REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École Nationale Polytechnique

Département d'Automatique
Laboratoire de Commande des Processus
Mémoire de projet de fin d'études
en vue de l'obtention du diplôme d'ingénieur d'état en Automatique

Optimisation Simultanée et Planification de Trajectoires pour une Formation d'Agents

Réalisé par :

Nour MITICHE

Zakaria Elhabib BOUZIT

Présenté et Soutenu publiquement le 12 Juillet 2021 devant le jury composé de :

Président	O. STIHI	MAA	ENP
Examinateur	E.M. BERKOUK	Pr.	ENP
Promoteur	M. TADJINE	Pr.	ENP
Promoteur	M. CHAKIR	MCA	ENP

ENP 2021

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École Nationale Polytechnique

Département d'Automatique
Laboratoire de Commande des Processus
Mémoire de projet de fin d'études
en vue de l'obtention du diplôme d'ingénieur d'état en Automatique

Optimisation Simultanée et Planification de Trajectoires pour une Formation d'Agents

Réalisé par :

Nour MITICHE

Zakaria Elhabib BOUZIT

Présenté et Soutenu publiquement le 12 Juillet 2021 devant le jury composé de :

Président	O. STIHI	MAA	ENP
Examinateur	E.M. BERKOUK	Pr.	ENP
Promoteur	M. TADJINE	Pr.	ENP
Promoteur	M. CHAKIR	MCA	ENP

ENP 2021

Dédicaces

A mes très chers parents, pour tous leurs sacrifices, soutien et amour, tout au long de mes études. Que Dieu les protège.

A Wissam, Ikram, Abdelhak, Bouchra et Cherifa.

à l'équipe Auto-mates, Rahim, Ramzi, Soheib, Abderahman, Mohammed, Anis, Oussama.

à la famille polytechnicienne, que j'ai rencontrée.

Je dédie ce travail.

Zakaria Elhabib BOUZIT

Dédicaces

Je dédie ce modeste travail à mes très chers parents à qui je dois tout, qui m'ont toujours soutenue et encouragée, et qui ont fait de moi la personne que je suis aujourd'hui. Jamais je ne vous remercierai assez.

À mes sœurs Rym, Zineb et Hiba,

et à tous mes ami(e)s.

Nour MITICHE.

Remerciements

Avant toute chose, nous louons Allah le Tout-Puissant qui nous a accordé le savoir, la patience et le courage pour réaliser ce modeste travail.

Nous tenons à exprimer nos vifs remerciements à nos encadreurs, Pr. Tadjine et Dr. Chakir.

Notre profonde gratitude va également au président et aux membres du jury, pour l'honneur qu'ils nous accordent en acceptant d'examiner notre travail.

Nous remercions nos enseignants, en particulier ceux du département d'automatique, pour leurs efforts voués à nous transmettre le savoir.

Nous tenons à exprimer notre très grande gratitude, et notre profonde affection à nos parents respectifs pour leurs encouragements, leur patience et leur grand soutien durant toutes ces années d'études.

Nous remercions enfin tout ceux qui ont contribué de près ou de loin à la réalisation de ce projet, plus particulièrement ceux qui partagent leurs savoir, expérience et travaux, gratuitement sur internet, au profit de la science.

ملخص

يعد تخطيط المسار أحد التقنيات الرئيسية لتسهيل القدرة على المناورة المستقلة للروبوتات المتنقلة. تصبح هذه المشكلة أكثر تعقيداً في المجالات التي تتطلب بنية متعددة الوكلاء لتكون قادرة على أداء المهام المعقدة للغاية بالنسبة إلى وكيل واحد. الهدف هو الحصول على مسارات مثالية خالية من الاصطدامات تضمن تحقيق الهدف المشترك وفقاً لمعايير معينة.

في ورقتنا البحثية ، نقدم حلين مختلفين: التعلم المركزي القائم على التعلم المعزز(DQN) ، والاستكشاف اللامركزي على أساس *RRT.

أخيرا نقترح حلا أفضل يعتمد على أمثلية Pareto في الهيكل اللامركزي، و تحسين أداء التعلم للحل المركزي.

تم حساب هذه الحلول في Python وتمت محاكاتها في MATLAB على الروبوتات التفاضلية المتنقلة.

الكلمات المفتاحية تخطيط المسار، الأنظمة متعددة الوكلاء، التعلم المعزز العميق، البحث عن الحل الأمثل، RRT.

Abstract

Path planning is one of the key technologies to facilitate autonomous maneuverability of mobile robots. This problem gets more complex in domains that require a multi-agent structure to be able to perform tasks that are too complex for a single agent. The goal is to obtain optimal collision-free trajectories that guarantee the accomplishment of the common goal according to particular criteria.

In our paper, we present two different solutions : centralized learning based on Reinforcement Learning (DQN), and decentralized exploration based on RRT*. We finally propose a better solution based on Pareto optimality in the decentralized structure, and an improvement in the learning performance of the centralized solution.

These solutions were computed in Python and simulated in MATLAB on mobile differential robots.

Keywords Path planning, Multi-Agent Systems, RRT, Deep Q-Learning, Optimization.

Résumé

La planification de trajectoire est l'une des technologies clés pour faciliter la manœuvrabilité autonome des robots mobiles. Cette problématique devient plus complexe dans les domaines qui nécessitent une structure multi-agent afin d'être en mesure d'effectuer des tâches trop complexes pour un agent unique. Le but est d'obtenir des trajectoires optimales sans collision qui garantissent l'accomplissement de l'objectif commun selon des critères particuliers.

Dans notre mémoire, nous présentons deux solutions différentes : l'apprentissage centralisé à base de Reinforcement Learning (DQN), et l'exploration décentralisée à base de RRT*. Nous avons fini par proposer une solution plus performante basée sur l'optimalité de Pareto dans la structure décentralisée, ainsi qu'une amélioration de l'apprentissage dans la solution centralisée.

Ces solutions ont été calculées sous Python, puis simulées sous MATLAB sur des robots différentiels mobiles.

Mots-Clés Planification de trajectoires, Systèmes Multi-Agents, RRT, Deep Q-Learning, Optimisation.

Table des matières

Table des figures

Liste des Abréviations

RRT : Rapidly-exploring Random Trees

RRT : Rapidly-exploring Random Trees

RL : Reinforcement Learning

MARL : Multi-Agent Reinforcement Learning

TD : Temporal Difference

PG : Policy Gradient

DDPG : Deep Deterministic Policy Gradient

MCTS : Monte-Carlo Tree Search

HAS : Hyper Action Space

GA : Genetic Algorithms

ACO : Ant Colony Optimization

PSO : Particle Swarm Optimization

PRM : Probabilistic Road Maps

MDP : Markov Decision Process

DQN : Deep Q-Network

QL : Q-Learning

SARSA : State-Action-Reward-State-Action

MG : Markov Games

CP : Centralized Policy

NN : Neural Network

Liste des Symboles

π : Policy

γ : Facteur de remise

α : Taux d'apprentissage

ϵ : Facteur d'exploration

δ : Solution réalisable

θ : Vecteur des paramètres du réseau de neurones

a : Action

s : State

r : Reward

T : Tree

Introduction Générale

Les systèmes robotiques autonomes continuent d'être sollicités pour effectuer une multitude de tâches. Comme pour les humains, les équipes d'agents autonomes sont capables d'accomplir plusieurs tâches en parallèle et, en outre, d'accomplir des tâches bien plus complexes qu'un seul agent.

Considérons, par exemple, un entrepôt automatisé tel que celui illustré dans la figure suivante.

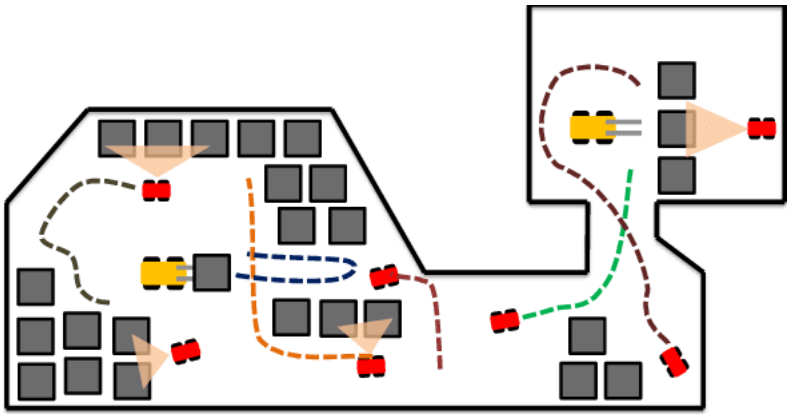


FIGURE 0.1: Une équipe de chariots élévateurs autonomes et de robots de guidage au sol opérant dans un environnement d'entrepôt contraint.

L'objectif global de l'équipe d'agents autonomes opérant dans cet entrepôt est de gérer efficacement le flux des stocks. À cette fin, les agents se voient attribuer des tâches telles que le tri et le déplacement des produits.

Cependant, comme l'illustre la figure, les agents doivent également éviter les collisions avec les objets de l'environnement et les autres agents pendant l'exécution de ces tâches.

Cela souligne une exigence évidente mais fondamentale pour pratiquement tout système multi-agent : la capacité de naviguer en toute sécurité dans l'environnement dans lequel les tâches doivent être exécutées.

Dans tout contexte pratique, les agents seront limités dans leur mobilité par des contraintes dynamiques, telles qu'un rayon de braquage minimum, ainsi que par des limites d'actionnement. Ces limites deviennent de plus en plus importantes lorsque des équipes autonomes sont déployées dans des environnements encombrés, y compris à l'intérieur. Par conséquent, tout planificateur de trajectoire multi-agent pratique doit tenir compte explicitement de ces contraintes afin de garantir une navigation sûre.

L'un des avantages majeurs des systèmes multi-agents est la possibilité de les utiliser pour accomplir des tâches qui pourraient mettre en danger la vie de l'homme si elles devaient se faire manuellement.

D'autre part, le caractère dangereux de certaines tâches fait qu'il serait compliqué voire impossible pour un agent unique de les effectuer seul. On peut citer comme exemples de tâches pareilles le déminage, la recherche et le sauvetage, et le nettoyage des déversements toxiques. La complexité du problème fera appel à une architecture multi-agent bien définie.

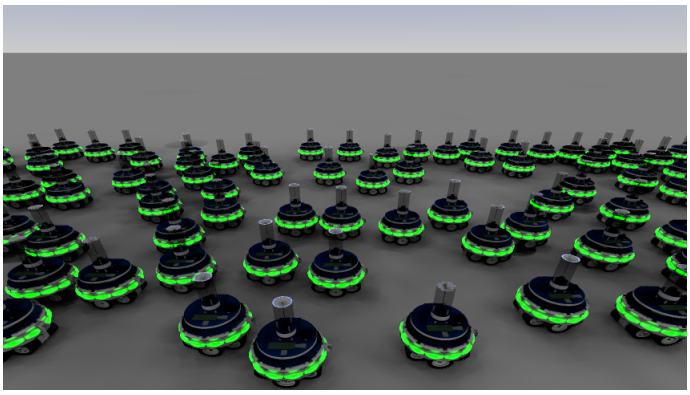


FIGURE 0.2: Formation d'agents pour la surveillance intelligente

À mesure que la taille de ces équipes augmente et qu'elles opèrent dans des environnements plus vastes (par exemple dans des missions ISR "Intelligence, Surveillance et Reconnaissance"[28], ou dans une tâche de recherche et

exploitation, ou des tâches d'extinction des feux de forêt..), la complexité du problème s'accroît considérablement, dépassant rapidement les capacités d'un planificateur unique et centralisé.

La communication nécessaire à un planificateur centralisé pour gérer tous les agents devient également prohibitive pour les problèmes à grande échelle. Cela motive le développement de planificateurs décentralisés, où chaque agent prend des décisions locales.

Cependant, cette approche n'est pas sans poser de problèmes, et tout planificateur décentralisé doit veiller à éviter les décisions locales contradictoires.

L'objectif de notre mémoire est d'étudier le problème de la planification de chemins dynamiquement faisables à travers des environnements complexes à deux dimensions (2-D) pour une équipe d'agents autonomes qui ont pour but d'atteindre des objectifs spécifiques.

Chapitre 1

Problématique

1.1 Positionnement du problème

La planification de trajectoire est l'un des problèmes les plus étudiés dans le domaine de la robotique. L'objectif principal de tout algorithme de planification de chemin est de fournir un chemin sans collision d'un état de départ à un état d'arrivée dans l'espace de configuration du robot.

Une représentation communément adoptée d'un agent aux fins de la planification de trajectoire est obtenue en considérant le robot comme un point mobile dans un espace approprié. Les obstacles de l'espace de travail sont également représentés dans le même espace en adoptant un mappage de transformation approprié. À cette fin, il est naturel de se référer aux coordonnées généralisées du système mécanique, dont la valeur identifie la configuration du robot. Cela permet d'associer à chaque pose du robot un point dans l'espace C (espace de configuration), qui est alors défini comme l'ensemble de toutes les poses admissibles que le robot peut prendre. L'espace de configuration est l'espace des articulations, l'espace dont les coordonnées sont l'état de chaque articulation du robot.

Bien que cette approche soit principalement utilisée en robotique pour la planification de la trajectoire d'un robot manipulateur, la représentation uniforme décrite et l'introduction de l'espace C permettent d'appliquer des algorithmes de planification qui ont été largement traités dans la littérature à un large éventail de problèmes de planification.

Lorsque le robot se déplace d'une configuration à une autre, un coût est

associé à ce mouvement. Dans un espace de configuration simple à 2 dimensions pour un robot planaire, le coût pour aller d'un endroit à un autre peut être mesuré en distance euclidienne.

Ce coût est ce qui détermine la qualité du chemin dans l'espace de configuration, si il est très élevé, cela signifie que le robot effectue trop de mouvements inutiles pour atteindre son objectif.

1.2 Hypothèses

Notre objectif étant centré sur la planification de trajectoires sans collisions de façon à optimiser le coût correspondant, i.e la longueur de la trajectoire dans une formation d'agents, nous avons établi des hypothèses simplificatrices concernant la nature de l'agent. Ainsi, dans les différents algorithmes que nous allons utiliser, la forme de l'agent importe peu. Il sera considéré comme un point matériel, et son dimensionnement sera traduit par un rayon de sécurité. La commandabilité et la stabilité de l'agent seront également présupposées.

1.3 Organisation

Dans notre mémoire, on a commencé par établir un bref survol des différentes approches dans le domaine de planification de trajectoires, plus particulièrement les approches classiques et heuristiques, dans le chapitre 2 "État de l'Art". On a ensuite choisi quelques algorithmes parmi les plus pertinents et les plus développés dans chacune des approches, qu'on a détaillés dans les chapitres 3 "Rapidly-Exploring Random Trees Planners" et 4 "Reinforcement Learning". Par la suite, dans le chapitre 5 "Implémentations et résultats", nous avons évalué l'approche classique (RRT) et heuristique (RL) en termes d'efficacité de solution dans les deux cas : pour un agent unique et multi-agent, et établi une étude comparative dans différents cas de figure pour en tirer des conclusions. Nous avons conclu par employer les solutions proposées dans une application réelle à base de robots mobiles dans un environnement 2D.

Chapitre 2

État de l'Art

2.1 Classification des approches existantes

2.1.1 Selon l'architecture de commande

2.1.1.1 Commande Centralisée

Une approche centralisée utilise des informations globales et planifie les trajectoires de tous les robots traités comme une seule entité. L'équipe de robots est considérée comme un système composite, auquel on applique un algorithme classique de planification de trajectoire pour un seul robot.

Dans l'algorithme centralisé introduit par Parsons et Canny [7], une décomposition de l'espace libre en cellules est d'abord calculée. Ensuite, il recherche le chemin dans le graphe d'adjacence résultant. En dépit de certains avantages liés au fait qu'elles sont complètes et optimales, le temps de calcul des méthodes centralisées augmente de façon exponentielle lorsque le nombre de robots augmente. D'autre part, les approches découplées sont rapides, mais parfois au prix d'une perte de complétude et d'optimalité. [2]

L'approche centralisée adopte une vision descendante du processus de synthèse du comportement. Ici, un agent central qui dispose d'un lien de communication duplex avec chaque membre du groupe observe simultanément les états des agents et de l'environnement, et traite cette base de données d'une manière conforme à l'objectif du groupe et aux contraintes de comportement.

Il génère ensuite des séquences synchronisées d'instructions d'action pour chaque membre.

Les instructions sont ensuite communiquées aux agents respectifs pour qu'ils modifient progressivement leurs trajectoires et atteignent leur destination en toute sécurité. Dans ce mode de comportement, la génération de la solution sans conflit satisfaisant les contraintes et les objectifs, commence par la construction de l'hyper espace d'action (HAS) du groupe.

Le HAS contient l'espace de toutes les actions ponctuelles admissibles que les agents peuvent tenter de projeter. Il est ensuite recherché pour trouver une solution qui est à son tour communiquée aux agents. Les agents exécutent la solution de manière réflexe, en espérant que leurs actions mèneront à la conclusion souhaitée.

Il est bien connu que, dans la vie réelle, toute solution générée par un mécanisme centralisé est de courte durée. La nature dynamique des environnements réels entraîne un décalage entre les conditions supposées au moment où le contrôleur commence à générer la solution, et les conditions réelles au moment où la solution est remise aux agents pour l'exécution.

Malgré la tentative d'atténuer ce problème en dotant les agents de capacités locales de détection et de prise de décision, les systèmes centralisés à grande échelle souffrent toujours de graves problèmes, dont les suivants :

— Presque tous les problèmes de planification et de contrôle centralisés sont connus pour être PSPACE-complets [1], avec une complexité dans le pire des cas qui croît exponentiellement avec le nombre d'agents.
— Les systèmes centralisés sont inflexibles dans le sens où la moindre modification des caractéristiques d'un ou plusieurs agents peut se traduire par une modification de l'ensemble du HAS. Cela rend nécessaire la répétition de la recherche coûteuse d'une solution. En retour, la propriété souhaitable que la taille de l'effort nécessaire pour ajuster le contrôle soit proportionnelle à la taille des changements dans ce cadre n'est pas satisfaite.

1. en théorie de la complexité informatique, un problème de décision est PSPACE-complet s'il peut être résolu en utilisant une quantité de mémoire polynomiale par rapport à la longueur de l'entrée (espace polynomial), où l'espace polynomial représente une façon de caractériser la complexité d'un algorithme. Si la complexité spatiale est polynomiquement bornée, on dit que l'algorithme est exécutable en espace polynomial. [Link6]

— Les systèmes centralisés ne sont pas robustes dans le sens où l'incapacité d'un agent à remplir son engagement envers le groupe pourrait entraîner l'échec de l'ensemble du groupe.

2.1.1.2 Commande Décentralisée

Une approche décentralisée décompose la tâche d'une équipe d'agents en sous-tâches, où chaque agent ne sera responsable que de sa propre solution et de l'espace qui l'entoure. La décomposition du problème multi-agent en un ensemble de problèmes mono-agent réduit considérablement la complexité de chaque problème.

Dans cette vue, un agent ne peut pas voir les états locaux et les actions locales des autres agents, et devra donc décider seul la prochaine action locale à prendre. Ainsi, chaque agent n'a qu'une vue partielle de l'état global du système, et différents agents ont différentes vues partielles. Bien sûr, cela ne signifie pas nécessairement que les agents sont isolés. Au contraire, un avantage important des agents coopératifs décentralisés est leur capacité à communiquer. Nous considérons la communication comme un moyen d'élargir la vue partielle d'un agent en échangeant des informations locales non observées par d'autres agents.

L'avantage majeur de cette technique est l'autonomie et l'indépendance de l'agent, et donc la simplicité de résolution. Cependant, les principaux inconvénients sont liés à une capacité de surveillance de l'espace d'agent inférieure par rapport à l'approche centralisée où on a accès à l'état globale de l'ensemble, ce qui signifie dans une tâche de planification de trajectoire que nos agents ne sont capables de prévoir des collisions possibles que dans un avenir à court terme,par exemple.

De ce fait, il doit y avoir une certaine forme de coordination. Par conséquent, de nombreuses stratégies de coordination ont été développées.

2.1.2 Selon l'exhaustivité "completeness"

2.1.2.1 Approche Classique (Exacte)

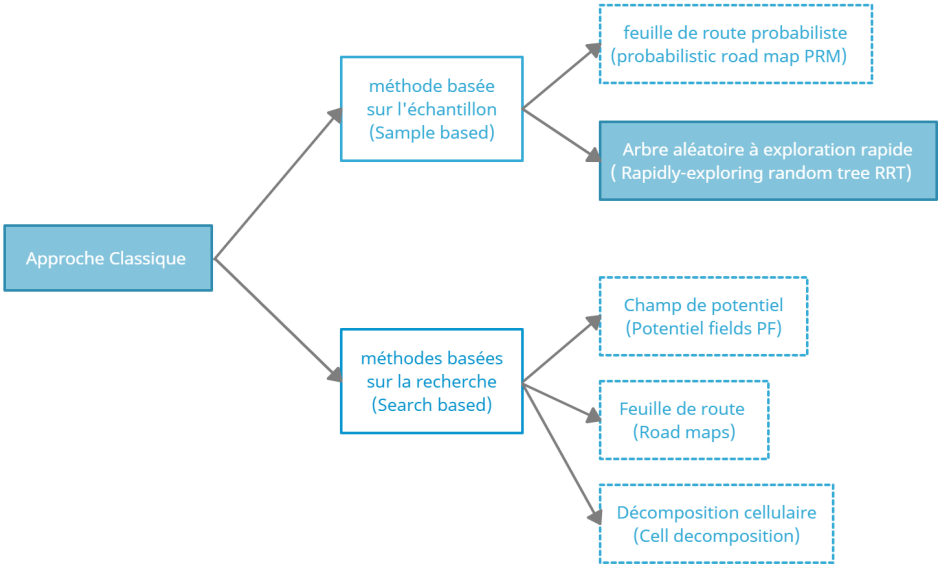


FIGURE 2.1: Classification des approches classiques

Dans les méthodes classiques, soit une solution est trouvée, soit il est prouvé qu'une telle solution n'existe pas. Le principal inconvénient de ces méthodes est leur intensité de calcul et leur incapacité à faire face à l'incertitude. Ces inconvénients rendent leur utilisation fragile dans les applications du monde réel. Ceci est dû aux caractéristiques naturelles de ces applications qui sont imprévisibles et incertaines.

La plupart de ces approches présentent les inconvénients suivants :

— L'enlisement dans des minimums locaux ;
— L'inefficacité en haute dimension ;
— La complexité élevée en haute dimension.

Les approches classiques de planification peuvent être classées comme suit : [4]

1. **Feuilles de route (Roadmap) :** Une feuille de route relie un ensemble de routes praticables par un robot dans son espace libre, tout en minimisant le nombre total de routes. Une feuille de route est une classe de cartes topologiques.

   C'est une union de lignes ou de courbes 1D qui sont appliquées à des espaces de configuration 2D avec des obstacles polygonaux. Étant donné un ensemble d'obstacles polygonaux, une carte routière trouve un chemin sans collision, puis un chemin reliant le départ et le but peut être construit. Le problème de la recherche d'un chemin à l'aide de cartes routières devient plus complexe proportionnellement au nombre de degrés de liberté.

2. **Décomposition cellulaire (Cell decomposition) :** La décomposition cellulaire permet de distinguer l'espace libre de l'espace occupé par des obstacles. L'espace libre est représenté par l'union de régions distinctes appelées cellules. Un exemple bien connu de décomposition cellulaire est la décomposition cellulaire trapézoïdale : il s'agit d'une cellule 2D qui a la forme d'un trapèze. Certaines de ces cellules ont la forme de triangles. Les cellules libres sont délimitées par des obstacles polygonaux. Une ligne qui joint deux points quelconques sur la limite d'une cellule trapézoïdale ne doit pas couper d'obstacles. [22]

3. **Champs de potentiel (Potential field) :** Cette méthode construit un champ de potentiel qui est élevé près des obstacles et faible au voisinage du but [29], [13]. Le robot est guidé vers la configuration cible tout en évitant les obstacles, en laissant sa configuration évoluer dans ce champ de potentiel.

   Autrement dit, le robot est attiré vers la configuration cible et repoussé des obstacles. Le vecteur gradient va pointer vers la direction qui maximise localement le champ de potentiel artificiel, et les variations locales du robot reflètent la structure de l'espace libre. Cette méthode permet un contrôle en temps réel, mais la possibilité de se retrouver piégé dans un minimum local du champ de potentiel empêche son utilisation dans des environnements très encombrés. [5]

Remarque : Ces méthodes ont l'avantage de garantir que le problème général de planification du mouvement est NP-complet [2], mais elles ont l'inconvénient d'être trop lentes pour être utilisées en pratique, surtout dans les problèmes à haute dimension, et de nécessiter une représentation explicite des obstacles, ce qui est très compliqué à obtenir dans la plupart des problèmes pratiques. [5]

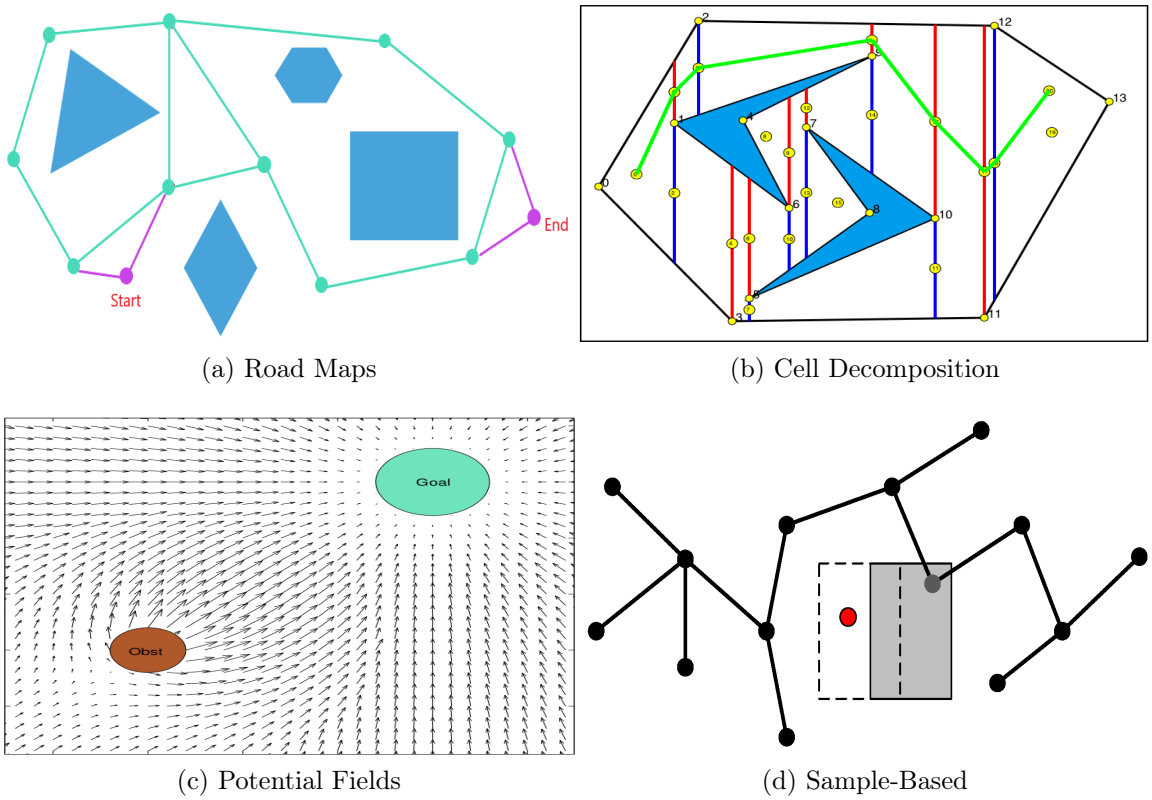
4. **Méthodes basées sur l'échantillonnage (Sample based) :**

   Les méthodes basées sur l'échantillonnage sont les mieux adaptées aux contraintes d'application réelle, que les autres méthodes.

   Contrairement aux algorithmes précédents, les planificateurs basés sur l'échantillonnage acceptent la complétude probabiliste, c'est-à-dire que le but peut ne pas être atteint en un temps fini, mais plus on se rapproche d'un temps infini plus la solution tends vers l'optimale. Entre-temps ils acceptent toute solution, pas nécessairement la solution optimale.

   De plus, ces méthodes négligent la représentation géométrique explicite de l'espace de configuration, et ils sont capables de traiter des robots ayant de nombreux degrés de liberté et de nombreuses contraintes[5].

2. Un problème est dit NP-complet (Nondeterministic Polynomial-time complete) si sa solution peut être trouvée et vérifiée en un temps polynomial



(a) Road Maps (b) Cell Decomposition

(c) Potential Fields (d) Sample-Based

FIGURE 2.2: Approches Classiques

En résumé, les approches classiques garantissent l'obtention de solution si elle existe, mais leurs principaux inconvénients qui les rendent inefficaces dans la pratique sont les suivants :

— Elles impliquent un coût de calcul élevé pour déterminer une trajectoire sans collision réalisable dans des dimensions élevées.

— Elles ont tendance à se coincer dans une solution optimale locale.

— La solution est assez compliquée lorsque l'environnement est dynamique et complexe.

Ces inconvénients empêchent leur utilisation dans des environnements complexes en présence de plusieurs agents[5].

2.1.2.2 Approche Heuristique

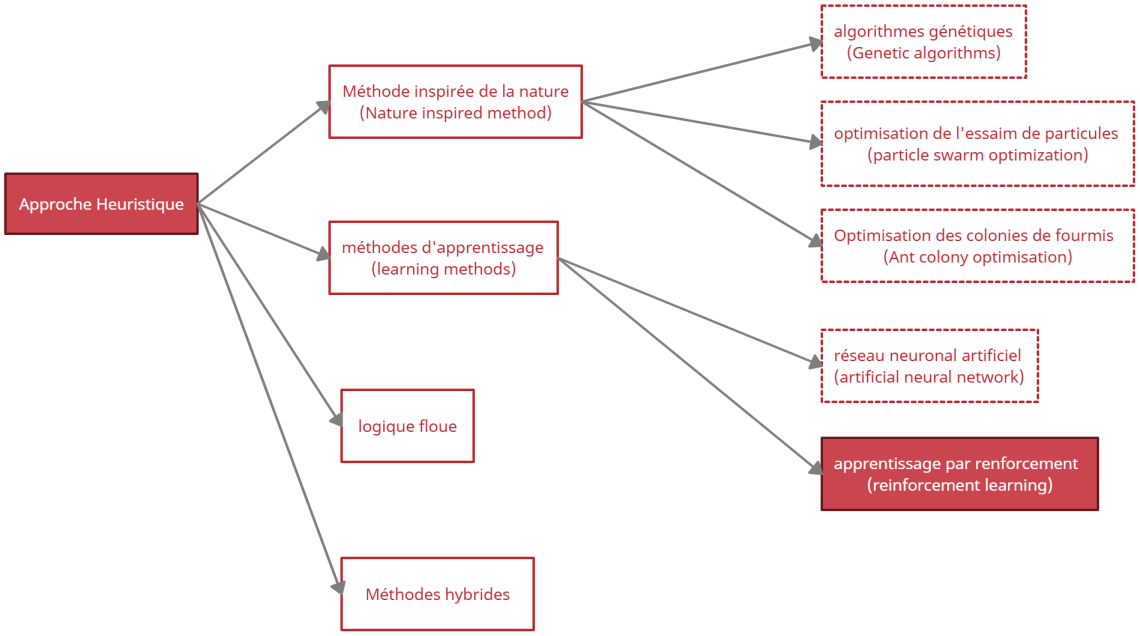


FIGURE 2.3: Classification des approches heuristiques

Comme mentionné précédemment, un problème de navigation est influencé par la précision du plan et de la technique de localisation. La nature dynamique et l'imprévisible des applications réelles rend souvent la tâche de navigation difficile.

En effet, dans de tels scénarios, le plan n'est pas fiable. Cheng et Zelinsky [10] ont suggéré l'utilisation d'approches heuristiques pour répondre aux contraintes des environnements du monde réel. Dans les approches heuristiques, un ensemble de comportements simples serait conçu de manière à résoudre des scénarios complexes.

Comme ces approches dépendent de la situation actuelle (état) et de l'ensemble de comportements conçus pour cet état, elles sont fiables dans les environnements dynamiques (vu qu'elles ne dépendent pas de la localisation et de la planification).

Même si ces approches sont fiables dans les environnements dynamiques, elles ont toujours des difficultés avec l'incertitude. Dans ces approches, l'incertitude provoque des défauts au niveau des états choisis (états irrésolubles), ce qui entraîne souvent des blocages dans les états des robots.

Certaines des approches classées comme méthodes heuristiques sont également appelées méthodes basées sur la population et le comportement dans certains ouvrages. Cela est dû au fait que ces méthodes, telles que GA (Genetic Algorithms), ACO (Ant Colony Optimization) [25], [35], [19] ; PSO (Particle Swarm Optimization) [14],[40] ; et RL (Reinforcement Learning) traitent un ensemble de solutions dans chacune de leurs itérations. Dans ces méthodes, à chaque itération, la stratégie est modifiée ou une nouvelle stratégie évolue par rapport aux précédentes. Contrairement aux méthodes classiques, les algorithmes basés sur les heuristiques ne garantissent pas l'obtention de la solution optimale, mais il est certain que s'ils trouvent la solution appropriée, ce sera en un temps plus court par rapport aux méthodes classiques. Il convient de noter que ces méthodes peuvent échouer dans l'obtention de la solution ou trouver une mauvaise solution (optimum local).[4]

Pour résoudre les inconvénients susmentionnés des approches classiques, des approches heuristiques ont été développées.[5]

2.2 Algorithmes de planification "sample-based"

2.2.1 Introduction

Les algorithmes de planification ont été largement traités dans la littérature, principalement en référence à un agent unique, abordant le problème de l'obtention d'une solution optimale et faisable dans des environnements connus et partiellement connus.

Un schéma efficace de planification de trajectoire est obtenu en considérant le robot comme un point mobile dans un espace approprié. Les obstacles de l'espace de travail sont également représentés dans le même espace en adoptant une transformation appropriée. À cette fin, il est naturel de se référer aux coordonnées généralisées du système mécanique, dont la valeur identifie la configuration du robot. Cela permet d'associer à chaque pose du robot un point dans l'espace C (espace des configurations), qui est alors défini comme l'ensemble de toutes les poses admissibles que le robot peut prendre.

2.2.2 Catégories

Ces algorithmes peuvent être divisés en deux catégories principales :

a. Les algorithmes de planification basés sur l'échantillonnage, qui effectuent une recherche qui sonde l'espace C à l'aide d'un schéma d'échantillonnage. Plus précisément, un ensemble de points est échantillonné dans l'espace libre et connecté afin de construire un arbre de trajectoires réalisables, qui sont ensuite utilisées pour déterminer la solution du problème de planification.

L'évitement des obstacles est réalisé en adoptant un module de détection des collisions qui élimine les trajectoires où une collision se produit de l'ensemble des trajectoires réalisables, ce qui simplifie la construction de l'arbre. Ce type d'algorithme est probabiliste complet, ce qui signifie que, s'il existe une solution réalisable, la probabilité qu'ils la trouvent tend vers 1, si le nombre d'échantillons tend vers l'infini. Certaines versions de ces algorithmes peuvent tenir compte d'un critère de coût évaluant la qualité des différentes trajectoires réalisables et sont également optimales, en probabilité.

Les planificateurs basés sur l'échantillonnage les plus importants à ce jour sont Probabilistic Road-Maps (PRM)[17] et Rapidly-exploring Random Trees (RRT) [34] .

b. Les algorithmes de planification basés sur la recherche, ayant pour but de trouver la séquence d'actions qui conduit l'agent d'un état initial à un état cible de manière optimale, en construisant un graphe d'états et en recherchant une solution dans les graphes [23], [26]. En particulier, ils nécessitent de discrétiser l'espace d'état continu en un ensemble fini d'états, de définir les actions possibles entre ces états et d'appliquer la recherche de graphe pour trouver une solution optimale. La solution trouvée dépend strictement du schéma de discrétisation adopté, et des efforts ont donc été déployés pour diviser efficacement l'espace d'état. Les algorithmes Feuilles de route, Décomposition cellulaire et champs de potentiel font partie de cette catégorie.

Ces deux familles d'algorithmes présentent de nombreuses différences : La première contient des algorithmes probabilistes complets, la seconde contient au contraire des algorithmes qui, si l'espace d'état est bien défini, garantissent la terminaison et la complétude de l'algorithme, c'est-à-dire que si un résultat est renvoyé c'est une solution, sinon l'algorithme renverra un échec. Dans ce mémoire, nous adoptons un algorithme basé sur l'échantillonnage pour les raisons suivantes :

i. Il permet de prendre en compte facilement les obstacles via un module de détection de collision ;
ii. Il permet de prendre en compte quelques propriétés propres à l'agent et les contraintes associées ;
iii. Il est typiquement plus rapide par rapport à un algorithme basé sur la recherche.

2.3 Le "Reinforcement Learning"

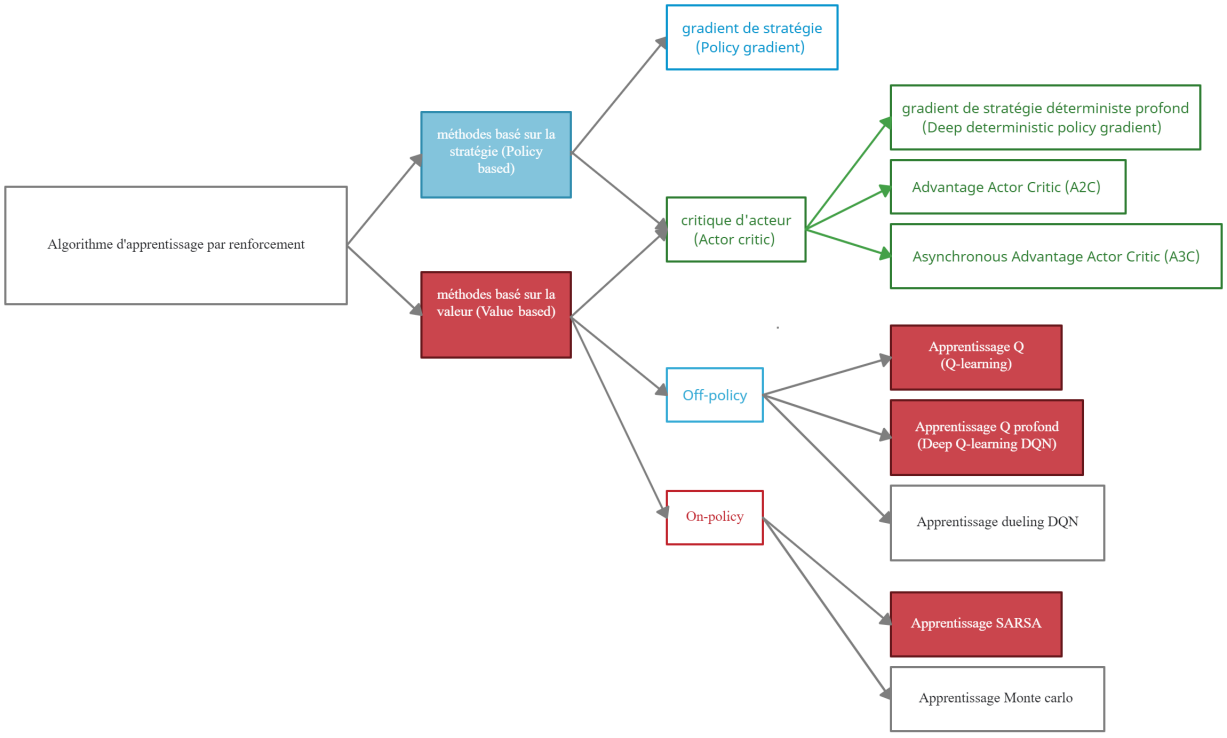


FIGURE 2.4: Classification des algorithmes d'apprentissage par renforcement

2.3.1 Introduction

L'apprentissage par renforcement (RL) est un domaine d'apprentissage automatique très dynamique en terme de théorie et d'application, qui fait partie des méthodes d'apprentissage heuristique, où il essaie d'apprendre la solution heuristique la plus efficace pour un environnement donné.

Les algorithmes d'apprentissage par renforcement étudient le comportement des agents dans des environnements et apprennent à optimiser leur comportement à partir une stratégie de récompense et punition.

Ces algorithmes peuvent être classés principalement en deux catégories : l'apprentissage par la valeur (value based) et l'apprentissage par la stratégie (policy based).[21]

2.3.2 Catégories du "Reinforcement Learning"

D'abord, il est nécessaire de détailler quelque définitions et terminologies.

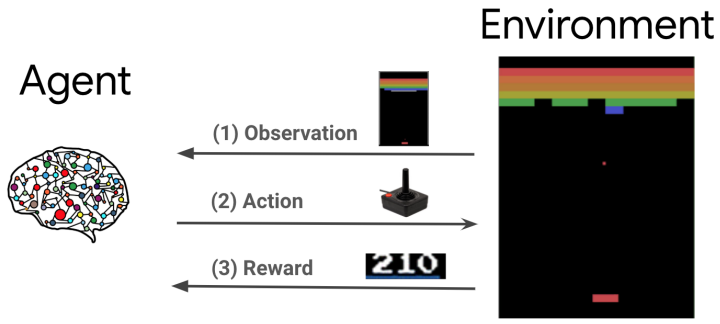


FIGURE 2.5: Interaction Agent-Environnement

- **Agent :** L'élément d'apprentissage qui effectue des actions dans un environnement, il reçoit un retour évaluant leurs actions.
- **Environnement :** Tout ce que l'agent peut interagir avec. L'environnement change lorsque l'agent effectue des actions, chaque changement de ce type est considéré comme une transition d'état.
- **État (Observation) :** Chaque scénario que l'agent rencontre dans l'environnement est formellement appelé un état. L'agent fait passer l'environnement d'un état à un autre en effectuant des actions.
  Il convient également de mentionner les états terminaux, qui marquent la fin d'un épisode. Il n'y a plus d'états possibles après qu'un état terminal ait été atteint, et un nouvel épisode commencera.
- **Actions :** Les actions sont les méthodes de l'agent qui lui permettent d'interagir et de modifier son environnement, et donc de passer d'un état à autre. Chaque action effectuée par l'agent donne lieu à une récompense de la part de l'environnement. La décision de l'action à choisir est prise par la stratégie.
- **Récompense (Reward) :** Une valeur numérique reçue par l'agent de la part de l'environnement en réponse directe aux actions de l'agent. L'objectif de l'agent est de maximiser la récompense globale qu'il reçoit au cours d'un épisode. Les récompenses sont donc la motivation dont l'agent a besoin pour adopter un comportement souhaité.

- **Épisode :** Tous les états qui se situent entre un état initial et un état terminal. Il est important de se rappeler que les différents épisodes sont complètement indépendants les uns des autres.
- **Stratégie (Policy π) :** La stratégie, désignée par π (ou parfois π(a|s)), est une correspondance entre un certain état s et les probabilités de sélection de chaque action possible étant donné cet état. Par exemple, une stratégie qui produit pour chaque état l'action avec la plus grande valeur Q attendue.
- **Les valeurs état-action Q(s,a) :** Une valeur Q représentent une évaluation du fait d'effectuer une action a dans un état s, elle vous indique donc à quel point il est bon d'effectuer une action spécifique dans un état spécifique. De meilleures actions auront des valeurs Q plus élevées.
- **Équation de Bellman :** Elle définit les relations entre un état donné (ou une paire état-action) à ses successeurs. Bien que de nombreuses formes existent, la plus courante habituellement rencontrée dans les tâches d'apprentissage par renforcement est l'équation de Bellman pour la valeur Q, qui est donnée par :

$$Q(s,a) = Expected\,[\, r + \gamma\, Q(s',a')\,] \tag{2.1}$$

où
— $Q(s,a)$: la valeur Q de l'état actuel s en effectuant l'action a.
— r : récompense de transition de l'état s à l'état s'.
— $Q(s',a')$: la valeur Q de l'état s' en effectuant l'action a'.
— a' : l'action qui provoque la transition de s à s'.
— γ : facteur de remise.

- **Facteur de remise (γ) :** détermine essentiellement à quel point l'agent se soucient des récompenses dans un avenir lointain par rapport à ceux dans un avenir immédiat.
- **Exploitation et exploration :** L'apprentissage par renforcement n'a pas de modèle à partir des-quel un agent peut apprendre, il doit crée leur propre expérience et apprends selon elle. Pour ce faire, l'agent doit essayer de nombreuses actions différentes dans de nombreux états différents afin d'essayer d'apprendre toutes les possibilités disponibles et de trouver le chemin qui maximisera sa récompense globale, c'est ce qu'on appelle l'exploration, car l'agent explore l'environnement.

D'autre part, si l'agent ne fait qu'explorer, il ne maximisera jamais la récompense globale, il doit aussi utiliser les informations qu'il a apprises pour y parvenir. C'est ce qu'on appelle l'exploitation, car l'agent exploite ses connaissances pour maximiser les récompenses qu'il reçoit.
Le compromis entre les deux est l'un des plus grands défis des problèmes d'apprentissage par renforcement, car les deux doivent être équilibrés afin de permettre à l'agent d'explorer suffisamment l'environnement, mais aussi d'exploiter ce qu'il a appris et de répéter le chemin le plus gratifiant qu'il a trouvé.

2.3.2.1 Méthodes basées sur la valeur "Value-Based"

Les méthodes RL basées sur la valeur sont conçues pour trouver une bonne estimation de la fonction Q pour toutes les combinaisons état-action. La stratégie optimale (approximative) peut ensuite être extraite en prenant les actions à valeurs Q maximale.

Les deux algorithmes value-based les plus populaires sont Q-learning et SARSA, où l'agent maintient une estimation de la fonction Q optimale. Lors de la transition de l'état actuel à l'état suivant en effectuant une action, l'agent reçoit une récompense, et met à jour la fonction Q(s,a) de cette transition, selon l'équation de Bellman (2.1), mais selon différentes stratégies.

Un autre algorithme populaire basé sur la valeur, mais qui suis une approche différente, est la recherche arborescente de Monte-Carlo (MCTS).

Une tâche importante concernant les méthodes basées sur la valeur est d'estimer la fonction de valeur associée à une stratégie donnée : cette tâche est appelée évaluation de stratégie, qui a été abordée par des algorithmes d'apprentissage par différence temporelle (TD) [3].

3. Temporal Difference, ou différence temporelle, est un agent qui apprend d'un environnement par épisodes, sans connaissance préalable de l'environnement. L'agent apprend donc par essais et erreurs où la fonction d'erreur TD rapporte la différence entre la récompense estimée à un état donné et la récompense réelle reçue

2.3.2.2 Méthodes basées sur la stratégie "Policy-Based"

Un autre type d'algorithmes RL procède différemment : au lieu d'apprendre une fonction de valeur qui indiquerait quelle action il faut prendre dans tel ou tel état, il recherche directement la fonction de stratégie qui décrit la succession d'actions à prendre, sans utiliser une fonction de valeur.

Cela signifie que nous essayerons directement d'optimiser notre fonction de stratégie π sans nous soucier d'une fonction de valeur. Nous allons directement paramétrer π qui est généralement estimée par des approximateurs de fonctions paramétrés comme les réseaux de neurones.

Par conséquent, l'idée la plus simple consiste à mettre à jour les paramètres de l'estimateur de π le long de la direction du gradient de la récompense à long terme maximale, ou ce qu'on appelle la méthode du gradient de stratégie (PG).[15]

Ensuite, diverses méthodes de gradient de stratégie, y compris REINFORCE[31], G(PO)MDP[12], et des algorithmes d'acteur-critique[38], ont été proposées en estimant le gradient de différentes manières.

2.3.3 État de l'Art du "Reinforcement Learning"

Dans cette section, on présente les algorithmes les plus récents dans l'apprentissage par renforcement.

2.3.3.1 Deep Q Learning

Dans cet algorithme, nous utilisons les DQN (Deep Q Networks) qui sont des réseaux de neurones profonds. Il s'agit d'un algorithme RL basé sur les valeurs.

L'idée de DQN est, plutôt qu'utiliser des itérations de valeur comme dans Q-learning pour déterminer les valeurs Q et trouver la fonction Q optimale, nous utilisons un approximateur de fonction pour estimer la fonction Q optimale, c'est-à-dire en utilisant des réseaux de neurones profonds.

L'objectif de ce réseau est d'approximer la fonction Q optimale qui satisfera l'équation de Bellman. La perte du réseau est déterminée en comparant les valeurs de Q prédites aux valeurs de Q cibles du côté droit de l'équation de Bellman. Une fois la perte calculée, le réseau met à jour les poids via une méthode d'optimisation (par exemple la descente de gradient stochastique et la rétropropagation) et c'est ainsi que la perte est minimisée. [21]

Des extensions de cette méthode sont développées en exploitant différentes architectures du réseau de neurones.

2.3.3.2 Actor critic

Les méthodes de critique de l'acteur sont des méthodes de TD (différence temporelle), qui ont une structure séparée pour représenter la stratégie indépendamment de la fonction de valeur. La structure de stratégie est appelée l'acteur, car elle est utilisée pour sélectionner les actions, et la fonction de valeur estimée est appelée la critique, car elle critique les actions effectuées par l'acteur.

L'apprentissage se fait en critiquant la stratégie de l'acteur, où la critique prend la forme d'une erreur de TD.

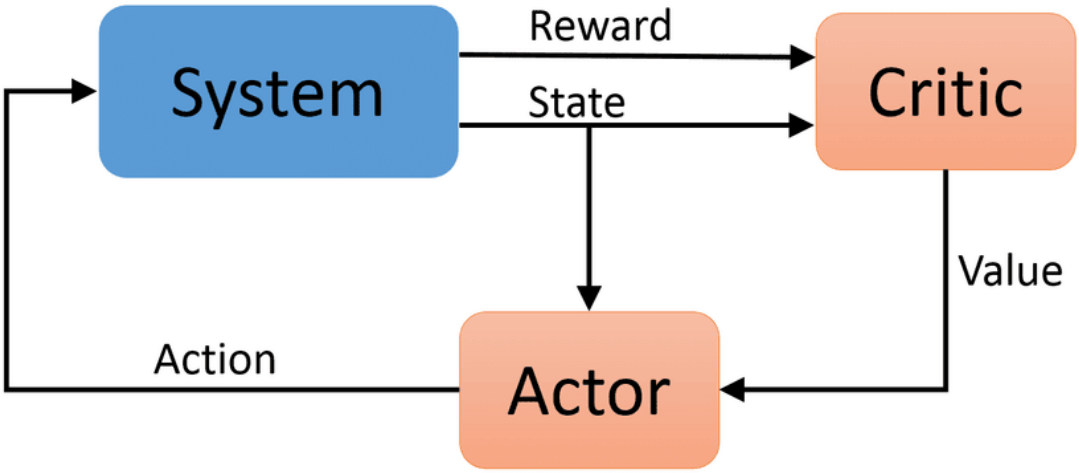


FIGURE 2.6: Structure d'apprentissage d'un algorithme Actor Critic

L'état de l'art de cette approche est A2C et A3C, qui signifie Advantage Actor-Critic et Asynchronous Advantage Actor-Critic, respectivement.

2.3.3.3 Policy gradient

Les méthodes de gradient de stratégie reposent sur l'optimisation de stratégie paramétrée par rapport au rendement attendu (récompense cumulative à long terme) par descente de gradient. Elle vise donc à modéliser et à optimiser la stratégie directement.

Lorsque la stratégie est généralement modélisée par une fonction paramétrée par rapport à θ, et que la valeur de la fonction de récompense dépend de cette stratégie, l'objectif est d'optimiser θ pour obtenir la meilleure récompense.

Une méthode de gradient de stratégie récente est Deep Deterministic Policy Gradient (DDPG), une technique Actor-critic qui apprend simultanément une fonction Q (en utilisant DQN) et une stratégie (en utilisant Policy gradient).

Conclusion Il existe d'autres nouveaux algorithmes effectifs pour traiter le problème d'apprentissage par renforcement, la plupart d'entre eux étant des extensions ou des combinaisons des trois méthodes précédentes, l'apprentissage Q profond qu'il s'agit d'un algorithme basé sur la valeur, le gradient de stratégie qui est un algorithme basé sur la stratégie, et la critique de l'acteur qui combine les deux (value based et policy based).

Chapitre 3

Optimisation par l'Algorithme "Rapidly-exploring Random Trees planners"

3.1 Cas d'un agent unique

3.1.1 L'algorithme RRT

3.1.1.1 Introduction

L'algorithme RRT calcule une solution réalisable δ en construisant un arbre T qui a pour nœud racine la configuration initiale de l'agent. À chaque étape de l'algorithme, une nouvelle configuration n est échantillonnée de manière aléatoire dans l'espace sans obstacles, et une trajectoire réalisable reliant le nœud échantillonné n au nœud le plus proche appartenant à l'arbre T est calculée par le biais de la fonction dite de direction [1].

Si cette trajectoire réalisable est sans collision, alors le nœud échantillonné est ajouté à l'arbre T. L'arbre est complètement construit si une configuration cible a été ajoutée comme nœud de l'arbre, ou si l'arbre a atteint sa longueur maximale. Une fois l'arbre complètement construit, si l'un des nœuds est la configuration cible, alors la trajectoire de l'arbre reliant la racine au nœud cible est notre solution δ, sinon le nœud cible est relié au nœud le plus proche appartenant à l'arbre T par une trajectoire réalisable, et la trajectoire ainsi

1. une fonction qui lie le nouveau nœud à l'arbre, si la trajectoire entre ce nœud et l'arbre est réalisable (sans collision)

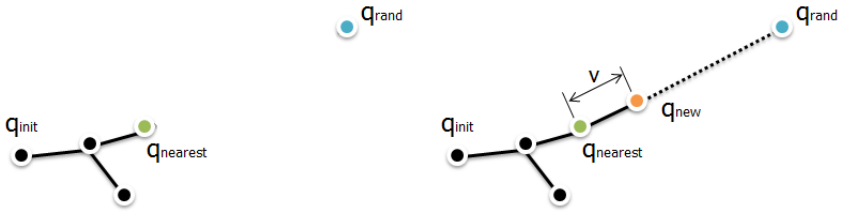


FIGURE 3.1: Procédure de l'algorithme RRT

obtenue reliant le nœud racine à la configuration cible est notre solution δ.

Cet algorithme, comme tous les autres algorithmes basés sur des échantillons, est probabiliste complet, ce qui signifie que la probabilité de trouver une solution réalisable δ avec cet algorithme tend vers 1 uniquement si le nombre d'échantillons tend vers l'infini. Nous avons une garantie de terminaison, car, même si l'arbre T ne s'approche jamais du point d'arrivée, l'arbre sera complètement construit, et une solution (peut-être pas sans collision) sera trouvée.

À noter que RRT ne garantit pas l'optimalité, même si un critère d'optimalité comme une fonction d'utilité est adopté par la fonction de direction pour sélectionner la trajectoire réalisable.

3.1.1.2 Pseudo-code et explication

RRT construit un arbre T = (E,V) dans l'espace de configuration où les nœuds V, sont des configurations sans collision choisies au hasard dans l'espace de configuration du robot. Chaque nouveau nœud ne crée qu'une seule arête E, de lui-même au nœud le plus proche, dans l'arbre.

Algorithm 1 $T = (V, E) \leftarrow RRT(q_{int})$ [6]

1.T ←InitializeTree()
2.T ←InsertNode(q_{init},T)
3.**while** $k < NumberNodes$ **do**
  4. q_{rand} ←RandomSample(k)
  5. q_{near} ←NearestNeighbor(q_{rand}, T)
  6. q_{new} ←Extend($q_{rand}, q_{near}, maxDistance$)
end

— Les lignes 1 et 2 de l'algorithme 1 initialisent l'arbre avec un nœud à l'emplacement de départ du robot $qinit$, et insèrent la position de départ dans l'arbre.
— L'arbre est formé en sélectionnant des échantillons aléatoires (nœuds), sans collision, puis faire croître l'arbre à ce nouveau nœud. L'algorithme continue à échantillonner et à ajouter de nouveaux nœuds et arêtes à l'arbre jusqu'à ce que le nombre maximal de nœuds autorisé soit atteint ou que le but soit trouvé.
— La ligne 4 de l'algorithme 1 sélectionne l'échantillon aléatoire et sans collision, q_{rand}.
— La ligne 5 trouve le plus proche voisin déjà dans l'arbre à partir duquel il faut croître.
— La ligne 6 est la fonction Extend(), décrite dans l'Algorithme 2, qui détermine si une nouvelle arête (branche) peut être ajoutée à l'arbre depuis le nœud le plus proche, q_{near}, jusqu'au nouveau nœud aléatoire, q_{rand}.
La longueur d'extension $maxDistance$ est une valeur prédéterminée appelée facteur de croissance. Le facteur de croissance peut avoir un effet significatif sur l'extension de l'arbre. Si le facteur de croissance est petit, alors l'arbre aura de nombreuses branches courtes. Si le facteur de croissance est élevé, l'arbre peut avoir de longues branches.
Il y a un compromis à faire en spécifiant le facteur de croissance. Si le facteur de croissance est trop faible, il faudra beaucoup plus de nœuds pour explorer l'espace de configuration et trouver la solution. Si le facteur de croissance est trop grand, l'extension de l'arbre échouera plus souvent, ce qui obligera l'algorithme à ré-échantillonner un nouvel emplacement.

Algorithm 2 $q_{new} \leftarrow$Extend$(q_{rand}, q_{near}, maxDistance)$ [6]

1.$q_{new} = min(maxDistance, \|q_{near} - q_{rand}\|)$
2.**if** $ObstacleFree(q_{new})$ **then**
    3. $V = V \cup q_{new}$
    4. $E = E \cup [q_{near}, q_{new}]$
    5. $return\ q_{new}$
else
    6. $noNodeAdded$
end

Lors de la tentative de croissance de l'arbre, la nouvelle branche est vérifiée pour les collisions en utilisant un planificateur local.

— La ligne 1 de l'algorithme 2, si la distance entre le nœud aléatoire q_{rand} et le voisin q_{near} est moins que la distance $maxDistance$, il devient le nouveau nœud q_{new}, sinon q_{rand} sera ajouté à une distance $maxDistance$ comme nouveau nœud q_{near}.

— ligne 2, Si ce nouveau nœud est sans collision, il sera ajouté comme sommet, une arête qui lie q_{new} et q_{near} sera ajouté également.

Lorsque le RRT sélectionne un échantillon aléatoire, l'emplacement du but est parfois choisi comme échantillon. Cela oblige l'arbre à essayer de se connecter au but à partir du nœud le plus proche du but. La sélection du nœud du but comme nouvel échantillon est basée sur une probabilité prédéterminée. Il est possible que l'arbre ait trouvé un nœud très proche du but, mais qu'il ne se connecte pas au but en raison de cette probabilité. Une autre option consiste à essayer de se connecter au but après chaque nœud inséré avec succès. En utilisant cette méthode, l'algorithme essaie de compléter le chemin aussi rapidement que possible.

3.1.2 L'Algorithme RRT*

3.1.2.1 Introduction

L'algorithme RRT* est dérivé de l'algorithme RRT, il hérite de toutes ses caractéristiques, mais fournit une garantie d'optimalité de la solution [33],[32]. L'algorithme suit les mêmes étapes que RRT mais ajoute la "phase de recâblage".

RRT* étend d'abord le plus proche voisin vers l'échantillon, cependant, il connecte le nouveau sommet, au sommet qui subit le coût cumulé minimum. RRT* étend également le nouveau sommet aux sommets afin de "recâbler" les sommets auxquels on peut accéder avec un coût moindre.

La figure ci-dessous décrit la procédure de recâblage.

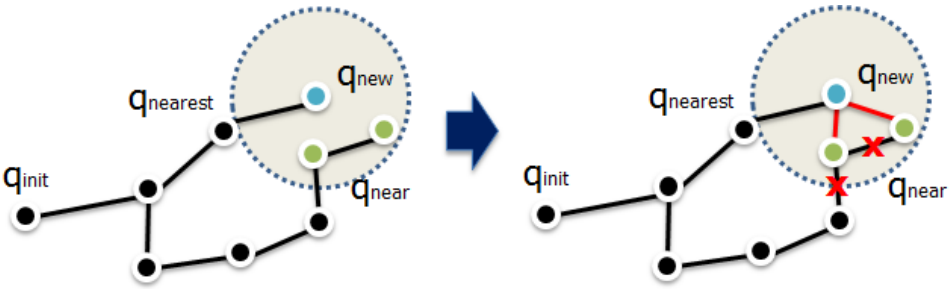


FIGURE 3.2: Procédure de l'algorithme RRT* (recâblage)

Cette modification dans l'algorithme donne la possibilité d'améliorer la solution, en détectant à chaque étape la meilleure connexion possible du nouveau nœud à l'arbre. Cela reste un algorithme de planification basé sur des échantillons comme RRT, avec toutes ses limites et ses faiblesses, mais maintenant nous avons que la probabilité de trouver une solution optimale tend vers 1, avec le nombre d'échantillons qui tend vers l'infini.

L'implémentation des algorithmes RRT et RRT* peut faire un compromis entre la qualité de la solution et le temps nécessaire pour l'obtenir. Lorsque la vitesse est d'une importance capitale, ces algorithmes offrent donc la possibilité de régler le temps nécessaire à l'obtention d'une solution. C'est l'une des raisons pour lesquelles nous les avons choisis comme algorithmes de planification multi-agent.

3.1.2.2 Pseudo-code et explication

L'algorithme RRT* commence de la même manière que le RRT :

Algorithm 3 $T = (V, E) \leftarrow RRT*(q_{int})$ [6]

1.$T \leftarrow$InitializeTree()
2.$T \leftarrow$InsertNode(q_{init},T)
3.**while** $k < NumberNodes$ **do**
  4. $q_{rand} \leftarrow$RandomSample(k)
  5. $q_{near}est \leftarrow$NearestNeighbor(q_{rand}, q_{near}, T)
  6. $q_{min} \leftarrow$ChooseParent($q_{rand}, q_{near}, q_{nearest}, maxDistance$)
  7. $T \leftarrow InsertNode(q_{min}, q_{rand}, T)$
  8. $T \leftarrow Rewire(T, q_{near}, q_{min}, q_{rand})$
end

— L'arbre est initialisé et l'emplacement de départ est ajouté à l'arbre dans les lignes 1 et 2. RRT* sélectionne également des échantillons aléatoires de la même manière que RRT.
— À la ligne 5 de l'algorithme 3, l'algorithme sélectionne le plus proche voisin de l'échantillon aléatoire. La fonction sélectionne également l'ensemble des nœuds, q_{near}, dans l'arbre qui sont dans le voisinage (selon un certain rayon de recherche) de l'échantillon aléatoire q_{rand}.
— La ligne 6 de l'algorithme 3 est la première différence majeure entre le RRT* et le RRT. Au lieu de sélectionner le plus proche voisin comme parent de l'échantillon aléatoire, la fonction ChooseParent() sélectionnera le meilleur nœud parent dans le voisinage des nœuds, qui minimise le coût totale.
— La ligne 8 est la deuxième différence majeure entre les RRT* et RRT. La fonction Rewire() étend le nouveau sommet q_{rand} aux sommets q_{near} afin de le recâbler avec un des nœuds q_{near}, auxquels on peut accéder avec un coût moindre.
— Les fonctions ChooseParent() et Rewire() modifient la structure de l'arbre de recherche par rapport à l'algorithme RRT. L'arbre généré par le RRT a des branches qui se déplacent dans toutes les directions. L'arbre généré par l'algorithme RRT* a rarement des branches qui se déplacent dans la direction du parent.

La fonction ChooseParent() garantit la création d'arêtes qui s'éloignent toujours de l'emplacement de départ.
— La fonction Rewire() modifie la structure interne de l'arbre pour s'assurer que les sommets internes n'ajoutent pas d'étapes inutiles sur un chemin découvert.

Les fonctions ChooseParent() et Rewire() garantissent que les chemins découverts seront sous-optimaux car ces fonctions minimisent toujours les coûts pour atteindre chaque nœud de l'arbre.[6]

3.2 Cas Multi-Agent

3.2.1 Introduction

Jusqu'à présent, les algorithmes qui ont été développés pour la planification d'un agent unique ont été décrits. En effet, dans un cadre multi-agent, le problème devient plus complexe.

Les algorithmes pour un système multi-agent dépendent principalement du type de solution que l'on recherche. En particulier, nous pouvons faire la distinction entre une configuration coopérative et une configuration non coopérative, puis adopter des outils d'optimisation.

Nous décrivons ici quelques exemples d'approches utilisées dans un contexte multi-agent.

— **Algorithme du feu de circulation :** Si une collision entre deux agents est prévue, l'un d'eux s'arrêtera et attendra que l'autre ait traversé son chemin. Il est tout à fait similaire au système de feux de circulation de nos carrefours. Le problème de cet algorithme est qu'il n'est pas toujours possible pour un agent de s'arrêter dans n'importe quelle position, car s'arrêter pourrait signifier violer une contrainte fondamentale, comme dans le cas d'un avion.

— **Poli RRT* (multi-agent RRT*) :** Il s'agit d'une extension de l'algorithme RRT* à un cadre multi-agent, en adoptant une approche basée sur les priorités. Les agents sont triés selon une hiérarchie donnée, et l'algorithme RRT* est exécuté pour chaque agent en suivant la priorité qui lui est attribuée : chaque agent calculera sa propre trajectoire en évitant tous les autres agents ayant une priorité supérieure à la sienne. Le problème de ce schéma est qu'il est difficile de trouver un ordre de priorité qui permette de déterminer une bonne solution. [27]

3.2.2 Solution proposée

3.2.2.1 Description du problème

Nous considérons N agents se déplaçant dans la même région. Chaque agent doit atteindre une certaine destination, en partant d'un point de départ sans entrer en collision avec d'éventuels obstacles, et en restant suffisamment éloigné des autres agents. La trajectoire conjointe qui évite les collisions avec les obstacles et préserve une certaine distance de sécurité entre toutes les paires d'agents est appelée sans collision. Nous supposons que chaque agent a une fonction d'utilité qui évalue la qualité de sa trajectoire, plus la valeur de la fonction d'utilité est élevée, meilleure est la trajectoire. L'objectif est alors de trouver une trajectoire conjointe sans collision qui respecte un critère bien défini qui assurera l'optimalité du système multi-agent.

3.2.2.2 Motivations

Pour résoudre ce problème, nous avons cherché une solution avec des caractéristiques spécifiques. Nous voulions une solution décentralisée pour répartir uniformément la charge de travail entre tous les agents avec un minimum d'informations échangées eux, pour réduire les possibilités de dysfonctionnements dûs aux erreurs de transmission. Nous voulions également trouver une solution aussi rapide que possible, et qui soit juste et acceptable pour tous les agents.

Les N agents vont d'abord chacun calculer la trajectoire qui leur permet d'atteindre leur point d'arrivée, en utilisant l'algorithme RRT*. Ensuite, ils enverront à tous les autres N-1 agents la trajectoire trouvée, et vérifieront si une collision est attendue, ou si la distance de sécurité n'est pas respectée entre deux ou plusieurs agents.

Si c'est le cas, le premier agent qui a détecté le danger le communiquera par un message d'avertissement. Tous les N agents concernés par la collision, une fois le signal reçu, vont recalculer leurs trajectoires en évitant les N-1 autres agents.

Pour recalculer la trajectoire, chaque agent va élaguer son propre arbre, en éliminant les branches de l'arbre qui risquent la collision.

Il s'agit d'une technique qui permet aux agents de réutiliser les données des calculs précédents, en évitant de se débarrasser des données utiles, et en économisant du temps et des ressources mémoire. Sans cette astuce, les temps de calcul augmenteraient trop.

Les nouvelles trajectoires trouvées sont comparées à la trajectoire optimale originale trouvée à la première étape, qui a été rejetée à cause de la collision, et chaque agent i envoie la mesure de performance Li(δ, δ'), avec δ la solution trouvée à la première étape, et δ' la solution trouvée à l'étape actuelle. Parmi tous les agents, celui qui détériore le moins sa trajectoire, qui a une mesure de performance plus faible, $Li(\delta, \delta')$, est choisi comme candidat pour résoudre le problème, ce qui confirmera le changement de trajectoire à tous les autres agents, et sera retiré de l'ensemble des N agents.

Les N-1 agents restants exécuteront à nouveau l'algorithme, en essayant d'identifier un autre agent comme candidat pour résoudre le problème. Jusqu'à ce que l'ensemble des agents N soit vide, ce qui signifie que chaque agent a trouvé une trajectoire sans collision reliant le but et le point de départ.

3.2.2.3 Notion d'optimalité de Pareto

En 1949, J. Nash a suggéré la notion d'une solution d'équilibre pour un jeu non coopératif, appelée récemment "le profil stratégique de l'équilibre de Nash". Depuis, cet équilibre est largement utilisé en économie, en sociologie, en sciences militaires et dans d'autres sphères de l'activité humaine.

Un optimum de Pareto peut être défini comme étant une allocation des ressources pour laquelle il n'existe pas une alternative dans laquelle tous les acteurs seraient dans une meilleure position.

3.2.2.4 Application de la Pareto-Optimalité dans notre cas

Une solution, ensemble de trajectoires sans collision $\delta = (\delta_1, \delta_2, \ldots, \delta_N)$, est dite Pareto-optimale si tout autre ensemble de trajectoires sans collision $\delta' = (\delta_1', \delta_2', \ldots, \delta_N')$ entraîne une dégradation d'une fonction dite fonction d'utilité d'au moins un agent, où cette fonction mesure les performances d'une solution.

Une solution Pareto optimale, définie ci-dessus, peut alors être le bon compromis entre une bonne solution sous-optimale, et l'effort pour la trouver. C'est ce qui motive notre choix de rechercher une trajectoire conjointe sans collision, optimale au sens de Pareto.

Fonction d'utilité choisie :

Pour la mesure de performance de l'agent i, on définit la fonction d'utilité Li(δ, δ') comme suit :

$$Li(\delta, \delta') = (n - n')/(100 * n') \tag{3.1}$$

où :
— n : nombre de nœuds dans la trajectoire actuelle ;
— n' : nombre de nœuds dans la trajectoire précédente.

Cette fonction représente combien la solution δ' détériore la solution δ pour l'agent i : plus la valeur Li(δ, δ') est élevée, plus la solution δ' est mauvaise par rapport à la solution δ.

Les seules garanties de cet algorithme est la terminaison. À chaque itération de l'algorithme, un candidat est toujours extrait de l'ensemble N, et supprimé. Par conséquent, l'ensemble des N agents est réduit de 1 à chaque itération. Inévitablement, l'ensemble N finira par se vider. Mais l'optimalité de Pareto de la solution n'est pas encore garantie.

Pour y remédier, un autre passage a été proposé. À chaque itération de l'algorithme, la mesure de performance de l'agent i, Li(δ, δ') est calculée, et comparée à $Li(\delta, \delta^{\sim})$, pour tous les N agents, avec δ la solution trouvée à la première étape, δ' la solution trouvée à l'étape actuelle, et $\delta^{\sim}$ la solution trouvée à l'étape précédente. Ainsi, chaque agent vérifie s'il a obtenu une amélioration et communique cette information à tous les agents restants, afin d'être sûr qu'au moins un agent améliore toujours sa solution à chaque étape. Si ce n'est pas le cas, une approche de retour en arrière est proposée. De N-1 agents on revient à N en ajoutant le précédent supprimé, comme si la dernière itération n'avait pas eu lieu, et on sélectionne un nouveau candidat, différent de ceux qui ont déjà été essayés.

Si tous les agents ont déjà été choisis au moins une fois, le premier qui a été choisi est sélectionné à nouveau. À noter que le processus de backtracking garantit toujours la terminaison de l'algorithme. En effet, à chaque itération, le nombre d'agents restera N ou deviendra N-1, sans jamais augmenter.

Maintenant qu'à chaque étape chaque agent vérifie si au moins l'un d'entre eux a amélioré sa solution, nous avons des garanties à la fois de la terminaison de l'algorithme et de l'optimalité Pareto de la solution.

3.2.2.5 Pseudo-Algorithme

1. Initialisation ;
2. Calcul des trajectoires individuelles de chaque agent ;
3. Vérifier si une collision est attendue.
4. Vérifier les agents actifs ;
5. Tant que des collisions sont attendues, et qu'il y a des agents actifs :
6. Chaque agent calcule sa trajectoire sans collision en évitant tous les autres agents et en reliant le but au point de départ. Calculer également les valeurs d'utilité ;
7. Création du vecteur contenant les valeurs d'utilité ;
8. Vérification si au moins une fonction d'utilité est dégradée ;
9. Sinon (les valeurs sont pires) : Retour au nœud précédent ;
10. Si tous les autres chemins ont déjà été essayés : Accepter celui qui a la plus faible valeur ;
    Sinon : Accepter le chemin avec la valeur la plus faible qui n'a pas été déjà essayé ;
11. Si tous les agents sont désactivés ou la solution est trouvée : fin.
    Sinon : retour au point 4.

Le cœur de notre solution est le calcul et la comparaison des mesures de performance.

L'idée est d'utiliser une simple méthode d'ordre de priorité, en résolvant le problème agent par agent, mais en les triant selon la mesure de performance. La décentralisation a été réalisée en donnant à chaque agent le droit de calculer sa propre trajectoire, et donc distribuer la charge de travail, ainsi chaque agent ne calculera qu'une seule fois un arbre complet, car si une collision est détectée, l'agent va rectifier sa trajectoire au voisinage de la collision tout en conservant le reste des nœuds non concernés.

Le MULTI Pareto-RRT* est un algorithme simple et rapide, basé sur la question "qui dépense le moins pour éviter les autres ?" pour hiérarchiser les différents agents, et trouver une solution optimale de Pareto facilement acceptable par tous les agents.[37]

Chapitre 4

Optimisation par "Reinforcement Learning"

Comme nous avions établi précédemment, malgré le fait que les approches classiques donnent la solution exacte si elle existe - un avantage indéniable-, leur besoin en temps et en volume de calculs dans des situations à un certain degré de complexité, sans oublier la complexité de la solution optimale offerte qui la rendrait non-réalisable, font qu'il serait impossible de les utiliser dans plusieurs problèmes réels, notamment ceux dont la dynamique impose une certaine rapidité. Pour ces raisons, une autre classe d'approches a été élaborée : celle des méthodes heuristiques, qu'on a brièvement introduites dans le chapitre 2. Dans ce chapitre, nous allons détailler la méthode du Reinforcement Learning.

4.1 Introduction

Un agent d'apprentissage par renforcement (RL) est un agent qui apprend en interagissant avec son environnement dynamique. À chaque pas de temps, l'agent perçoit l'état de l'environnement et entreprend une action, qui fait passer l'environnement à un nouvel état.

Un signal de récompense scalaire évalue la qualité de chaque action, et l'agent doit maximiser la récompense cumulative tout au long de l'interaction. Le feedback RL (la récompense) est moins informatif que dans l'apprentissage supervisé, où l'agent recevrait les actions correctes à entreprendre (ces informations ne sont malheureusement pas toujours disponibles).

Le feedback RL est cependant plus informatif que dans l'apprentissage non supervisé, où il n'y a pas de feedback explicite sur les performances. Des algorithmes bien définis, dont la convergence est prouvée, sont disponibles pour résoudre l'apprentissage par renforcement pour un agent unique. La simplicité et la généralité du cadre rendent le RL intéressant également pour l'apprentissage multi-agent.[16]

4.2 Cas d'un agent unique

Un agent d'apprentissage par renforcement est modélisé pour prendre des décisions séquentielles en interagissant avec l'environnement. L'environnement est généralement formulé sous la forme d'un processus de décision de Markov (MDP) actualisé à horizon infini.

4.2.1 Processus de Décision de Markov

En tant que modèle standard, le Processus de Décision de Markov (MDP) a été largement adopté pour caractériser la prise de décision d'un agent dans un environnement, avec une observabilité totale des états du système.

Un MDP est défini par un n-uplet (S, A, P, R, γ), où S et A désignent respectivement les espaces d'états et d'actions, $P : S \times A \to \Delta(S)$ désigne la probabilité de transition de tout état $s \in S$ vers tout état $s' \in S$ pour toute action donnée $a \in A$.

$R : S \times A \times S \to$R est la fonction de récompense qui détermine la récompense immédiate reçue par l'agent pour une transition de s vers s'. $\gamma \in$[0,1) est le facteur de remise qui échange les récompenses instantanées et futures.

À chaque instant t, l'agent choisit d'exécuter une action a(t) face à l'état du système s(t), ce qui entraîne la transition du système vers s(t+1), et l'agent reçoit une récompense instantanée R(s(t), a(t), s(t+1)) pour cette transition.

Le but de la résolution du MDP est donc de trouver une stratégie π : S $\to$ Δ(A), qui lie tout état $s \in S$ à une action $a \in A$ qu'il faut prendre dans cet état, de telle sorte que la récompense cumulée soit maximisée.

En vertu de la propriété markovienne, la stratégie optimale peut être obtenue par des approches de programmation dynamique (DP) ou d'apprentissage par renforcement (RL), où en DP la fonction de récompense est connue par l'agent, alors dans RL l'agent apprend selon son expérience seulement.

Lorsque l'on tente de résoudre un MDP par RL, on peut choisir entre deux méthodes principales : calculer les fonctions de valeur ou les valeurs Q de chaque état et choisir les actions en fonction de celles-ci, ou calculer directement une stratégie qui définit les probabilités que chaque action soit entreprise en fonction de l'état actuel, et agir en fonction de celle-ci.[15]

4.2.2 Les algorithmes Q-Learning et SARSA

Chaque algorithme d'apprentissage par renforcement doit suivre une certaine stratégie afin de décider des actions à effectuer à chaque état. Cependant, la procédure d'apprentissage de l'algorithme ne doit pas tenir compte de cette stratégie pendant l'apprentissage. Les algorithmes qui se préoccupent de la stratégie qui a donné lieu à des décisions d'action dans les états précédents sont appelés algorithmes on-policy, tandis que ceux qui l'ignorent sont appelés off-policy. Un algorithme off-policy bien connu est le Q-Learning, car sa règle de mise à jour utilise l'action qui donnera la valeur Q la plus élevée, alors que la stratégie réelle utilisée pourrait restreindre cette action ou en choisir une autre. La variante avec stratégie de Q-Learning est connue sous le nom de SARSA, où la règle de mise à jour utilise l'action choisie par la stratégie suivie.

4.2.2.1 Q-Learning

Q-Learning est un algorithme off-policy pour l'apprentissage par différence temporelle. Il a été prouvé qu'avec un entraînement suffisant pour toute stratégie ϵ-greedy [1], l'algorithme converge avec une probabilité de 1 vers une approximation proche de la fonction Q optimale.

1. Greedy Policy et ϵ - Greedy Policy : Greedy policy est une stratégie où l'agent effectue constamment l'action qui est censée produire la récompense attendue la plus élevée. Une telle stratégie ne permettra pas à l'agent d'explorer du tout.

Afin de permettre une certaine exploration, une stratégie ϵ-greedy est souvent utilisée à la place : un facteur (nommé ϵ) dans la plage de [0,1], désigne le taux d'exploration est sélectionné, et avant de sélectionner une action, un nombre aléatoire dans la plage de [0,1] est sélectionné. Si ce nombre est supérieur à ϵ, l'action greedy est sélectionnée, mais s'il est inférieur, une action aléatoire est sélectionnée.

Équation de Bellman correspondante : Une formulation de l'équation de Bellman pour le Q-learning est la suivante :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \, max \, Q(s_{t+1}, a') - Q(s_t, a_t)\right] \tag{4.1}$$

où :

s_t, a_t, r_t : état actuel, action effectuée dans cet état et récompense obtenue, respectivement.

$max \, Q$: la récompense maximale qui peut être obtenue dans l'état suivant.

a' : l'action qui donne Q max pour l'état suivant(récompense maximale attendue) quelle que soit notre stratégie.

γ : facteur de remise, également défini entre 0 et 1.

α : le taux d'apprentissage, fixé entre 0 et 1, définit le taux de mise à jour des valeur Q. Le fixer à 0 signifie que les valeurs Q ne sont jamais mettre à jour, donc que rien n'est appris. La définition d'une valeur élevée, telle que 0.9, signifie que l'apprentissage peut se produire rapidement.

L'objectif de l'agent est d'apprendre l'équation 4.1, optimale.

Pseudo-code et explication :

Algorithm 4 Q-Learning [Link8]

1: Initialize **Q(s,a)** arbitrarily
2: Repeat (for each episode)
3: Choose **a** from **s** using policy derived from **Q**
4: (e.g., ϵ - greedy)
5: Take action **a**, observe **r**, **s'**
6: **Q(s,a)** $\leftarrow$ **Q(s,a)** + α **[r+** γ max_{α}**,Q(s',a') - Q(s,a)]**
7: s $\leftarrow$ s'
8: until s is terminal

On peut reformuler ces étapes plus simplement comme suit :

1. Initialiser la table des valeurs Q.
2. Observer l'état actuel s.
3. Choisir une action a pour cet état en se basant sur l'une des stratégies de sélection d'action existantes (par exemple ϵ-greedy).
4. Effectuer l'action et observer la récompense r, ainsi que le nouvel état s'.
5. Mettre à jour la valeur Q de l'état en utilisant la récompense observée et la récompense maximale possible pour l'état suivant. La mise à jour est effectuée selon la formule (4.1)
6. Répéter le processus jusqu'à ce qu'un état terminal soit atteint.[Link8]

4.2.2.2 SARSA

L'algorithme SARSA est un algorithme On-Policy pour l'apprentissage par différence temporelle. La principale différence entre cet algorithme et le Q-Learning est que la récompense maximale pour l'état suivant n'est pas nécessairement utilisée pour mettre à jour les valeurs Q. En revanche, une nouvelle action et donc une nouvelle récompense sont sélectionnées en utilisant la même stratégie que celle qui a déterminé l'action originale. Le nom SARSA vient du fait que les mises à jour sont effectuées en utilisant le quintuplé $Q(s, a, r, s', a')$,

où : s, a sont l'état et l'action actuels, r est la récompense observée dans l'état suivant et s', a' sont la nouvelle paire état-action.

Équation de Bellman correspondante : une formulation de l'équation de Bellman pour SARSA est la suivante :

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{4.2}$$

où :

a_{t+1} : l'action de l'état suivant, choisie en fonction de la stratégie avec laquelle nous avons travaillé.

$Q(s_{t+1}, a_{t+1})$: La récompense de l'état suivant, en effectuant l'action a_{t+1}.

Les autres éléments de l'équation ont la même signification que dans Q-Learning. L'objectif de l'agent est d'apprendre l'équation 4.2, optimale.

Pseudo-code et explication

La forme procédurale de l'algorithme SARSA est comparable à celle du Q-Learning :

Algorithm 5 SARSA [Link8]

Initialize **Q(s,a)** arbitrarily
  Repeat (for each episode) :
  Initialize **s**
    Choose **a** from **s** using policy derived from **Q**
  (e.g., ϵ - greedy)
  Repeat (for each step of episode) :
  Take action **a**, observe **r**, **s'**
    Choose **a'** from **s'** using policy derived from **Q**
  (e.g., ϵ - greedy)
  Q(s,a) $\leftarrow$ **Q(s,a)** + α **[r+** γ max_{α}**,Q(s',a') - Q(s,a)]**
s $\leftarrow$ **s' , a** $\leftarrow$ **a'**
until s is terminal

Comme nous pouvons le voir, deux étapes de sélection d'action sont nécessaires, pour déterminer la paire état-action suivante en même temps.

SARSA utilise la stratégie de comportement de l'agent (c'est-à-dire la stratégie utilisée par l'agent pour générer de l'expérience dans l'environnement, qui est généralement $\epsilon - greedy$) pour sélectionner une action supplémentaire a_{t+1}, puis utilise $Q(s_{t+1}, a_{t+1})$ en tant que récompense futurs attendus.

Q-learning n'utilise pas la stratégie de comportement pour sélectionner une action supplémentaire a'. Au lieu de cela, il estime les rendements futurs attendus comme $max\ Q(s_{t+1}, a')$.

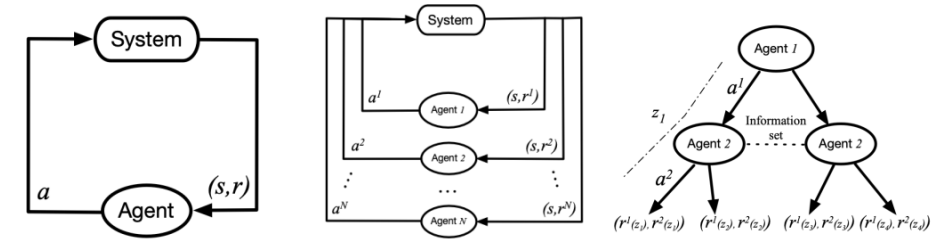
4.3 Cas Multi-Agent

4.3.1 Introduction

Le Multi-Agent Reinforcement learning (MARL) traite également des problèmes de prise de décision séquentielle, mais en présence de plusieurs agents. En particulier, l'évolution de l'état du système aussi bien que la récompense reçue par chaque agent sont influencées par les actions conjointes de tous les agents. Plus intriguant encore, chaque agent a sa propre récompense à long terme à optimiser, qui devient alors une fonction des stratégies de tous les autres agents.

Le MARL peut être considéré comme une fusion de l'apprentissage par renforcement, la théorie des jeux et des techniques plus générales de recherche de stratégie [16]. Par conséquent, un tel problème peut être modélisé par plusieurs cadres.

En général, il existe deux cadres théoriques apparemment différents mais étroitement liés pour les MARL : les jeux de Markov/stochastiques et les jeux de forme extensive [15], comme nous le verrons plus loin. L'évolution des systèmes dans les différents cadres peut être illustrée de cette manière :



(a) Markov decision process (b) Markov game (c) Extensive-form game

FIGURE 4.1: Différents cadres du MARL

4.3.2 Markov et jeux de forme extensive

4.3.2.1 Jeux de Markov :

Une généralisation directe des MDP qui permet de capturer l'imbrication de multiples agents est les jeux de Markov (MGs), également connus sous le nom de jeux stochastiques[18]. Le cadre des MGs est utilisé depuis longtemps dans la littérature pour développer des algorithmes MARL.[24]
Nous en présentons la définition formelle ci-dessous.

Définition :
Un jeu de Markov est défini par un tuple $(N,\ S,\ \{A_i\}_{i\in N},\ P,\ \{R_i\}_{i\in N}, \gamma)$ où N désigne l'ensemble agents, S désigne l'espace d'état observé par tous les agents, A_i désigne l'espace d'action de l'agent i.

Soit A = $A_1 \times ... \times A_N$, alors P : S × A → Δ(S) désigne la probabilité de transition de tout état s∈ S vers tout état s' ∈ S pour toute action conjointe a ∈ A.
R_i : S × A × S → $\mathbb{R}$ est la fonction de récompense qui détermine la récompense immédiate reçue par l'agent i pour une transition de s vers s', et $\gamma \in [0\,;1)$ est le facteur de remise.

Au temps t, chaque agent i ∈ N exécute une action a_i, selon l'état du système s_t. Le système passe ensuite à l'état s_{t+1}, et récompense chaque agent i par $R_i(s_t,a_t,s_{t+1})$.

L'objectif de l'agent i est d'optimiser sa propre récompense à long terme, en trouvant la stratégie π_i : S → Δ (A_i).

Le concept de solution d'un MG diffère de celui d'un MDP, puisque la performance optimale de chaque agent est contrôlée non seulement par sa propre stratégie, mais aussi par les choix de tous les autres participants du jeu.

4.3.2.2 Jeux de forme extensive

Un jeu de forme extensive est une spécification d'un jeu en théorie des jeux, permettant la représentation explicite d'un certain nombre d'aspects clés, tels que la séquence des mouvements possibles des joueurs, leurs choix à chaque point de décision, les informations (éventuellement imparfaites) dont dispose chaque joueur sur les mouvements de l'autre joueur lorsqu'il prend une décision, et leurs gains pour tous les résultats possibles du jeu. Ce cadre partage le même aspect que celui du jeux de Markov, sauf qu'il est plus commun de l'utiliser pour analyser les jeux dynamiques, cependant Markov est dédié pour les jeux répétées. [9][8]

4.3.3 Algorithmes d'apprentissage multi-agent

Les algorithmes MARL peuvent être classés selon plusieurs dimensions, dont certaines, comme le type de tâche, découlent des propriétés des systèmes multi-agent en général. D'autres, comme la stratégie d'apprentissage et l'architecture de commande.

4.3.3.1 Classification des algorithmes MARL

1. **Selon le type de tâches :**

   Les tâches qu'une formation d'agents doit réaliser peuvent être classées comme : coopérative, compétitive ou mixte. Selon l'objectif de la formation, cet objectif peut être défini comme une matrice de récompense des agents présents en formation.

   Les jeux compétitifs sont également appelés jeux à somme nulle, car la somme des matrices de récompense des agents est une matrice nulle. Les jeux mixtes sont également appelés jeux à somme générale, car il n'y a pas de contrainte sur la somme des récompenses des agents. Dans notre étude, on s'intéresse aux jeux coopératifs.

   Dans un jeu stochastique (de Markov) entièrement coopératif, les agents ont la même fonction de récompense ($r_1 = ... = r_n$) et l'objectif d'apprentissage est de maximiser le rendement actualisé commun. Si un contrôleur centralisé était disponible, la tâche se réduirait à un processus de décision de Markov, dont l'espace d'action serait l'espace d'action conjoint du jeu stochastique. Dans ce cas, l'objectif pourrait être atteint, par exemple, en apprenant les valeurs optimales de l'action conjointe avec Q-learning, et en utilisant ensuite une stratégie $\epsilon - greedy$.

   D'autre part, les agents peuvent être des décideurs indépendants, et un problème de coordination se pose même si tous les agents apprennent en parallèle la fonction Q optimale commune. Il peut sembler que les agents pourraient utiliser des stratégies $\epsilon - greedy$ appliquées à Q^* pour maximiser le rendement commun, cependant, dans certains états, plusieurs actions conjointes peuvent être optimales. En absence de mécanismes de coordination supplémentaires, différents agents peuvent rompre ces liens entre les multiples actions conjointes optimales de différentes manières, et l'action conjointe résultante peut être sous-optimale.

2. **Selon la stratégie :**

Après avoir fixé le type de tâche à résoudre, un système multi-agent a deux choix possibles de stratégies pour effectuer cette tâche : une stratégie centralisée ou décentralisée.

Le problème de la génération de plans est un problème clé de la collaboration multi-agent. Sur la base des différentes vues du système utilisées dans les approches de planification, on trouve celles qui utilisent une vue centralisée et celles qui utilisent une vue décentralisée.

Typiquement, la vue centralisée est celle du concepteur du système. Le comportement des agents est souvent considéré collectivement, en termes d'actions communes. Dans ce cas, un plan qui spécifie la stratégie de résolution de problème des agents est souvent décrit par une correspondance entre les états globaux du système et les actions conjointes des agents. Nous appelons cela une stratégie multi-agent centralisée.

En revanche, la vue décentralisée est souvent celle d'un agent situé dans le système, qui traite une observation partielle de l'état global du système et prend des décisions locales. Dans ce cas, un plan est souvent une correspondance entre les observations locales et les actions locales, et nous appelons cela une stratégie décentralisée. Il est clair que dans cette vision chaque agent a besoin de sa propre stratégie, contrairement à la vision centralisée où une stratégie centralisée spécifie les actions communes des agents.

En termes de prise de décision, la vue centralisée du système peut être décrite par un modèle de processus de décision de Markov multi-agent.
Il s'agit d'un processus de décision de Markov standard qui se compose d'un ensemble d'états globaux S, d'un ensemble d'actions conjointes A (chaque action conjointe spécifie une action pour chaque agent), d'une matrice de probabilité de transition $Pr(s|s', a)$ - la probabilité que le système passe de l'état s à l'état s' après l'action conjointe a, et d'une fonction de récompense r qui spécifie l'utilité globale reçue lors d'une transition.

Ce cadre correspond à un modèle de calcul où, à toute étape t, le système est décrit par son état global actuel s, qui est constitué des états locaux actuels des agents. Le système entreprend ensuite une action conjointe et évolue vers l'un des états globaux suivants.

Dans ce cadre, une stratégie centralisée (CP), qui est une correspondance entre les états globaux et les actions conjointes, est précisément une stratégie pour un processus de décision de Markov. [30], [1]

La figure ci-dessous montre comment la résolution de problèmes s'effectue dans le cadre d'une telle stratégie (dans un système à deux agents X et Y). L'utilité attendue du CP peut être calculée à l'aide d'un algorithme standard d'évaluation de stratégie pour les processus de décision de Markov.

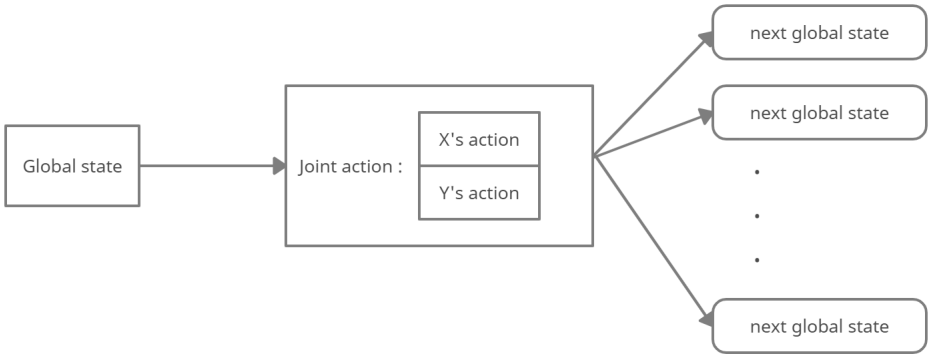


FIGURE 4.2: Stratégie centralisée à deux agents

Dans la vue décentralisée, un agent ne peut pas voir les états locaux et les actions locales des autres agents, et doit décider de l'action locale suivante sur la sienne. Ainsi, chaque agent n'a qu'une vue partielle de l'état global du système, et les différents agents ont des vues partielles différentes. Un moyen d'élargir la vue partielle d'un agent est d'échanger des informations locales non observées par d'autres agents.

Le système est modélisé par chaque agent ayant un espace d'état individuel, son propre ensemble d'actions locales et sa propre mesure de probabilité de transition d'état local, mais utilise une fonction de récompense globale pour relier les effets des actions dans les agents. Ainsi, il ne s'agit pas d'un MDP standard.

La figure suivante montre le modèle de calcul dans cette vision décentralisée : à toute étape t, chaque agent décide d'abord de l'action locale à entreprendre. Ensuite, il décide s'il est nécessaire de communiquer lorsque l'action est terminée.

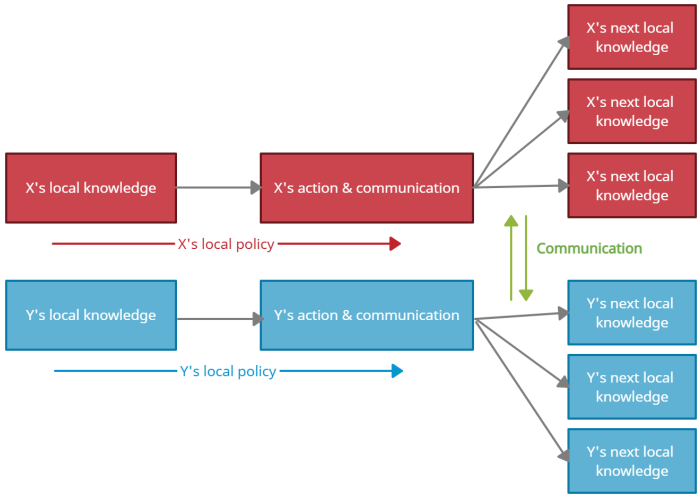


FIGURE 4.3: Stratégie décentralisée à deux agents

La principale différence entre un CP et un DP est qu'un CP prend l'état global comme point de départ mais dans le DP, l'état global n'est pas automatiquement observé par les agents. Cela rend DP beaucoup plus complexe que CP. Cependant, la plupart des systèmes multi-agent sont distribués par nature et les agents sont généralement autonomes, ce qui signifie que chaque agent est seul décideur.

Cela ne signifie pas que les politiques centralisées ne sont pas valides. En fait, ils représentent simplement des solutions ancrées dans des perspectives différentes.

4.3.4 Solution proposée

4.3.4.1 Description du problème :

Considérons N agents se déplaçant dans le même environnement, où l'objectif est d'effectuer une tâche coopérative, déplacement en formation, en partant d'un point de départ sans entrer en collision avec d'éventuels obstacles. Chaque agent de la formation doit communiquer sa stratégie avec les autres agents, ce qui fait appel à la nécessité de coordination.

4.3.4.2 Motivation :

Pour résoudre ce problème, nous avons cherché une solution avec des caractéristiques spécifiques. Nous voulions une solution centralisée où l'ensemble d'agents de formation sera considéré comme un agent unique global, pour calculer une seule solution, une fois, pour l'ensemble. Chaque agent aura accès à l'état des autres agents ainsi que leurs stratégies, pas besoin d'autre échange d'information ou communication.

Tous les agents participant à la tâche coopérative partagent un objectif commun et une seule fonction de récompense. Dans la prochaine session, nous allons décrire un schéma qui satisfait ces propriétés.

Les N agents vont d'abord construire un hyper espace d'action et d'état (HAS) de l'ensemble, le HAS contient l'espace de toutes les actions conjointes admissibles que les agents peuvent tenter, et l'espace d'état globaux qui englobe les états des N agents. Cet espace sera modélisé par un MDP (processus de décision de Markov standard) avec une fonction de récompense commune. Finalement ce MDP se résout avec un algorithme d'apprentissage par renforcement à un agent.

4.3.4.3 Centralized Q-Learning

Une première solution pour résoudre ce MDP est d'utiliser le Q-learning comme agent global de la formation. Cependant une évaluation de performances de cette solution a fait appel à des problèmes de performance.

Q-learning aura une dégradation de performances proportionelle au nombre d'agents présents dans la tâche. De plus, il ne peut gérer que des espaces d'états et d'actions discrets, et ne peut pas fonctionner avec des états et actions continus, car l'apprentissage de la fonction Q se fait par calcul explicite des valeurs Q pour toutes les paires état-action.

Habituellement, Q-learning a besoin d'approximations de fonctions. Par exemple, des réseaux neuronaux pour associer des valeurs Q estimées, au lieu de calculer la table Q complète pour toutes les paires état-action possibles.

4.3.4.4 Centralized DQN (Deep Q-learning)

Dans cette solution, la fonction Q est représentée par un réseau de neurones profond, qui apprend la fonction Q-optimale.

- Où s'intègrent les réseaux de neurones ?[Link1]

Les réseaux de neurones sont des approximateurs de fonctions, qui sont particulièrement utiles dans l'apprentissage par renforcement lorsque l'espace d'état ou l'espace d'action sont trop grands pour être complètement connus.

Un réseau de neurones peut être utilisé pour approximer la fonction de valeur. C'est-à-dire que les réseaux neuronaux peuvent apprendre à mapper des états sur des valeurs Q. Plutôt que d'utiliser une table de recherche pour stocker, indexer et mettre à jour tous les états possibles et leurs valeurs correspondantes, ce qui est impossible dans le cas de problèmes de grande dimension, nous pouvons entraîner un réseau de neurones sur des échantillons de l'espace d'état pour apprendre à prédire leurs valeurs.

La figure ci-dessous illustre la différence d'apprentissage de la fonction Q, entre Q-learning et DQN.

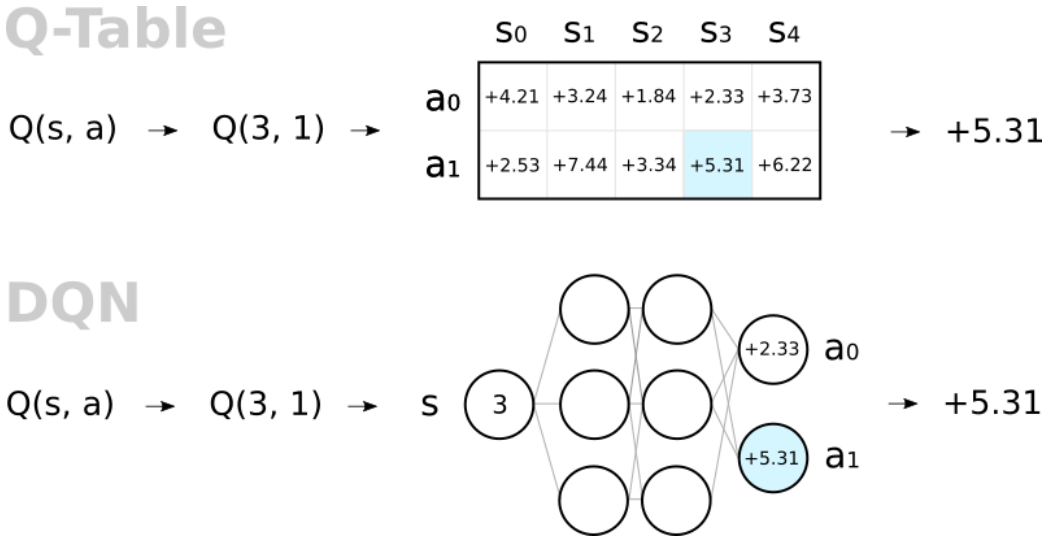
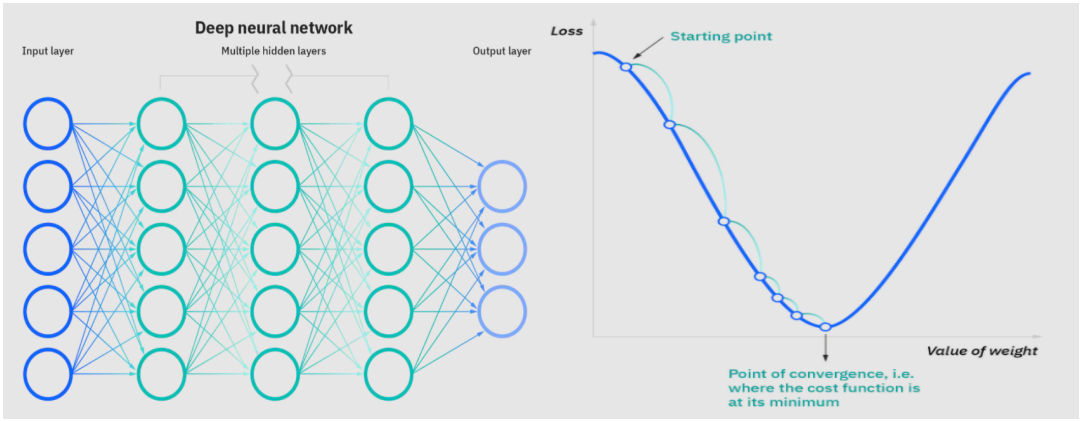


FIGURE 4.4: Apprentissage de la fonction Q dans le cas du Q-Learning et DQN

- L'apprentissage des réseaux de neurones :[Link4]

Comme tous les réseaux de neurones, des coefficients sont utilisés pour approximer la fonction reliant les entrées aux sorties. Leur apprentissage consiste à trouver les bons coefficients, ou poids (*Weight*), en ajustant ces poids de façon itérative le long du gradient (ou une autre méthode d'optimisation) qui promettent de minimiser une fonction de coût qui présente l'erreur d'apprentissage du réseau (*Loss*).



(a) (b)

FIGURE 4.5: Apprentissage du réseau de neurones

- Fonction objectif (Loss) :

C'est une fonction qui sert de critère pour déterminer la meilleure solution à un problème d'optimisation. Le problème d'optimisation devient un problème de maximisation (ou minimisation) de cette fonction objectif.

- Backpropagation :

La rétropropagation du gradient est une méthode pour calculer le gradient de l'erreur pour chaque neurone d'un réseau de neurones, de la dernière couche vers la première.

- Algorithme du gradient :

La descente de gradient est un algorithme d'optimisation itératif pour trouver un minimum d'une fonction coût. L'idée est de faire des pas répétés dans la direction opposée de la pente (ou pente approximative) de la fonction au point actuel, car c'est la direction de la descente la plus raide.

Deep Q-Learning[11] [39]

Soit un réseau de neurones dit $Q\ Network$ qui, pour un état donné s, produit un vecteur de valeurs d'action $Q(s,a\,;\theta i)$, où θ_i sont les paramètres du réseau. L'objectif de ce réseau est d'apprendre à prédire la fonction Q-optimale.

L'apprentissage du $Q\ Network$ se fait par entraînement sur des échantillons d'expérience dits *Experience Replay* [2], que l'agent construit pendant un certain temps, en interagissant avec son environnement. L'expérience est de la forme :

$$e = (\text{état actuel } s, \text{ état suivant } s', \text{ récompense } r, \text{ action } a).$$

Ensuite, l'optimisation de $Q\ Network$ se fait en minimisant l'erreur de Bellman, en utilisant la récursion :

$$L_i(\theta_i) = (y_i^{DQN} - \ Q(s, a; \theta_i))^2 \tag{4.3}$$

Et y_i^{DQN}, la valeur cible de $Q(s, a; \theta_i)$, est obtenue par :

$$y_i^{DQN} = r + \gamma\ max\ Q(s', a'; \theta_{i-1}) \tag{4.4}$$

Cependant, on n'a toujours pas le terme $max\ \ Q(s', a'; \theta_{i-1})$. Pour y remédier, Mnih et al. (2015) ont proposé l'utilisation d'un deuxième réseau dit $Target\ Network$ pour la prédiction de cette valeur.

Le réseau $Target\ Network$, avec les paramètres θ_{i-1}, est le même que le réseau $Q\ Network$, (même architecture et paramètres initiaux) sauf que ses paramètres sont fixes, et pour chaque pas de temps T, ils sont mis à jour en prenant les paramètres de $Q\ Network$ comme nouveaux paramètres de $Target\ Network$ (θ_{i-1}=θ_i).

2. **Experience Replay :** Comme les tâches d'apprentissage par renforcement n'ont pas d'ensembles d'entraînement pré-générés à partir desquels elles peuvent apprendre, l'agent doit conserver des enregistrements de toutes les transitions d'état qu'il a rencontrées afin de pouvoir en tirer des enseignements ultérieurement. La mémoire utilisée pour stocker ces données est souvent appelé Experience Replay.[36]

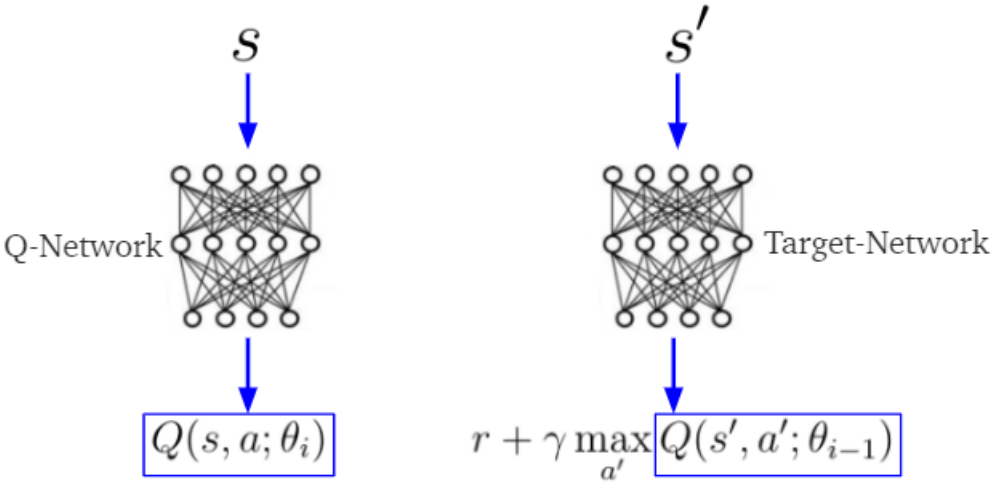


FIGURE 4.6: Séparation de la tâche d'apprentissage en deux réseaux

Prenons l'exemple d'un état s comme entrée de *Q Network* et s' (état suivant) comme entrée de *Target Network*. Vu que les deux réseaux partagent le même réseau initial, et qu'il n'existe qu'une seule étape entre eux dans l'algorithme, alors en mettant à jour les poids des deux réseaux à chaque étape, l'optimisation sera instable et le *Q Network* revient à courir après sa propre queue. Il est donc nécessaire de fixer le réseau *Target Network* quelques pas de temps, puis de mettre à jour ses poids avec les poids du réseau *Q Network*, ce qui va stabiliser l'apprentissage.[3]

Algorithme de DQN : [Link9] On peut résumer la solution sous les étapes suivantes :

1. Initialiser la capacité de *Replay memory* pour le stockage des échantillons d'expérience ;
2. Initialiser le réseau *Q Network* avec des poids aléatoires ;
3. Cloner le réseau *Q Network* et l'appeler le réseau *Target Network* ;
4. Pour chaque épisode :
   Initialiser l'état de départ ;
5. Pour chaque pas :
   Sélectionner une action (Par exploration ou exploitation) ;

6. Exécuter l'action sélectionnée ;
7. Observer la récompense et l'état suivant ;
8. Stocker l'expérience dans *Replay memory* ;
9. Transmettre un échantillon d'expérience au réseau *Q Network* ;
10. Calculer la perte entre les valeurs Q de sortie de *Q Network* et les valeurs Q cibles :

    - Nécessite un passage par le réseau *Target Network* pour l'état suivant ;
11. La descente de gradient met à jour les poids dans le réseau de *Q Network* pour minimiser l'erreur d'apprentissage ;
12. Après un certain nombre de pas de temps, les poids du réseau *Target Network* sont mises à jour avec les poids du réseau *Q Network*.

Le schéma ci-dessous résume la solution.

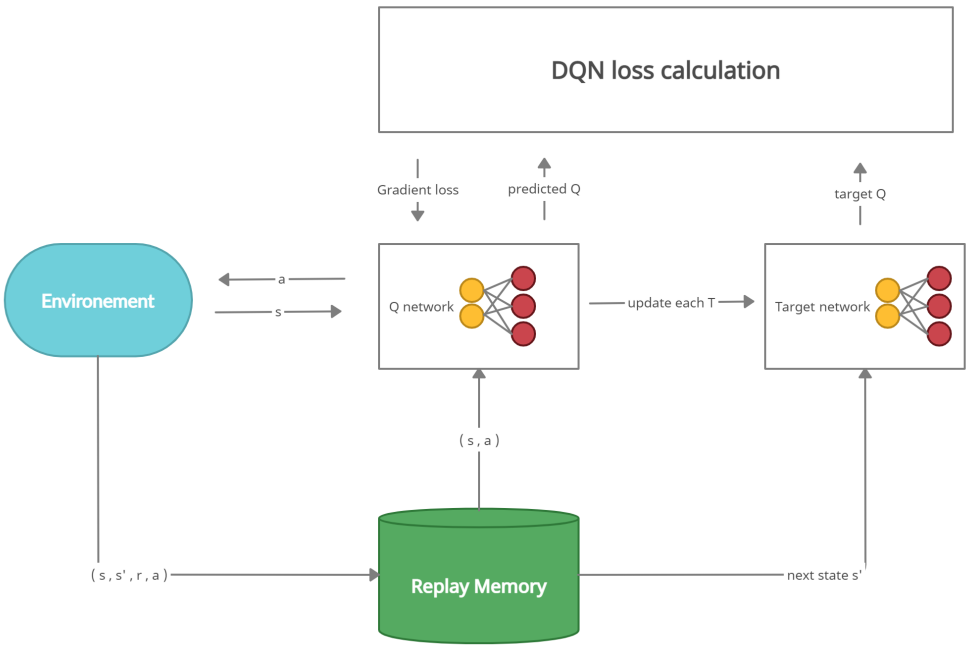


FIGURE 4.7: Apprentissage par DQN

Chapitre 5

Implémentations et résultats

5.1 Approche classique

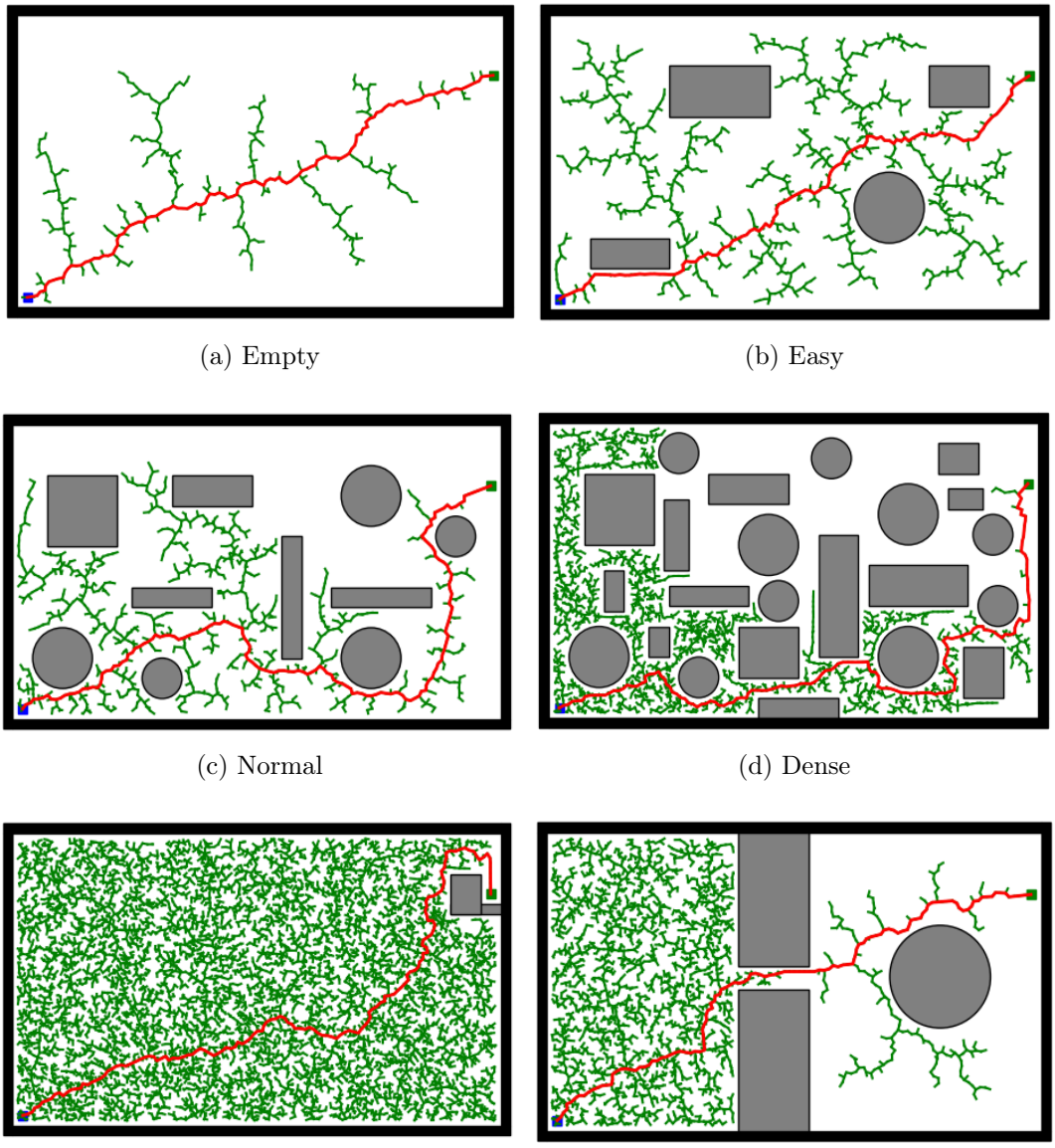
Nous avons effectué des simulations afin d'observer les résultats donnés par les algorithmes RRT et RRT* dans le cas d'un agent unique, ainsi que pour un groupe d'agents décentralisés. Nous avons essayé par la suite d'exploiter la notion de l'optimalité de Pareto afin de pouvoir comparer et en tirer des conclusions.

5.1.1 Agent unique

Premièrement, nous avons implémenté RRT et RRT* pour la planification de trajectoire d'un seul agent, dans un environnement 2D pour différents degrés de complexité, dans le but de tester et évaluer l'efficacité de solution obtenue dans des cas de figures qui simulent bien un environnement réel.

Ensuite, une étude comparative entre RRT et RRT* est détaillée. L'évaluation se concentre sur le coût de la solution obtenue, qui est la longueur de la trajectoire, et l'effort nécessaire pour obtenir cette solution. Nous avons choisi d'illustrer cet effort par le temps d'exécution qu'a nécessité chaque algorithme.

RRT :



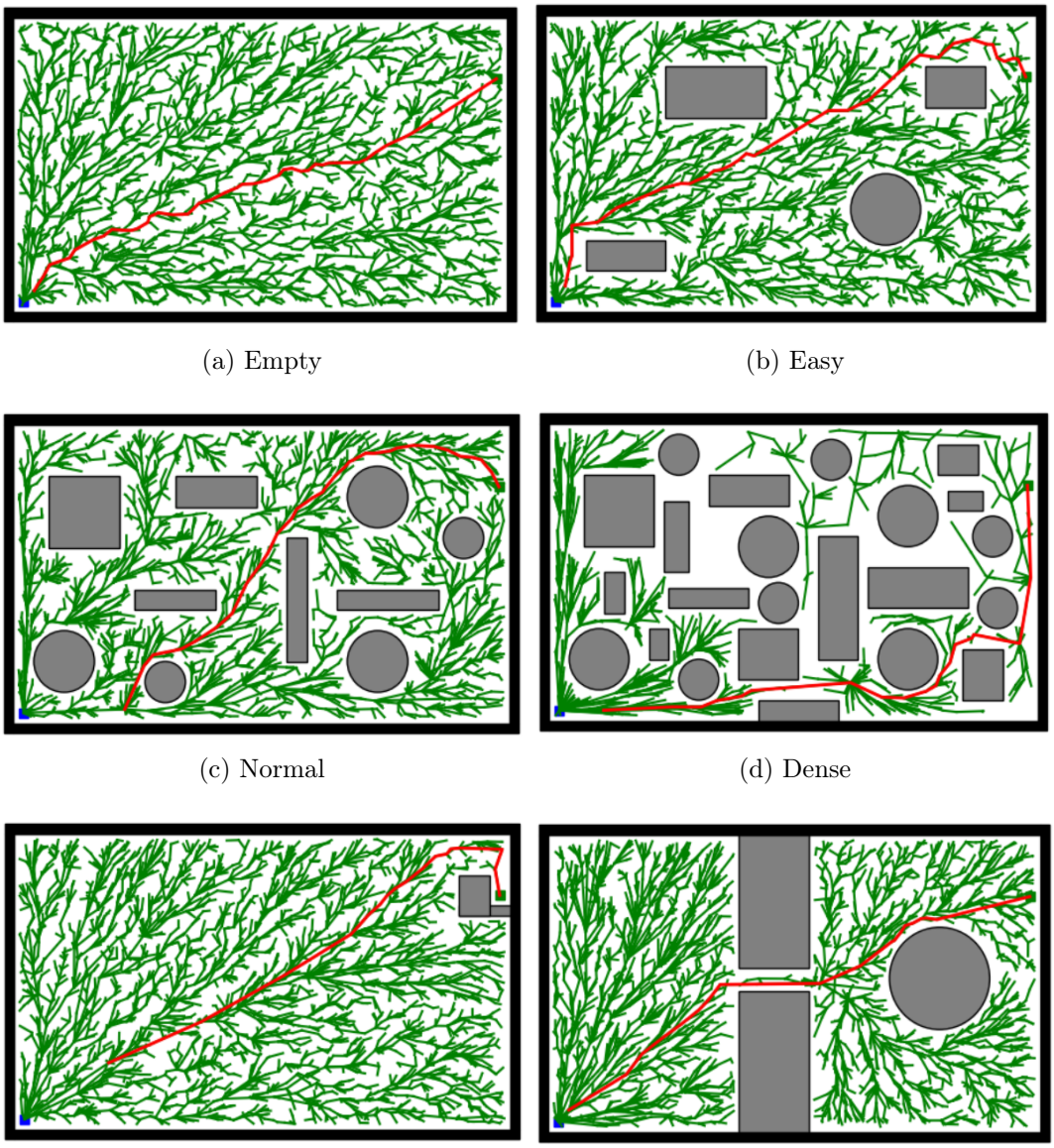
(a) Empty (b) Easy

(c) Normal (d) Dense

(e) Concave (f) Narrow

FIGURE 5.1: Trajectoires obtenues par l'algorithme RRT pour différentes Maps

RRT* :



(a) Empty (b) Easy

(c) Normal (d) Dense

(e) Concave (f) Narrow

FIGURE 5.2: Trajectoires obtenues par l'algorithme RRT* pour différentes Maps

Étude comparative :

— Par rapport à la longueur de trajectoire

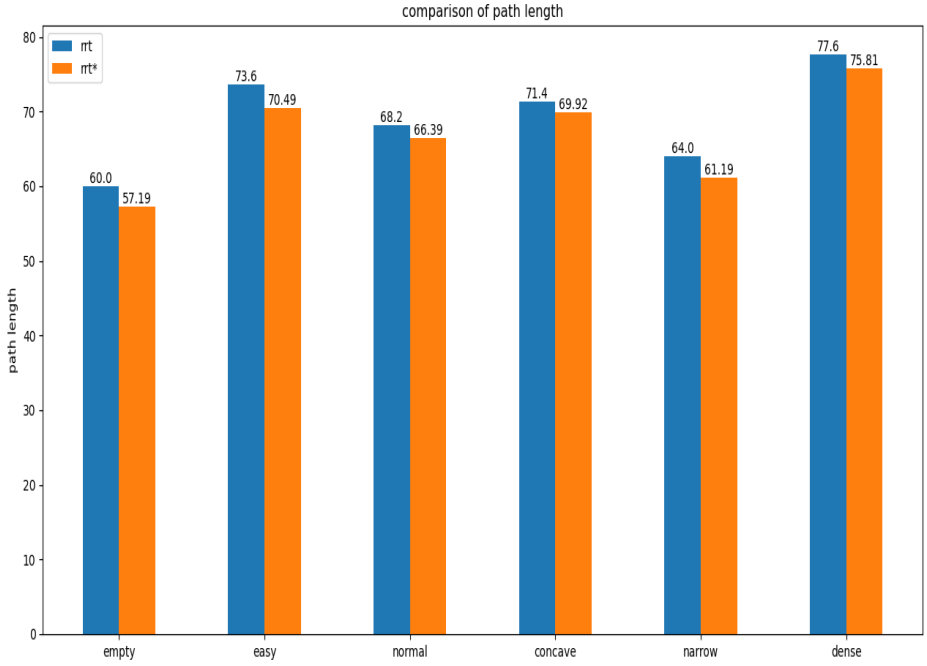


FIGURE 5.3: Comparaison entre la longueur de la trajectoire générée par RRT et celle générée par RRT*

Dans tous les cas de figure, et peu importe la complexité de l'environnement, RRT* trouve toujours une trajectoire plus courte (ce qui se traduit par un coût moins important) que celle obtenue par RRT. La différence de coût varie entre environ 1 et 3 unités de distance.
Cela a confirmé ce que nous avions prévu, car l'optimalité de RRT* est déjà prouvée théoriquement.

— **Par rapport au temps d'exécution**

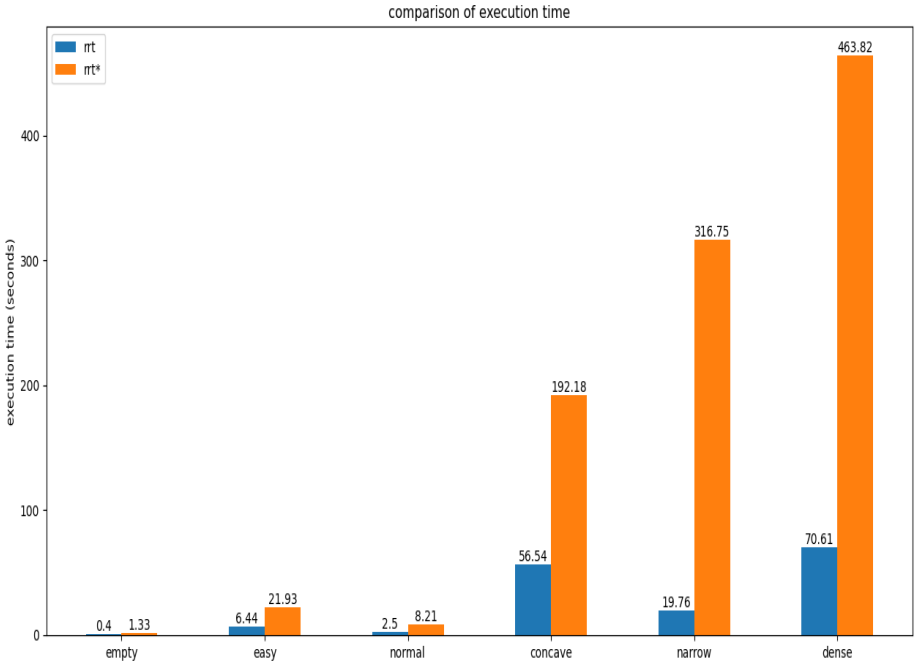


FIGURE 5.4: Comparaison entre le temps d'exécution de RRT et celui de RRT*

Dans tous les environnements testés, RRT* fournit un effort de calcul plus important que RRT : la différence est proportionnelle à la complexité de l'environnement.
Cette différence arrive à 6.5 minutes d'écart entre les temps d'exécution nécéssités par chacun des deux algorithmes.

— **Par rapport au nombre total de nœuds générés**

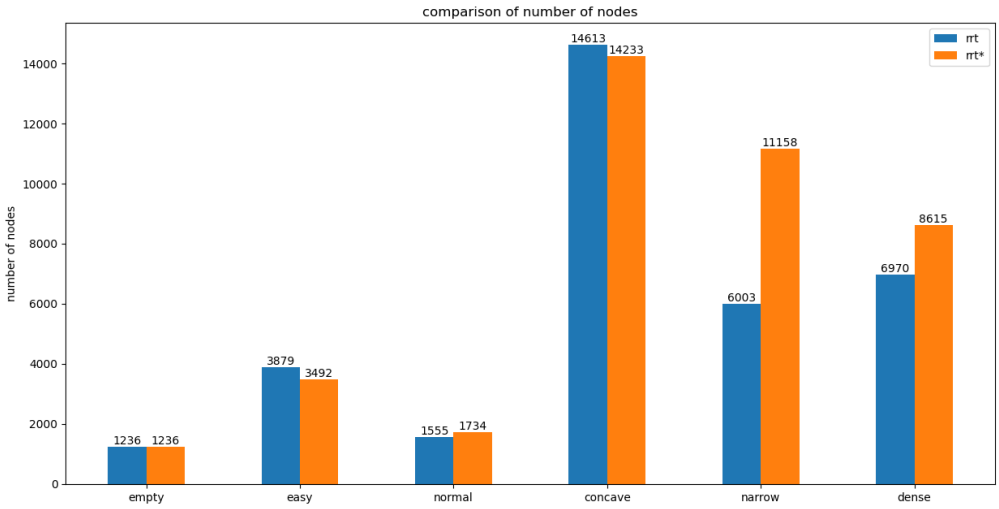


FIGURE 5.5: Comparaison entre le nombre total de nœuds générés par RRT et par RRT*

— **Par rapport au nombre de nœuds générés et non utilisés dans la trajectoire optimale**

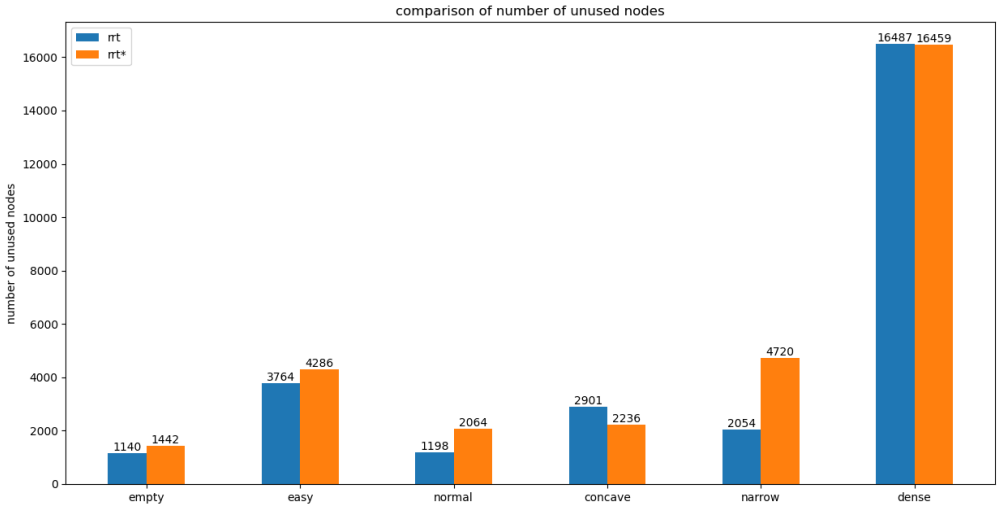


FIGURE 5.6: Comparaison entre le nombre de nœuds non utilisés par RRT et ceux par RRT*

On peut résumer les résultats précédents comme suit :

Map	Iterations n°	RRT			RRT*		
		Runtime	Path length	Total nodes n°	Runtime	Path length	Total nodes n°
Empty	1237	0.4019	60.0	1236	1.3324	57.19	1236
Easy	5334	6.4457	73.6	3879	21.9328	70.49	3492
Normal	2614	2.5003	68.2	1555	8.2199	66.39	1734
Concave	15990	56.5497	71.4	14613	192.1818	69.92	14233
Narrow	14973	19.7641	64.0	6003	316.75	61.19	11158
Dense	28664	70.6140	77.60	6970	463.8273	75.81	8615

FIGURE 5.7: Comparaison entre RRT et RRT*

Conclusion :

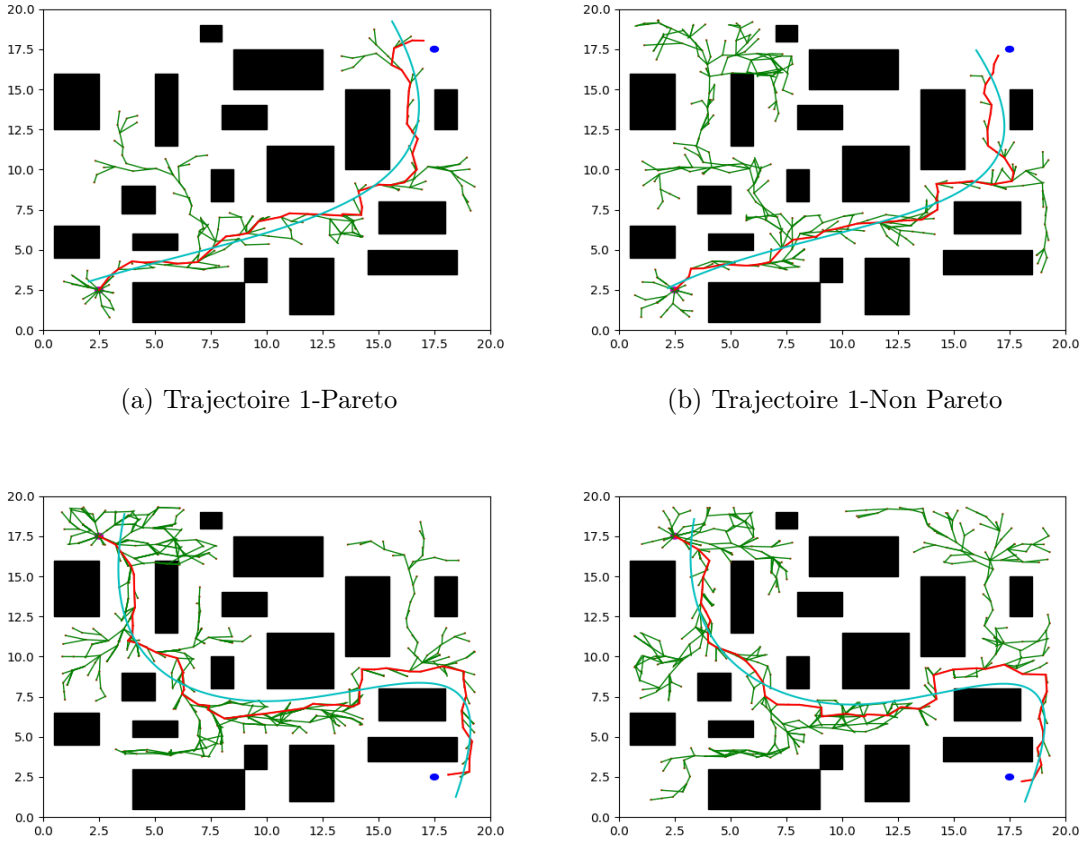
— RRT* nécessite plus de temps de calcul/exécution que RRT ;

— RRT* donne une trajectoire meilleure (plus courte) que RRT pour un nombre donné d'itérations : l'algorithme RRT* converge plus rapidement que RRT vers l'optimum.

— Le nombre total de nœuds générés par chacun de ces deux algorithmes diffère selon l'application/l'environnement de simulation.

5.1.2 Multi-agent

5.1.2.1 Multi-RRT* : cas Pareto-Optimal / Non Pareto-Optimal



(a) Trajectoire 1-Pareto

(b) Trajectoire 1-Non Pareto

(c) Trajectoire 2-Pareto

(d) Trajectoire 2-Non Pareto

FIGURE 5.8: Trajectoires obtenues par les algorithmes à base de Pareto et Non-Pareto

Commentaire :

Dans les mêmes circonstances (même map, point de départ et cible), on voit bien la différence d'effort d'exploration effectué par la solution Non-Pareto par rapport à Pareto qui nécessite moins d'exploration de la map pour trouver la solution.

Comparaison d'exécution :

Pareto :

iteration : 1

Goal reached

Goal reached

Goal reached

collision detected at : 65s at distance : 0.64857122036874

collision : [True, False, True]

trajectory 1 changed - Agent 1 desactivated

Bad utility - Reactive Agent 1

trajectory 3 changed - Agent 3 desactivated

collision : [False, False, False]

[Finished in 1001.4s]

Non Pareto :

iteration : 1

Goal reached

Goal reached

Goal reached

collision detected at : 67s at distance : 0.19112166361573

collision : [True, False, True]

trajectory 1 changed

iteration : 2

collision detected at : 127s at distance : 0.8615731984475

collision : [True, True, False]

trajectory 1 changed

collision : [False, False, False]

[Finished in 4552.1s]

Commentaires :

— On remarque qu'au bout de la première itération, les deux algorithmes (Pareto et Non-Pareto) ont détecté une collision entre les trajectoires 1 et 3. Donc l'un des deux agents (1 ou 3) sera choisi pour changer sa trajectoire, l'autre devra être désactivé.

— Dans le cas où l'optimalité de Pareto est respectée, on voit qu'en premier lieu l'algorithme choisit de désactiver l'agent 1 mais ce choix finit par détériorer la fonction d'utilité. L'algorithme revient sur son choix, réactive l'agent 1 et désactive l'agent 3. Cela finit par éliminer les collisions tout en respectant la condition d'optimalité de Pareto.

— Dans le cas Non-Pareto, l'algorithme commence par désactiver l'agent 1. Ceci élimine en effet le problème initial (collision entre les trajectoires 1 et 3), mais en crée un autre (collision entre les trajectoires 1 et 2). Il passe à l'itération suivante, où il choisit de désactiver l'agent 2, ce qui élimine enfin toutes les collisions.

— On notera donc que dans cet exemple, l'algorithme respectant les fonctions d'utilité soumises au critère défini par l'optimalité de Pareto a trouvé la solution en une seule itération, alors que l'algorithme original l'a trouvée en deux itérations et en prenant plus que 4 fois le temps qu'a pris l'algorithme précédent.

On peut résumer les différences entre les résultats obtenus par l'algorithme multi RRT* dans les cas Pareto et Non-Pareto, par rapport au temps d'exécution ainsi que la longueur des trajectoires obtenues, comme suit :

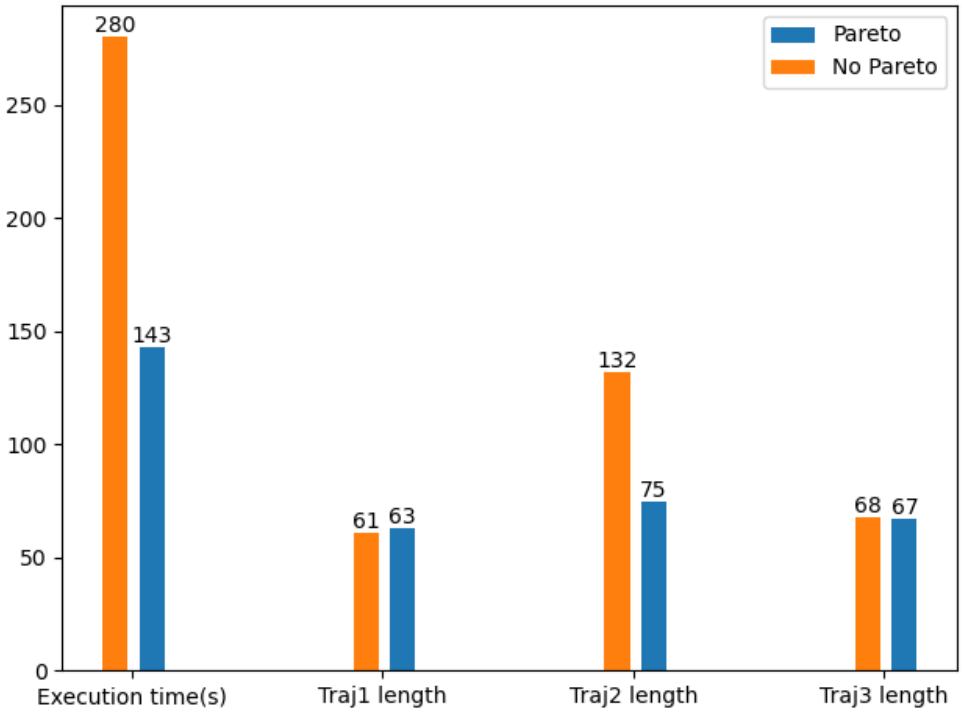


FIGURE 5.9: Pareto / Non-Pareto : Temps d'exécution et Longueur des trajectoires obtenues

On a pu constater également, d'après les résultats des simulation dans les figures 5.8 et 5.9, que l'algorithme à base d'optimalité de Pareto a tendance à explorer moins la map et cibler mieux l'emplacement de ses nœuds. Par ailleurs, il nécessite un temps d'exécution plus court que l'algorithme Multi RRT*, tout en donnant des trajectoires plus ou moins similaires.

Conclusion

— L'optimalité de Pareto améliore la solution MA-RRT* décentralisée en termes de temps d'exécution.

— L'optimalité de Pareto fait en sorte que l'algorithme RRT* cible mieux les trajectoires proposées, en évaluant leur qualité selon les fonctions d'utilité définies.

Simulation de la solution

La simulation est faite sous MATLAB, où chaque agent est représenté par un point mobile dont le dimensionnement est représenté par un rayon de sécurité de 4m (safe distance = 4).

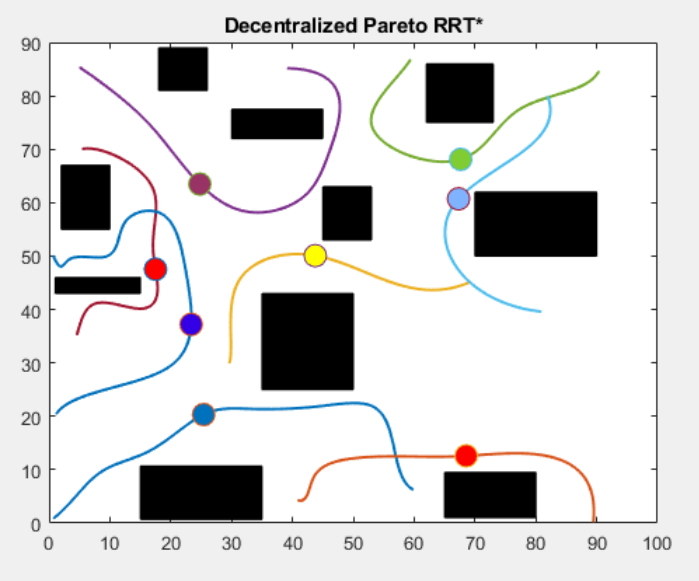


FIGURE 5.10: Simulation de la solution multi RRT* pareto

5.1.3 Conclusion - Approches Classiques

Ayant opté pour des algorithmes de type sample-based, nous avons choisi pour nos simulations les algorithmes RRT et RRT*.

D'après les résultats des différentes simulations dans le cas d'un agent unique, on a pu constater que l'algorithme RRT* est meilleur en termes de qualité de solutions et de vitesse de convergence, quoique relativement plus lent en temps d'exécution par rapport à l'algorithme RRT.

Ayant toutefois estimé que le temps d'exécution de l'algorithme RRT* reste raisonnable, on l'a choisi pour une structure multi-agent décentralisée, qu'on a détaillée dans le chapitre 3.

Le choix de la structure décentralisée ayant naturellement soulevé le problème de priorité (quel agent devrait changer sa trajectoire lorsqu'une collision est prévue). Si le critère de choix de priorité est mal formulé, l'algorithme risque de faire de mauvais choix.

Pour y remédier, nous avons choisi d'introduire la notion d'optimalité de Pareto : lors de la détection d'une collision, une modification dans l'une des trajectoires ne serait acceptée que si cette modification améliorerait au moins la trajectoire d'un autre agent. Ceci nous garantit une qualité de solution acceptable au long terme.

Cette amélioration a effectivement eu comme conséquence de converger plus vite vers la meilleure solution sans collision, et ce en prenant à la fois moins d'itérations et moins de temps de calcul. En effet, l'élimination des mauvais choix de priorité évite des recalculs inutiles de trajectoires : les modifications sont mieux ciblées et la solution finale est atteinte plus rapidement.

5.2 Approche heuristique

5.2.1 Agent unique

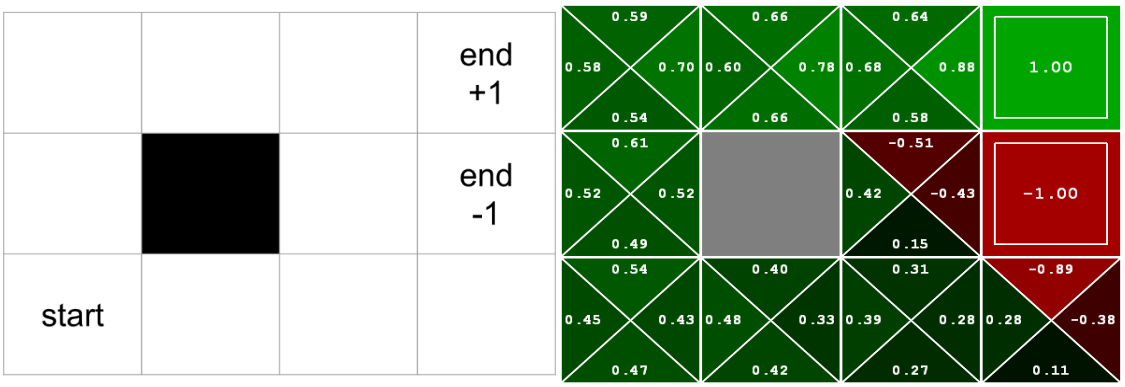
Dans cette partie, on cherche à résoudre le problème d'apprentissage lié à la planification de trajectoire d'un seul agent dans un environnement quelconque, où l'objectif principal est de trouver une trajectoire optimale. L'optimalité de la solution est définie par le fait de maximiser la récompense totale cumulée, ce qui revient à minimiser le nombre d'actions (déplacements) inutiles. En d'autres termes, obtenir la solution optimale revient à trouver la stratégie qui permet à l'agent d'atteindre sa cible par le chemin le plus court.

Q-Learning et SARSA : Nous avons tenté de résoudre le problème d'apprentissage en utilisant les algorithmes Q-learning et SARSA pour quelques environnements.

Pour une première étude, nous avons évalué la convergence vers la stratégie optimale, ensuite la rapidité de convergence ainsi que les caractéristiques de stratégie pour Q-learning et SARSA.

Description d'un MDP 'carte de grille' : [20]
Un environnement carte de grille est un environnement simple pour simuler la planification de trajectoire, où cette dernière peut être extraite de la succession des cellules par lesquelles l'agent passe.

Dans l'environnement ci-dessous, l'agent doit trouver le chemin le plus court du point de départ $start$ au point cible $end + 1$. L'agent a 4 actions possibles à effectuer : up , $down$, $left$, $right$.



(a) Environnement (b) Table-Q optimale

FIGURE 5.11: Exemple de Carte de grille

L'agent a pour chaque état 4 valeurs Q pour chaque action possible dans cet état. On voit bien que le chemin optimal qui lie *start* à $end + 1$ est obtenu en effectuant les actions à valeur Q maximale dans chaque état.

Description du MDP étudié :

Dans notre cas, l'environnement est une carte de grille constituée des cellules :

- Cellule verte : Objectif à atteindre, avec récompense positive.
- Petit carré vert : Notre agent.
- Cellule noire : Obstacle.
- Cellule rouge : Cellule "piège", avec une récompense négative.

Chaque épisode a un nombre d'étapes non limité. La fin de l'épisode est définie par l'atteinte de l'objectif. L'agent s'entraîne à un certain nombre d'épisodes.

Notre agent peut effectuer uniquement 4 actions : up, down, left, right. L'objectif de l'agent est d'aller de sa position initiale à la position cible par le chemin le plus court. Plus vite il termine sa mission, plus élevée sera sa récompense.

Chaque passage d'un état à un autre sera récompensé (ou plutôt dans ce cas, pénalisé) par une petite récompense négative, de sorte que l'agent est encouragé à minimiser le nombre de déplacements.

L'entraînement de l'agent Q learning et SARSA se fait par la même stratégie : $\epsilon - greedy$.

— **Étude de convergence vers l'optimum :**

Dans cette partie, l'objectif est d'atteindre la cible le plus rapidement possible, pour deux différentes configurations d'environnements, chacune ayant son propre degré de complexité.
Le premier environnement (Map1) illustre le cas d'un environnement complexe.

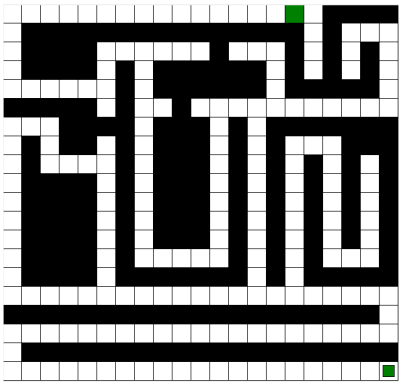


FIGURE 5.12: Map 1

Les graphes ci-dessous illustrent l'évolution de la récompense cumulée maximale de l'agent par épisode.

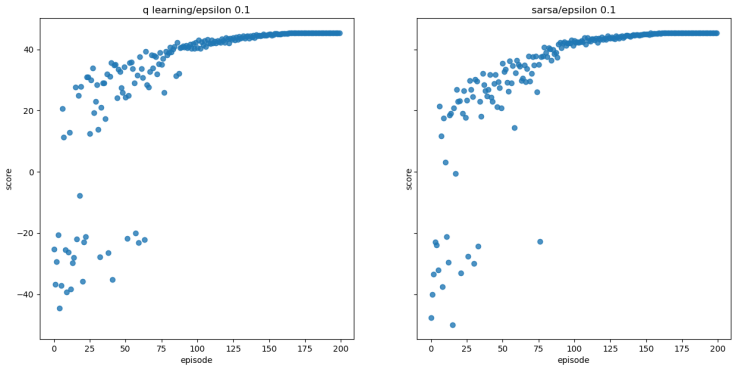


FIGURE 5.13: Évaluation de la convergence vers l'optimum, Map1

À noter que la convergence signifie que la stratégie de l'agent devient très proche de la stratégie optimale, où la courbe d'apprentissage devient constante et n'augmente plus, ce qui veut dire qu'on a atteint le score maximal (récompense) possible.
Pour les premiers épisodes d'entraînement, le score total de l'épisode est faible. Cela est dû au fait que l'agent vient de découvrir son environnemt, il tend donc beaucoup plus à explorer en premier lieu.
Après environ 150 épisodes, la stratégie de l'agent converge vers l'optimale.

Le deuxième environnemt (Map2) illustre le cas où l'objectif sera atteint au long terme, où la cible est beaucoup plus loin. Cela a pour but de tester la rapidité de l'agent à converger vers la stratégie qui lui fera obtenir la récompense cumulée maximale.

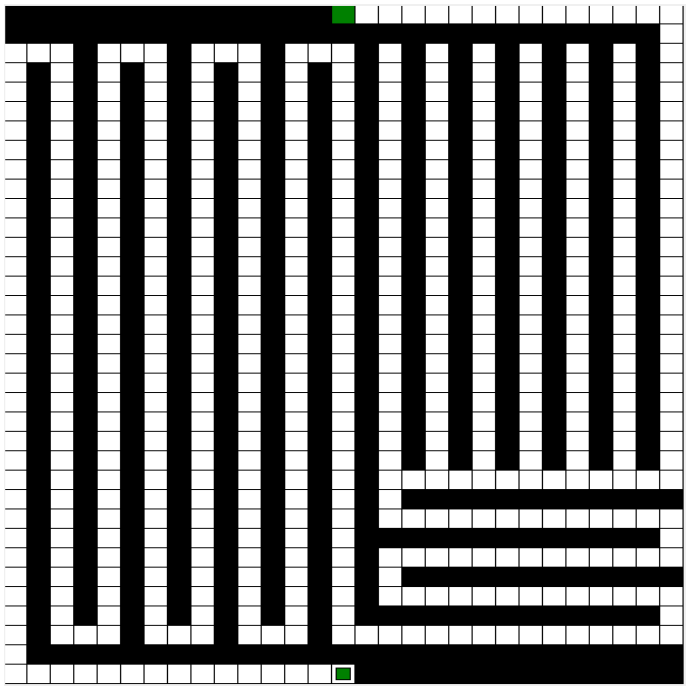
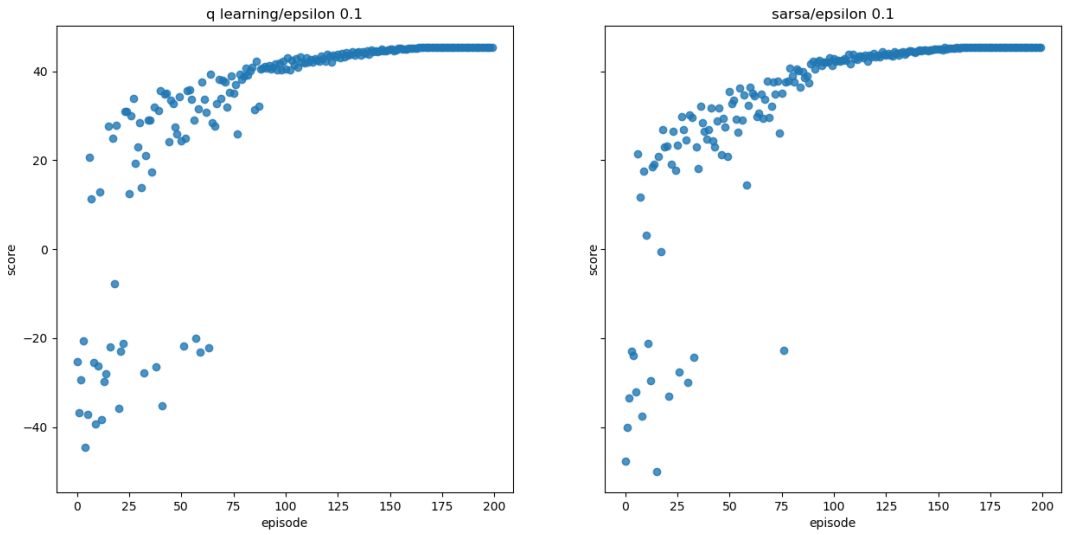


FIGURE 5.14: Map 2

Les graphes ci-dessous illustrent l'évolution de la récompense cumulée maximale de l'agent par épisode.



(a)

FIGURE 5.15: Évaluation de la convergence vers l'optimum, Map2

L'agent découvre son environnemt durant les premiers épisodes, puis il commence à converger.
Après environ 125 épisodes, la stratégie de l'agent converge vers la stratégie optimale.
À noter que dans un environnement pareil, où l'objectif à atteindre est très loin par rapport à l'objectif sous-optimal proche de l'agent, le facteur de remise joue un rôle important sur la vitesse de convergence de l'agent, car il contrôle l'importance des récompenses futures par rapport aux récompenses immédiates.

Conclusion sur la convergence :
En pratique, on considère qu'un algorithme d'apprentissage par renforcement converge lorsque la courbe d'apprentissage devient plate et n'augmente plus.
Et dans les cas que nous avons élaboré dans notre évaluation, Q-Learning et SARSA finissent par converger vers l'optimal peut importe la complexité de l'environnement.

En théorie, il a été prouvé que Q-Learning et SARSA convergent inévitablement vers la solution optimale. Cependant, d'autres éléments doivent être pris en compte car cela dépend du cas de figure et de la configuration adoptée.
En bref, deux conditions doivent être remplies pour garantir la convergence dans la limite, ce qui signifie que la stratégie deviendra arbitrairement proche de la stratégie optimale après une période de temps arbitrairement longue :

1. Décroître le taux d'apprentissage jusqu'à ce qu'il tende vers zéro, mais pas trop rapidement ;
2. Chaque paire état-action doit être visitée infiniment souvent. Ceci a une définition mathématique précise : chaque action doit avoir une probabilité non nulle d'être sélectionnée par la stratégie dans chaque état, c'est-à-dire $\pi(\text{s, a}) > 0$ pour toutes les paires (s, a). En pratique, l'utilisation d'une stratégie ϵ-greedy décroissante (où $\epsilon > 0$) permet de s'assurer que cette condition est satisfaite.

Ces conditions ne disent cependant rien sur la vitesse à laquelle la stratégie se rapprochera de la stratégie optimale.

— **Étude de vitesse de convergence et nature de stratégie :**

Dans cette partie, l'objectif est d'atteindre la cible le plus rapidement possible, mais en présence de pièges à récompense négative. Le chemin optimal doit passer par ces cellule pièges, pour deux configuration d'environnements.
Le premier environnement (Map1) illustre le cas où l'agent doit impérativement passer par les cellules pièges pour atteindre la cible.

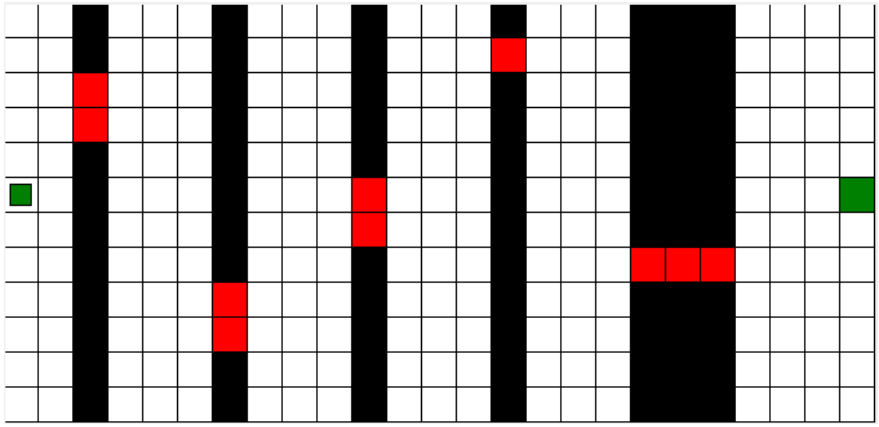


FIGURE 5.16: Map 1

Les graphes ci-dessous illustrent l'évolution de la récompense cumulée maximale de l'agent par épisode.

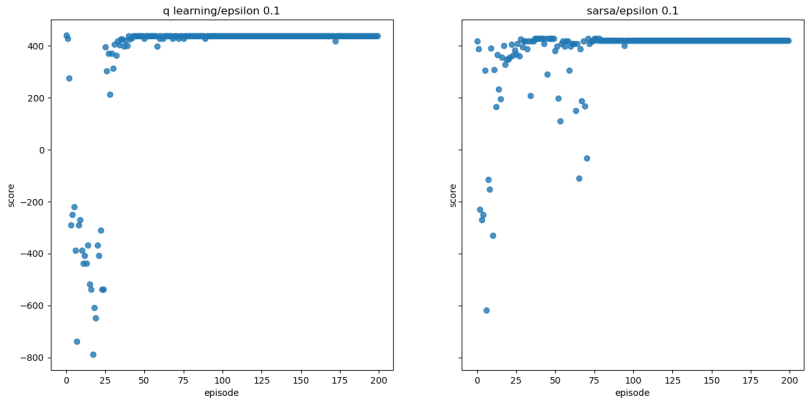


FIGURE 5.17: Évaluation de la vitesse de convergence, Map1

On remarque que dans l'intervalle [25,75] épisodes, le Q-learning a tendance à se stabiliser sur la stratégie optimale rapidement, tandis que SARSA continue à explorer relativement plus longtemps, autrement dit il hésite à se stabiliser immédiatement sur l'optimum qu'il a obtenu. C'est pour cette raison que le Q-learning converge vers l'optimal plus rapidement que SARSA.
Dans cette partie, l'environnement (Map2) diffère au précèdent dans le sens où l'agent a cette fois le choix d'atteindre sa cible en évitant les cellules piège, néanmoins ce chemin est plus long et sous-optimal. Le chemin optimal est celui qui passe par les cellule piège.

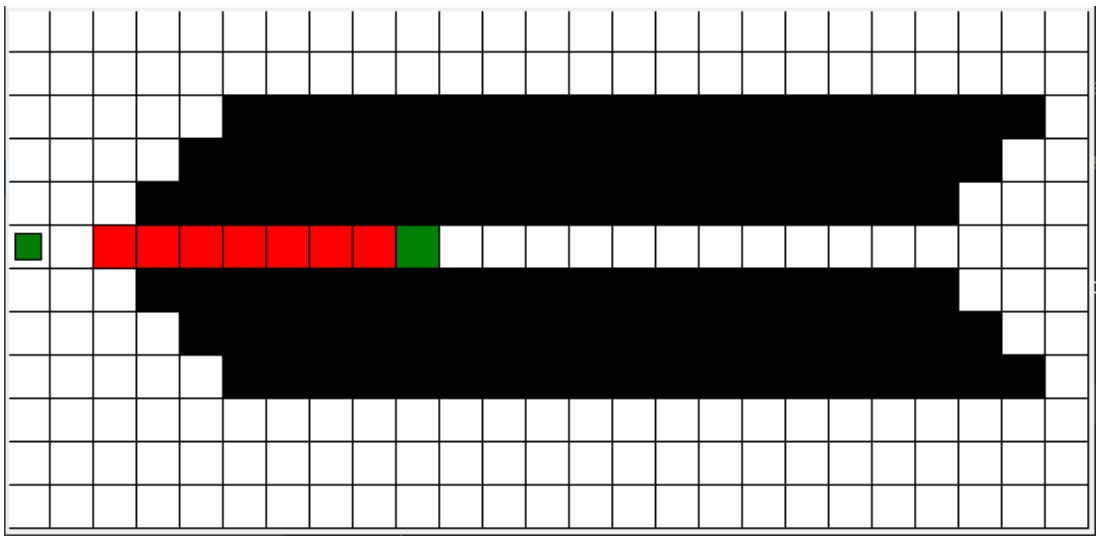


FIGURE 5.18: Map 2

Les graphes ci-dessous illustrent l'évolution de la récompense cumulée maximale de l'agent par épisode.

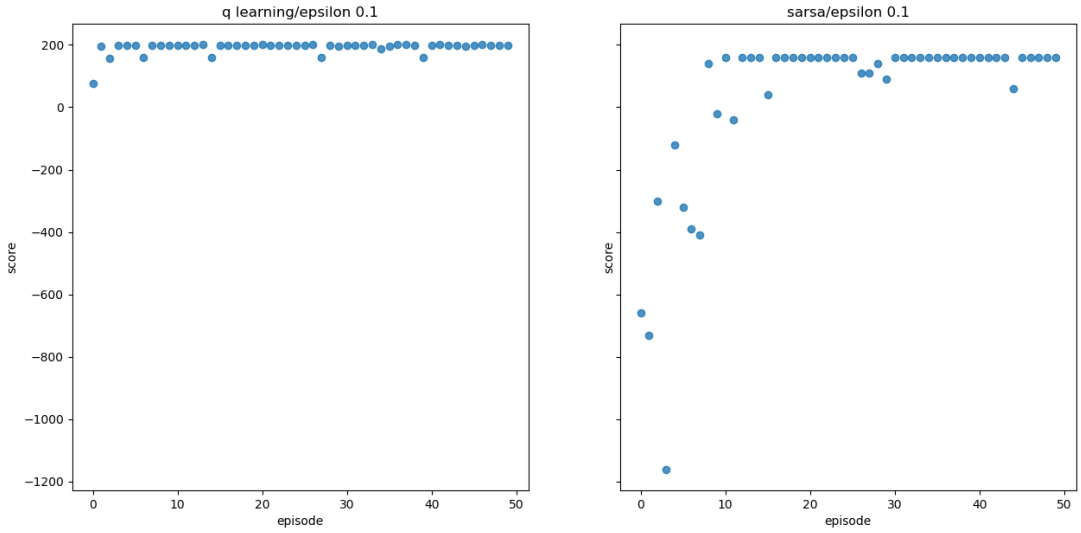


FIGURE 5.19: Évaluation de vitesse de convergence, Map2

Comme dans la Map1, le Q-learning converge plus rapidement, ce qui signifie qu'il est le premier à prendre le chemin à cellules rouges. Autrement dit, le Q-learning est le premier à prendre le risque (où le risque ici est illustré par les cellules rouges à récompense négative) même s'il existe un chemin moins risqué mais plus long.
Contrairement au Q-Learning, SARSA prend moins d'actions "audacieuses", mais il éventuellement finit par converger vers le chemin risqué optimal.

Conclusion sur la nature des stratégies :

— QL converge vers l'optimum plus rapidement que SARSA. Les résultats sont conformes aux attentes car SARSA choisit de jouer de manière plus sûre (la sécurité est représentée ici par le fait d'éviter les cases piège ayant des récompenses négatives) par rapport à QL. Par conséquent, il peut prendre des mesures moins radicales au cours du jeu, ce qui entraîne le besoin de plus de parties (épisodes) pour converger vers la stratégie optimale.

— SARSA (on-policy) apprend les valeurs des actions par rapport à la stratégie qu'il suit, tandis que le Q-Learning (off-policy) le fait par rapport à la stratégie avide. Sous certaines conditions communes, ils convergent tous deux vers la fonction de valeur réelle, mais à des vitesses différentes.

— QL est un algorithme plus agressif : il apprend directement la stratégie optimale, alors que SARSA est plus conservateur. L'exemple classique est la marche près du bord d'une falaise. QL prendra le chemin le plus court parce qu'il est optimal (avec le risque de tomber), tandis que SARSA prendra le chemin le plus long vu qu'il est le plus sûr (pour éviter une chute inattendue).

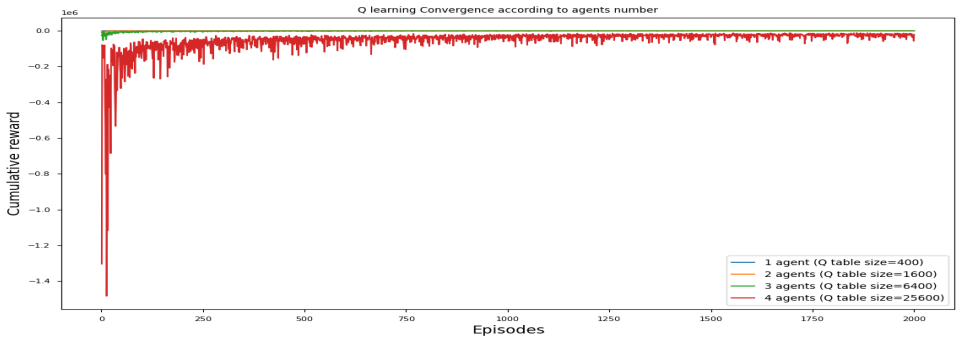
— En pratique, si on veut entraîner rapidement un robot en simulation, QL serait la meilleure option pour le faire. Cependant, si les erreurs sont coûteuses (défaillance minimale inattendue), alors la formation dans le monde réel fait de SARSA la meilleure option.

5.2.2 Multi-agent

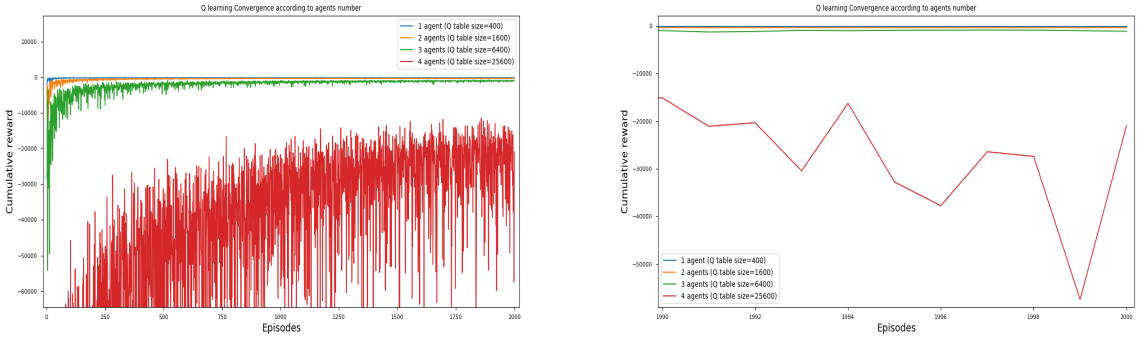
Pour une solution centralisée, nous avons d'abord généralisé le Q-learning comme solution multi-agent. Mais il s'est avéré que cette solution pose quelques problèmes que nous allons détailler dans la section suivante. Une solution proposée pour résoudre ces problèmes va également être détaillée par la suite.

5.2.2.1 Centralized Q-Learning

Dans cette partie, une extension du Q-learning dans le cas multi-agent est proposée, où le MDP est défini par un espace d'actions contenant toutes les actions conjointes possibles, et un espace d'état global qui englobe les états de tous les agents. Nous avons fait une évaluation de performance de cette solution : le graphe ci-dessous illustre la convergence de Q-learning selon le nombre d'agents. Nous avons ajouté deux images agrandies de ce graphe afin de mettre en évidence la différence de convergence entre les 4 cas.



(a)



(b) (c)

FIGURE 5.20: Convergence du QL dans le cas de 1,2,3 et 4 agents

Commentaires

Pour plus que 3 agents, et sur une petite map de dimension 10x10, l'apprentissage QL trouve des difficultés à converger. En termes de ressources de calcul et de mémoire de stockage requis pour la table Q, cette solution n'est pas pratique.

En général, le Q-learning n'est pas vraiment une bonne solution dans le cas où nous aurions affaire à un grand espace d'états et d'actions, car le calcul explicite de la table Q complète pour toutes les paires état-action existantes devient épuisant.

5.2.2.2 Q-Learning et DQN

Le problème de ressources et mémoire nécessaires pour Q-learning nous oriente à penser à une solution qui nécessiterait moins de ressources de calcul. Ce problème est posé en premier lieu à cause du calcul explicite de toutes les paires action-état possibles dans le Q-Learning. Pour cette raison, nous avons cherché une alternative qui se baserait sur l'apprentissage d'une approximation de la fonction Q, au lieu du calcul explicite.

L'un des meilleurs moyens d'approximation, qui a bien prouvé son efficacité, est le réseau de neurones.

Les réseaux de neurones ont une sorte d'universalité qui fait que peu importe ce qu'est la fonction qui lie la paire action-état que l'on veut approximer, il existe un réseau qui peut approximer le résultat de cette fonction. Ce résultat est valable quel que soit le nombre d'entrées et de sorties du réseau.

Une étude comparative entre Q-learning et DQN est faite par la suite.

— **Par rapport à la vitesse de convergence optimale :**

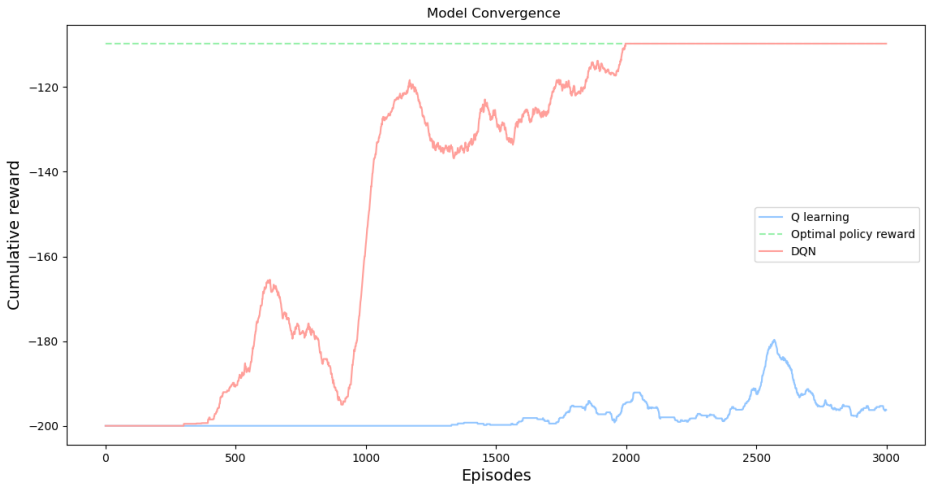


FIGURE 5.21: Comparaison de convergence entre QL et DQN

— **Par rapport au temps de calcul :**

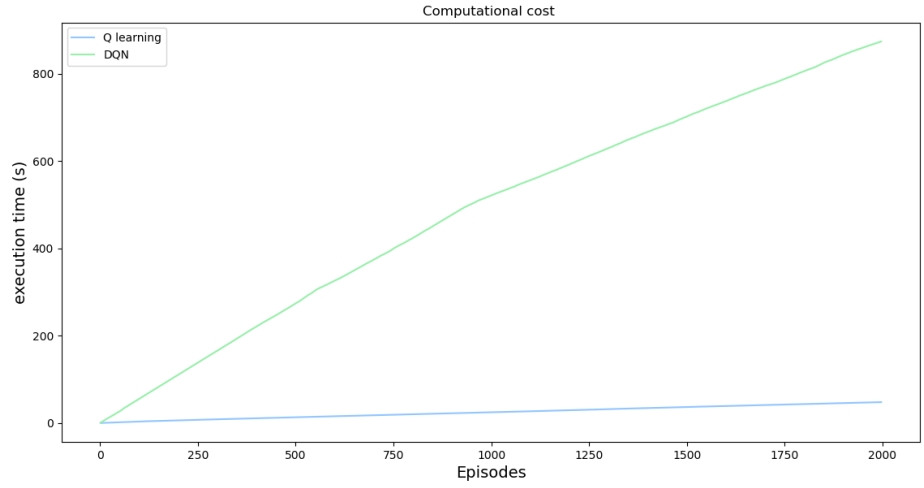


FIGURE 5.22: Comparaison entre les durées d'exécution de QL et DQN

Commentaires

— On note que QL n'a toujours pas trouvé la solution après 3000 épisodes, alors que DQN a pu converger vers la solution optimale au bout de 2000 épisodes.

Cette évaluation est faite pour le cas d'un agent unique, et un environnement moyennement complexe. Plus la complexité de l'environnement augmente, plus la différence de performance entre QL et DQN augmente.

— En revanche, pour un entraînement de 2000 épisodes, DQN a nécéssité 875 secondes, alors que QL n'a pris que 48 secondes.

Conclusions

— Avec DQN, nous pouvons apprendre des environnements beaucoup plus complexes car :

1. DQN utilise un réseau de neurones qui peut prendre des actions qu'il n'a jamais vues auparavant. Avec l'apprentissage Q, si un certain scénario se présente et qu'il est en dehors de toutes les combinaisons discrètes qui lui ont été définies, il va prendre une action aléatoire. Alors qu'un réseau de neurone peut faire l'analogie avec des situations similaires et peut agir en conséquent, ce qui fait sa force dans le cas d'environnements complexes.
2. La taille de la mémoire requise augmente avec la complexité de l'environnement : La moindre augmentation dans la taille de la table Q (en augmentant le nombre d'agents ou bien la complexité de l'environnement) va engendrer une explosion de la quantité de mémoire nécessaire pour maintenir la table Q.

— Pour ces deux raisons, les réseaux de neurones sont bien meilleurs, mais ils prennent également beaucoup plus de temps à s'entraîner. Donc si QL prend quelques minutes, DQN peut prendre des heures d'entraînement. Mais l'avantage ici est que même si DQN nécessite parfois beaucoup de temps pour apprendre, il ne nécessitera qu'un petit espace mémoire.

— Le DQN peut vraiment résoudre différents types d'environnements, alors que l'apprentissage Q est plutôt destiné aux environnements moins complexe. Donc il n'est pas très commun de trouver des applications réelles et avancée à base de Q-Learning, en particulier parce qu'il peut juste traiter des environnements discrets alors que les applications du monde réel sont généralement continues.
— Ainsi, pour les applications multi-agent centralisées, où l'espace d'action croît de manière exponentielle avec l'augmentation du nombre d'agents, et pour les applications du monde réel où les actions des agents ne se limitent pas à se déplacer vers le haut, le bas, la gauche et la droite, QL est pratiquement inutile.

5.2.2.3 Centralized DQN

Dans cette partie, une extension de Deep Q-Networks pour le cas multi-agent est proposée, où le MDP est définie comme précédemment.

Exemple d'application et MDP :

Map1 : Pour cet environnement, l'objectif pour les 3 agents (vert, rouge, bleu) est trouver un chemin qui arrive à la cible (en bas) le plus rapidement possible, tout en gardant la formation à 3.

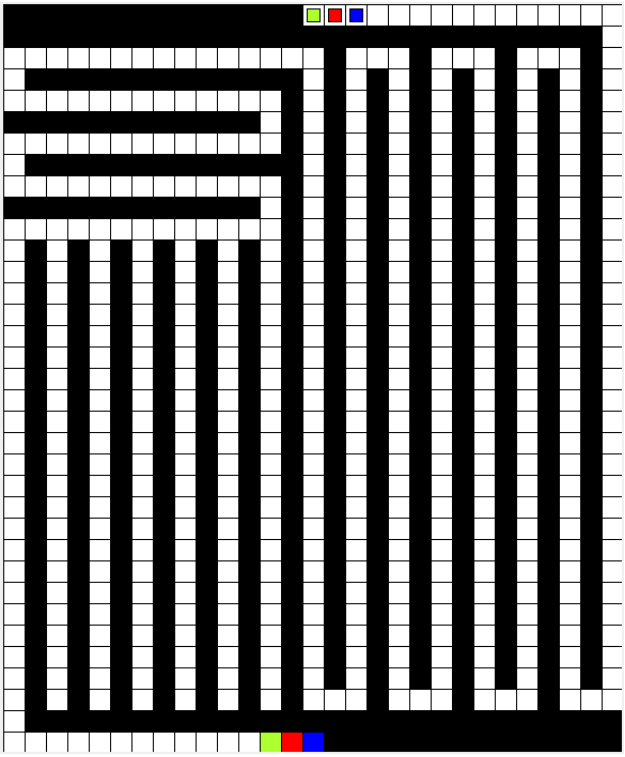


FIGURE 5.23: Map 1

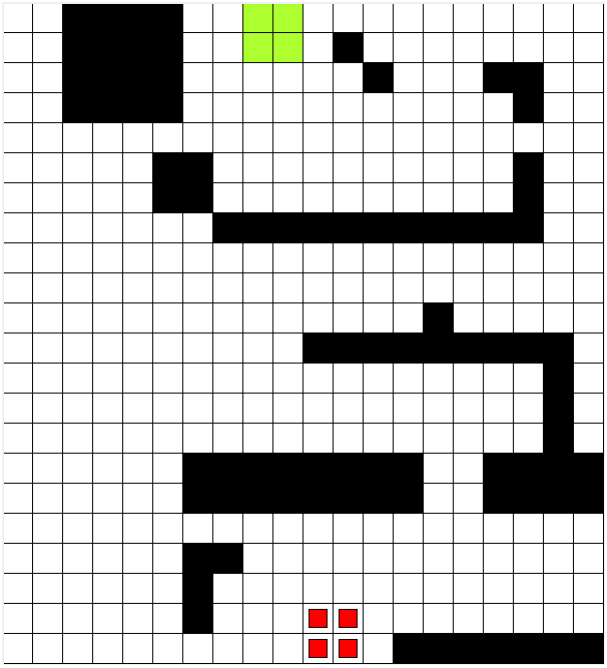
Description du MDP :

État global : x1, x2, x3, y1, y2, y3

Actions : (up,up,up),(up,up,down)... 64 actions jointes.

Récompense : si les agents prennent des actions qui gardent la formation, la récompense est positive (+0.001), sinon pour chaque action hors-formation elle est négative (-0.5). Si la formation arrive à la cible, une récompense positive (+50) sera attribuée. L'agent global est encouragé à minimiser ses actions, par une récompense négative (-0.004) pour chaque action effectuée.

Map2 : Pour cet environnement, l'objectif pour les 4 agents rouges est de trouver un chemin qui arrive à la cible verte le plus rapidement possible, tout en gardant la formation à 4.



(a)

FIGURE 5.24: Map 2

Description du MDP :

État global : x1, x2, x3, x4, y1, y2, y3, y4

Actions : (up, up, up, up), (up, up, up, down)... 256 actions jointes.

Récompense : pareil que dans l'environnement Map1.

- Architecture de solution :

La solution DQN proposée a l'architecture de réseau de neurone décrite ci-dessous, où n représente le nombre d'agents.

Le réseau est composé d'une couche d'entrée qui prend les états globaux de l'agent centralisé comme entrées, 2 couches cachées de 30 et 24 nœuds respectivement, et une couche de sortie où chaque nœud représente une valeur Q d'une action jointe.

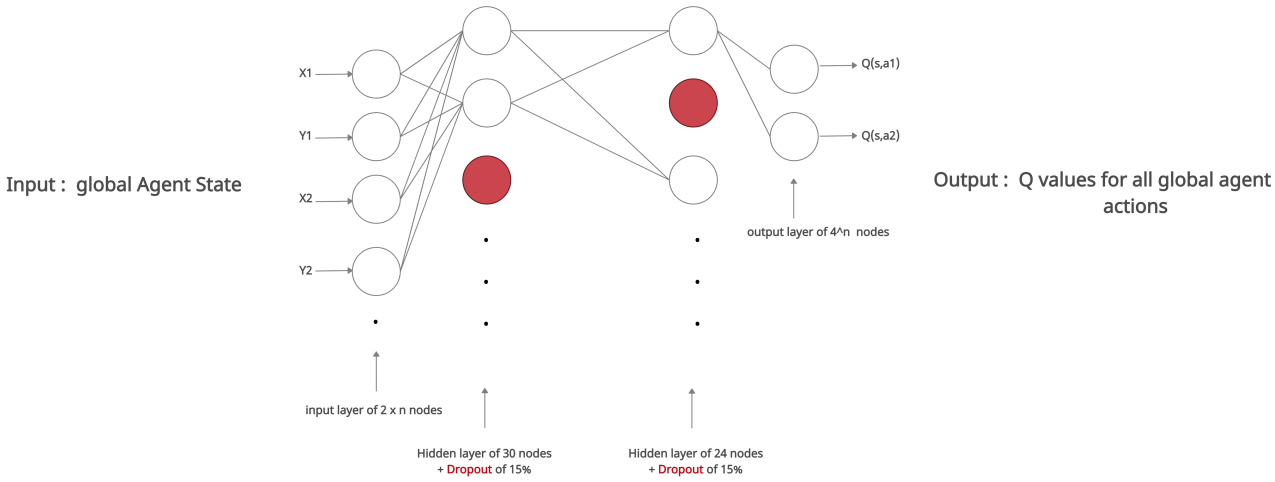


FIGURE 5.25: Architecture proposée de Q-Network et Traget-Network

Une évaluation de performance de cette solution est faite selon le nombre d'agents.

— **Évaluation par rapport au nombre d'épisodes nécessaire pour arriver à l'optimum :**

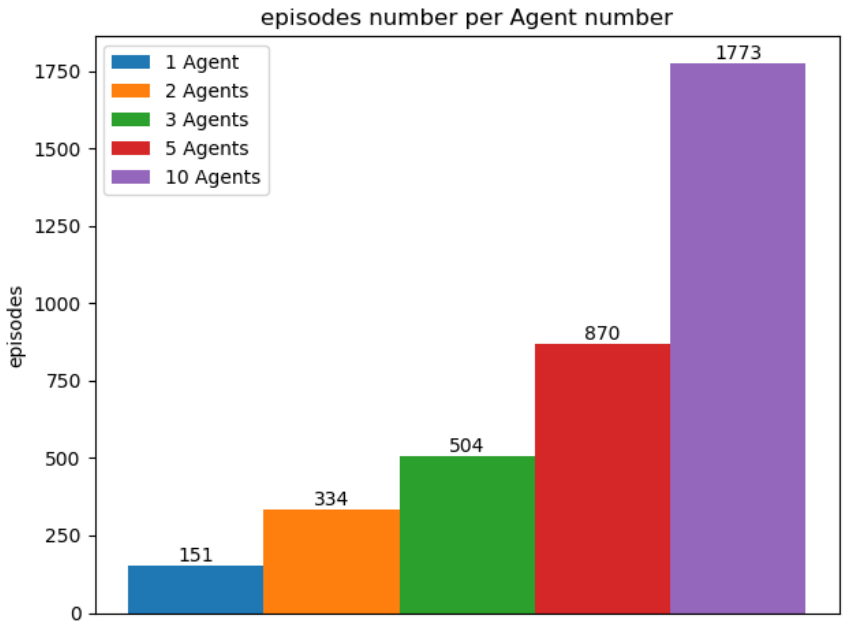


FIGURE 5.26: Convergence du DQN dans le cas de 1,2,3,5 et 10 agents

DQN apprend rapidement en terme d'épisodes, même pour une formation à 10 agents. En effet, environ 1770 épisodes sont suffisants pour résoudre le MDP.

— **Évaluation par rapport au temps nécessaire pour arriver à l'optimum :**

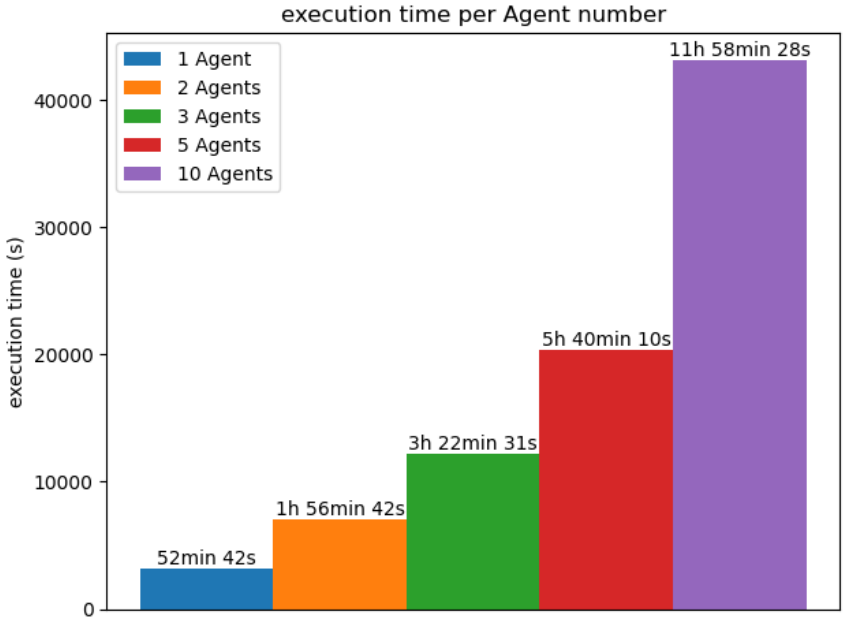


FIGURE 5.27: Temps d'exécution dans le cas de 1,2,3,5 et 10 agents

Le temps de calcul augmente considérablement en augmentant le nombre d'agents. Pour 10 agents, l'exécution prend environ 12 heures.

Conclusion

La solution proposée résout le problème de ressources et mémoire de calcul, mais elle souffre d'un temps de calcul qui augmente largement en augmentant le nombre d'agents.

Dans certains environnements et pour des applications complexes, le Deep Q-learning peut prendre des jours d'entraînement. Ce problème peut être résout par des techniques qui minimisent le temps d'entraînement du réseau de neurones.

Amélioration de l'Apprentissage DQN :

Notre implémentation du Deep Q-Network pour multi-agent souffre d'un temps d'exécution énorme, cela est principalement dû à l'entraînement du DQN qui nécessite deux réseaux de neurones. Donc une amélioration sur l'architecture des réseaux utilisées peut faire la différence. Dans cette section, nous avons tester quelques astuces pour réduire le temps d'entraînement.

1. Le bon choix de la fréquence de mise à jour des paramètres du modèle est primordial. Si on met à jour les poids du modèle trop souvent (par exemple après chaque étape), l'algorithme apprendra très lentement alors que peu de choses auront changé. Dans notre cas, nous avons fini par choisir une fréquence de mise à jour des poids du réseau $QNetwork$ toutes les 50 étapes, ce qui a aidé l'algorithme à s'exécuter beaucoup plus rapidement.
2. Le bon choix de la fonction objectif ($Lossfunction$) du réseau de neurone aide également l'agent à mieux apprendre, comme nous pouvons le constater d'après la comparaison des fonctions objectif selon la convergence de l'erreur du réseau $QNetwork$ dans la figure ci-dessous.

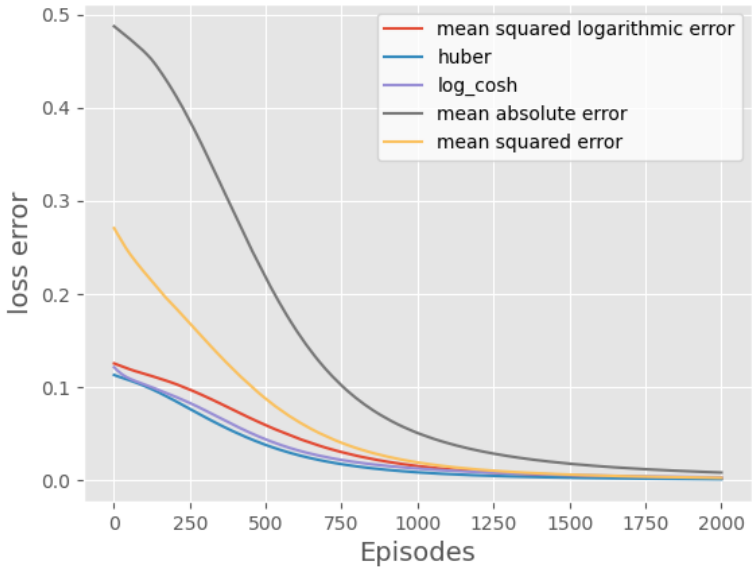


FIGURE 5.28: Choix de la fonction objectif

3. Définir la fréquence correcte pour copier les poids du réseau *Q Network* vers le réseau *Target Network* permet également d'améliorer les performances d'apprentissage. Nous avons initialement essayé de mettre à jour le réseau cible tous les N épisodes, ce qui s'est avéré moins stable car les épisodes peuvent avoir un nombre d'étapes différent : il pourrait y avoir environ 100 étapes dans un épisode et plus que 1000 étapes dans un autre. Nous avons constaté que la mise à jour du réseau *Target Network* toutes les 450 étapes semblait fonctionner relativement bien.
4. L'utilisation du bon algorithme d'optimisation de la fonction objectif choisie *huber*, influence également la vitesse de convergence de l'erreur d'apprentissage, comme on peut le constater ci-dessous à travers la comparaison d'algorithme d'optimisation selon la convergence de l'erreur du réseau *Q Network*.

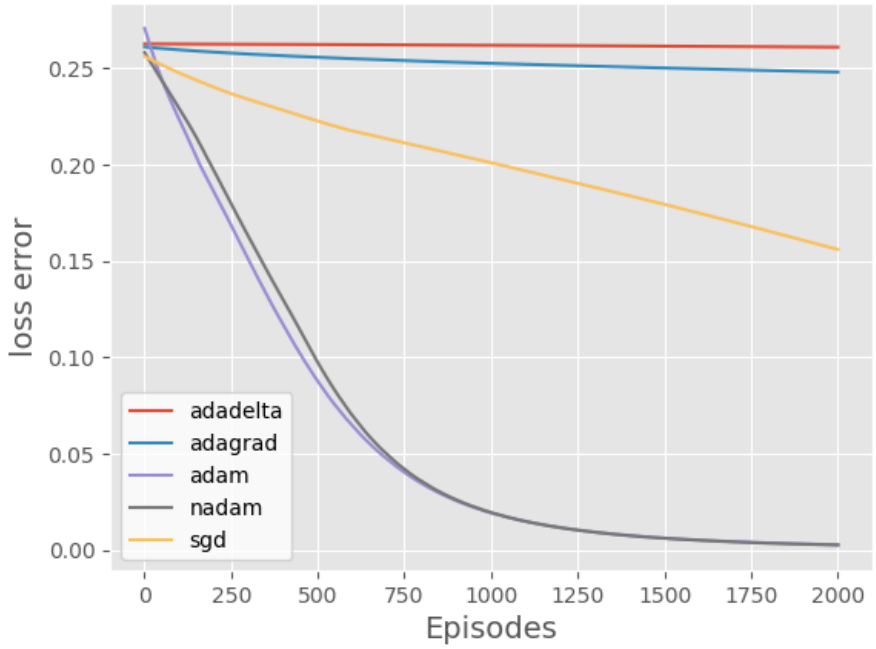


FIGURE 5.29: Choix de l'algorithme d'optimisation

5. L'architecture du réseau, y compris son degré de complexité influence également la vitesse d'apprentissage du réseau. Ci-dessous une comparaison de différentes architectures (nombre de nœuds dans la couche - en utilisant le *Dropout* [1]) selon la convergence de l'erreur du réseau *Q Network*.

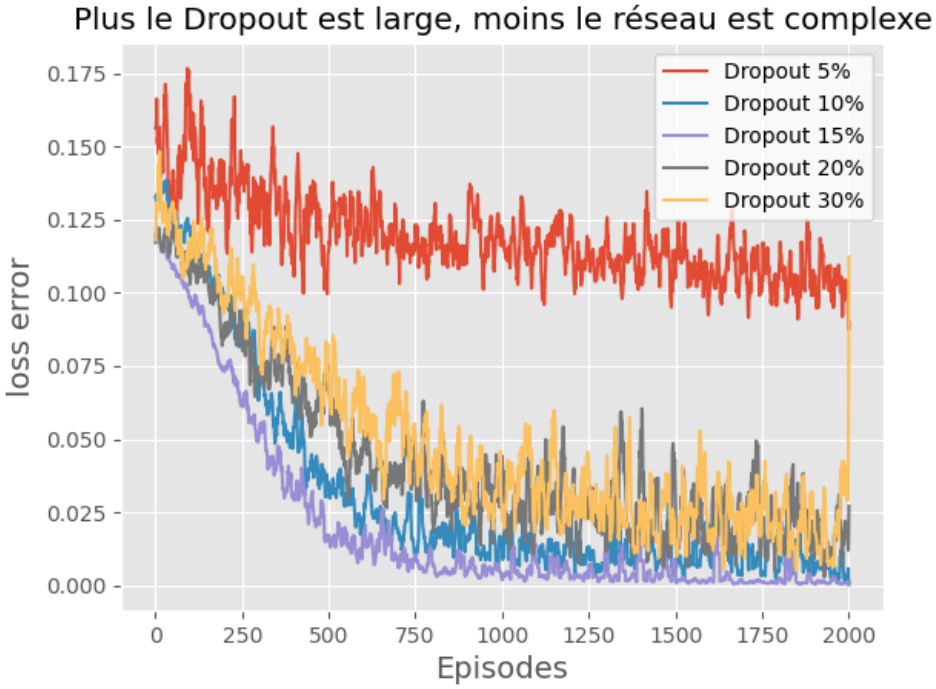


FIGURE 5.30: Complexité du réseau

Conclusion :

Nous avons testé l'amélioration d'architecture de la solution avec les paramètres suivants :

— Fréquence de mise à jours des poids du *Target Network* : 450 étapes.
— Fréquence de mise à jours des poids du *Q Network* : 50 étapes ;
— Utilisation de fonction objectif : *Huber objectif function* ;
— Algorithme d'optimisation : *Adam* et un *Dropout* de 15%.

1. Le Dropout est une technique qui est destinée à empêcher le sur-apprentissage sur les données d'entraînement en abandonnant des unités dans un réseau de neurones. Cela force le modèle à éviter de trop apprendre, en induisant une complexité non nécessaire.

Ce qui nous a donné les résultats suivants :

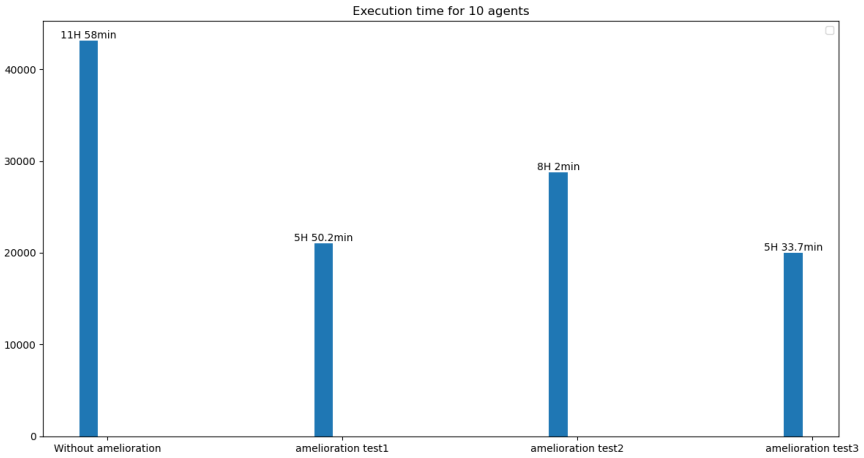


FIGURE 5.31: Amélioration de la solution en termes de temps d'exécution

L'histogramme montre que l'amélioration a effectivement diminué le temps d'exécution de la solution. Pour une formation de 10 agents, le temps d'exécution est passé de 12h à environ de 6h30min.

Simulation de la solution

La simulation est faite sous MATLAB, où chaque agent est représenté par un point mobile dont le dimensionnement est représenté par un rayon de sécurité de 4m (safe distance = 4). Les trajectoires obtenue sont interpolées par des fonctions *Spline cubique* pour les rendre lisses.

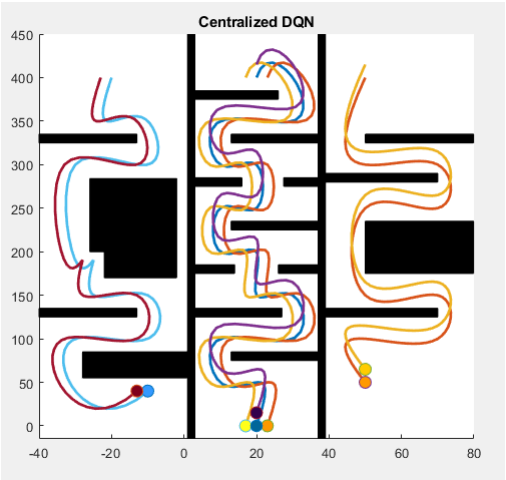


FIGURE 5.32: Simulation de la solution améliorée à base de DQN

5.2.3 Conclusion - Approche Heuristique

Un problème d'apprentissage par renforcement dans le cas d'un agent unique peut être résolu par $Q-learning$ et $Sarsa$ vu que la convergence vers l'optimum est vérifiée. Toutefois, le choix entre ces deux algorithmes dépend des exigences du problème en question. Dans notre cas, il s'agit d'entraînements en simulation : nous avons donc fini par choisir $Q-learning$ pour bénéficier de sa vitesse de convergence.

Pour le cas multi-agent, nous avons étendu notre solution à un agent basée sur le $Q-learning$ à une structure centralisée. Cependant, cette solution souffre du problème de dimensionnalité que cause le calcul explicite des valeurs de la Q-table, qui demande des ressources énormes. Les exigences de cet algorithme augmentent de façon dramatique pour la moindre augmentation dans le nombre d'agents.

Une solution pour les problèmes susmentionnés avec le $Q-learning$ centralisé est de remplacer le calcul explicite par des approximateurs de fonctions. Les réseaux de neurones présentant une solution efficace et fiable, on a opté pour l'algorithme DQN qui va éliminer le problème du calcul explicite en apprenant une fonction approximative à l'aide de réseaux de neurones.

La solution DQN centralisée peut gérer à la fois le nombre d'agents participant à la tâche, ainsi que la complexité de l'environnement. Cependant, cette solution souffre d'un temps de calcul énorme, qui a atteint 12H pour 10 agents. Pour remédier au problème du temps de calcul, on a proposé d'améliorer l'architecture des réseaux de neurones de la solution, ainsi que ses paramètres. La solution améliorée a diminué le temps d'exécution de 12H à environ 6H30.

Simulation sur des robots mobiles :

La simulation est faite sur des robots différentiels sous *Matlab*, en utilisant le package *Robotics system toolbox*.

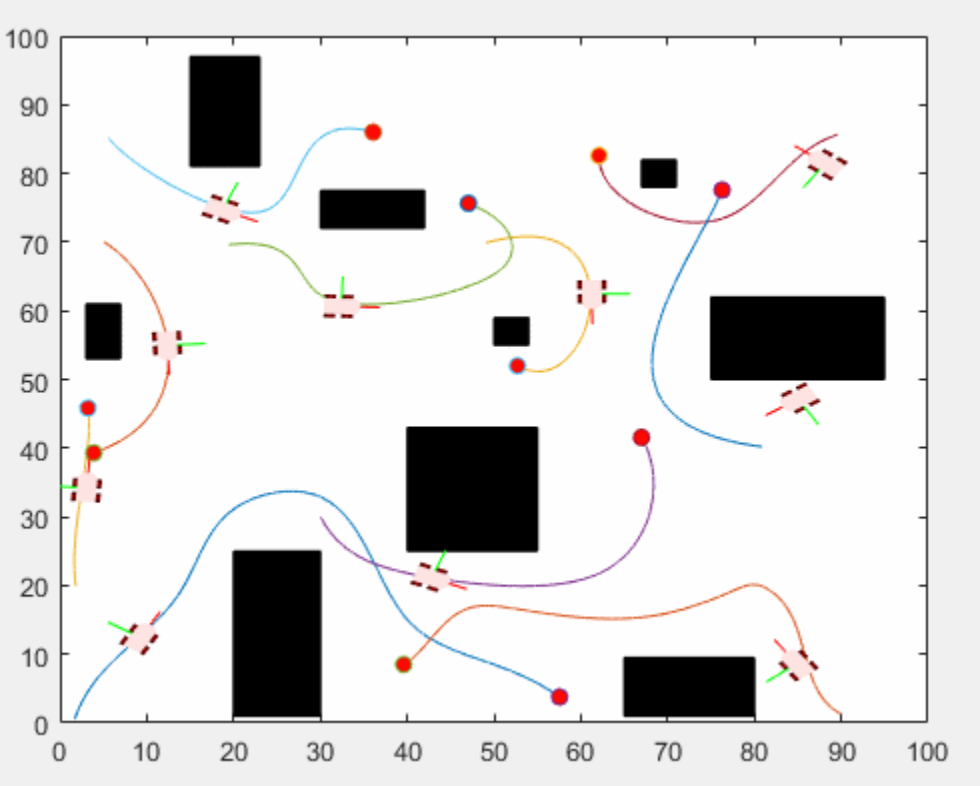


FIGURE 5.33: Solution décentralisée

Le modèle du robot utilisé est *differentiel Drive Kinematics*. Ce dernier crée un modèle de véhicule à entraînement différentiel pour simuler une dynamique de véhicule simplifiée. Ce modèle se rapproche de celui d'un véhicule avec un seul essieu fixe et des roues séparées par une largeur de voie spécifiée. Les roues peuvent être entraînées indépendamment. La vitesse et le cap du véhicule sont définis à partir du centre de l'essieu. L'état du véhicule est défini comme un vecteur à trois éléments, [x y theta], avec une position xy globale, spécifiée en mètres, et un cap du véhicule, theta, spécifié en radians. Il est possible de calculer les états dérivés dans le temps du modèle en utilisant la fonction dérivée avec les commandes d'entrée et l'état actuel du robot. La solution décentralisée génère des trajectoires dynamiquement faisable et relativement meilleures que celles générées par la solution centralisée.

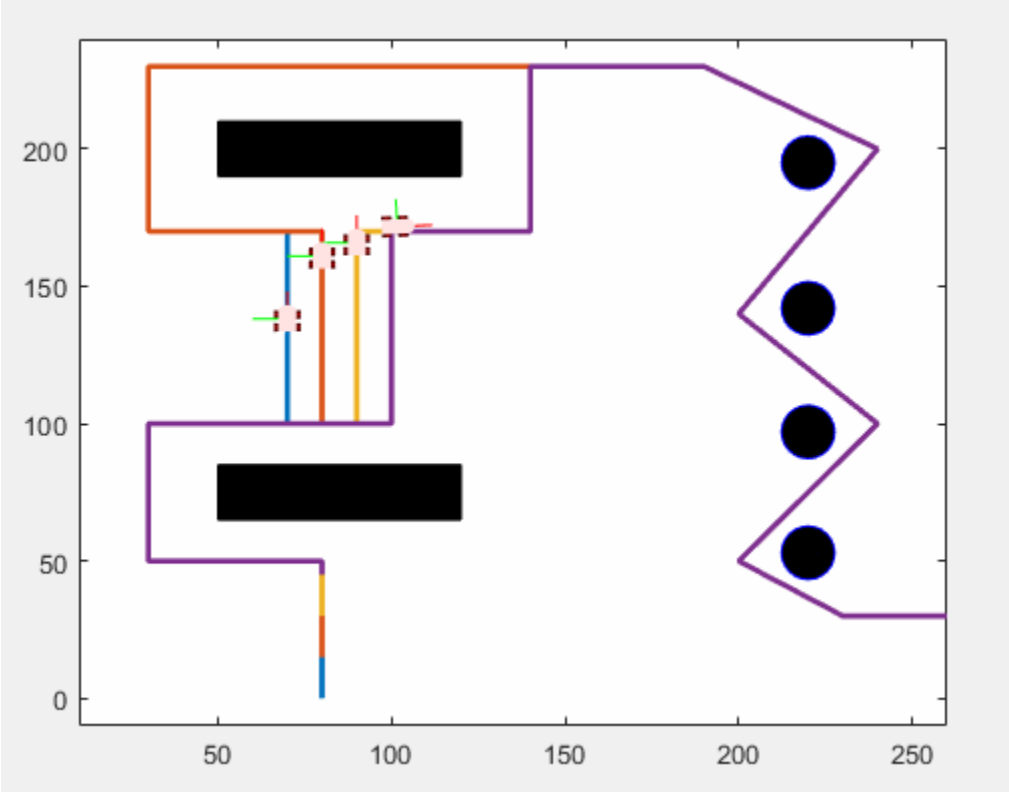


FIGURE 5.34: Solution centralisée

Cela est dû à la nature de l'implémentation de RRT*, qui prend en compte les caractéristiques dynamiques de l'agent, alors que l'implémentation par Reinforcement Learning se concentre sur l'apprentissage où l'agent n'a que 4 mouvements possibles. Cela est uniquement causé par la façon dont l'implémentation est définie, donc une implémentation meilleure ou une amélioration des trajectoires obtenues pourrait y remédier.

Conclusion Générale

Motivés par l'engouement croissant autour des systèmes multi-agent, ce mémoire avait pour ambition de contribuer à un domaine qui combine l'automatique à cette thématique, à savoir l'optimisation dans la planification de trajectoires dans le cas d'un agent unique, puis dans une formation d'agents.

En premier lieu, nous avons tenté de survoler les différentes approches dans le domaine de la planification de trajectoires, plus particulièrement les approches classiques et heuristiques.

Dans l'approche classique, nous avons choisi les méthodes basées sur l'échantillonnage (sample-based). On a testé pour différentes circonstances les algorithmes RRT et RRT*, et établi un bilan comparatif. L'algorithme RRT* ayant donné de meilleurs résultats dans le cas d'un agent unique, nous l'avons choisi par la suite pour le cas multi-agent en adoptant une structure décentralisée et tenté de l'améliorer en termes de performances en introduisant l'optimalité de Pareto.

Dans l'approche heuristique, nous avons choisi le Reinforcement Learning. On a testé d'abord pour un agent unique dans différentes circonstances les algorithmes SARSA, Q-Learning et DQN. Pour notre application, nous avons adopté une structure centralisée pour notre formation et il s'est avéré que DQN était le meilleur, voire le seul choix judicieux grâce aux avantages qu'apporte le calcul basé sur les réseaux de neurones. Présentant toutefois un inconvénient en termes de temps d'exécution, nous avons tenté vers la fin d'améliorer cette solution en jouant sur l'architecture des réseaux de neurones, ainsi que les hyperparamètres de l'apprentissage.

Dans la perspective de continuité de ce travail, nous proposons les alternatives suivantes :

— Explorer les différentes techniques d'apprentissage par imitation, comme partie de l'apprentissage par renforcement pour résoudre le problème de planification.
— Généraliser la solution proposée à base de l'optimalité de Pareto dans le cas 3D pour une éventuelle application à base de drônes.
— Introduire la notion d'incertitude au niveau de modèle du mouvement du robot ainsi que l'emplacement des obstacles.
— Envisager d'améliorer l'architecture de la solution centralisée (Dueling Networks, Residual Networks, ..).
— Explorer la possibilité de solution hybride qui combine les architectures centralisée et décentralisée.

Bibliographie

[1] D.D.Lee V.Kumar A.Ribeiro A.Khan, C.Zhang. Scalable centralized deep multi-agent reinforcement learning via policy gradients, 2018.

[2] A.Mahdavi and M.Carvalho. Distributed coordination of autonomous guided vehicles in multi-agent systems with shared resources. *IEEE, SoutheastCon, Huntsville, AL, USA, pp. 1-7*, 2019.

[3] S.Blackwell C.Alcicek R.Fearon A.De Maria V.Panneershelvam M.Suleyman C.Beattie S.Petersen S.Legg V.Mnih K.Kavukcuoglu D.Silver A.Nair, P.Srinivasan. Massively parallel methods for deep reinforcement learning. *ResearchGate*, 2015.

[4] Adham Atyabi and David Powers. Review of classical and heuristic-based navigation and path planning approaches. *International Journal of Advancements in Computing Technology (IJACT)*, 5 :1–14, 2013.

[5] F. Valero C. Llopis-Albert, F. Rubio. Optimization approaches for robot trajectory planning. *Multidisciplinary Journal for Education*, 2018.

[6] H.La D.Connell. Dynamic path planning and replanning for mobile robots using rrt. pages 1429–1434, 2017.

[7] J.Canny D.Parsons. A motion planner for multiple mobile robots. *ResearchGate*, 1996.

[8] N.Immerman S.Zilberstein D.S.Bernstein, R.Givan. The complexity of decentralized control of markov decision processes. *Mathematics of Operations Research 27(4), 819–840*, 2002.

[9] C.Amato F.A.Oliehoek. Concise introduction to decentralized pomdps, vol.1. *Springer*, 2016.

[10] A.Zelinsky G.Cheng. A physically grounded search in a behavior based robot. *ResearchGate*, 1997.

[11] D.Silver H.Van Hasselt, A.Guez. Deep reinforcement learning with double q-learning. *ResearchGate*, 2015.

[12] P.L.Bartlett J.Baxter. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 2001.

[13] J.C.Latombe. Robot motion planning. *Kluwer Academic Publisher*, 1991.

[14] R.C.Eberhart J.Kennedy. Particle swarm optimization. *Proceedings of ICNN'95 - International Conference on Neural Networks, pp. 1942-1948 vol.4*, 1995.

[15] T.Basar K.Zhang, Z.Yang. Multi-agent reinforcement learning : A selective overview of theories and algorithms, 2019.

[16] B.De Schutter L.Busoniu, R.Babuska. Multi-agent reinforcement learning : An overview. *Springer International Publishing*, 2010.

[17] J.C.Latombe M.H.Overmars L.E.Kavraki, P.Svestka. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation, vol. 12, no. 4, pp. 566-580.*, 1996.

[18] L.S.Shapley. Stochastic games. *Proceedings of the National Academy of Sciences*, 1953.

[19] O.Castillo R.Sepulveda P.Melin M.A.P.Garcia, O.Montiel. Path planning for autonomous mobile robot navigation with ant colony optimization and fuzzy cost evaluation. *Journal of Central South University of Technology à Changsha, Chine*, 2009.

[20] M.Arango. Deep q-learning explained. *ResearchGate*, 2018.

[21] D. Mehta. State-of-the-art reinforcement learning algorithms. *International Journal of Engineering Research and Technology (IJERT)*, 2019.

[22] Zanele G.N. Mkhize. Motion planning algorithms for autonomous robots in static and dynamic environments. Master's thesis, University of JOHANNESBURG, South Africa, 2011.

[23] Ferguson M.Likhachev. Planning long dynamically feasible maneuvers for autonomous vehicles. *The International Journal of Robotics Research*, 2009.

[24] M.L.Littman. Markov games as a framework for multi-agent reinforcement learning. *International Conference on Machine Learning*, 1994.

[25] M.W.Dunnigan M.M.Mohamad, N.K.Taylor. Articulated robot motion planning using ant colony optimisation. *3rd International IEEE Conference Intelligent Systems, pp. 690-695*, 2006.

[26] A.Kelly M.Pivtoraiko, R.A.Knepper. Differentially constrained mobile robot motion planning in state lattices. *Journal of Field Robotics*, 2009.

[27] L.Bascetta M.Ragaglia, M.Prandini. Multi-agent poli-rrt. *Springer International Publishing*, 2016.

[28] Office of the Secretary of Defense. Unmanned aircraft systems roadmap. 2005.

[29] O.Khatib. Real time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotics Research*, 1986.

[30] V.Lesser P.Xuan. Multi-agent policies : From centralized ones to decentralized ones. pages 1098–1105, 2002.

[31] R.J.Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Kluwer Academic Publishers, Boston*, 1992.

[32] A.Perez E.Frazzoli S.Teller S.Karaman, M.R.Walter. Anytime motion planning using the rrt*. *IEEE International Conference on Robotics and Automation, pp. 1478-1483*, 2011.

[33] E.Frazzoli S.Karaman. Optimal kinodynamic motion planning using incremental sampling-based methods. *49th IEEE Conference on Decision and Control (CDC), pp. 7681-7687*, 2010.

[34] S.M.Lavalle. Rapidly-exploring random trees : A new tool for path planning. *ResearchGate*, 1998.

[35] A.Sloman T.Guan-Zheng, H.E.Huan. Global optimal path planning for mobile robot based on improved dijkstra algorithm and ant system algorithm. *Journal of Central South University of Technology à Changsha, Chine*, 2006.

[36] I.Antonoglou D.Silver T.Schaul, J.Quan. Prioritized experience replay. 2015.

[37] P. Verbari. Multi-rrt* : A sample-based algorithm for multi-agent planning. Master's thesis, ING - Scuola di Ingegneria Industriale e dell'Informazione, 2017.

[38] V.Gao V.Konda. Actor-critic algorithms. *ResearchGate*, 2000.

[39] D.Silver A.Graves I.Antonoglou D.Wierstra M.Riedmiller V.Mnih, K.Kavukcuoglu. Playing atari with deep reinforcement learning. 2013.

[40] N.Li Y.G.Cen Y.Q.Qin, D.B.Sun. Path planning for mobile robot using the particle swarm optimization with mutation operator. *Proceedings of 2004 International Conference on Machine Learning and Cybernetics (IEEE Cat. No.04EX826), 2004, pp. 2473-2478 vol.4*, 2004.

Liens

[Link1] A beginner's guide to deep reinforcement learning. https://wiki.pathmind.com/deep-reinforcement-learning.

[Link2] Deep reinforcement learning for maze solving. https://www.samyzaf.com/ML/rl/qmaze.html.

[Link3] Deep reinforcement learning : The tour de flags test case. https://samyzaf.com/ML/tdf/tdf.html.

[Link4] Neural networks. https://www.ibm.com/cloud/learn/neural-networks.

[Link5] Pathplanning. https://github.com/zhm-real/PathPlanning.git.

[Link6] Pspace-complete. https://wikimili.com/en/PSPACE-complete.

[Link7] Q-learning-gridworld. https://github.com/ludobouan/Q-learning-gridworld.git.

[Link8] Reinforcement learning - algorithms. https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html.

[Link9] Reinforcement learning - goal oriented intelligence. https://deeplizard.com/learn/video/xVkPh9E9GfE.

[Link10] Rrt. https://github.com/p-akanksha/RRT.git.