1 Rue de la Chebarde, 63170,
Aubière, France

Rapport d'élève ingénieur
Stage de 2ème année
Filière : Modélisation Mathématique et Sciences des Données

Outil d'Optimisation et Apprentissage par renforcement multi-agent pour le problème de planification en ligne d'un service mobile à la demande

Présenté par :
Célio Lucas LEMES DE MEDEIROS

Responsable ISIMA : Viet Hung NGUYEN
Responsable LIMOS : Viet Hung NGUYEN

Date de la soutenance : 08/07/2024
Durée du projet : 4 mois

Campus des Cézeaux. 1 rue de la Chebarde. TSA 60125. 63170 Aubière

Remerciements

Tout d'abord, je voudrais remercier M. Viet NGUYEN qui a proposé et dirigé ce travail en tant que tuteur, ainsi que le professeur Rafael COLARES qui a également suivi le travail réalisé pendant le stage. De plus, je tiens à remercier le laboratoire de recherche LIMOS pour avoir accueilli ce travail de stage.

Table des figures

Liste des tableaux

Résumé

Ce travail est le résultat du stage de 2ème année à l'école d'ingénieurs Institut Supérieur d'Informatique, de Modélisation et de leurs Applications (ISIMA).

L'étude qui sera présentée implique un problème d'apprentissage par renforcement multi-agent (MARL) basé sur listes de préference. Ce problème découle de l'application d'agents qui reçoivent des Commandes et visent à les satisfaire en listant leurs préférences, de sorte que les Commandes primaires de chacun soient satisfaites compte tenu des informations de l'environnement et de la décision de l'algorithme d'affectation.

L'objectif de ce rapport est de présenter le problème, de le contextualiser et de montrer une application initiale pour ce contexte. Dans ce sens, un environnement d'apprentissage par renforcement a été développé à partir de zéro et modélisé en fonction des informations initiales, en y intégrant l'algorithme d'affectation (Gale-Shapley), et enfin un algorithme d'apprentissage basé sur les réseaux de neurones a été appliqué.

Le développement a été réalisé en Python à l'aide d'une machine propre.

Mots-Clès : MARL, listes de préférences, Gale-Shapley, MADDPG, réseaux neuronaux, Python.

Resumo

Este trabalho é o resultado do estágio do 2º ano na escola de engenharia Institut Supérieur d'Informatique, de Modélisation et de leurs Applications (ISIMA).

O estudo que será apresentado envolve um problema de aprendizado por reforço multiagente (MARL) baseado em listas de preferências. Este problema decorre da aplicação de agentes que recebem Comandas e visam atendê-las listando suas preferências, de modo que as Comandas primárias de cada um sejam satisfeitas considerando as informações do ambiente e a decisão do algoritmo de afetação.

O objetivo deste relatório é apresentar o problema, contextualizá-lo e mostrar uma aplicação inicial para esse contexto. Nesse sentido, um ambiente de aprendizado por reforço foi desenvolvido do zero e modelado de acordo com as informações iniciais, integrando o algoritmo de afetação (Gale-Shapley) e, finalmente, um algoritmo de aprendizado baseado em redes neurais foi aplicado.

O desenvolvimento foi realizado em Python utilizando uma máquina própria.

Palavras-Chave : MARL, listas de preferências, Gale-Shapley, MADDPG, redes neurais, Python.

Table des matières

Introduction

Ce rapport décrit une étude qui s'est déroulée du début avril 2024 à la fin août 2024 à l'Institut Supérieur d'Informatique, de Modélisation et de leurs applications (ISIMA) sous la responsabilité du Dr. Viet NGUYEN et le Dr. Rafael COLARES. Ce travail décrit le Travaux d'Etudes et Recherche (TER) réalisé au Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes (LIMOS) pendant la deuxième année à l'ISIMA, et il est partie intégrante du cursus de Double Diplôme ISIMA - UFMG.

Le problème d'apprentissage par renforcement multi-agent (MARL) basé sur des listes de préférences est un problème qui comprend deux éléments fondamentaux : des services étant demandés et des ressources limitées pour répondre à ces demandes, telles que le nombre de personnes travaillant, les véhicules disponibles, le temps de service restant, etc. C'est un cas courant, que l'on peut rencontrer dans divers types de services, comme les services de taxi ou de livraison. Pour la modélisation de ce problème, le contexte de la recharge d'hydrogène a été pris en compte, où des véhicules sont responsables de réapprovisionner des points nécessitant une attention tout au long d'une journée de travail.

Ainsi, le travail à réaliser consiste à modéliser et à mettre en œuvre un environnement d'apprentissage par renforcement en Python, de manière à ce qu'il se rapproche des conditions réelles du cas étudié, en simulant ses attributs et son fonctionnement. Cependant, afin de permettre des tests plus rapides, certains attributs n'ont pas été pris en compte et d'autres ont été simplifiés pour faciliter la mise en œuvre. Par conséquent, après la création de l'environnement, des études utilisant des algorithmes d'apprentissage deviennent possibles pour que la prise de décision (actions) de la part des véhicules (agents) soit optimisée et automatisée.

En conséquence, ce travail sera divisé en deux parties : la première consistera en la modélisation, la construction de l'environnement et de ses caractéristiques, et la seconde portera sur la mise en œuvre d'un algorithme d'apprentissage dans ce même environnement multi-agent. Cette approche permettra de contextualiser le problème, de caractériser l'environnement construit et de montrer comment l'algorithme d'apprentissage doit être considéré dans ce cas.

1 Présentation du stage

1.1 Présentation du laboratoire

Situé à Clermont-Ferrand, en France, le LIMOS (Laboratoire d'Informatique, de Modélisation et d'Optimisation des Systèmes), fondé en 2001, est une Unité Mixte de Recherche (UMR 6158) associée à l'Université Clermont Auvergne et au CNRS (Centre National de la Recherche Scientifique). Spécialisé dans l'optimisation, la modélisation des systèmes, l'apprentissage automatique, les réseaux de capteurs et l'interopérabilité des données.

Le laboratoire collabore activement à des projets nationaux et internationaux avec des partenaires académiques et industriels. Par ses contributions scientifiques, le développement de nouvelles technologies et la formation des chercheurs et étudiants, le LIMOS joue un rôle crucial dans l'avancement des sciences et technologies de l'information et de la communication.

Figure 1 – LIMOS

La forme d'organisation du laboratoire est représentée par l'organigramme à la Figure 2, qui montre le fonctionnement de l'institution de recherche. Le travail qui a été conçu dans ce rapport est lié au professeur Viet NGUYEN, dans le domaine de l'optimisation combinatoire.

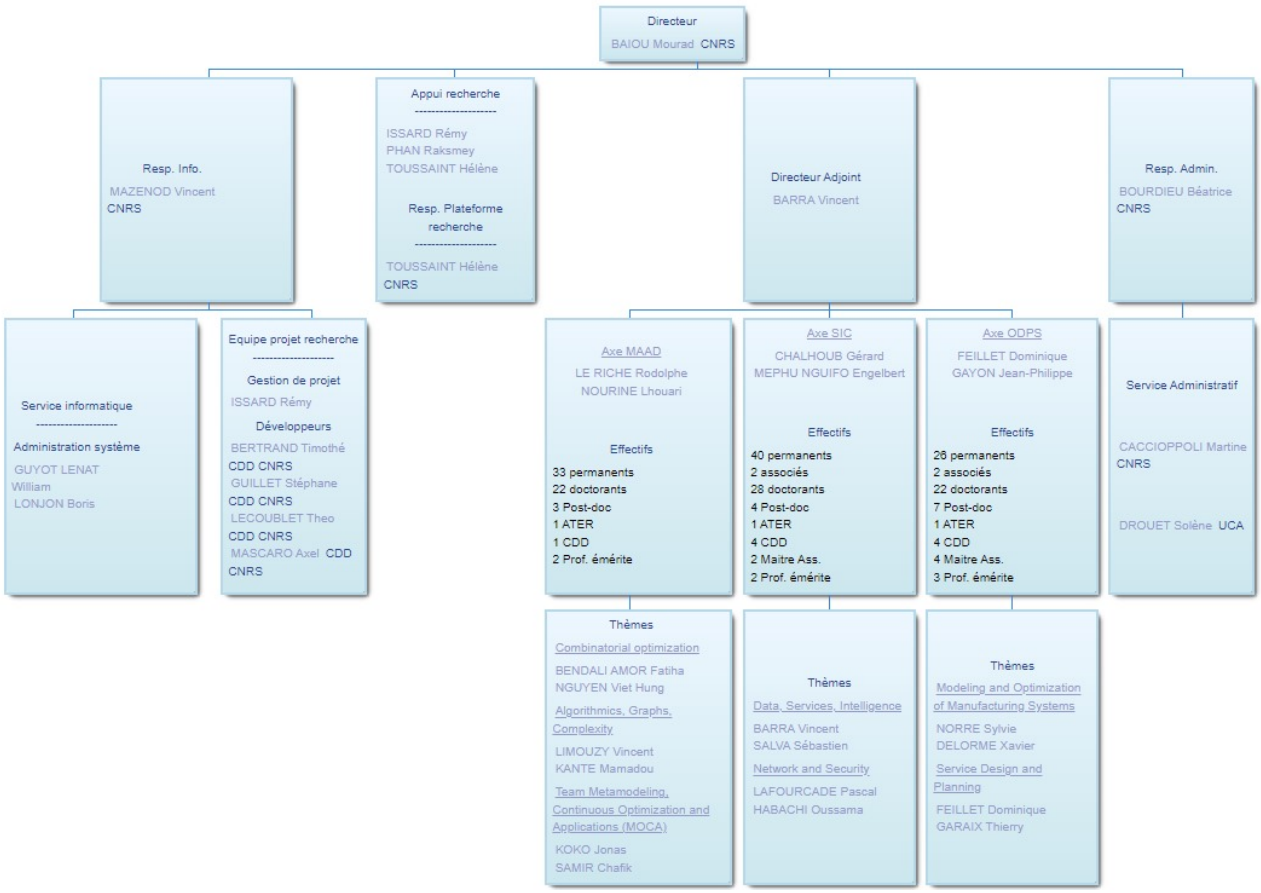

FIGURE 2 – Organigramme LIMOS. Source : https ://limos.fr/organigramme

1.2 Présentation du sujet

L'apprentissage par renforcement est compris par [1] comme une situation où l'apprenant ne sait pas quelles actions entreprendre dans un contexte donné et découvrira à partir des récompenses qui seront données pour chaque action entreprise. Les actions n'impactent pas toujours uniquement le moment présent où elles sont prises, mais peuvent également affecter les situations et actions futures et, par conséquent, les récompenses reçues. Ce système de tentative et d'erreur et les récompenses sont deux caractéristiques très importantes pour comprendre ce qu'est l'apprentissage par renforcement.

Dans ce sens, ce domaine d'étude ouvre la possibilité de former des agents (apprenants) à prendre des actions qui maximisent leur récompense à long terme, étant donné un environnement délimité par ses caractéristiques et son objectif spécifique. Dans ce cadre, nous avons quatre éléments définis par [1] :

1. **Politique :** Définit le comportement (les actions) de l'agent dans un environnement pour chaque état s à un temps t. En général, les politiques peuvent être stochastiques,

spécifiant la probabilité de chaque action.

2. **Signal de Récompense :** Cet élément représente le signal que l'environnement envoie à l'agent à chaque pas de temps, représenté par un nombre. Les récompenses sont le moyen d'indiquer à l'agent si l'événement survenu a été bénéfique ou néfaste pour lui et, à partir de là, de suggérer un changement de politique si la valeur reçue a été faible, dans le but de recevoir de meilleures récompenses à l'avenir.
3. **Fonction de Valeur :** Alors que la récompense indique la qualité de l'événement dans un sens immédiat, la fonction de valeur indique ce qui est bon à long terme. Cette valeur indique dans un état combien il peut espérer accumuler de récompenses à l'avenir, en tenant compte des états probables à atteindre et de leurs récompenses respectives.
4. **Modèle :** Le modèle représente une simulation du comportement de l'environnement et est capable de prédire, donné un état s et une action a, quel sera le prochain état de l'agent et sa récompense r. Ces modèles sont utilisés pour déterminer quelle action entreprendre en considérant les situations futures possibles sans nécessairement les avoir expérimentées.

Avec ces principaux concepts en tête, l'objectif de ce travail est de modéliser un système d'apprentissage par renforcement et d'implémenter un algorithme d'apprentissage qui fournira un modèle de cet environnement aux agents.

1.3 Contexte du sujet

Dans le contexte du monde réel, il existe divers services qui sont liés au problème d'affectation entre utilisateurs/clients et prestataires de services. Parmi ceux-ci, nous pouvons citer les taxis, où de nombreuses personnes dispersées demandent le service et un véhicule doit être envoyé pour répondre à cette demande, le service de livraison, où les livreurs reçoivent une demande de se rendre dans différents restaurants pour récupérer la nourriture et la livrer au client, ainsi que les services d'urgence, tels que les pompiers, la police et les ambulances, qui reçoivent une ou plusieurs demandes d'assistance et un des véhicules disponibles à travers la ville doit être mobilisé pour répondre à l'urgence. Dans ce sens, le travail réalisé est lié à ce type de situation, mais dans un contexte de recharge d'hydrogène, où des clients demandent un service de recharge et des véhicules sont disponibles pour effectuer le travail.

Par conséquent, cette situation implique deux éléments fondamentaux : Commandes et Véhicules. Chacun d'eux a ses propres attributs qui seront importants pour la modélisation de l'environnement en code. Pour que l'implémentation soit réalisable, certains attributs trouvés dans le monde réel ont été écartés dans un premier temps. Ainsi, les attributs

considérés pour chacun des éléments étaient les suivants :

Élément	Attributs
Commandes	Position
	Prix
	Durée du service
Véhicules	Position
	Quantité d'hydrogène disponible
	Temps de travail restant
	Qualité du service

TABLE 1 – Description des commandes et des véhicules

Ainsi, le scénario sera que les Commandes établiront leur liste de préférence de Véhicules à partir des attributs de chacun, et les Véhicules établiront leur liste de préférence des Commandes également en fonction des attributs de chacune d'elles. Pour définir l'affectation entre Commandes et Véhicules, la décision sera prise par l'algorithme de décision de Gale-Shapley, qui sera chargé de recevoir les listes de préférence des Véhicules et des Commandes et de réaliser l'attribution entre eux. La Figure 3 illustre ce processus :

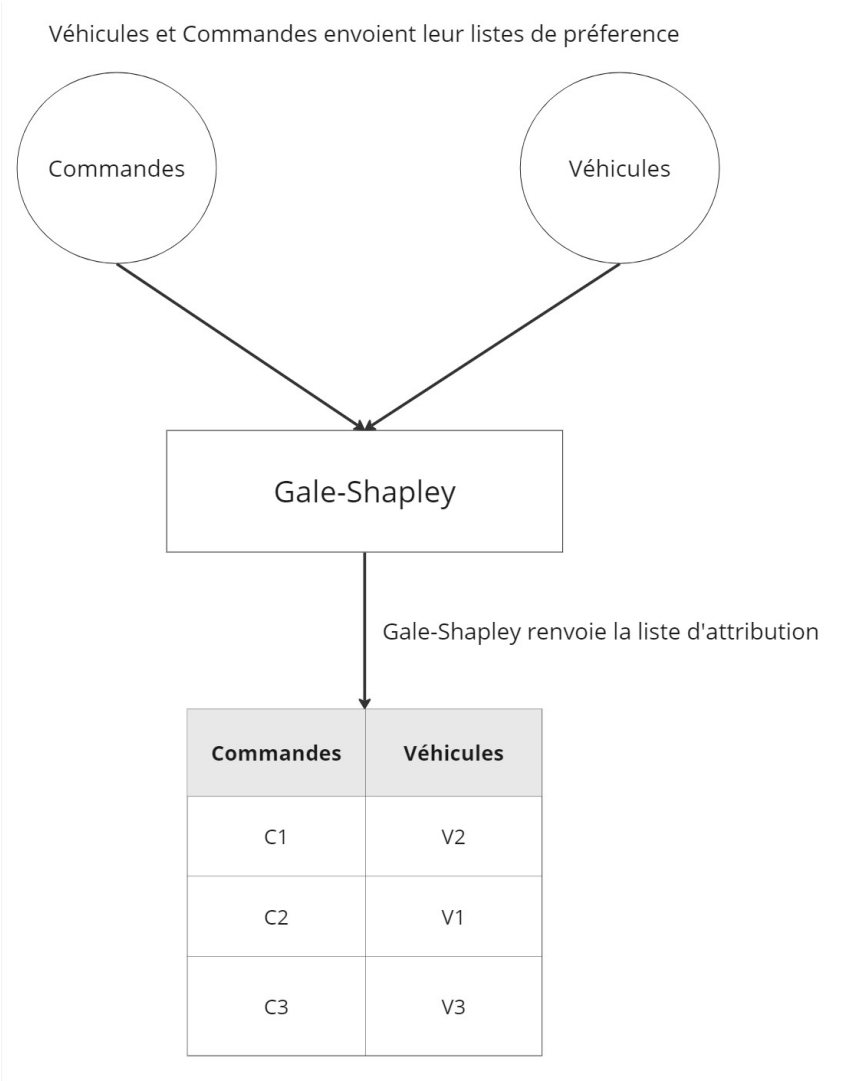

FIGURE 3 – Interaction des Véhicules et des Commandes avec l'algorithme d'affectation

Il est important de noter que les listes de préférences seront définies par les Commandes et les Véhicules à partir des poids attribués à chaque attribut présent en chacun d'eux. Une somme pondérée de la valeur des attributs avec les poids respectifs attribué seront utilisées pour définir la préférence de chaque Commande et Véhicule.

La Figure 4 et l'équation (1) montrent comment les Commandes vont classer les Véhicules par ordre décroissant :

Commandes	Poids
C1	[0.8, 0.5, 0.2, 0.6]
C2	[0.5, 0.6, 0.3, 0.7]
C3	[0.3, 0.8, 0.7, 0.2]

FIGURE 4 – Exemple de vecteur de poids des Commandes

Voici un exemple de la valeur attribuée au Véhicule V_1 pour la Commande C_1 :

$$\begin{aligned} & 0.5 \times 0.5 \text{ (Hydrogène disponible)} \\ & - 0.8 \times 0.2 \text{ (Distance)} \\ & + 0.2 \times 0.6 \text{ (Temps restant de travail)} \\ & + 0.6 \times 0.5 \text{ (Qualité du service)} \\ & = 0.51 \end{aligned} \tag{1}$$

À partir du vecteur de poids que chaque Commande définit, elles vont générer une liste individuelle de préférences. La Figure 5 illustre le résultat :

Commandes	Liste de Préference
C1	1. V2 2. V1 3. V3
C2	1. V1 2. V3 3. V2
C3	1. V3 2. V2 3. V1

FIGURE 5 – Exemple de liste de préférences ordonnée par Commandes à partir du vecteur de poids

Pour définir la liste de préférences de chaque Véhicule, le processus est similaire. Chaque Véhicule détermine son vecteur de poids et ordonne de façon décroissante une liste de préférences en fonction de la valeur des attributs des Commandes. Dans ce cas, la différence est que le vecteur de poids des Commandes est fixe et défini manuellement, tandis que celui des Véhicules constitue les actions des agents dans notre environnement et sera donc modifié ultérieurement par un algorithme d'apprentissage. La Figure 6 et l'équation (2) illustrent le processus de classement :

Vehicules	Poids
V1	[0.33, 0.5, 0.5]
V2	[0.2, 0.8, 0.3]
V3	[0.8, 0.3, 0.5]

FIGURE 6 – Exemple de vecteur de poids des Vehicules

Voici un exemple de la valeur attribuée à la Commande C_1 par le Véhicule V_1 :

$$\begin{aligned} & 0.5 \times 0.6\,(\text{Prix}) \\ & - 0.33 \times 0.8\,(\text{Distance}) \\ & - 0.5 \times 0.4\,(\text{Durée du service}) \\ & = -0.164 \end{aligned} \tag{2}$$

Au terme de ce processus, chaque Véhicule possédera également sa propre liste de préférences par rapport aux Commandes, comme illustré à la Figure 7 :

Commandes	Liste de Préference
C1	1. V2 2. V1 3. V3
C2	1. V1 2. V3 3. V2
C3	1. V3 2. V2 3. V1

FIGURE 7 – Exemple de liste de préférences ordonnée par Vehicules à partir du vecteur de poids

Comme dernière information importante concernant la modélisation de l'environnement, l'algorithme d'affectation fonctionne avec les deux listes de préférences en priorisant une, comme illustré dans le travail de recherche [2]. Ainsi, dans ce travail, la priorité est donnée aux Commandes, traitées comme des clients dans le monde réel. Autrement dit, l'algorithme essaie avant tout de satisfaire la préférence principale de chaque Commande. Cependant, cela n'est pas toujours possible étant donné l'existence des préférences des Véhicules et il se peut que plusieurs Commandes préfèrent le même Véhicule. Ainsi, le besoin d'apprentissage se fait sentir, car l'objectif est de faire en sorte que les Véhicules apprennent à ordonner leur liste de préférences à partir des informations des Commandes. Même si l'algorithme privilégie les Commandes lors de l'affectation, ils doivent savoir quelle sera la préférence principale de chaque Commande afin que cette priorité soit satisfaite et pour qu'ils puissent choisir les commandes que les souhaitent.

2 Description du travail réalisé

2.1 Description de l'environnement d'apprentissage

Dans cette partie, il sera décrit l'environnement d'apprentissage et l'algorithme utilisé pour résoudre le problème d'affectation entre Commandes et Véhicules dans le contexte de recharge d'hydrogène multi-agent. Cet environnement repose sur la mise en œuvre de classes représentant les Véhicules et les Commandes, et utilise l'algorithme de Gale-Shapley pour réaliser des affectations optimales.

Tout d'abord, chaque classe construite sera présentée et décrite pour comprendre la structure du projet, puis un diagramme sera montré indiquant les relations entre elles. De plus, pour écrire ce code, la documentation de la bibliothèque d'environnements pour l'apprentissage par renforcement, Gymansium [3], a été utilisée comme référence.

2.1.1 Classe Vehicule

Titre de la classe : Vehicle

Attributs

- **name :** str
  - La chaîne de caractères représentant le nom du Véhicule.
- **position :** Array[float]
  - Array représentant la position en coordonnées x et y.
- **hydrogen :** float
  - Nombre représentant la quantité d'hydrogène disponible.
- **remaining_working_time :** float
  - Nombre représentant le temps disponible pour le travail.
- **quality_of_service :** float
  - Nombre représentant la qualité du travail du véhicule.
- **weights :** Array[float]
  - Array représentant le poids accordé à chaque attribut de commande par le Véhicule.

— **preference :** List[Tuple(str, float)]

  — Liste de tuples représentant le nom de la Commande et la valeur qui lui est attribuée en multipliant les poids par les attributs.

— **is_matched :** bool

  — Booléen indiquant si le Véhicule a déjà été affecté à une Commande.

— **job :** Command

  — Indique la Commande à laquelle le Véhicule a été affecté.

— **index :** int

  — Index de la liste des préférences.

— **score :** float

  — Valeur attribuée par la somme pondérée des attributs de la Commande.

Méthodes

— **propose()** → Tuple[str, float]

  — Elle consiste à envoyer un tuple contenant le nom d'une des Commandes et son score.

— **is_available()** → bool

  — Envoi d'un booléen indiquant si le Véhicule est toujours sans Commande ou non.

— **update_score(new_score : float)** → None

  — Il suffit de mettre à jour le score.

— **reset_index()** → None

  — Réinitialise l'index.

2.1.2 Classe Commande

Titre de la classe : Command

Attributs

— **name :** str

  — La chaîne de caractères représentant le nom de la Commande.

— **position :** Array[float]

  — Array représentant la position en coordonnées x et y.

— **price :** float

  — Nombre représentant le prix de la Commande.

— **duration :** float

  — Nombre représentant la durée du travail.

— **weights :** Array[float]

  — Array représentant le poids accordé à chaque attribut des Véhicules par la Commande.

— **preference :** List[Tuple(str, float)]

  — Liste de tuples représentant le nom du Véhicule et la valeur qui lui est attribuée en multipliant les poids par les attributs.

— **is_matched :** bool

  — Booléen indiquant si la Commande a déjà été affecté à une Véhicule.

— **vehicle :** Vehicle

  — Indique le Véhicule auquel la Commande a été affecté.

— **index :** int

  — Index de la liste des préférences.

— **score :** float

  — Valeur attribuée par la somme pondérée des attributs du Véhicule.

Méthodes

— **propose()** → Tuple[str, float]

  — Elle consiste à envoyer un tuple contenant le nom d'un des Véhicules et son score.

— **is_available()** → bool

  — Envoi d'un booléen indiquant si la Commande est toujours sans Véhicule ou non.

— **update_score(new_score : float)** → None

  — Il suffit de mettre à jour le score.

— **reset_index()** → None

  — Réinitialise l'index.

2.1.3 Classe de l'algorithme d'affectation

Titre de la classe : AssigmentsVehicle

Attributs

- **assigments :** Dict[str, Union[Vehicle, Command]]
  - Dictionnaire représentant les Commandes et les Véhicules affectées par l'algorithme.
- **commands :** List[Command]
  - Liste des Commandes à affecter par l'algorithme.
- **vehicles :** List[Vehicle]
  - Liste des Véhicules à affecter par l'algorithme.
- **assigment_count :** int
  - Nombre d'affectations effectuées par l'algorithme.

Méthodes

- **assign(vehicle_name : str, command_name : str)** → None
  - Elle consiste à affecter une Commande avec un Véhicule.
- **unassign(vehicle_name : str, command_name : str)** → None
  - Elle s'agit d'annuler une affectation tout au long des itérations effectuées par l'algorithme.
- **reset()** → None
  - Réinitialise toutes les informations relatives aux affectations entre les Véhicules et les Commandes.
- **match()** → List[Set[str]]
  - Envoi d'une liste de correspondances définies par l'algorithme d'affectation (Gale-Shapley).
- **sets()** → List[Set[str]]
  - Méthode utilisée pour renvoyer la liste des affectations dans match().

2.1.4 Classe d'environnement d'apprentissage

Titre de la classe : MultiHydrogenRecharge

Attributs

- **num_vechicles :** int
  - Nombre de Véhicules placés dans l'environnement.
- **num_commands :** int

  - Nombre de Commandes placés dans l'environnement.
- **observation_space :** spaces.Dict
  - Dictionnaire de la bibliothèque du gym pour représenter les attributs des Commandes et des Véhicules définis dans le Table 1.
- **action_space :** spaces.Box
  - Méthode utilisée à partir de la bibliothèque du gym pour définir l'espace d'action continu des agents (Véhicules) dans l'environnement.
- **vehicles :** List[Vehicle]
  - Il s'agit d'une liste d'objets de la classe Vehicle en fonction du paramètre num_vehicles.
- **commands :** List[Command]
  - Il s'agit d'une liste d'objets de la classe Command en fonction du paramètre num_commands.
- **match_assigments_vehicle :** AssigmentVehicle
  - C'est un objet de la classe AssigmentVehicle qui sera chargé de faire correspondre les Commandes et les Véhicules à l'aide de l'algorithme de Gale-Shapley.

Méthodes

- **_get_observation()** → Array[float]
  - Array de valeurs du type float représentant les attributs des Véhicules et des Commandes, caractérisant l'état de l'environnement.
- **reset()** → Dict[int, np.ndarray]
  - Elle réinitialise les informations relatives à l'environnement et renvoie un dictionnaire de arrays créé en fonction du nombre d'agents (num_véhicules).
- **step(actions : np.ndarray)** → Dict[int, np.ndarray], List[int], Dict[int, bool]
  - Cette méthode reçoit les actions que les agents doivent entreprendre sous la forme d'un array et renvoie leur prochain état en fonction des actions entreprises, de la récompense pour chacune d'entre eux et du fait qu'ils sont tombés ou non dans un état terminal.

Le diagramme de la Figure 8 représente la relation entre les classes construites :

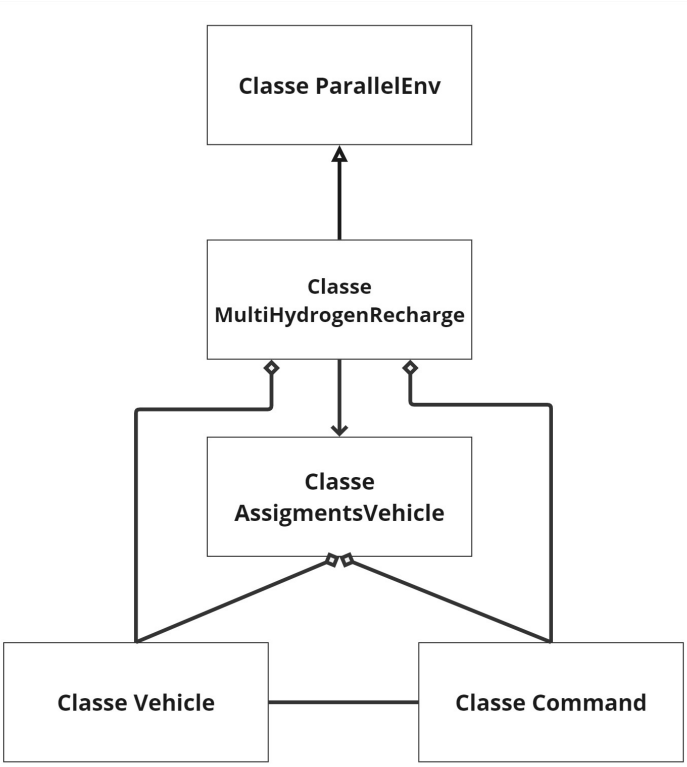

FIGURE 8 – Relations entre les classes du projet

Ce diagramme montre qu'il existe une relation d'association entre les Véhicules et les Commandes, une agrégation d'objets des classes Vehicle et Command à la fois dans la classe AssignmentsVehicle (Gale-Shapley) et dans la classe de l'environnement d'apprentissage. Une dépendance de l'environnement d'apprentissage par rapport à l'algorithme d'affectation et, enfin, une héritage de la classe ParallelEnv de la bibliothèque PettingZoo [4] pour la création d'environnements d'apprentissage multi-agents.

En plus des classes présentées, trois fonctions indépendantes ont également été construites pour que l'environnement fonctionne comme il a été modélisé. Ces fonctions sont : calculate_distance(position1 : float, position2 : float) (3), calculate_vehicle_score(command : Command, weights : np.ndarray, position : float) (4) et calculate_command_score(vehicle : Vehicle, weights : np.ndarray, position : float) (5).

Les calculs de chacune de ces fonctions se font mathématiquement de la manière suivante :

$$d(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2} \tag{3}$$

$$V_s = \text{prix} \cdot w_1 - \text{distance} \cdot w_2 - \text{durée} \cdot w_3 \tag{4}$$

$$
\begin{aligned}
C_s = {} & \text{quantité d'hydrogène} \cdot w_1 - \\
& \text{distance} \cdot w_2 + \\
& \text{temps de travail restant} \cdot w_3 + \text{qualité du service} \cdot w_4
\end{aligned} \tag{5}
$$

2.2 Description de l'algorithme d'apprentissage utilisé

Pour choisir l'algorithme d'apprentissage par renforcement à utiliser dans ce travail, il a été référencée l'article [5], qui décrit les principales méthodes utilisées dans ce domaine et montre de manière expérimentale comment le MADDPG (Multi-Agent Deep Deterministic Policy Gradient) a obtenu de meilleurs résultats en termes de convergence et de stabilité dans les expériences réalisées.

Selon la modélisation de l'environnement, en donnant aux actions un espace continu dans l'intervalle de [0, 1], des algorithmes comme le DQN (Deep Q-Network) ne deviennent pas efficaces en raison de leur besoin d'obtenir une distribution de probabilité pour chacune des actions présentes dans les états afin de définir une politique optimale π, comme présenté dans [1].

Ainsi, l'algorithme utilise le concept d'Actor-Critic, présent dans les algorithmes de policy gradient, qui, au lieu de calculer la distribution de probabilité pour chaque action, apprend en fait uniquement les statistiques de la distribution de probabilité [1]. Dans ce cas, l'Actor représente la politique de chaque agent et le Critic la valeur de la fonction d'action à partir des observations et des actions prises par tous les agents. Ce système est représenté dans le schéma fourni dans [5] :

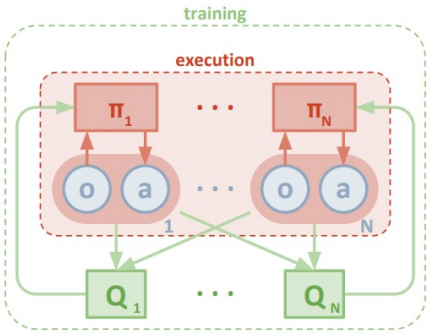

FIGURE 9 – Schéma pour l'entraînement et exécution de l'algorithme d'apprentissage

Ce schéma démontre que l'exécution de l'algorithme est décentralisée, c'est-à-dire que chaque agent observe son propre état et prend une action en fonction de sa propre poli-

tique. Cependant, la valeur Q (Q-Value) de chacun des agents est centralisée, en prenant en compte l'état et les actions de tous les agents. Ainsi, chaque agent prend connaissance de l'action choisie par les autres pour réaliser l'apprentissage.

Grâce à ce concept, et comme démontré dans l'article [5], cet algorithme devient applicable aussi bien dans un environnement compétitif que coopératif. Dans le cas étudié ici, il s'agit d'un environnement coopératif, car chaque Véhicule a pour objectif d'obtenir la Commande qui le souhaite, sans nécessairement être en compétition avec les autres.

L'algorithme complet tel que présenté dans [5] est le suivant :

Algorithm 1 Multi-Agent Deep Deterministic Policy Gradient pour N agents

for épisode = 1 à M **do**
  Initialiser un processus aléatoire $\mathcal{N}$ pour l'exploration des actions
  Recevoir l'état initial x
  for $t = 1$ à la longueur maximale de l'épisode **do**
    pour chaque agent i, sélectionner l'action $a_i = \mu_{\theta_i}(o_i) + \mathcal{N}_t$ par rapport à la politique actuelle et l'exploration
    Exécuter les actions $a = (a_1, \ldots, a_N)$ et observer la récompense r et le nouvel état x'
    Stocker (x, a, r, x') dans la mémoire tampon D
    $x \leftarrow x'$
    for chaque agent $i = 1$ à N **do**
      Échantillonner un minibatch aléatoire de S échantillons (x^j, a^j, r^j, x'^j) depuis D
      Définir $y^j = r_i^j + \gamma Q_{\mu'_i}(x'^j, a'^1, \ldots, a'^N) \mid a'^k = \mu'_k(o_k^j)$
      Mettre à jour le critique en minimisant la perte : $L(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_{\mu_i}(x^j, a_1^j, \ldots, a_N^j)\right)^2$
      Mettre à jour l'acteur en utilisant le gradient de politique échantillonné :
      $\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\mu_i(o_i^j)\nabla_{a_i} Q_{\mu_i}(x^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i=\mu_i(o_i^j)}$
    end for
    Mettre à jour les paramètres du réseau cible pour chaque agent i : $\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$
  end for
end for

Dans l'article où l'algorithme MADDPG a été présenté, les paramètres utilisés pour construire les réseaux neuronaux ont également été fournis, ce qui a été reproduit dans ce travail :

— **Optimizer :** Adam
— **Learning Rate :** 0.01
— **Tau :** 0.01
— **Gamma :** 0.95
— **Taille du Replay Buffer :** 10^6
— **Batch Size :** 1024

Un seul paramètre a été personnalisé :

— **Learning Frequency :** 5

3 Conception de la solution

3.1 Fonctionnement de l'environnement d'apprentissage

Dans la classe MutliHydrogenRecharge, il existe une logique pour attribuer des valeurs aux attributs présentés dans les Commandes et les Véhicules ainsi que pour définir la manière dont ils interagissent tout au long de la simulation. Il est donc nécessaire de fournir une explication de ce processus pour comprendre le résultat de ce travail et comment il a été atteint.

Comme il n'existait pas de données réelles d'une entreprise, les attributs ont été générés de manière aléatoire dans un intervalle déjà normalisé, allant de [0, 1]. Cette décision a été prise afin qu'aucun attribut n'ait une influence disproportionnée par rapport à un autre dans le choix des préférences par les Commandes et les Véhicules, et pour augmenter l'efficacité de l'apprentissage.

Pour l'initialisation des attributs des Véhicules, un des éléments du vecteur de valeurs sera choisi de manière aléatoire pour chaque agent. Ainsi, une des possibilités suivantes est assumée pour chaque attribut :

Attributs	**Valeurs**
Position	[0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
Hydrogen	[0.1, 0.3, 0.5, 0.7, 0.9]
Temps de travail restant	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
Qualité de service	[0.2, 0.6, 1.]

TABLE 2 – Vecteurs de possibilités pour les attributs des Véhicules

Les Commandes suivent le même processus de définition de leurs attributs :

Attributs	**Valeurs**
Position	[0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
Prix	[0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
Durée de service	[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]

TABLE 3 – Vecteurs de possibilités pour les attributs des Commandes

Les poids définis par les Commandes pour classer les Véhicules sont basés sur le vecteur :

$$[1, 1, 0, 0] \tag{6}$$

L'idée est que chaque Commande valorise deux attributs des Véhicules et que les deux autres ne soient pas pris en compte dans le calcul du score, de sorte qu'à chaque étape de l'environnement, les attributs valorisés soient différents, en mélangeant les 0 et les 1 du vecteur pour chacune des Commandes. Cela conduit à une plus grande diversité de préférences de Véhicules parmi les Commandes, rendant ainsi la tâche d'apprentissage plus nécessaire et se rapprochant du monde réel, où différents clients ont des profils variés.

Quant aux actions des Véhicules, elles sont définies par un vecteur de trois éléments, avec une valeur comprise entre 0 et 1 pour chacun d'eux, déterminant le poids de chaque attribut de la Commande lors du calcul du score. Ainsi, l'objectif est que les Véhicules apprennent à attribuer des poids plus importants aux attributs qui valorisent la Commande qui souhaite être desservie.

Dans cet environnement, l'état est défini par les éléments suivants :

— Positions des Véhicules
— Positions des Commandes
— Valeurs d'hydrogène pour chaque Véhicule
— Temps de travail restant pour chaque Véhicule
— Qualité de service de chaque Véhicule
— Prix des Commandes
— Durée du service pour chaque Commande
— Poids définis par chaque Commande

Cela fait en sorte que l'état soit entièrement observable par les agents dans l'environnement, c'est-à-dire que les véhicules possèdent leurs propres informations ainsi que celles des autres. Le calcul de la taille de la dimension de l'état est donné de la manière suivante :

$$\begin{aligned}\text{Dimension de l'État} = {} & 2 \cdot \text{Nombre de Véhicules} + 2 \cdot \text{Nombre des Commandes} \\ & + 1 \cdot \text{Nombre de Véhicules} + 1 \cdot \text{Nombre de Véhicules} \\ & + 1 \cdot \text{Nombre de Véhicules} + 1 \cdot \text{Nombre des Commandes} \\ & + 1 \cdot \text{Nombre des Commandes} + 4 \cdot \text{Nombre des Commandes} \quad (7)\end{aligned}$$

À chaque étape effectuée dans l'environnement, c'est-à-dire une action réalisée par les agents, chaque Véhicule reçoit une récompense selon le tableau de récompenses suivant :

Position de la Commande affectée	Récompense
1	10
2	-20
3	-50
4 ou plus	-100

TABLE 4 – Tableau des récompenses des agents

L'idée est que le Véhicule ne reçoit une récompense positive que lorsqu'il est affecté par sa préférence primaire, et qu'à partir de la deuxième position, il reçoit une valeur de récompense de plus en plus faible, constituant une pénalisation pour le classement effectué. Il est important de souligner qu'il ne sera pas toujours possible d'obtenir la préférence primaire même avec l'apprentissage, car dans plusieurs situations, il y aura des Commandes que peuvent préférer le même Véhicule, ce qui empêche que tous les Véhicules soient la préférence primaire de certaines Commandes.

À partir de la Commande attribuée au Véhicule, deux attributs changent : sa position est mise à jour pour celle de la Commande attribuée pour l'étape suivante et la quantité d'hydrogène disponible diminue proportionnellement à la durée du service demandé par la Commande, ce qui fait que les Commandes avec une durée plus longue consomment plus d'hydrogène disponible dans les Véhicules.

À chaque étape, de nouvelles Commandes sont générées, entraînant un échange d'informations chaque fois que les services demandés sont satisfaits. Enfin, dans cet environnement, il n'y a pas eu d'implémentation d'un état terminal en raison du fait qu'il s'agit d'une version initiale de la modélisation de l'environnement ; cependant, dans une situation réelle, un état pourrait être considéré comme terminal lorsque le Véhicule atteint une quantité d'hydrogène disponible de 0, ce qui l'empêcherait de satisfaire davantage de Commandes.

3.2 Entraînement et test de l'algorithme d'apprentissage

Pour implémenter l'algorithme d'apprentissage décrit dans le chapitre précédent, la bibliothèque AgileRL a été utilisée, en suivant la documentation [6] pour fournir les paramètres de manière cohérente en appliquant l'environnement d'apprentissage construit.

Lors de l'apprentissage, les variables suivantes ont été fournies :

Variables de la solution

— **device :** torch.device
  — Définit le dispositif à utiliser pour traiter les opérations tensorielles de la bibliothèque PyTorch.
— **state_dim :** np.ndarray
  — En fonction de ce qui est renvoyé par l'environnement, il définit la dimension de l'état observé par les agents. Dans le cas de ce qui a été mis en œuvre, il renvoie les positions de chaque Véhicule, les positions des Commandes, la quantité d'hydrogène disponible pour chaque Véhicule, le temps de travail restant pour chaque Véhicule, la qualité de service de chaque Véhicule, les prix des Commandes, la durée de travail de chaque Commande et les poids définis pour chacune des Commandes.
— **action_dim :** np.ndarray
  — Il définit la dimension des actions de chaque Véhicule, qui sera donnée par le nombre d'attributs fournis par les Commandes, en donnant un poids entre 0 et 1 à chacun d'entre eux.
— **discrete_actions :** bool
  — Variable booléenne qui indique si l'environnement a des actions discrètes ou continues, dans ce cas la valeur est False.
— **max_action :** np.ndarray
  — Il représente la valeur maximale qu'une action peut être entreprise par les agents, dans ce cas représentée par le vecteur de 1.
— **min_action :** np.ndarray
  — Il représente la valeur minimale à laquelle une action peut être entreprise par les agents, représentée dans ce cas par le vecteur de 0.
— **n_agents :** int
  — Fournit le nombre d'agents créés dans l'environnement, qui sera donné par le nombre de Véhicules choisis.
— **agents_ids :** List[str]
  — Il s'agit de la liste des identifiants de chaque agent créé dans l'environnement.
— **field_names :** List[str]
  — Il s'agit de la liste des champs à créer dans la mémoire de l'environnement afin que l'algorithme d'apprentissage puisse accéder à ces données et effectuer l'apprentissage via des réseaux neuronaux. Ces champs sont les suivants : état, action, récompense, état suivant et done (représentant si l'agent est tombé dans un état terminal ou non).

A partir des variables précédentes, un objet de l'algorithme MADDPG a été créé, comme suit :

```
agent = MADDPG(state_dims=state_dim,
                action_dims=action_dim,
                one_hot=False,
                n_agents=n_agents,
                agent_ids=agent_ids,
                max_action=max_action,
                min_action=min_action,
                discrete_actions=discrete_actions,
                device=device)
```

En outre, la variable mémoire a également été créée afin que l'algorithme d'apprentissage puisse l'utiliser pour apprendre à partir des données stockées.

— **memory :** MultiAgentReplayBuffer
  — Il s'agit de la structure de données qui sera utilisée par l'algorithme d'apprentissage pour stocker les expériences (transition état-action-récompense), afin de les échantillonner de manière aléatoire et de procéder à l'apprentissage.

Afin d'exécuter la boucle d'apprentissage et de simuler l'environnement plusieurs fois, il est nécessaire de définir le nombre d'étapes pour chaque épisode et le nombre d'épisodes à exécuter pour que l'algorithme puisse obtenir des données et s'entraîner sur celles-ci. Dans ce travail, les variables suivantes ont été construites pour la simulation :

— **episodes :** int
  — Nombre d'épisodes à exécuter avec l'algorithme, fixé à 10000 pour la simulation.

— **max_steps :** int
  — Nombre d'étapes qui seront exécutées au cours de chaque épisode. Ce ratio peut être considéré comme le nombre de fois où les commandes seront générées pour les Véhicules à servir au cours d'une journée (épisode), défini comme 10.

— **epsilon :** float
  — C'est la valeur qui fait l'epsilon, la variable responsable de la probabilité que l'agent prenne une décision aléatoire ou une décision renvoyée par la prédiction de l'algorithme d'apprentissage, fixée au début à 1,0.

— **epsilon_end :** float
  — Il s'agit de la valeur finale d'epsilon, fixée à 0,01.

— **eps_decay :** float

— Il s'agit du facteur de multiplication de l'epsilon, qui détermine sa décroissance au fil des épisodes, de sorte qu'au fur et à mesure que ceux-ci se déroulent, la probabilité de prendre une action aléatoire est plus faible et la prédiction faite par l'algorithme est davantage utilisée.

Pour calculer la convergence de l'apprentissage, la moyenne des récompenses est enregistrée tous les 200 épisodes, afin qu'au long du nombre d'épisodes établi comme 10000, il soit possible d'observer la courbe de l'augmentation de la moyenne des récompenses reçues.

À la fin de l'entraînement de l'algorithme, il est sauvegardé dans le répertoire sous une variable appelée maddpg_agent pour que le test puisse être réalisé. Lors du test, 10000 épisodes sont générés à nouveau, où les actions sont prises à 100% par l'algorithme, afin de vérifier son efficacité sur de nouvelles données, c'est-à-dire qui n'ont pas été utilisées pour l'entraînement.

4 Mise en œuvre du projet

Pour la réalisation de ce travail, le sujet de recherche a été proposé par le professeur Viet NGUYEN au laboratoire LIMOS, en collaboration avec un autre collègue de la 2ème année d'ingénierie du programme d'échange Brafitec, Marcelo ISIDORO. Le professeur Rafael COLARES a également été présent pendant le développement du projet.

Lors de la proposition du sujet, le professeur Viet NGUYEN a orienté les premières tâches afin que mon collègue et moi puissions commencer le projet et travailler ensemble pour atteindre l'objectif final de la recherche. Il a été convenu que je me concentrerais sur la partie apprentissage et que mon collègue se concentrerait sur l'algorithme d'affectation, de sorte qu'au cours du stage, nous pourrions fusionner ces éléments dans l'environnement d'apprentissage, qui serait construit en collaboration.

La Figure 10 montre le diagramme de Gantt idéal pour le travail réalisé :

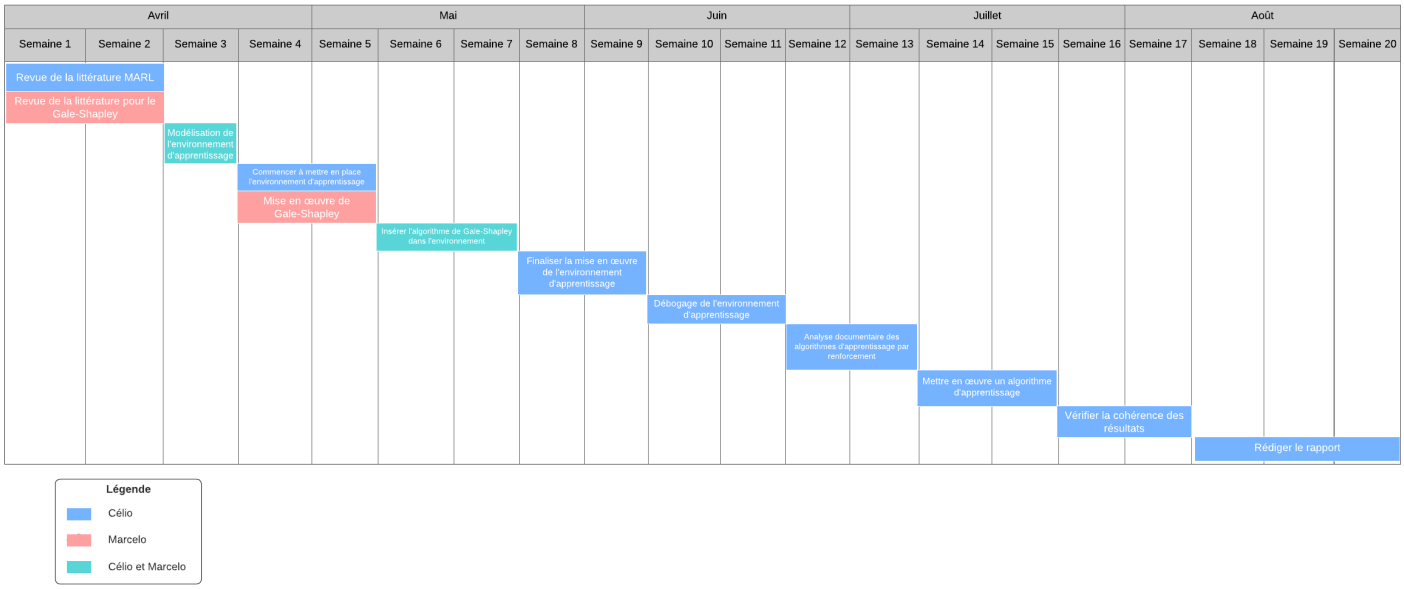

FIGURE 10 – Diagramme de Gantt du Stage - Ideal

Cependant, certains défis sont apparus au cours de cette période et la planification ne s'est pas déroulée comme prévu idéalement. La Figure 11 montre le diagramme de Gantt réel des étapes suivies pendant le stage :

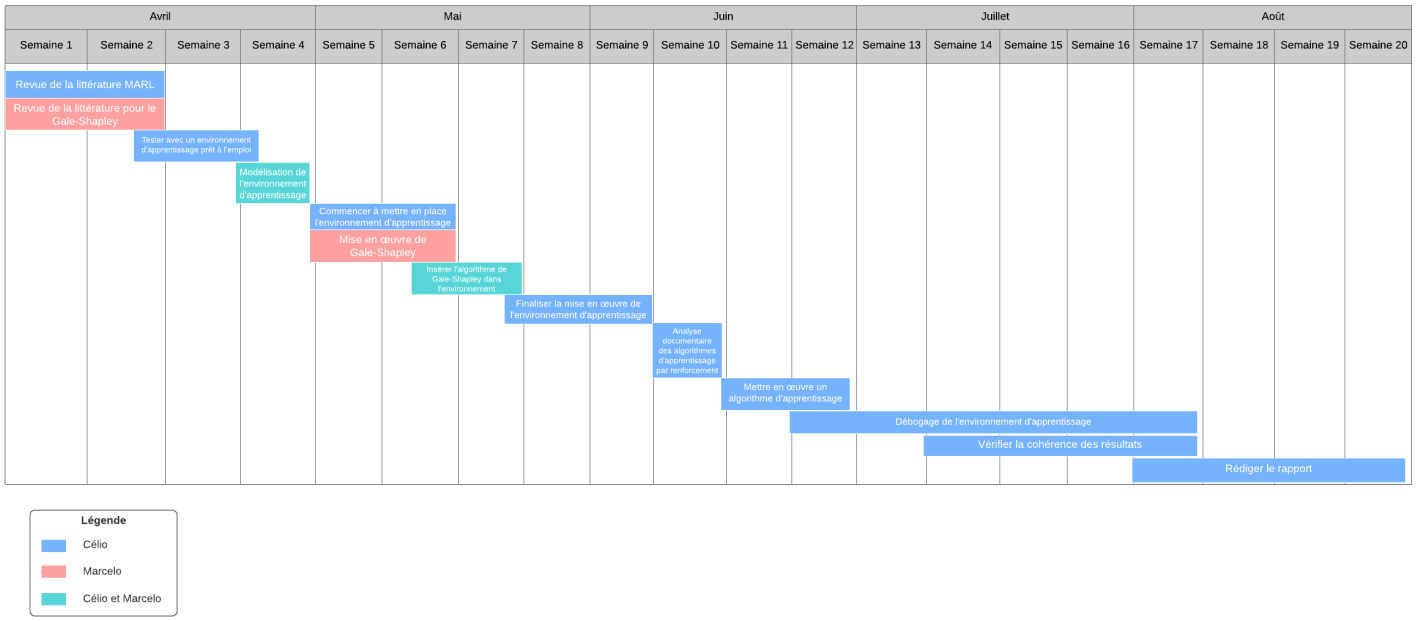

FIGURE 11 – Diagramme de Gantt du Stage - Réel

Comme observé, dans la phase initiale, il a d'abord été question de comprendre ce qui devait être fait, s'il était possible d'utiliser un environnement existant et d'appliquer simplement les algorithmes ou s'il était nécessaire de construire notre propre modélisation. À partir de là, l'environnement a été modélisé et implémenté, mais pour appliquer l'algorithme d'apprentissage, il est devenu nécessaire de réaliser certaines adaptations.

Ensuite, l'algorithme d'affectation a été intégré et les tests pour la réalisation de l'apprentissage ont également commencé à être effectués. Cependant, le fonctionnement de l'algorithme d'apprentissage n'était pas cohérent, et la phase de débogage a été entamée, ce qui a pris un temps considérable dans ce stage et n'avait pas été prévu initialement. La phase finale de débogage et de vérification des résultats a été cruciale, car ce n'est qu'une fois terminée que l'environnement a pu être considéré comme totalement prêt et la convergence de l'algorithme a été prouvée.

Par conséquent, ce travail a nécessité de nombreuses recherches, allant de la manière de développer un environnement d'apprentissage, comprendre comment intégrer l'algorithme d'affectation à l'environnement, réaliser une revue bibliographique des algorithmes d'apprentissage, et lors de la vérification des résultats, trouver les raisons pour lesquelles il n'y avait pas de convergence, pour finalement aboutir au débogage final.

5 Résultats et Discussion

Pour les résultats du travail, une représentation graphique des récompenses obtenues par l'algorithme au cours des épisodes a été utilisée, ainsi qu'une comparaison avec les résultats obtenus dans les mêmes circonstances avec la prise de décision aléatoire.

Le graphique ci-dessous montre la courbe de convergence de l'apprentissage de l'algorithme lors de son entraînement :

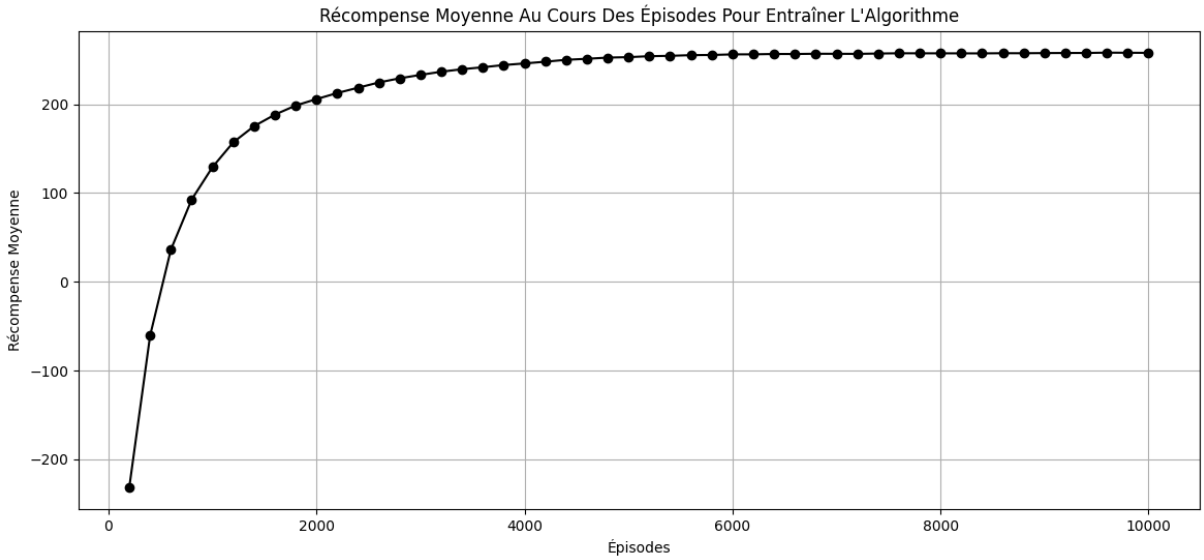

FIGURE 12 – Graphique des récompenses reçues lors de l'apprentissage de l'algorithme

Un graphique a également été généré pour comparer la décroissance de l'epsilon (probabilité qu'une action aléatoire se produise) par rapport à la récompense moyenne reçue.

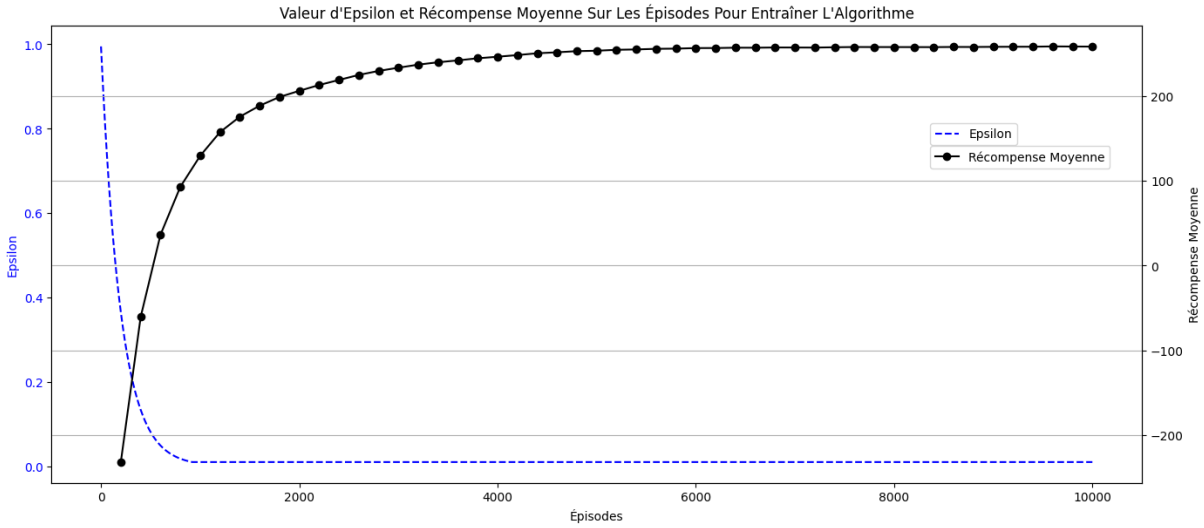

FIGURE 13 – Graphique de la décroissance d'epsilon et de la moyenne des récompenses reçues

Maintenant, l'application de l'algorithme entraîné dans une simulation de test :

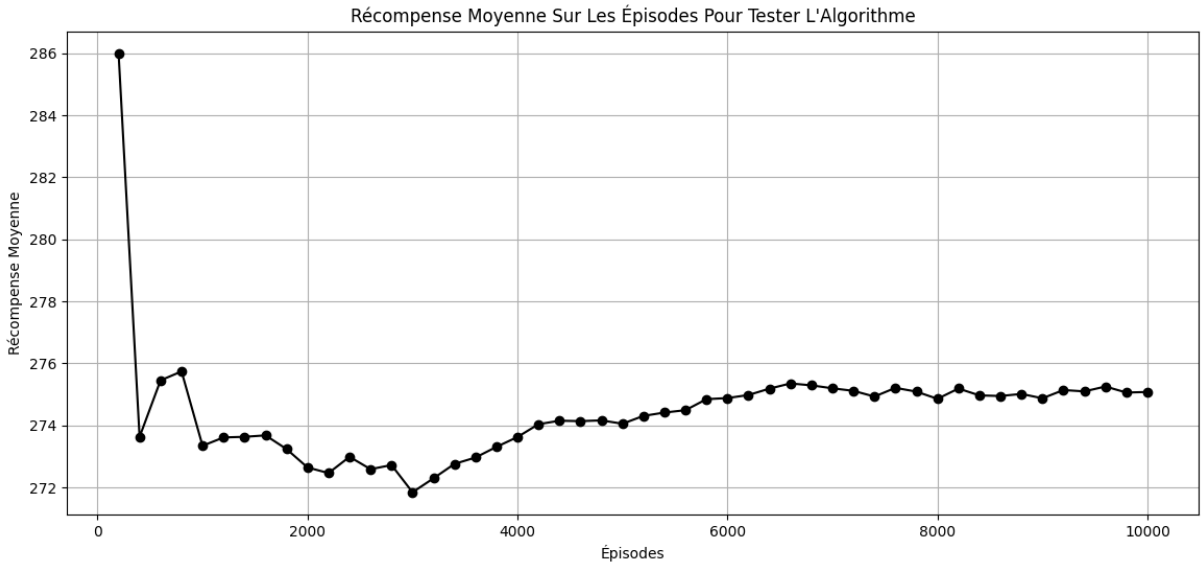

FIGURE 14 – Graphique des récompenses reçues lors du test de l'algorithme

Le résultat des actions aléatoires est indiqué ci-dessous :

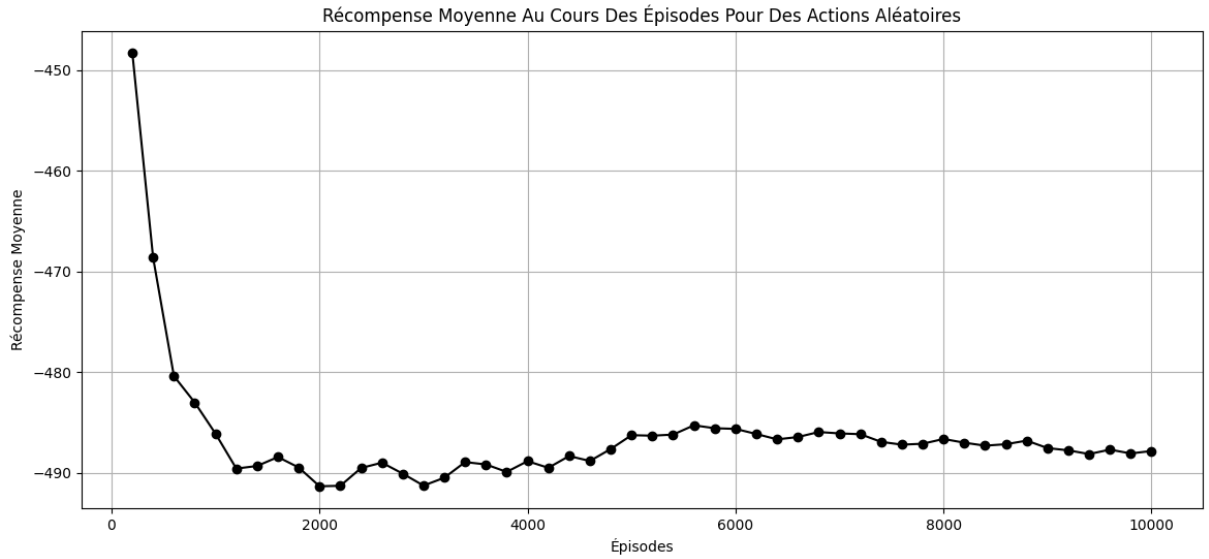

FIGURE 15 – Graphique des récompenses reçues pour des actions aléatoires

De cette manière, il est possible de visualiser le grand avantage de l'utilisation de cet algorithme d'apprentissage par renforcement dans le contexte de l'environnement créé, accordant des récompenses bien plus élevées aux agents de manière constante.

Ce résultat peut être visualisé de manière descriptive en comparant les deux tableaux ci-dessous :

TABLE 5 – Statistiques descriptives pour le test de l'algorithme

Statistique	**Valeur**
Nombre d'échantillons	50.000000
Moyenne	274.473514
Écart-type	1.944757
Minimum	271.840000
1er Quartile	273.611875
Médiane	274.665302
3e Quartile	275.091687
Maximum	286.000000

TABLE 6 – Statistiques descriptives pour les actions aléatoires

Statistique	**Valeur**
Nombre d'échantillons	50.000000
Moyenne	-486.453218
Écart-type	6.453300
Minimum	-491.320000
1er Quartile	-488.965724
Médiane	-487.411389
3e Quartile	-486.205556
Maximum	-448.250000

En comparant les deux moyennes, on observe une augmentation de 177,23% de la valeur moyenne de la récompense reçue par les agents dans le test et un écart-type plus faible.

En analysant la récompense moyenne dans les deux scénarios au fil des épisodes, il est possible d'affirmer que, pour l'algorithme entraîné, les véhicules obtiennent une récompense de 27,44 par step, ce qui, d'après le Tableau 4, démontre que pratiquement 3 des 4 véhicules construits dans ce scénario parviennent à être affectés par leur préférence principale. En revanche, pour les actions aléatoires, la récompense moyenne par step est de -48,64, ce qui indique que 2,43 véhicules prennent la préférence secondaire.

Pour reproduire les résultats obtenus dans cette expérimentation, il est possible d'accéder au dépôt du code sur GitHub, via le lien : https ://github.com/celiolucaslm/Multi-Agent-Hydrogen-Recharge. Les paramètres utilisés pour obtenir les résultats présentés étaient *num_vehicles* égal à 4, *num_commands* égal à 4 et *seed* pour reproduire la même aléatoire égal à 42.

Ainsi, avec l'étape de validation du test de l'algorithme, l'objectif de l'environnement implémenté a été atteint, avec une convergence de l'apprentissage et un avantage significatif par rapport aux actions aléatoires.

6 Conclusion

Le travail réalisé durant ce stage est considéré comme terminé et les objectifs fixés au début des activités ont été atteints. Il a été déterminé de construire l'environnement, de mettre en œuvre les algorithmes d'affectation et d'apprentissage, et de réaliser la convergence de l'apprentissage pour les agents, ce qui a été accompli et présenté dans ce rapport. Les difficultés techniques rencontrées tant pour la mise en œuvre de l'environnement que pour la réalisation de la convergence de l'apprentissage ont été résolues durant le processus de travail déterminé, ce qui a permis d'acquérir une compréhension approfondie du sujet traité.

L'intérêt de cette étude était de tester une approche différente de l'apprentissage par renforcement et d'apporter une situation contemporaine du monde réel à simuler dans le domaine de l'étude MARL. Grâce à ce travail, plusieurs connaissances ont été élargies et découvertes, telles que l'architecture d'un environnement d'apprentissage, la compréhension des algorithmes existants pour entraîner les agents et l'utilisation de concepts fondamentaux en programmation tels que la programmation orientée objet.

Pour ce qui avait été proposé initialement, le travail a été finalisé, cependant, un grand défi demeure : améliorer l'environnement construit en ajoutant de nouveaux attributs aux éléments (Commandes et Véhicules) et intégrer une base de données réelle à l'environnement, ce qui rendra le projet encore plus robuste et applicable pour la résolution d'un problème 100% pratique du monde réel.

7 Réferences Bibliographiques

[1] Sutton, R. S., et Barto, A. G., « Reinforcement Learning : An Introduction », 2018.

[2] Teo, C.-P., Sethuraman, J., et Tan, W.-P., « Gale-Shapley Stable Marriage Problem Revisited : Strategic Issues and Applications », 2001.

[3] « https ://gymnasium.farama.org », 2024.

[4] « https ://pettingzoo.farama.org », 2024.

[5] Lowe, R., Wu, Y., Tamar, A., Harb, J., Abbeel, P., et Mordatch, I., « Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments », 2017.

[6] «https ://docs.agilerl.com/en/latest/», 2024.