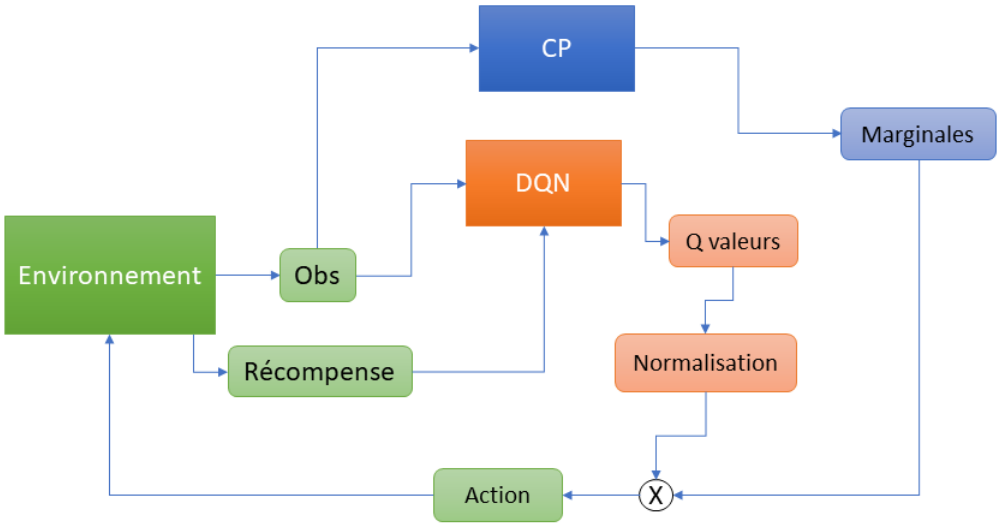

Figure 3.8 Schéma de fonctionnement de la deuxième version de l'agent avec marginales dans le processus de prédiction

La seconde version utilise la CP-BP (noté agent CP-BP plus loin). On dégage aussi des actions «interdites», celles dont la marginale est égale à 0. Lorsque l'agent doit prédire la prochaine action à l'aide du réseau de neurones, on procède en deux étapes comme illustrées dans la Figure 3.8.

D'abord on récupère les Q valeurs liées à toutes les actions et on leur soustrait la Q valeur minimale. On obtient donc des valeurs uniquement positives ou nulles. Ensuite, on multiplie ces nouvelles valeurs par les marginales associées aux actions. On choisit l'action qui a la valeur finale maximale. On a besoin de rendre les Q valeurs positives pour que toutes les valeurs du produit le soient. Toutes nos marginales sont positives. Plus une marginale est grande plus une action a un bon potentiel. Donc le produit permet de pousser en avant les actions qui ont une bonne marginale. En revanche, si on faisait le produit avec des Q valeurs négatives on pourrait avoir des actions à grandes marginales qui ne seraient jamais choisies car le produit serait très négatif donc désavantagé. Puisque l'on fait un produit, les actions qui ont une marginale de 0 ou la pire Q valeur ne peuvent être choisies.

3.4 Mise en oeuvre

Pour récupérer les marginales, on utilise miniCPBP qui est codé en java. Le langage le plus utilisé et donc le plus pratique pour faire du RL est le python. La totalité de l'état de l'art en RL pour le planning est en python et utilise le OpenAI gym pour ses environnements. On a besoin que ces deux langages communiquent entre eux et se passent des informations. C'est le python qui détient le plan partiel qui permet à miniCPBP de calculer ses marginales. C'est le java qui calcule les marginales dont le python a besoin pour avancer. Pour ce projet, la communication se fait par fichier texte interposé, ce qui ralentit forcément la manoeuvre.

Le python écrit ses plans partiels dans des fichiers .dat et le java écrit les marginales obtenues dans un fichier .txt. De plus, les automates pour toutes les grilles produits par du code Python sont eux aussi enregistrés dans des fichiers texte mis à disposition du java qui les relit à chaque fois pour construire la représentation du problème. Si on reconstruisait miniCPBP en python, il est probable que l'on puisse faire accélérer nos agents qui l'utilisent.

CHAPITRE 4 RÉSULTATS ET DISCUSSION

Dans ce chapitre, on présente les résultats de l'évaluation des différents agents décrits à la section 3.3. On commencera par observer les performances brutes des différents types d'agents puis on s'intéressera plus en détail aux agents qui fonctionnent bien. Enfin on discutera de la pertinence de ces résultats et de comment les améliorer.

On cherche à savoir si l'ajout d'information issue de la CP peut améliorer un agent de RL classique. On lui met cette information à disposition de plusieurs manières et on regarde si sa performance, sa vitesse d'entraînement et sa capacité à généraliser sont améliorées. Une fois entraîné, est-il capable de se passer de la CP ?

4.1 Résultats

Les expériences pour les agents qui utilisent la CP dans la récompense et dans les observations sont menées sur la branche Cedar de Compute Canada. Les commandes SLURM utilisées sont en Annexe A. Les expériences pour les autres agents sont menées sur un ordinateur portable avec un CPU AMD Ryzen 5 5600H 3,30 GHz, 16 GB de RAM et un GPU NVIDIA GeForce RTX 3060. Ces expériences ont été menées en local pour avoir plus facilement accès aux modèles une fois entraînés. On fait également toujours tourner l'agent témoin sur la même machine que l'agent évalué.

Pour mesurer la performance de nos agents, on les fait tous tourner sur les mêmes environnements. On utilise 9 environnements de complexité croissante. On fait faire à l'agent 1000 épisodes sur chaque environnement et on regarde la proportion de ces épisodes où notre agent a réussi la tâche avec succès, c'est-à-dire qu'il a peint toutes les cases qui devaient être peintes. Puisque certains agents n'ont pas une performance très stable, dans ce cas-là, on entraîne 5 instanciations de cet agent et on les évalue toutes sur 1000 épisodes. La valeur reportée dans les graphes est une moyenne des valeurs de performance.

Pour mesurer la capacité de transfert de l'apprentissage de nos agents, on fait deux expériences. On commence par complètement entraîner chacun de nos agents sur un exemplaire en particulier et enregistrer les poids du réseau qui ont été appris. La première expérience consiste à regarder la capacité de nos agents à résoudre, sans changer les poids, un autre exemplaire similaire. On utilise encore une fois la même métrique.

Pour nos agents avec la CP dans le processus de décision, on veut savoir si le réseau de neurones entraînés avec la CP est toujours capable de prendre de bonnes décisions lorsque la CP est retirée. La seconde expérience consiste donc cette fois-ci à garder le même exemplaire que celui sur lequel les poids ont été entraînés et à regarder si un agent DQN normal sans support de la CP pouvait faire bon usage des poids entraînés.

4.1.1 Performances brutes

Pour l'agent avec les marginales dans les observations, on donne autant d'itérations d'entraînement que pour l'agent sans marginales. Puisque le calcul des marginales à chaque pas prend du temps on entraîne les agents pendant 24h, ce qui donne à peu près 300 000 pas de temps. Comme vu sur la figure 4.1, les performances de cet agent ne décollent pas lors de ces 300 000 itérations : il semble que l'agent n'arrive pas à utiliser l'information qui lui est fournie. On notera qu'il faut environ 20 minutes pour entraîner l'agent témoin sur ces mêmes grilles et pour obtenir des performances bien meilleures. On a également essayé de donner comme observations uniquement les marginales, mais les résultats ne sont pas meilleurs. Pourtant les marginales proposées sont bonnes même après beaucoup de mouvements sans intérêt. L'information est bonne, mais l'agent n'arrive pas à l'interpréter. Il reste donc coincé dans des épisodes sans intérêt qui terminent très vite sur une action de peinture au mauvais endroit. Un épisode typique est move_up, paint_down. On voit bien que ce qui a été appris n'est pas très intéressant.

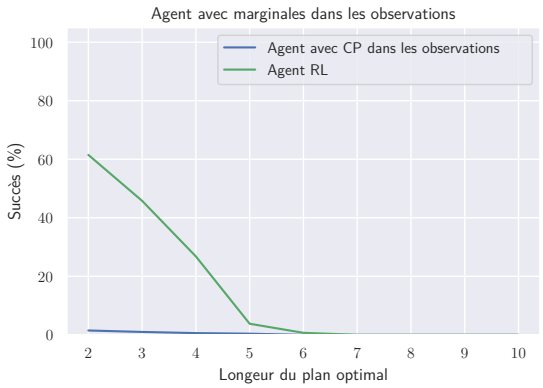

Figure 4.1 Performances de l'agent avec les marginales dans les observations comparées à celles d'un agent témoin.

Pour l'agent avec les marginales dans la récompense, on procède de la même façon. 300 000 itérations seulement en à peu près 24h pour l'entraînement de tous les agents. On teste différentes valeurs de α entre 0 et 10, 0 étant l'agent témoin puisqu'il ne change rien aux

récompenses. On observe en Figure 4.2 que la performance des différents agents n'excède jamais la performance de l'agent témoin de beaucoup. En revanche, lorsque α prend une valeur trop grande, les performances sont franchement dégradées. L'information fournie à cet endroit-là n'aide pas ou peu l'agent à s'améliorer.

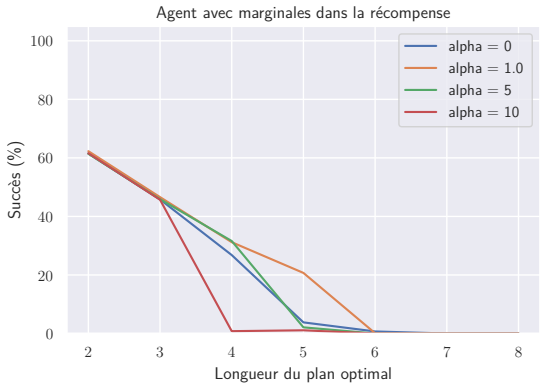

Figure 4.2 Performances de l'agent avec marginales dans la récompense pour différentes valeurs de α comparées à celles d'un agent témoin.

Pourtant encore une fois les marginales restent bonnes tout au long de l'épisode. L'agent a pourtant toujours tendance à faire des actions de mouvement inutiles vers les murs. Un épisode typique pour la grille 6 est :

Mouvement	m right	m right	m down	m right	m right	m right	p down	p down
Récompense	-5.9	-6.9	-5.5	-12	-12	-12	9.5	-100

Les récompenses sont correctes et semblent pousser dans la bonne direction. Les actions inutiles sont bien découragées. On dirait que l'agent n'a pas encore saisi les subtilités de quand il doit peindre et quand il doit s'abstenir et il se déplace un peu sans but. C'est le même genre d'épisode typique que l'agent sans CP, donc on dirait qu'il ne prend pas trop en compte l'information portée par la CP.

Les deux agents qui utilisent la CP dans le processus de décision nécessitent beaucoup moins de pas de temps pour être entraînés. Puisque les actions qui mettent fin à l'épisode sont interdites, ils ont une beaucoup plus haute probabilité de trouver les cases à peindre et donc moins d'itérations d'entraînement. On ne leur donne que 8000 pas de temps. On donne en revanche à l'agent témoin au moins 800 000 pas de temps pour s'entraîner contrairement à précédemment. On veut être sûr que son entraînement est terminé. C'est pour ça que la courbe de l'agent témoin est différente pour ces agents sur la Figure 4.3.

Comme on peut le voir dans la Figure 4.3, les deux agents surpassent largement les performances de l'agent témoin. La différence de performance entre les deux agents qui utilisent

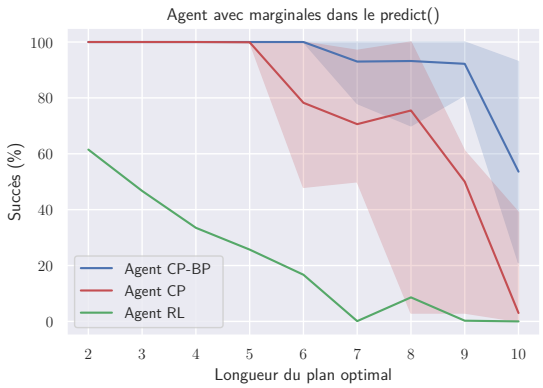

Figure 4.3 Performances des agents avec CP dans le processus de décision comparées à celles d'un agent témoin

la CP montre que l'information portée par les marginales est plus forte que juste "quelles actions sont interdites ou inutiles". Elles poussent l'agent dans la bonne direction et l'aident à trouver la solution.

Pour ces deux agents, les performances sur les grilles plus difficiles sont très variables. On voit sur la Figure 4.3 la zone d'incertitude, elle contient toute la zone entre les valeurs extrêmes obtenues sur les 5 essais. Tous les entraînements se déroulent dans les mêmes conditions avec les mêmes hyper paramètres, mais les poids de départ du réseau étant aléatoires, le résultat l'est aussi. On peut donc avoir un agent bien meilleur ou bien pire que les valeurs moyennes selon la chance. Par exemple, pour la grille 7 et l'agent CP, les performances observées varient entre 48 et 100.

En termes de vitesse d'entraînement, il y a deux types. Les agents avec les marginales dans la récompense et les observations ont été entraînés pendant 24h environ par grille. L'agent témoin prend entre 10 et 20 minutes pour être entraîné. Les agents avec CP dans le processus de décision n'ont que 8000 pas d'entraînement ce qui est moins que la valeur de learning_starts, toutes les actions d'entraînement sont donc tirées au sort ce qui est très rapide. Ces agents sont donc très rapides à entraîner, un peu moins d'une seconde en moyenne.

On notera ici les performances de la CP seule. Pour la grille la plus complexe, la CP finit son exécution complète en 1.5 seconde. La solution en elle-même de longueur 10 est trouvée en 12 ms et 1 fail. La plus grande partie du temps d'exécution est passé à vérifier s'il n'existe pas de plan plus court avant de trouver une solution puis à vérifier que la solution est optimale après l'avoir trouvée.

En revanche, pour ce qui est de l'évaluation des agents, seul l'agent témoin est rapide. Pour générer les 1000 épisodes d'évaluation, il ne faut que quelques secondes à l'agent témoin. Alors que pour nos autres agents, selon la longueur des épisodes et donc la performance cela prend entre 5 minutes et 3h30.

4.1.2 Généralisation et Transfert

On ne s'intéresse au transfert et à la généralisation que pour les agents déjà prometteurs, c'est-à-dire les agents qui utilisent la CP pendant le processus de décision.

On commence par regarder si l'apprentissage est généralisable à d'autres grilles sans ré entraîner l'agent sur les grilles en question. On prend donc un agent que l'on entraîne sur une grille en particulier et on enregistre les poids de son réseau de neurones à la fin de l'entraînement. Puisque certaines grilles ont des dimensions différentes, les réseaux de neurones entraînés sur différentes grilles peuvent avoir une dimension d'entrée différente. On coupe donc nos grilles en deux groupes liés à leur dimension.

Pour les grilles de 2 à 6, de taille 3x3, on entraîne nos trois agents sur les grilles 2 et 6 et on les teste sur toutes les grilles. Pour les grilles 7 à 10, de taille 4x4, on fait la même chose avec les grilles 7 et 10 (les grilles sont détaillés en Figure 3.2). Les résultats sont reportés dans le Tableau 4.1.

modèle/grille	2	3	4	5	6
CP BP/grille 2	100	100	100	72	99
CP BP/grille 6	100	100	100	100	100
CP/grille 2	100	100	100	87	99
CP/grille 6	100	100	91	100	77
RL/grille 2	59	1.3	0.01	0.03	0
RL/grille 6	15	10	3	4	11

modèle/grille	7	8	9	10
CP BP/grille 7	98	95	70	87
CP BP/grille 10	100	100	76	93
CP/grille 7	97	100	3	10
CP/grille 10	54	75	29	47
RL/grille 7	0.1	0	0	0
RL/grille 10	0	0	0	0

Tableau 4.1 Performances en pourcentage de réussite de modèles entraînés sur une grille en particulier sur les autres grilles

Puisque la performance de nos agents est variable même pour un entraînement de la même longueur sur la même grille, dans le Tableau 4.1 on rapporte dans la case liée à la grille d'entraînement le pourcentage de succès obtenu par l'agent après entraînement. On a entraîné de nouveaux agents, donc ils peuvent se placer n'importe où par rapport à la performance moyenne. On peut ainsi voir que pour les agents CP et CP-BP l'agent entraîné sur la grille 7 est particulièrement bon par rapport à la performance moyenne rapportée plus haut. De même pour l'agent CP BP entraîné sur la grille 10.

On observe dans le Tableau 4.1 que nos agents qui utilisent la CP généralisent beaucoup mieux que l'agent témoin. On peut notamment regarder les agents entraînés sur la grille 2. Là où l'agent RL entraîné avait une performance correcte sur la grille la plus simple, il ne transmet quasiment rien quand il faut résoudre des grilles plus complexes. Quand il est entraîné sur la grille la plus complexe une petite partie de cette performance se transmet dans les grilles plus simples. En revanche, les agents CP et CP-BP gardent de très bonnes performances même quand ils sont entraînés sur les grilles les plus simples de leur groupe.

C'est lié au fait que les marginales utilisées dans la décision ne sont pas liées à la grille d'entraînement. Elles restent donc de qualité égale même si on change la grille.

On notera également que pour les agents CP-BP qui sont particulièrement bons par rapport à la moyenne, ils restent particulièrement bons sur les autres grilles du second groupe ce qui n'est pas le cas des agents CP. Quand les performances sont meilleurs que la moyenne, l'agent a appris à se laisser entraîner par les marginales. Ainsi, il est performant sur toutes les grilles. Alors que pour l'agent CP, il doit trouver la bonne direction tout seul.

Ensuite, on s'intéresse au transfert c'est-à-dire à la capacité de nos agents à fonctionner sans la CP pour les guider. Les poids entraînés ont-ils une valeur intrinsèque ou sont-ils uniquement utiles quand ils sont couplés à la CP ? On prend donc nos mêmes agents entraînés sur des grilles en particulier. On donne ces poids enregistrés à un DQN sans entraînement et on évalue ses performances sur la grille d'entraînement.

On obtient les performances d'un agent complètement sans entraînement, 0% de réussite. Un épisode typique que ce soit pour l'agent CP ou l'agent CP-BP ressemble à {move_up, move_left, move_down, paint_up}.

4.1.3 Résultats pour le choix des hyperparamètres

Nous détaillons ici les performances de l'agent RL seul lorsque l'on change les hyperparamètres dont on a étudié l'effet sur les performances dans quelques tableaux. Tout d'abord on peut voir dans le tableau 4.2 que changer les valeurs de récompense n'a que peu changé les performances de l'agent RL. On a donc choisi la deuxième version de récompense car les performances sont globalement légèrement meilleures et parce que les valeurs paraissaient meilleurs pour l'usage avec les marginales dans la récompense. Il fallait s'assurer que les récompenses de peinture restent positives et que l'écart avec la récompense de mouvement reste significative.

Ensuite, pour l'étude d'hyperparamètres, le choix s'est porté sur l'hyperparamètre qui donnait les meilleurs résultats. C'est-à-dire 1 pour tau comme vu dans le tableau 4.3. Le choix est

Récompenses	2	3	4	5	6	7	8	9	10
-1, 1	60.3	46.3	33.7	25.7	17.2	0.1	4.8	0.5	0.0
-5, 10	60.9	44.4	34.1	24.3	15.2	0.0	8.2	1.6	0.9

Tableau 4.2 Performances de l'agent RL selon les valeurs de récompense pour toutes les grilles

moins clair pour target_update_interval. Il a fallu faire une étude complémentaire avec les agents qui utilisent la CP BP pour faire un choix plus éclairé vu dans le tableau 4.4. On choisit finalement target_update_interval = 10 000 car il s'agit des meilleures performances cumulées quand on regarde les deux agents à la fois.

tau	7	8	9	10
0.5	0.26	1.54	0.08	0.02
0.75	0.04	4.48	0.04	0
1	0.10	8.60	0.25	0

Tableau 4.3 Performances de l'agent RL selon la valeur de tau pour les grandes grilles

target_update_interval	7	8	9	10
5000	2.40 45.0	5.12 82.8	0.06 57.8	0 60.0
10 000	0.10 93.0	8.60 93.2	0.25 92.2	0 53.6
20 000	0.32 84.2	2.18 75.6	0.04 79.2	0.02 62.2
50 000	0.54 58.2	0.58 83.6	0.18 73.2	0.04 24.6

Tableau 4.4 Performances de l'agent RL selon la valeur de target_update_interval pour les grandes grilles. En noir l'agent RL, en rouge l'agent CP BP

4.2 Discussion

En résumé, deux de nos agents ont des résultats intéressants. Ils sont plus performants et généralisent mieux sur des grilles proches de leur grille d'entraînement que l'agent témoin. En revanche, ils ne sont pas plus rapides à entraîner et sont beaucoup plus lents en inférence après l'entraînement. Sur les grilles difficiles, l'agent témoin prend 20 secondes pour générer 10 000 épisodes alors que dans le même temps les agents CP en génèrent 2. Leur utilisation dans le vrai monde paraît donc compromise.

Tout d'abord, puisque les performances en transfert sont aussi mauvaises on peut se demander si les agents CP et CP-BP apprennent tout court. On veut savoir si les performances sont uniquement liées aux marginales forçant l'agent dans une direction ou si l'agent apprend à aller dans la bonne direction avec l'assistance des marginales. Pour l'agent CP-BP, jusqu'à

la grille 6, l'agent n'apprend rien. Après 10 pas de temps d'entraînement, il est déjà à 100% de réussite. En revanche, pour les grilles plus complexes, on peut observer un peu d'apprentissages comme dans la figure 4.4. De la même façon, l'agent CP n'a besoin d'apprendre qu'à partir de la grille 6.

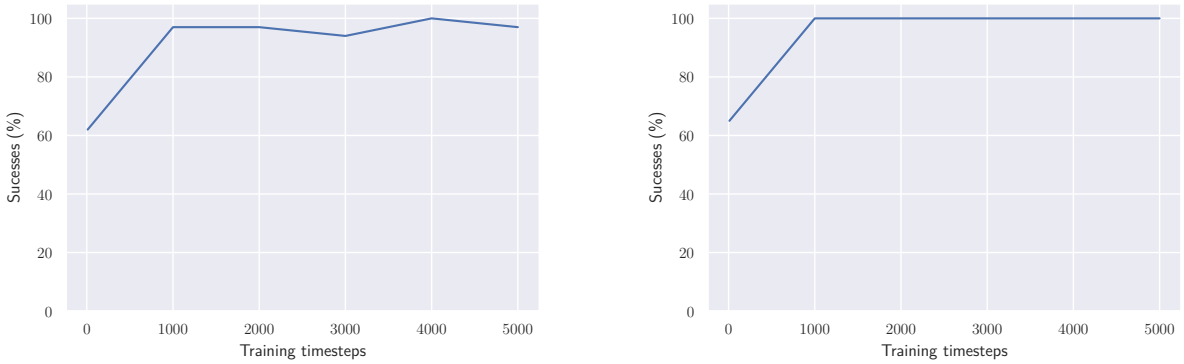

Figure 4.4 Performances au cours de l'entraînement d'un agent CP-BP sur la grille 7 à gauche et d'un agent CP sur la grille 6 à droite

Ensuite, on remarque que les seuls agents qui ont du succès sont ceux où l'on force l'agent à prendre en compte l'information issue de la CP. Même en mettant les marginales dans la source principale d'information pour l'entraînement, la récompense, on n'obtient pas de différence notable dans les performances. Pire encore, quand l'information est à disposition dans les observations, elle semble juste embrouiller l'agent et les performances sont encore pires.

Donc, à moins de lui forcer la main, il semble que l'agent DQN ne sache pas faire usage de ce genre d'informations.

De plus, pour les agents où l'information est seulement mise à disposition, nous avons limité la durée d'entraînement à 24h. Il est entièrement possible que, si l'on pouvait pousser l'entraînement plus longtemps, ces agents obtiennent de meilleures performances que l'agent témoin. Une courbe classique d'apprentissage est croissante jusqu'à atteindre un plateau. Rien ne nous garantit de la position du plateau de ces agents. Il semble quand même raisonnable de les écarter au vu de la durée d'entraînement qui serait nécessaire.

CHAPITRE 5 CONCLUSION

Dans ce mémoire, on cherchait à regarder si, dans le domaine de la planification IA, un agent d'apprentissage par renforcement pouvait faire bon usage d'informations issues de la programmation par contraintes. On utilise un modèle de programmation par contraintes capable de résoudre le problème par lui-même et on lui demande de transmettre des informations en cours de route à l'agent RL. On a testé trois emplacements où placer cette information pour que l'agent d'apprentissage par renforcement puisse l'utiliser. Une de ces méthodes a des résultats intéressants. Les agents obtiennent de meilleures performances que l'agent témoin et généralisent de façon correcte même sur les grilles les plus compliquées. Même entraînés sur des grilles simples, ils sont toujours capables de résoudre des grilles plus compliquées. Ces agents sont entraînés en beaucoup moins de pas de temps que l'agent témoin. Ces pas de temps ne sont pas plus longs puisqu'ils ne nécessitent pas de faire appel à la CP donc l'entraînement est finalement beaucoup plus rapide.

Cependant, ces agents sont beaucoup plus lents à l'inférence que les méthodes traditionnelles, là où l'agent traditionnel peut résoudre 1000 problèmes en quelques secondes, cela pourrait prendre des heures à nos agents qui utilisent la programmation par contraintes. De la même façon, la programmation par contraintes seule peut résoudre ces mêmes 1000 problèmes, garantie sans erreur, en une minute. Nos agents ne sont donc pas suffisamment efficaces pour avoir une chance d'être utilisés en application réelle. En outre, nos agents ne peuvent pas se passer du modèle CP même après entraînement. Si on retire la CP, l'agent devient complètement inutile et incapable de résoudre le problème sur lequel il a été entraîné. S'ils avaient pu s'en passer, ils pourraient être franchement plus rapides et donc plus aptes à être utilisés dans la vraie vie.

Pour ce qui est des agents qui n'avaient pas de très bonnes performances, nous n'avons aucune garantie que plus d'entraînement ne leur permettrait pas de dépasser les performances de notre agent témoin. Il est possible que la saturation de leur entraînement se fasse plus haut que celle de l'agent témoin.

Dans le futur, il pourrait être intéressant de réitérer ces expériences avec un solveur de CP basé sur python. On pourrait alors hybrider plus facilement entre les packages d'apprentissage machine de python et la programmation par contraintes. Il est assez probable que cela rendrait nos agents beaucoup plus rapides s'ils n'avaient pas besoin d'écrire et de lire autant de fichiers. Il faudrait aussi tester la performance de nos agents sur d'autres tâches de planification classique pour confirmer la véracité de nos résultats. La différence est-elle liée à la tâche en

particulier ou se retrouverait-elle ailleurs ?

Enfin, il serait plus intéressant de tester nos agents sur des grilles en partie inconnues ou avec d'autres éléments d'incertitude. Le problème sans incertitude est déjà très facilement résolu par la programmation par contraintes seule. Il faut donc complexifier le problème pour que l'usage d'un agent d'apprentissage par renforcement soit nécessaire. Si dans ces cas-là, l'agent de RL est toujours efficace, il serait beaucoup plus utile en pratique.

RÉFÉRENCES

Babaki, Behrouz, Gilles Pesant et Claude-Guy Quimper. « Solving Classical AI Planning Problems Using Planning-Independent CP Modeling and Search ». In : *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS 2020, Online Conference [Vienna, Austria], 26-28 May 2020*. Sous la dir. de Daniel Harabor et Mauro Vallati. AAAI Press, 2020, p. 2-10. url : https://aaai.org/ocs/index.php/SOCS/SOCS20/paper/view/18512.

Barták, Roman et Daniel Toropila. « Reformulating Constraint Models for Classical Planning. » In : jan. 2008, p. 525-530.

Barto, Andrew G et Sridhar Mahadevan. « Recent advances in hierarchical reinforcement learning ». In : *Discrete event dynamic systems* 13.1 (2003), p. 41-77.

Beek, Peter van et Xinguang Chen. « CPlan : A Constraint Programming Approach to Planning ». In : (juil. 2000).

BELLMAN, RICHARD. « A Markovian Decision Process ». In : *Journal of Mathematics and Mechanics* 6.5 (1957), p. 679-684. issn : 00959057, 19435274. url : http://www.jstor.org/stable/24900506.

Brélaz, Daniel. « New Methods to Color the Vertices of a Graph ». In : *Commun. ACM* 22.4 (avr. 1979), p. 251-256. issn : 0001-0782. doi : 10.1145/359094.359101. url : https://doi.org/10.1145/359094.359101.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang et Wojciech Zaremba. « Openai gym ». In : *arXiv preprint arXiv :1606.01540* (2016).

Dechter, R., B. Bidyuk, Robert Mateescu et E. Rollon. « On the Power of Belief Propagation : A Constraint Propagation Perspective ». In : *Heuristics, Probability and Causality. A Tribute to Judea Pearl*. Heuristics, Probability and Causality. A Tribute to Judea Pearl. College Publications, jan. 2010, p. 137-158. url : https://www.microsoft.com/en-us/research/publication/on-the-power-of-belief-propagation-a-constraint-propagation-perspective/.

Do, Minh Binh et Subbarao Kambhampati. « Planning as constraint satisfaction : Solving the planning graph by compiling it into CSP ». In : *Artificial Intelligence* 132.2 (2001), p. 151-182. issn : 0004-3702. doi : https://doi.org/10.1016/S0004-3702(01)00128-X. url : https://www.sciencedirect.com/science/article/pii/S000437020100128X.

Feng, Dieqiao, Carla Gomes et Bart Selman. « Solving Hard AI Planning Instances Using Curriculum-Driven Deep Reinforcement Learning ». In : *Proceedings of the Twenty-Ninth*

International Joint Conference on Artificial Intelligence, IJCAI-20. Sous la dir. de Christian Bessiere. Main track. International Joint Conferences on Artificial Intelligence Organization, juil. 2020, p. 2198-2205. doi : 10.24963/ijcai.2020/304. url : https://doi.org/10.24963/ijcai.2020/304.

Fern, Alan, Roni Khardon et Prasad Tadepalli. « The first learning track of the international planning competition ». In : *Machine Learning* 84.1 (juil. 2011), p. 81-107. issn : 1573-0565. doi : 10.1007/s10994-011-5234-y. url : https://doi.org/10.1007/s10994-011-5234-y.

Gentleman, R. et V. J. Carey. « Unsupervised Machine Learning ». In : *Bioconductor Case Studies*. New York, NY : Springer New York, 2008, p. 137-157. isbn : 978-0-387-77240-0. doi : 10.1007/978-0-387-77240-0_10. url : https://doi.org/10.1007/978-0-387-77240-0_10.

Ghallab, Malik, Dana Nau et Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

Hart, Peter E., Nils J. Nilsson et Bertram Raphael. « A Formal Basis for the Heuristic Determination of Minimum Cost Paths ». In : *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), p. 100-107. doi : 10.1109/TSSC.1968.300136.

Hasselt, Hado van, Arthur Guez et David Silver. « Deep Reinforcement Learning with Double Q-Learning ». In : *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (mar. 2016). url : https://ojs.aaai.org/index.php/AAAI/article/view/10295.

Hastie, Trevor, Robert Tibshirani et Jerome Friedman. « Overview of supervised learning ». In : *The elements of statistical learning*. Springer, 2009, p. 9-41.

Kask, Kalev, Rina Dechter et Vibhav Gogate. « Counting-Based Look-Ahead Schemes for Constraint Satisfaction ». In : *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*. Sous la dir. de Mark Wallace. T. 3258. Lecture Notes in Computer Science. Springer, 2004, p. 317-331. doi : 10.1007/978-3-540-30201-8_25. url : https://doi.org/10.1007/978-3-540-30201-8%5C_25.

Lee, Junkyu, Michael Katz, Don Joven Agravante, Miao Liu, Tim Klinger, Murray Campbell, Shirin Sohrabi et Gerald Tesauro. « AI Planning Annotation for Sample Efficient Reinforcement Learning ». In : *arXiv preprint arXiv :2203.00669* (2022).

Lopez, Adriana et Fahiem Bacchus. « Generalizing GraphPlan by Formulating Planning as a CSP ». In : (juin 2003).

Michalski, Ryszard Stanislaw, Jaime Guillermo Carbonell et Tom M Mitchell. *Machine learning : An artificial intelligence approach*. Springer Science & Business Media, 2013.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, I. Antonoglou, Daan Wierstra et Martin Riedmiller. *Playing Atari with Deep Reinforcement Learning*. 2013. doi : 10.48550/ARXIV.1312.5602. url : https://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski et al. « Human-level control through deep reinforcement learning ». In : *nature* 518.7540 (2015), p. 529-533.

Pesant, G., Claude-Guy Quimper et Alessandro Zanarini. « Counting-Based Search : Branching Heuristics for Constraint Satisfaction Problems ». In : *J. Artif. Intell. Res.* 43 (2012), p. 173-210.

Pesant, Gilles. « A constraint programming primer ». In : *EURO Journal on Computational Optimization* 2.3 (2014), p. 89-97. issn : 2192-4406. doi : https://doi.org/10.1007/s13675-014-0026-3. url : https://www.sciencedirect.com/science/article/pii/S2192440621000289.

— « From Support Propagation to Belief Propagation in Constraint Programming ». In : *J. Artif. Intell. Res.* 66 (2019), p. 123-150. doi : 10.1613/jair.1.11487. url : https://doi.org/10.1613/jair.1.11487.

Raffin, Antonin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus et Noah Dormann. « Stable-Baselines3 : Reliable Reinforcement Learning Implementations ». In : *Journal of Machine Learning Research* 22.268 (2021), p. 1-8. url : http://jmlr.org/papers/v22/20-1364.html.

Rossi, Francesca, Peter van Beek et Toby Walsh. *Handbook of Constraint Programming*. USA : Elsevier Science Inc., 2006. isbn : 9780080463803.

Schaul, Tom, John Quan, Ioannis Antonoglou et David Silver. « Prioritized Experience Replay ». In : (2015). doi : 10.48550/ARXIV.1511.05952. url : https://arxiv.org/abs/1511.05952.

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, L. Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan et Demis Hassabis. « Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm ». In : *ArXiv* abs/1712.01815 (2017).

Sutton, Richard S. et Andrew G. Barto. *Reinforcement Learning : An Introduction*. Second. The MIT Press, 2018. url : http://incompleteideas.net/book/the-book-2nd.html.

Watkins, Christopher JCH et Peter Dayan. « Q-learning ». In : *Machine learning* 8.3 (1992), p. 279-292.

Zanarini, Alessandro et Gilles Pesant. « Solution counting algorithms for constraint-centered search heuristics ». In : *Constraints* 14.3 (2009), p. 392-413.

Zanarini, Alessandro, Gilles Pesant et Michela Milano. « Planning with soft regular constraints ». In : *ICAPS Workshop on Preferences and Soft Constraints in Planning* (2006), p. 73-78.

ANNEXE A SLURM SCRIPT POUR CEDAR

```
#!/bin/bash
#SBATCH –mail-user=dana.chaillard@polymtl.ca
#SBATCH –mail-type=END
#SBATCH –gres=gpu :4 #Request GPU
#SBATCH –cpus-per-task=8
#SBATCH –account=def-pesantg
#SBATCH –mem=4000M
#SBATCH –time=23 :50 :00
echo CP RL a=1
echo CP RL map 6
module load python scipy-stack java
unset JAVA_TOOL_OPTIONS
source env/bin/activate
python main.py 250000 1 out1.txt
```