Semestre 9 en Intelligence Artificielle

Apprentissage par Renforcement

Projet : Développement d'un Robot de Nettoyage dans Différents Environnements

---

Auteurs
Bonhomme Romain
Pierre Romain

Professeur
Akka Zemmari

Année Académique 2024-2025

1 Démarche de Modélisation

1.1 Environnements

Pour réaliser les différentes tâches demandées, il a fallu développer trois environnements différents, un hangar, un entrepôt et un garage. Etant donné que tous ces environnements ont quasiment la même structure, la suite comprendra une explication générale de la modélisation avec des détails concernant la spécificité des différents environnements. Ces environnements ont été développés sous forme de *Markov Decision Process* (MDP), une modélisation mathématique utilisée pour représenter un problème de décision séquentielle dans un environnement incertain. Un MDP est défini par un ensemble d'états S, un ensemble d'actions A, des probabilités de transition $P(s'|s,a)$ qui décrivent la dynamique de l'environnement, et une fonction de récompense $R(s,a,s')$ associant des gains aux transitions. Le but est de trouver une politique $\pi(a|s)$, c'est-à-dire une règle qui prescrit l'action optimale pour chaque état afin de maximiser la récompense cumulée à long terme, pondérée par un facteur d'actualisation γ.

Les environnements sont créés comme des classes, définissant :

- une hauteur et une largeur de la grille (paramètres *height* et *width*),
- un nombre d'obstacles (*number_of_holes*),
- un nombre d'objets à collecter (*number_of_materials* pour le Hangar, *number_of_uncleaned_areas* pour l'Entrepôt et *number_of_recharging_stations* pour le Garage),
- des états représentés par des tuples contenant la position (i,j) et un état booléen pour chaque objet (objets de nettoyage à collecter pour le Hangar et cellules à nettoyer pour l'entrepôt).

Cette méthode permet de rendre le code facilement réutilisable, car tous ces paramètres peuvent être donnés au constructeur de la classe. Les différentes actions possibles (haut, bas, gauche, droite) sont définies sous forme de tuples *UP*, *DOWN*, *LEFT*, *RIGHT*. Les obstacles (*bad_states*) et les positions des objet spécifiques sont placés aléatoirement dans la grille. Finalement, les coordonnées de l'état initial (coin supérieur gauche pour le Hangar et l'Entrepôt, coin inférieur gauche pour le Garage) et final (coin inférieur droit pour le Hangar et coin supérieur droit pour l'Entrepôt) sont encodées.

La dernière partie de l'initialisation concerne les probabilités de transition et les récompenses :

- **Probabilités de transition :** Un dictionnaire est créé avec des clés de type *(state, action, new_state)* initialement remplies de 0, puis mises à jour selon la logique des déplacements que peut faire l'agent ; à savoir 0 si le new_state est hors de la grille ou sur un obstacle et 1 dans les autres cas .
- **Récompenses :** Un dictionnaire similaire est créé pour définir les récompenses des transitions. Les valeurs attribuées dépendent de l'état atteint après chaque action :

- 3 si *new_state* est l'état terminal et que toutes les cellules particulières ont été visitées (pour le Hangar et l'Entrepôt),
- 2 si *new_state* est une case contenant un objet de nettoyage qui n'a pas encore été collecté,
- 1 si *new_state* est une borne de recharge dans l'environnement du Garage,
- -1 si *new_state* est invalide (hors de la grille ou un obstacle) ou si l'agent se rend sur la case de sortie sans avoir visité toutes les cellules particulières (pour le Hangar et l'Entrepôt),
- 0 dans tous les autres cas.

La fonction *take_action* simule une transition en appliquant une action à un état donné. Si le nouvel état est valide, il est retourné ; sinon, l'agent reste dans l'état courant. La collecte des objets est gérée en modifiant le troisième élément du tuple *state* pour refléter le changement de statut (de *False* à *True*). Cette particularité du code des environnements leur permet d'être totalement indépendants de l'agent markovien utilisé, rendant le code modulaire

Deux méthodes permettent de visualiser l'environnement et les politiques :

- *print_board* : affiche la grille avec des symboles pour les obstacles (X), les cases propre à chaque environnement (M pour les objets de nettoyage du Hangar, U pour les cellules sales de l'Entrepôt et T pour les bornes de recharge du Garage), l'état initial (S) et l'état terminal (T (sauf pour le Garage où les états terminaux sont les bornes de recharge)).
- *print_policy* : affiche la politique optimale sous forme de flèches indiquant les actions préférées pour chaque position, dépendant de l'état actuel.

Cette approche modulaire et flexible permet de simuler les différents environnements et d'évaluer les performances d'agents dans ce cadre défini.

1.2 Agents

Pour résoudre ce problème, deux approches ont été essayées, une dite 'model-free' et l'autre dite 'model-based'. La première est une approche dans laquelle l'agent apprend directement une politique ou une fonction de valeur en se basant uniquement sur les expériences issues de l'interaction avec l'environnement, sans chercher à construire ou exploiter un modèle explicite des dynamiques de l'environnement (transitions et récompenses). À l'inverse, la seconde approche repose sur la construction explicite d'un modèle de l'environnement, qui décrit les probabilités de transition entre les états et les récompenses associées aux actions. Ce modèle est ensuite utilisé pour planifier, en simulant des trajectoires et en optimisant la politique en fonction des prédictions qu'il génère. Dans la suite, la structure de ces deux approches seront expliquées.

1.2.1 Model-Free Agent

L'agent Monte Carlo repose sur le concept d'apprentissage par simulation, où des épisodes sont générés pour explorer l'environnement et accumuler des données sur les récompenses associées à chaque action. L'implémentation commence par la définition d'un constructeur, qui initialise les paramètres essentiels tels que l'environnement *mdp*, le taux d'exploration ϵ et le facteur d'actualisation γ. Deux structures de données importantes sont créées : *Q*, un *defaultdict* qui associe chaque paire état-action à une estimation de la valeur $Q(s, a)$, et *returns*, qui stocke la liste des retours cumulés pour chaque paire état-action. La politique initiale est générée en attribuant aléatoirement une action possible à chaque état, en utilisant la fonction *random.choice*.

La méthode principale pour interagir avec l'environnement est *generate_episode*, qui simule une trajectoire en suivant la politique actuelle. À chaque étape, l'agent utilise une politique ϵ-greedy pour choisir une action : soit une action aléatoire (exploration, avec probabilité ϵ), soit l'action optimale selon les valeurs actuelles de *Q* (exploitation, avec probabilité $1 - \epsilon$). L'environnement répond en fournissant un nouvel état et une récompense. Ces états, actions et récompenses sont enregistrés dans une liste d'épisode, jusqu'à ce que l'état terminal soit atteint ou que le nombre maximum de pas (*max_steps*) soit dépassé.

La mise à jour de *Q* repose sur la méthode Monte Carlo classique. Lors de chaque épisode, le retour cumulé G est calculé en partant de la fin vers le début de la trajectoire. Si une paire état-action est rencontrée pour la première fois dans cet épisode, G est ajouté à la liste des retours associés dans *returns*, et la valeur $Q(s, a)$ est mise à jour en prenant la moyenne des retours accumulés. Pour garantir que chaque paire état-action n'est mise à jour qu'une seule fois par épisode, un ensemble (*visited*) est utilisé.

Afin d'améliorer constamment la politique, l'agent recalcule périodiquement la meilleure action pour chaque état à partir des valeurs *Q* actualisées. Cette amélioration, implémentée dans *update_policy*, assure que la politique converge vers une solution optimale à mesure que l'apprentissage progresse.

Pour analyser les performances de l'agent et valider la politique apprise, la classe *MonteCarloAgent* inclut deux fonctions d'affichage. La méthode *print_policy* affiche la politique sous forme d'une grille où chaque case indique la direction optimale pour l'état correspondant (il y a donc $2^{nb_o bjets}$ flèches par case, étant donné que chaque objet peut-être soit ramassé (True) soit pas encore (False) et la correspondance de ces flèches avec la valeur des indicateurs booléens est affichée lors de l'exécution, un exemple de lecture de résultat est affiché à la FIG.2. De même, *print_value_function* calcule la valeur optimale pour chaque état à partir des valeurs *Q* et les représente dans un format visuel.

Ces mécanismes permettent à l'agent de naviguer efficacement dans un environnement inconnu tout en optimisant les récompenses. En ajustant les paramètres ϵ et γ, ainsi que le nombre d'épisodes, l'agent peut s'adapter à des scénarios variés, allant de l'exploration initiale à l'exploitation complète des connaissances acquises.

1.2.2 Model-Based Agent

L'agent de 'Value Iteration' repose sur un processus d'optimisation dynamique visant à trouver la politique optimale pour un environnement donné. Cette méthode est déterministe et utilise une équation de Bellman pour mettre à jour les estimations des valeurs d'état. L'implémentation commence par un constructeur, qui initialise les paramètres essentiels, notamment l'environnement *mdp*, le facteur d'actualisation γ, et un seuil de convergence θ. Deux structures principales sont définies : V, un dictionnaire qui stocke les valeurs estimées pour chaque état, et *policy*, qui contient une action initialement arbitraire pour chaque état.

La fonction *get_max_action_value* calcule la meilleure valeur d'action et l'action correspondante pour un état donné. Si l'état est terminal ou appartient à un ensemble d'états indésirables (*bad_states*), la valeur est immédiatement fixée à zéro. Pour les autres états, la méthode évalue chaque action possible en utilisant les probabilités de transition et les récompenses associées à cette action, ainsi que la valeur actualisée des états suivants. La valeur d'une action est calculée comme la somme pondérée des récompenses immédiates et des valeurs futures, multipliées par les probabilités de transition. L'action maximisant cette valeur est sélectionnée comme la meilleure action.

L'entraînement de l'agent repose sur la méthode *train*, qui implémente l'algorithme d'itération sur les valeurs. À chaque itération, la valeur de chaque état est mise à jour en fonction de la valeur maximale calculée par *get_max_action_value*. Après la mise à jour, le changement maximal (*delta*) entre les anciennes et les nouvelles valeurs des états est calculé. Si ce changement devient inférieur à θ, l'algorithme considère que la convergence est atteinte et l'apprentissage s'arrête. L'algorithme garantit ainsi une amélioration progressive des estimations des valeurs des états et de la politique associée.

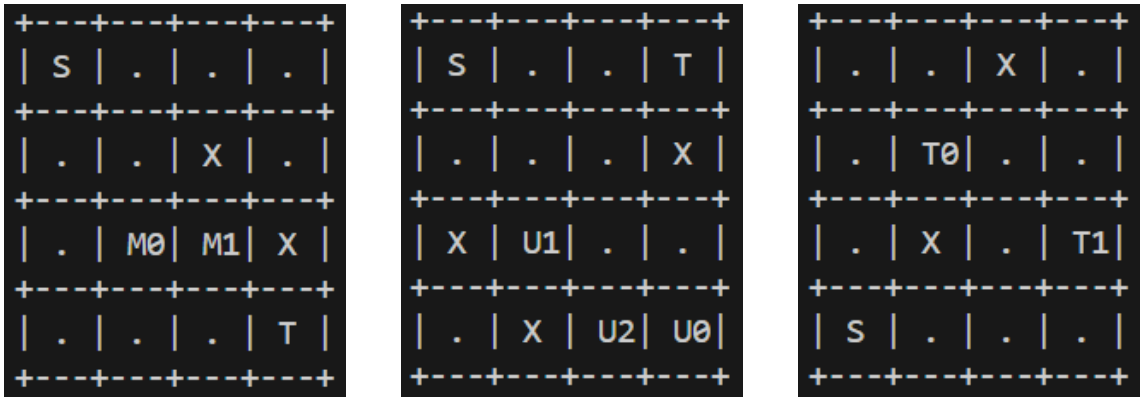
La méthode *train* inclut également un mécanisme pour détecter la convergence, affichant le nombre d'itérations nécessaires et le *delta* final, ce qui permet de vérifier la stabilité de l'algorithme. Les résultats de l'apprentissage sont ensuite utilisés pour générer une politique optimale, où chaque état est associé à l'action maximisant la valeur calculée.

Pour faciliter l'interprétation des résultats, la classe *ValueIterationAgent* inclut deux méthodes d'affichage. La méthode *print_policy* représente graphiquement la politique optimale sous forme de directions associées à chaque état. La méthode *print_value_function*, quant à elle, affiche les valeurs finales des états sous une forme visuelle adaptée.

Grâce à l'itération sur les valeurs, cet agent peut efficacement résoudre les environnements en identifiant des politiques optimales, tout en garantissant une convergence contrôlée par le seuil θ. En ajustant les paramètres γ et θ, l'agent peut s'adapter à divers environnements, que ce soit pour privilégier des récompenses immédiates ou futures.

2 Expériences Réalisées

Dans un premier temps, il semblait important de visualiser concrètement à quoi ressemblaient les différents environnements modélisés. Leurs représentations peut être observées sur la FIG.1.



(a) Représentation de l'instance HangarWorldMDP(4, 4, 2, 2), représentant l'environnement Hangar comme une grille de taille 4×4, avec 2 obstacles et 2 objets de nettoyage à récupérer.

(b) Représentation de l'instance EntrepotWorldMDP(4, 4, 3, 3), représentant l'environnement Entrepôt comme une grille de taille 4×4, avec 3 obstacles et 3 cellules sales à nettoyer.

(c) Représentation de l'instance GarageWorldMDP(4, 4, 2, 2), représentant l'environnement Garage comme une grille de taille 4×4, avec un obstacle et 2 stations de recharge.

Figure 1 – Représentation des trois environnements du projet que sont le hangar, l'entrepôt et le garage.

Pour la visualisation, chaque cellule possède un nombre de flèches qui équivaut au nombre de combinaisons possibles des n booléens correspondant aux n objets (objets de nettoyage pour le Hangar, cellules sales pour l'Entrepôt). Ainsi si il y a par exemple 2 objets, 4 états sont possibles par coordonnées ((false, false) : Aucun objet ramassé/nettoyé, (true, false) : l'objet 0 est ramassé/nettoyé, (false, true) : l'objet 1 est ramassé/nettoyé, (true, true) : les 2 objets sont ramassés/nettoyés. Pour suivre l'avancée du robot il faut donc commencer par la première flèche (tout est false) et selon la case rencontrée se rendre au numéro de flèche correspondant. Dans l'exemple précédent, si l'agent passe par l'objet 1 il faudra alors lire la troisième flèche à partir de cette cellule.

La FIG.2 représente une sortie du code lorsque l'agent Monte Carlo est appliqué à l'environnement du Hangar. En la regardant, le chemin suivi, en partant du coin supérieur gauche sera (en se basant sur la configuration observée sur la FIG.1a) : droite (première flèche), bas (première flèche), bas (première flèche), droite (deuxième flèche), bas (quatrième flèche), droite (quatrième flèche). A titre informatif, la même expérience a été réalisée avec l'agent se basant sur la politique de 'Value Iteration' et ces résultats peuvent être observés sur la FIG.3.

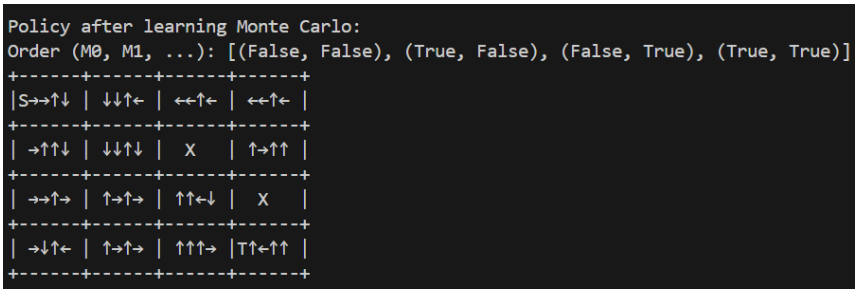


Figure 2 – Sortie du code lors de l'exécution de l'agent Monte Carlo sur un environnement HangarWorldMDP(4, 4, 2, 2), la première flèche de chaque case correspond aux états où cette case est considérée sans qu'aucun des objets de nettoyage n'aient été ramassés, la deuxième flèche correspond aux états considérés lorsque seul le premier objet a été ramassé, la troisième flèche correspond aux états considérés lorsque seul le second objet a été ramassé et enfin la quatrième flèche correspond aux états considérés lorsque les 2 objets ont été ramassés.

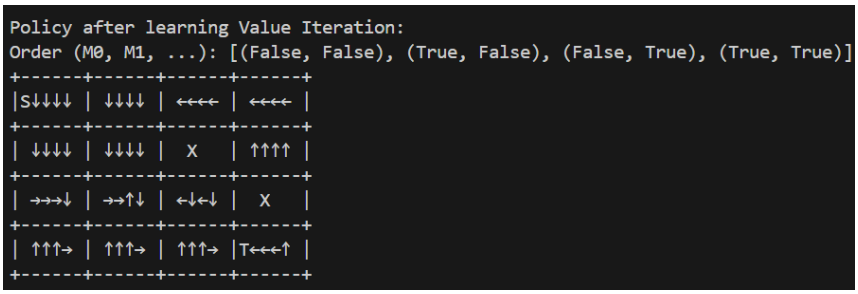


Figure 3 – Sortie de l'agent de 'Value Iteration' sur le Hangar.

La FIG.4 représente une sortie du code lorsque l'agent Monte Carlo est appliqué à l'environnement de l'Entrepôt. En la regardant, le chemin suivi, en partant du coin supérieur gauche sera (en se basant sur la configuration observée sur la FIG.1b) : droite (première flèche), bas (première flèche), bas (première flèche), droite (troisième flèche), droite (troisième flèche), bas (troisième flèche), gauche (quatrième flèche), haut (huitième flèche), haut (huitième flèche), haut (huitième flèche), droite (huitième flèche). A titre informatif, la même expérience a été réalisée avec l'agent se basant sur la politique de 'Value Iteration' et ces résultats peuvent être observés sur la FIG.5.

La FIG.6 représente une sortie du code lorsque l'agent Monte Carlo est appliqué à l'environnement du Garage. En la regardant, le chemin suivi, en partant du coin inférieur gauche sera (en se basant sur la configuration observée sur la FIG.1c) : haut, haut, droite pour arriver sur la borne $T0$. A titre informatif, la même expérience a été réalisée avec l'agent se basant sur la politique de 'Value Iteration' et ces résultats peuvent être observés sur la FIG.7.

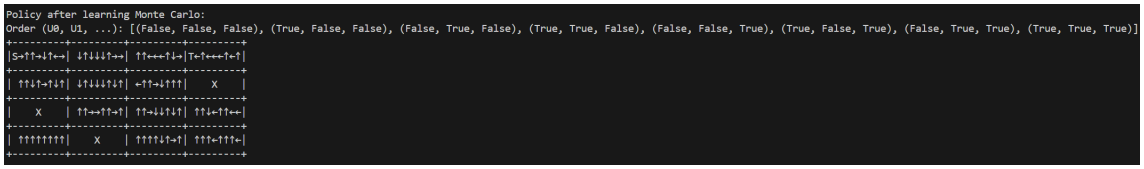


Figure 4 – Sortie du code lors de l'exécution de l'agent Monte Carlo sur un environnement EntrepotWorldMDP(4, 4, 3, 3), la première flèche de chaque case correspond aux états où cette case est considérée sans qu'aucune des cellules sales n'aient été nettoyées, la deuxième flèche correspond aux états considérés lorsque seule la première cellule sale a été nettoyée, la troisième flèche correspond aux états considérés lorsque seule la deuxième cellule sale a été nettoyée, la quatrième flèche correspond aux états considérés lorsque les deux premières cellules sales ont été nettoyées, ..., et enfin la huitième flèche correspond aux états considérés lorsque toutes les cellules sales ont été nettoyées.

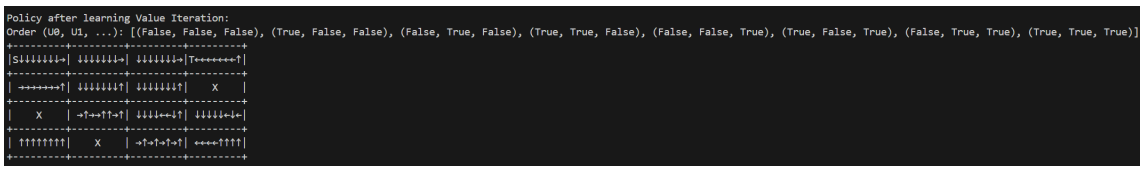


Figure 5 – Sortie de l'agent de 'Value Iteration' sur l'Entrepôt.

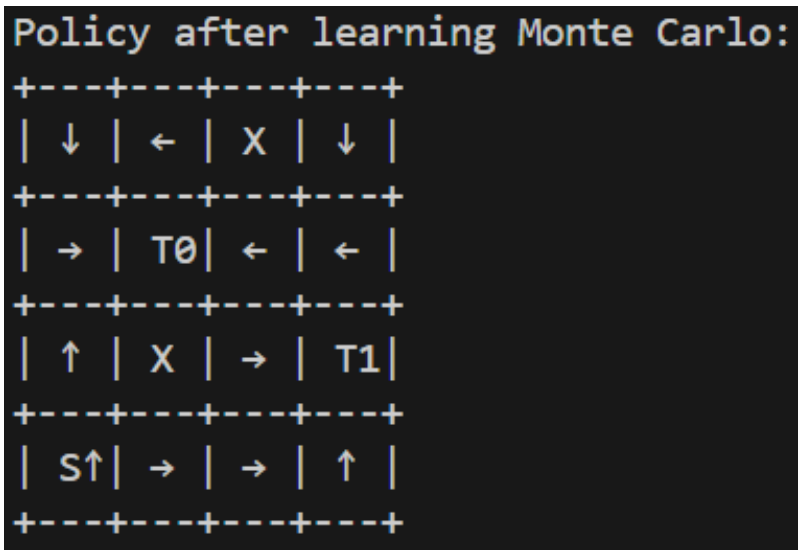


Figure 6 – Sortie du code lors de l'exécution de l'agent Monte Carlo sur un environnement GarageWorldMDP(4, 4, 2, 2), la flèche correspondant à la direction prise en quittant la cellule considérée.

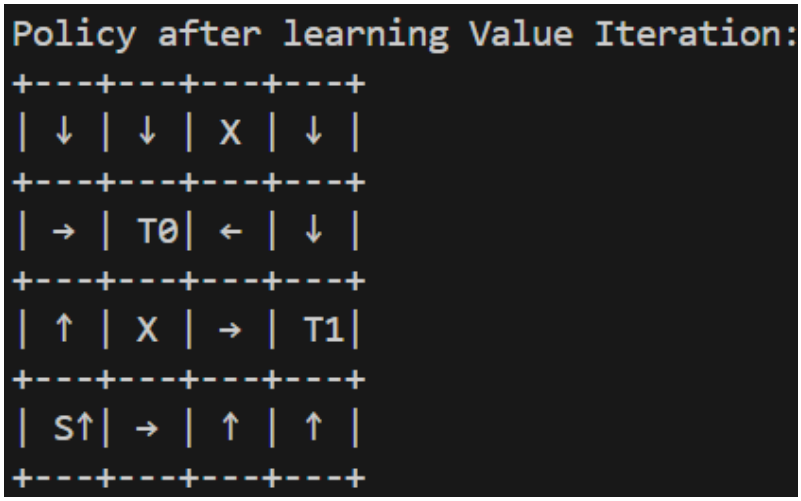


Figure 7 – Sortie de l'agent de 'Value Iteration' sur le Garage.