IA Jeux Tour Par Tour

RAPPORT DE PROJET

Aurélien Castel – Léa Da Costa – Kevin Seri – Nicolas Guiblin

Sommaire

I. Synthèse

Le projet avait pour but de créer une application modulaire pouvant jouer à des jeux en tour par tour. Actuellement, l'application permet de jouer au jeu de Nim, au Tictactoe et au Quoridor.

Pour qu'un utilisateur humain puisse jouer face à un adversaire, plusieurs algorithmes ont été créés, notamment Minimax[i], Q-learning[ii] et SARSA[iii]. Ces deux derniers peuvent même apprendre : ce sont des algorithmes d'apprentissage par renforcement[iv] ou « reinforcement learning ».

La modularité de l'application permet de créer de nouveaux jeux et algorithmes tout en préservant le bon fonctionnement des fonctionnalités. Le projet peut donc être utilisé sous forme d'API[v] : un utilisateur peut incorporer les classes et les fonctions dans son code.

II. Méthodologie

1. Méthodes utilisées

La méthode Agile[vi] et Data Driven Architecture[vii] ont été mise en place durant la réalisation du projet.

En ce qui concerne la méthode agile, des dates limites et des rendez-vous réguliers ont été fixés avec notre tuteur.

Quant à la méthode Data Driven Architecture, elle permet à l'application de respecter le principe de modularité. L'application fonctionne tant que les jeux et algorithmes respectent leurs structures respectives. Cela permet de pouvoir charger n'importe quel jeu avec n'importe quel algorithme créé.

2. Répartition des tâches

Le projet a été divisé en plusieurs parties pour répartir chacune d'elle à chaque membre du groupe :

- Aurélien : structure en API et les algorithmes
- Léa : structure en API et le jeu Quoridor
- Kevin : le jeu Quoridor et l'algorithme d'optimisation Minimax
- Nicolas : le jeu de Go et le Tictactoe

Vous pouvez visualiser l'avancée progressive des tâches du projet par le diagramme de Gantt en annexe.

3. Rétrospectives des rendez-vous avec le tuteur

L'un des principes de la méthode Agile repose sur l'écoute des demandes du client, ici notre tuteur de projet. Voici les rétrospectives des rendez-vous avec notre tuteur :

Date	Sujet du rendez-vous	Décision prise
17/10	Premières discussions sur le projet	Écrire le projet en python et faire un projet porté sur les jeux et l'intelligence artificielle
24/10	Introduction du jeu du Quoridor par le tuteur	Mettre en place le jeu dans le projet
07/11	Explication de l'algorithme Minimax	Apprentissage et documentation sur l'algorithme
14/11	Approbation du cahier des charges	Approbation OK
21/11	Les premiers jeux en python	Faire un jeu de Nim en python
28/11	L'algorithme Minimax	Commencer à écrire l'algorithme Minimax et faire un jeu de Tictactoe
19/12	Le jeu Quoridor	Répartir les tâches du groupe
27/12	La vision globale du projet et les différentes tâches à effectuer	Répartition des tâches du groupe pour les vacances
23/01	Compte rendu après vacances : le Quoridor, Minimax, les algorithmes d'apprentissage et le jeu de Go	Reprendre un nouveau rendez-vous plus tard pour le rapport et le Quoridor
04/02	Le rapport et les algorithmes d'apprentissage	Continuer à écrire le rapport, avancer sur le Quoridor et préparer un dernier rendez-vous avant les vacances
07/02	Avancée du jeu Quoridor	Répartition des tâches du groupe pour les vacances

4. Tableau PPN

Ce tableau sert à visualiser l'ensemble des connaissances acquises à l'IUT qui ont pu être utiles à notre projet pour chacun des membres.

Notions/Membres	Aurélien	Léa	Kevin	Nicolas
Modélisations mathématiques	x			
Conception et programmation objet avancées	x	x	x	x
Algorithmique avancée	x	x	x	x
Analyse et méthodes numériques	x			
Graphes et langages	x	x	x	x
Bases de la conception orientée objet	x	x	x	x
Bases de la programmation orientée objet	x	x	x	x
Anglais et Informatique	x			
Méthodologie de la production d'applications	x	x	x	x
Algèbre linéaire	x	x	x	x
Mathématiques discrètes	x	x	x	x
Fondamentaux de la communication	x	x	x	x
Gestion de projet informatique	x	x	x	x

III. Projet

Le projet a été construit autour de la donnée mais aussi des algorithmes d'apprentissage. Il a donc été séparé en deux groupes : les agents[viii] et leur environnement[ix] (les jeux). C'est un principe récurrent dans tout type de projet en intelligence artificielle.

Un agent est un résolveur de problèmes pouvant effectuer certaines actions. L'environnement est la « résidence » d'un agent. Un environnement fournit des réponses à un agent en fonction des actions qu'il effectue.

1. Choix du langage et bibliothèques utilisées

Le python est le langage utilisé pour coder l'application. Il permet à la fois la programmation objet et la programmation impérative.

La librairie Mathplotlib a été utilisée. Elle permet la création de diagrammes servant à représenter graphiquement les données.

La librairie Numpy utilisée est destinée à manipuler des matrices ou des tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

Enfin, la librairie turtle (intégrée par défaut à python) est utilisée pour faire des dessins graphiques.

2. Les agents : les algorithmes utilisés

2.1. Algorithme d'optimisation : Minimax

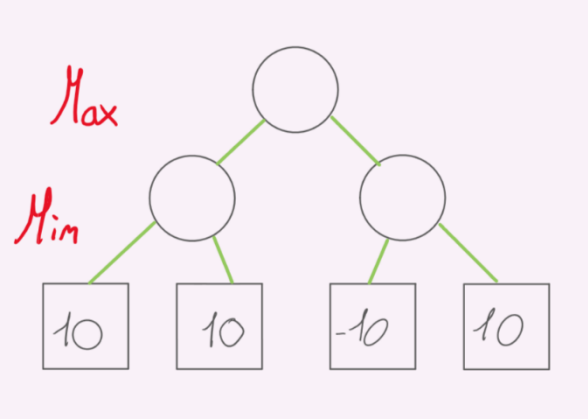
2.1.1. Minimax

L'algorithme Minimax est un algorithme d'optimisation[x] destiné à trouver le meilleur choix possible dans un environnement compétitif entre deux agents.

L'algorithme créé un arbre de toutes les possibilités d'états[xi] du jeu selon un état donné et selon les coups possibles. Les nœuds de l'arbre représentent donc un état du jeu, les nœuds finaux ou feuilles de l'arbre, des états finaux du jeu (victoire, défaite ou match nul) et entre chaque nœud ou branche de l'arbre se trouvent les coups joués pour arriver à un état.

L'algorithme se déroule donc en deux phases, d'une part un parcours descendant jusqu'à un état final (ou feuille de l'arbre) et d'autre part un parcours ascendant des valeurs finales jusqu'en haut de l'arbre (les valeurs finales dans un jeu sont par exemple 10, -10 et 0 pour respectivement une victoire, une défaite ou un match nul).

La particularité de Minimax par rapport à d'autres algorithmes de création d'arbres est l'asymétrie de chaque niveau de l'arbre.



Exemple d'arbre Minimax

En effet, à chaque niveau de l'arbre on change la manière de remonter les valeurs d'en dessous. Soit en voulant remonter la valeur minimale (sur l'image : on remonte ici 10 au nœud de gauche et -10 à droite) soit en remontant la valeur maximale (comme montré sur l'image, ici on prend 10 sur le nœud de gauche et -10 sur celui de droite, donc au sommet on remonte 10).

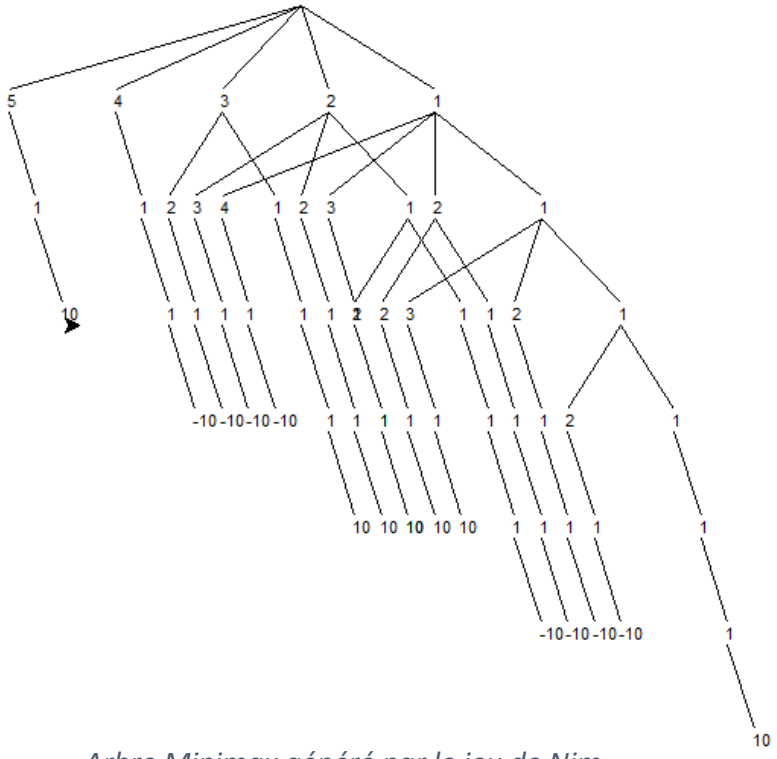
Des appels de fonctions successifs sont réalisés : la fonction max cherche le maximum des valeurs de nœuds d'en dessous et pour chaque nœud la fonction min est appelée, cherchant le minimum des valeurs des nœuds d'en dessous. Le processus est réitéré en utilisant la fonction max, et ainsi de suite.

L'ordre des appels a une importance, une succession d'appels de fonctions « min, max » ou bien « max, min » changera le comportement de l'agent. Dans le cas des jeux en tour par tour cela change le comportement de la manière suivante : l'agent peut vouloir gagner ou bien vouloir laisser gagner l'adversaire.

Ici, les appels de fonctions sont organisés de cette façon : max puis min si l'agent Minimax joue en deuxième et min puis max s'il joue en premier. Cela s'explique par le fait que l'arbre doit toujours rester le même. Le joueur en haut de l'arbre commence par min et celui d'après par max. Le fait de choisir un arbre commençant par max ou min n'a pas d'importance mais l'alternance des deux en a. Cela permet d'avoir une dualité. On peut le représenter comme un joueur voulant jouer le meilleur coup (max) alors que l'autre veut choisir le pire coup pour que l'adversaire perde (min).

Un algorithme tel que Minimax est donc performant car il prend le meilleur choix en créant l'ensemble des possibilités du jeu. Mais il n'est pas rapide : créer tous les nœuds du jeu prend du temps, d'autant plus que Minimax n'est pas optimisé. Autrement dit, il y a plus de nœuds finaux dans l'arbre que d'états finaux dans le jeu. Ce problème vient du fait qu'une succession de coups dans un jeu peut avoir le même état final qu'une autre succession de coups. Des algorithmes tels que Monte-Carlo ou élagage Alpha-Beta contournent ce problème de différentes manières (en coupant l'arbre entier ou seulement certaines branches).

2.1.2. Le programme de visualisation d'arbre Minimax



Arbre Minimax généré par le jeu de Nim

Le programme de visualisation a été écrit avec la librairie turtle pour l'une de nos premières versions de Minimax où l'on a séparé la génération de l'arbre et la remontée des feuilles. Cela permet de facilement créer l'arbre pour ensuite le visualiser.

Cette partie du projet est à part, en dehors de l'API. Mais il aurait été intéressant d'inclure le programme de visualisation dans le Minimax de l'API pour visualiser l'arbre de tous les jeux.

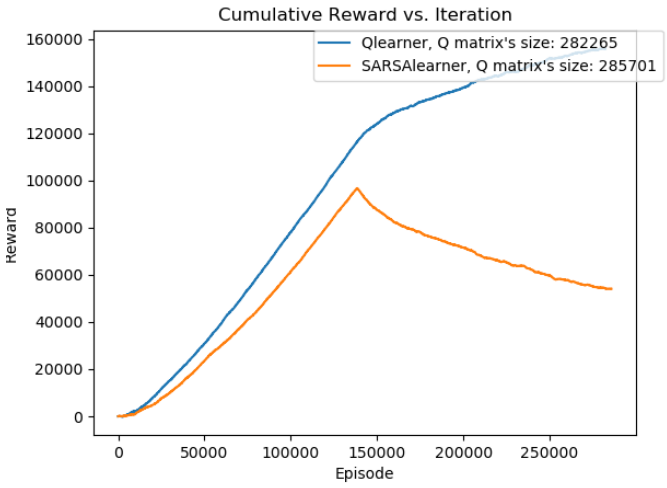
2.2. Algorithmes d'apprentissage : Q-learning, SARSA

Assez proche de l'algorithme Minimax, les algorithmes d'apprentissage ne créent pas tous l'arbre car s'entrainant sur chaque état qu'ils parcourent. Par rapport à Minimax, ces algorithmes sont au contraire très versatiles, pouvant servir dans de nombreuses situations. Il n'est pas nécessaire de connaitre l'environnement pour les faire fonctionner, mais il faut néanmoins un système de récompense[xii] pour orienter notre algorithme dans ses actions.

Les algorithmes d'apprentissages sont caractérisés comme des matrices représentées par les états de leur environnement (par exemple le plateau du jeu) et les actions possibles (par exemple jouer un coup dans un jeu).

Il y a néanmoins quelques particularités dans notre cas dans le sens où se sont des jeux en tour par tour :

- Il est nécessaire d'avoir un agent adverse face à l'algorithme d'apprentissage, le meilleur choix étant un algorithme de choix aléatoire afin de parcourir le plus d'états différents. Il est intéressant d'expérimenter avec l'agent adverse, c'est-à-dire de faire des entrainements entre deux algorithmes d'apprentissages ou bien faire des entrainements progressifs.



Entrainement progressif de 30000 parties jouées

L'exemple ci-dessus illustre un entrainement progressif, commençant par un adversaire aléatoire pour les deux algorithmes puis en les entrainant l'un contre l'autre. Comme ils se partagent les récompenses, les courbes sont symétriques. Au final, un entrainement avec un l'algorithme aléatoire donne toujours les meilleurs résultats. Entrainer deux algorithmes d'apprentissage l'un contre l'autre les rendent excellents pour des choix précis. Mais du fait qu'ils ne vont pas expérimenter beaucoup d'états, l'entrainement n'est pas suffisant pour jouer face à un humain.

- L'environnement fait en sorte que l'algorithme d'apprentissage récupère les récompenses à la fin. Mais on pourrait aussi bien récupérer les récompenses au cours d'un entrainement (par exemple en donnant une récompense s'il a fait un bon coup au cours du jeu).

Comment cette matrice est-elle formée ?
Voici la formule sur laquelle se base chaque mise à jour de valeur de la matrice Q :

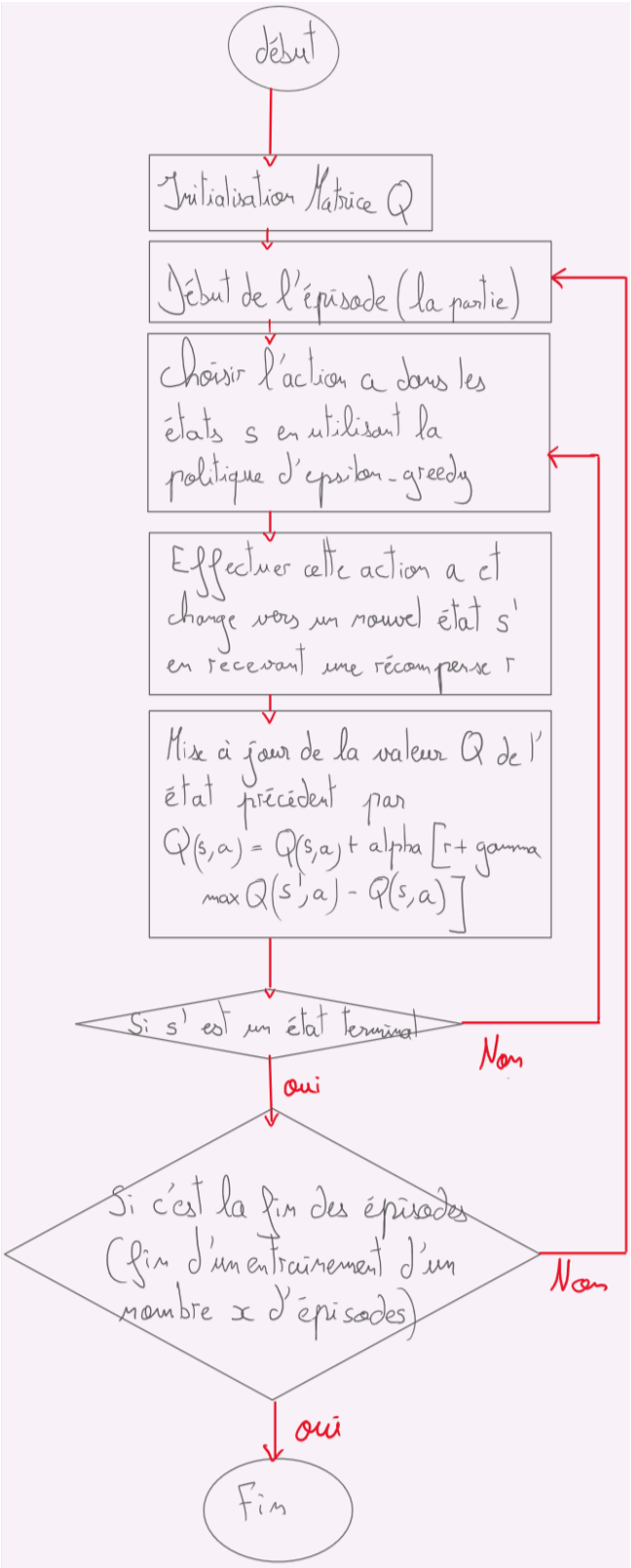
$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)}^{\text{new value (temporal difference target)}} \overbrace{- \underbrace{Q(s_t, a_t)}_{\text{old value}}}^{\text{temporal difference}}$$

La formule Q

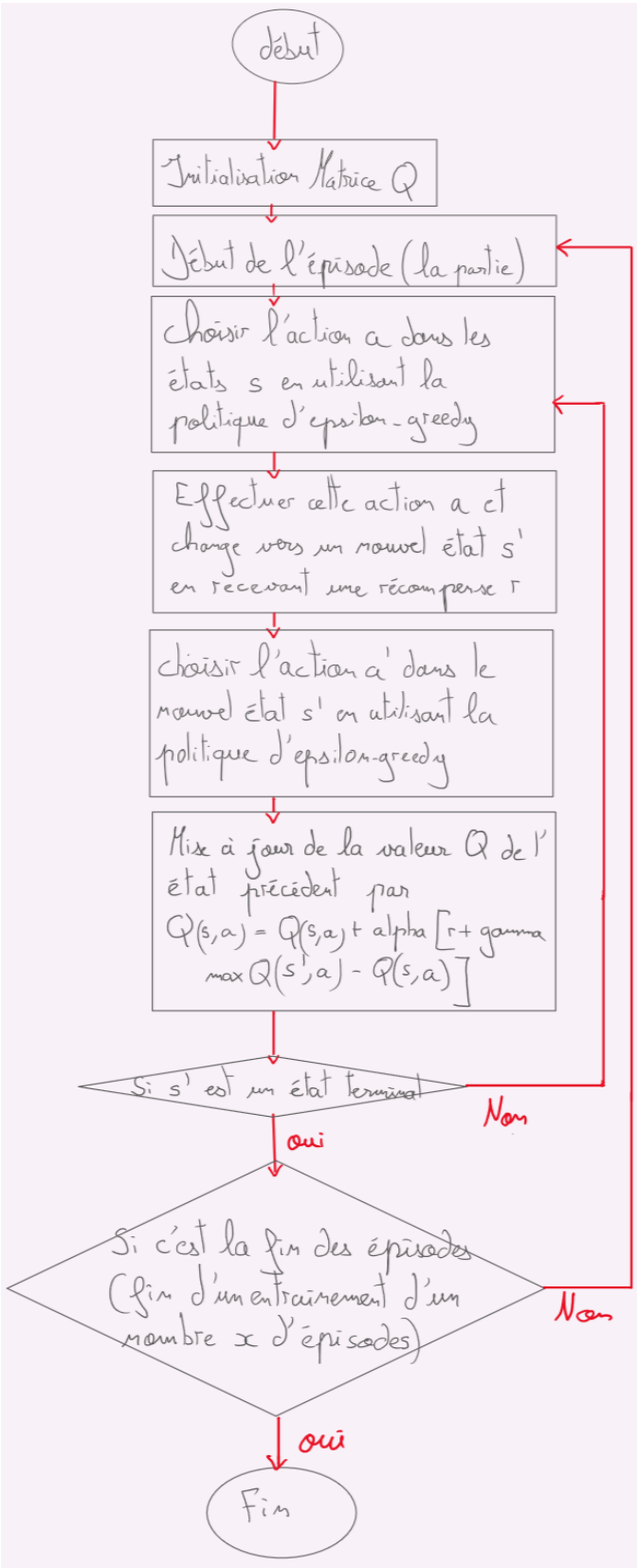
Avec :

- $Q^{new}(s_t, a_t)$: la nouvelle valeur qu'on met à jour dans la matrice Q aux coordonnées s (état, state) à (action) au temps t.
- $Q(s_t, a_t)$: l'ancienne valeur dans la matrice Q aux coordonnées s (état, state) à (action) au temps t.
- α : la vitesse d'apprentissage, qui est une constante comprise entre 0 et 1 et qui est définie au début (par exemple 0.5). Plus elle est élevée, plus on donne de l'importance à ce qu'on apprend. α peut aussi être variable (comme par exemple être importante au début et peu élevée au fil du temps).
- r_t : la récompense qui peut être soit contenue dans une matrice, soit donnée dynamiquement par l'environnement à chaque nouvel état rencontré où l'on effectue une action.
- γ : le facteur d'actualisation qui est une constante comprise entre 0 et 1 et qui détermine l'importance des récompenses futures. Ici, il est intéressant de mettre gamma à 0.9 pour faire intervenir les récompenses les plus lointaines.
- $\max_a Q(s_{t+1}, a)$: la meilleure action en prenant la valeur maximale d'une ligne (la valeur maximale d'un état en fonction des actions).

Les deux algorithmes d'apprentissage qui ont été mis en place pour notre projet sont le Q-learning et le SARSA.

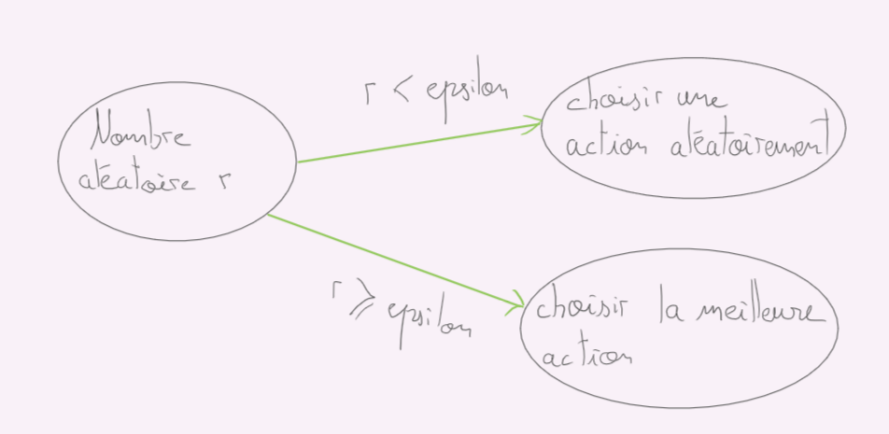


Algorigramme de Q-learning du projet



Algorigramme de SARSA du projet

Une politique[xiii] d'epsilon-greedy est utilisée pour ces deux algorithmes. Cela sert à ce qu'au début (lorsque la matrice vient d'être initialisée), l'algorithme puisse choisir des actions aléatoirement. ε va être décrémenté au fur et à mesure pour que l'algorithme puisse se baser sur la matrice plutôt qu'aléatoirement.



Méthode de choix d'action avec une politique d'epsilon-greedy

Comme les algorigrammes au-dessus le montrent, une politique d'epsilon-greedy est utilisée dans le Q-learning et, tout en actualisant la valeur du Q, l'action maximale est retenue. Il en va de même dans le cadre du SARSA : tout en mettant à jour la valeur du Q, des mesures sont prises au travers de l'utilisation de la politique d'epsilon-greedy.

L'algorithme SARSA est appelé algorithme « online » (ou on-policy) alors que le Q-learning est appelé algorithme « offline » (ou off-policy). Cela signifie qu'en général, l'algorithme SARSA est utilisé lorsqu'il intervient pendant son entrainement alors que le Q-learning est plus utile une phase d'entrainement est d'abord réalisé avant la phase d'utilisation.

3. L’environnement : les jeux

3.1. Le jeu de Nim, Tictactoe : la structure des jeux

En résumé, dans le jeu de Nim, on dispose des paquets d'allumettes sur une table. Chaque joueur, à tour de rôle, prend le nombre d'allumettes qu'il veut (au moins une) dans un des paquets. Le gagnant est celui qui prend la dernière allumette.

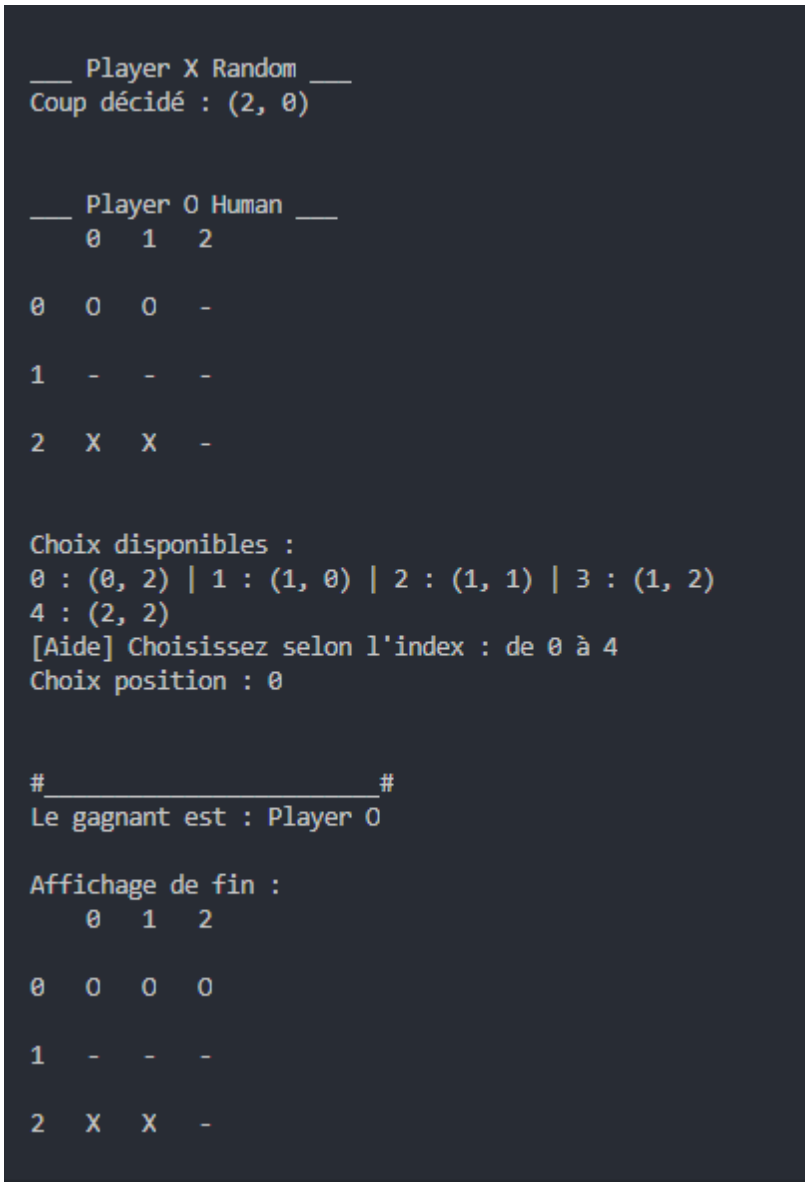
```
___ Player1 Qlearner ___
Les actions possibles :
 [1, 2, 3]
Les valeurs de la matrice Q :
 [-0.59819906  3.81018556  6.67525074]
Les actions possibles après update :
 [1, 2, 3]


___ Player2 Human ___
Allumettes restantes : 7
Choix disponibles :
0 : 1 | 1 : 2 | 2 : 3
[Aide] Choisissez selon l'index : de 0 à 2
Tirage allumettes : 2


___ Player1 Qlearner ___
Les actions possibles :
 [1, 2, 3]
Les valeurs de la matrice Q :
 [ 8.18178878 -4.87240548 -3.14554826]
Les actions possibles après update :
 [1, 2, 3]
```

Affichage des actions choisies par le joueur Qlearner et le joueur Human au cours d’une partie

Le Tictactoe, aussi appelé « morpion », est un jeu se pratiquant à deux joueurs au tour par tour dont le but est de créer en premier un alignement. Ils doivent remplir chacun leur tour une case de la grille avec le symbole qui leur est attribué : O ou X.



Affichage des deux derniers coups choisis par le joueur Random (X) et le joueur Human (O)

Ce sont les premiers jeux qui ont été mis en place pour le projet et qui ont déterminé comment devaient être construits les autres jeux, et plus particulièrement, le Tictactoe car plus complet et pouvant ainsi servir de meilleur modèle pour d'autres jeux.

Les informations des joueurs du côté des agents (précisément dans la partie *TurnBased*) et celles de la partie jeux sont autonomes afin de permettre au créateur d'un jeu de gérer ces informations comme il le souhaite. Par exemple, dans le Tictactoe on caractérise les pions des joueurs dans *player_informations*. Cela permet de faire un inventaire des joueurs : on pourrait par exemple y stocker leur score, leurs cartes s'il s'agissait d'autres jeux. La liste *players* permet de faire la liaison entre les agents et les joueurs (*agent_informations* dans le dossier utilities).

Dans un jeu, il peut aussi y avoir des phases et des changements d'actions à certains moments du jeu. Même si le Tictactoe n'en n'a pas, il a été conçu de façon à préparer cette structure à d'autres jeux.

De ce fait, la méthode *play_move()* sert à changer l'environnement (le plateau) selon les interactions des joueurs. Elle peut avoir plusieurs méthodes internes selon le nombre de phases. Le changement de phase est géré par un *switch* sur l'attribut *currentphase*, celui-ci changeant si tous les joueurs sont passés à la phase suivante du jeu.

La méthode *valid_move()* permet de donner toutes les actions que les joueurs peuvent faire en renvoyant une liste de tous les coups possibles. Comme *play_move()* cette méthode peut renvoyer des listes d'actions différentes selon la phase du jeu.

La méthode *winner()* sert à la fois à récupérer le gagnant du jeu mais aussi à indiquer si la partie est terminée, auquel cas elle renvoie None.

Enfin, la méthode *print_game()* permet de récupérer un string caractérisant l'environnement pour les algorithmes d'apprentissage mais aussi permettant de visualiser le plateau lorsque l'agent est un utilisateur humain.

3.2. Quoridor

Ce jeu peut se jouer à 2 ou à 4. Il se compose d'un plateau de dimensions 9x9 ainsi que de 4 pions. Si la partie rassemble 2 joueurs, alors chaque joueur dispose de 10 barrières. Sinon, chaque joueur dispose de 5 barrières. Le but du jeu est d'atteindre en premier la ligne opposée à sa ligne de départ.

Au début, chacun des joueurs pose son pion au milieu de sa ligne de départ. Puis à tour de rôle, chacun déplace son pion d'une case vers la droite, la gauche, le haut ou le bas, ou pose une barrière afin de ralentir l'adversaire. Cependant, il doit toujours lui laisser un chemin pour lui permettre d'atteindre sa ligne d'arrivée. Les pions doivent contourner les barrières, ce qui crée un labyrinthe dont il faut sortir.

Les informations de chaque joueur sont représentées par la classe Joueur. Elle contient la position du joueur, le nombre de murs restant à placer, sa ligne d'arrivée et son pion. La classe Plateau est l'environnement dans lequel les joueurs évoluent.

Le plateau de jeu est représenté par une matrice. Celle-ci est initialisée lors de la création du plateau. Les joueurs se distinguent dans la matrice par des lettres ('A' ou 'B'). Les cases représentant leurs lignes d'arrivées sont respectivement renseignées par des '7' et par des '5'. Les cases sur lesquelles ils peuvent se déplacer correspondent aux '0'. Les 'm' qui composent la matrice sont en fait les cases dédiées à la pose des murs. Lorsqu'ils sont posés, ces murs sont signifiés par des '1'.

```
#____________________#
Le gagnant est : Joueur A

Affichage de fin :

      0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
     ---------------------------------------------------------
 0 |  5  5  5  5  5  5  5  5  5  5  5   5   5   5   5   5   5
 1 |  0  m  0  1  0  m  0  m  0  m  0   m   0   m   0   m   0
 2 |  m  m  m  1  m  m  m  m  m  m  m   m   m   m   m   m   m
 3 |  0  m  0  m  0  1  0  m  0  m  0   1   0   m   0   m   0
 4 |  m  m  m  m  m  1  m  m  m  m  m   1   m   m   m   m   m
 5 |  0  m  0  m  0  m  0  m  0  m  0   m   0   m   0   m   0
 6 |  m  m  m  m  m  m  m  m  m  m  m   m   m   m   m   m   m
 7 |  0  m  0  m  0  m  0  m  0  m  0   m   0   m   0   m   0
 8 |  m  m  m  1  m  m  m  m  m  m  m   m   m   m   m   m   m
 9 |  0  m  0  1  0  m  0  m  0  m  0   m   0   m   0   1   0
10 |  m  m  m  m  m  m  m  m  m  m  m   m   m   m   m   1   m
11 |  0  m  0  m  0  m  0  m  0  m  0   m   0   m   0   m   0
12 |  m  m  m  m  m  m  m  m  1  1  m   m   m   m   m   1   1
13 |  0  m  0  m  0  m  0  m  0  m  0   m   0   m   0   m   0
14 |  m  m  m  m  m  m  m  m  m  m  m   m   m   m   m   m   m
15 |  0  m  0  m  0  m  0  m  0  m  0   m   0   m   0   m   0
16 |  m  m  m  m  m  m  m  m  m  m  m   m   m   m   m   1   1
17 |  0  m  0  m  0  m  0  m  0  m  B   m   0   m   0   m   0
18 |  7  7  7  7  7  7  A  7  7  7  7   7   7   7   7   7   7
Games played: 0
```

Affichage du plateau de jeu après une victoire du joueur A

Contrairement aux jeux précédemment abordés, le Quoridor possède différentes phases de jeu : une phase de déplacement et une phase pour la pose de mur. L'action menée par le joueur dépend de la phase dans laquelle il se trouve. La méthode *play_move()* permet la réalisation de cette action. Elle comporte donc deux méthodes, à savoir, *phase_deplacement()* et *phase_pose_murs()*. L'appel de ces méthodes s'effectue au moyen de l'attribut *currentphase* de la classe Plateau. La valeur de cet attribut est elle-même déterminée par l'attribut *numerophase* de cette même classe. Ici, le déroulement choisi est le suivant : les joueurs déplacent leurs pions tour à tout durant 3 phases avant de poser chacun un mur pendant la 4ème phase, et ainsi de suite, jusqu'à ce qu'il ne leur reste plus aucun mur à poser (il s'agit de la 41ème phase), auquel cas ils se déplacent jusqu'à la fin de la partie.

La fin de la partie est notamment précisée par la méthode *winner()*. Celle-ci vérifie si l'un des joueurs est arrivé à sa ligne d'arrivée ou bien s'il a été bloqué par son adversaire (appel à la méthode *blocage()* [1]). Elle renvoie ce joueur (le gagnant) si l'une de ces deux conditions est remplie. Le cas échéant, la méthode renvoie *None* pour signaler que la partie est toujours en cours. Lorsqu'une fin de jeu est détectée et qu'un gagnant est désigné, une nouvelle partie est lancée pour entrainer à nouveau nos agents.

[1] Afin de détecter les situations qui peuvent bloquer un joueur entre des murs, la méthode *blocage()* représente le plateau sous forme d'une matrice d'adjacence, dont les sommets correspondent aux cellules destinées à accueillir les pions (y compris les lignes d'arrivées qui sont regroupées en un sommet chacune). La matrice d'adjacence est contenue dans une liste de listes. On parcourt alors tous ses sommets en utilisant l'algorithme de parcours en profondeur (récursif) à partir de la position du joueur dans le but de trouver la sortie (la ligne d'arrivée qui lui est associée).

La classe Plateau dispose également d'une méthode *valid_moves()* qui retourne tous les coups autorisés. Ceux-ci sont définis en fonction de l'état du plateau, du joueur, de sa position et de la phase dans laquelle il évolue.

- Durant la phase de déplacement, pour chacune des directions possibles (rappel: haut, bas, gauche, droite), on vérifie si le joueur peut se déplacer (appel à la méthode *check_moves()* [1]). Si c'est le cas, alors on ajoute cette direction à une liste représentant les coups autorisés.

[1] La méthode *check_moves()* vérifie que la direction choisie par le joueur le déplace sur une case dédiée à cet effet, qu'un mur ne le sépare pas de sa destination, que le joueur adverse ne s'y trouve pas déjà (dans ce cas il doit lui passer par-dessus selon les mêmes conditions) et bien sûr que son choix ne le fasse pas dépasser les extrémités du plateau.

- Durant la phase de placement de mur, pour chacune des cases destinées à recevoir un mur (liste obtenue grâce à la méthode *numMurs()* [2]), on vérifie si le joueur peut y poser un mur (appel à la méthode *check_laying_walls()* [3]). Si c'est le cas, alors on ajoute le numéro de cette case à une liste représentant les coups autorisés.

[2] La méthode *numMurs(*) attribue à chaque cellule dédiée à la pose d'un mur un numéro ainsi que ses coordonnées.

[3] La méthode *check_laying_walls()* vérifie que deux cases adjacentes (verticalement ou horizontalement) dédiées à la pose d'un mur sont disponibles.

La méthode *play_move()* présentée ci-dessus utilise la méthode *valid_moves()* afin de déterminer si le choix du joueur fait partie des coups autorisés et de permettre (si réponse positive) le déplacement du joueur (appel à la méthode *seDeplacer()*) ou la pose d'un mur (appel à la méthode *poserMur()*) en modifiant l'état du plateau de jeu.

Le jeu du Quoridor utilise lui aussi la méthode *print_game()*, pour les mêmes raisons que précédemment. Elle renvoie un string comportant le plateau de jeu ainsi que les coordonnées de chacune des cases.

3.3. Jeu de Go

Le plateau de jeu traditionnel se compose d'un goban sur le lequel est tracé un quadrillage de 19x19 lignes, soit 361 intersections, et de pierres qui sont soit noires, soit blanches.

Le go se joue à deux. Celui qui commence joue avec les pierres noires et l'autre avec les blanches. A tour de rôle, les joueurs posent une pierre de leur couleur sur une intersection inoccupée du goban ou bien ils passent.

Les aspects techniques importants :

- Le territoire est un ensemble d'une ou plusieurs intersections inoccupées voisines de proche en proche, délimitées par des pierres de même couleur.
- Une chaîne est un ensemble d'une ou plusieurs pierres de même couleur voisines de proche en proche.
- Les libertés d'une chaîne sont les intersections inoccupées voisines des pierres de cette chaîne.
- Le principe de capture apparaît lorsqu'un joueur supprime la dernière liberté d'une chaîne adverse, il la capture en retirant du goban les pierres de cette chaîne.
- Le principe de répétition se définie lorsqu'un joueur, en posant une pierre, redonne au goban un état identique à l'un de ceux qu'il lui avait déjà donné le tour précédent, ce qui est interdit.
- La partie s'arrête lorsque les deux joueurs passent consécutivement. On compte alors les points. Chaque intersection du territoire d'un joueur lui rapporte un point, ainsi que chacune de ses pierres encore présentes sur le goban.

Dans le jeu développé ici, les règles ne sont pas exactement les mêmes. Le principe de répétition est permis, et, pour la fin de partie, seules les pierres présentent sur le goban rapportent des points au joueur.

Le plateau de jeu correspond à une matrice, les cases disponibles sont représentées par un point, les pierres du joueur jouant les pierres noires par un "A" et le joueur jouant les pierres blanches par un "B".

Le constructeur initialise les joueurs, le statut du jeu et la matrice.

Afin de vérifier à chaque pose d'une pierre si une capture est réalisée, une première méthode est appelée afin de savoir si une pierre ennemie est adjacente horizontalement ou verticalement. Ensuite, c'est la seconde méthode « *verifCapture()* » qui détermine, avec son algorithme, si une capture est réalisée. L'algorithme utilise le principe du fil d'Ariane, en parcourant la chaîne, de pierres, de l'adversaire et en utilisant une pile stockant les chemins encore non explorés de la chaîne. Ainsi, si en parcourant pierre par pierre la chaine de l'adversaire et que l'une des pierres ce celle-ci est adjacente à une case vide, la capture ne sera pas possible. On peut comparer ceci à un labyrinthe : si on trouve une sortie, on ne se fait pas capturer.

Les autres méthodes :

- *afficheJeu()* permet d'afficher l'état du plateau sur la console.
- *verrifChoix()* permet de vérifier si le placement choisi par le joueur, pour sa pierre, est une case disponible sur le plateau.
- *placerPierre()* permet de placer une pierre sur le plateau.
- *findeJeu()* permet de vérifier si les deux joueurs ont passés leur tour successivement.
- *victoire()* permet de déterminer qui remporte la partie, et renvoie le joueur gagnant.
- *tour()* permet au joueur de réaliser l'action qu'il souhaite, placer une pierre ou passer.

4. La modularité de l'application et la mise en place sous forme d'API

L'application a été construire autour des données qui lui sont entrées, c'est-à-dire autour des jeux et des agents qui y jouent.

Tout d'abord, les agents qui doivent apprendre sont initialisés. Tous les coups jouables du jeu leur sont donnés pour initialiser la matrice Q. Quant aux états du jeu (les lignes de la matrice), ils sont ajoutés au fur et à mesure en récupérant le plateau de jeu (ce qui permet de contrer aussi les problèmes d'optimisation de Minimax car la matrice est dynamique selon le nombre d'états que l'agent apprenant a pu rencontrer lors de son apprentissage).

Pour réunir les deux, l'utilisateur doit utiliser la méthode *TurnBased_episodes()* permettant de faire des itérations de la fonction TurnBased qui permet de faire une partie d'un jeu. Durant la partie, les agents sont questionnés tour à tour avec la méthode *choose_move(),* qui permet de récupérer une action parmi les actions possibles à ce moment dans le jeu (en leur donnant en paramètre une liste représentant les coups jouables).

Tous les agents suivent la classe abstraite « agent » pour les catégoriser en tant que tel. Ensuite, les agents d'algorithme d'apprentissage suivent la même classe abstraite pour les catégoriser en tant que « learner ». Cela permet la différenciation des agents et la mise à jour des matrices Q de chaque learner.

L'environnement doit lui respecter certaines contraintes afin de bien fonctionner avec les agents, comme par exemple avoir une liste *players* caractérisant le nombre de joueurs, des méthodes *play_move(), valid_moves(), winner()* et *print_game().*

Pour mieux comprendre les résultats, des diagrammes sont mis en place à chaque fin d'entrainement. L'utilisateur du projet peut changer le nombre d'itérations en entrant les arguments -t suivis du nombre d'itérations voulu. Au travers des arguments, il peut aussi sauvegarder et charger des agents apprenant.

Une personne qui veut utiliser ce projet en tant qu'API utilise simplement la méthode *TurnBased_episodes()* et importe les classes des agents. Il peut alors créer ses propres agents ou jeux (en prenant l'exemple du main.py ou en le modifiant).

IV. Résultat final et attendu

L'ensemble des fonctionnalités nécessaires fixées lors de la création du cahier des charges ont été remplies (voir les spécifications fonctionnelles du cahier des charges).

Il aurait donc été possible d'apporter des fonctionnalités supplémentaires à ce projet selon différentes voies : de nouveaux algorithmes, de nouveaux jeux, des programmes de visualisation.

Il aurait également été intéressant d'améliorer l'algorithme Minimax, qui reste peu efficace excepté sur de petits jeux, en utilisant l'algorithme Monte-Carlo ou Alpha-Beta.

Le jeu de Go aurait pu être finalisé, ce qui aurait permis d'avoir un jeu complexe et long d'apprentissage pour les algorithmes d'apprentissages par renforcement.

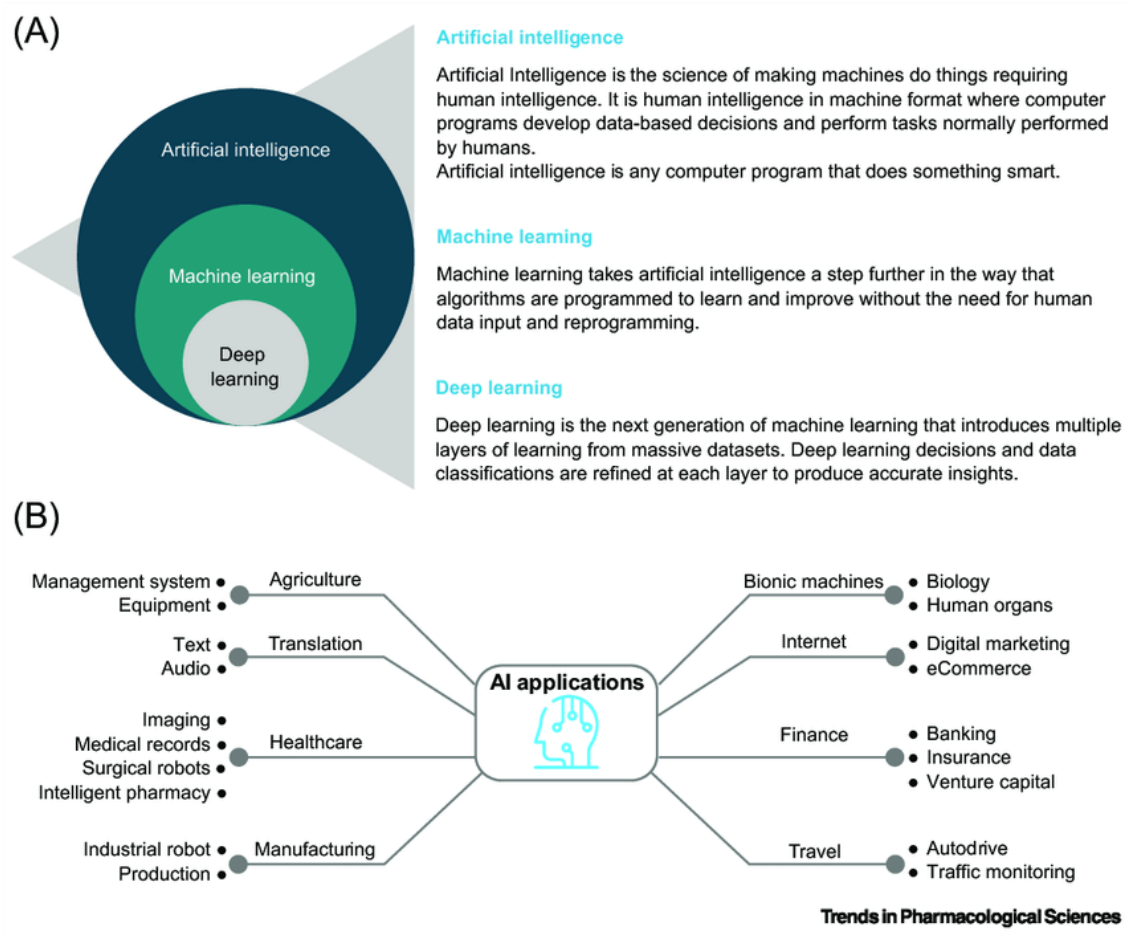
Un autre aspect qui aurait pu être davantage développé aurait été la visualisation des données.

V. Conclusions

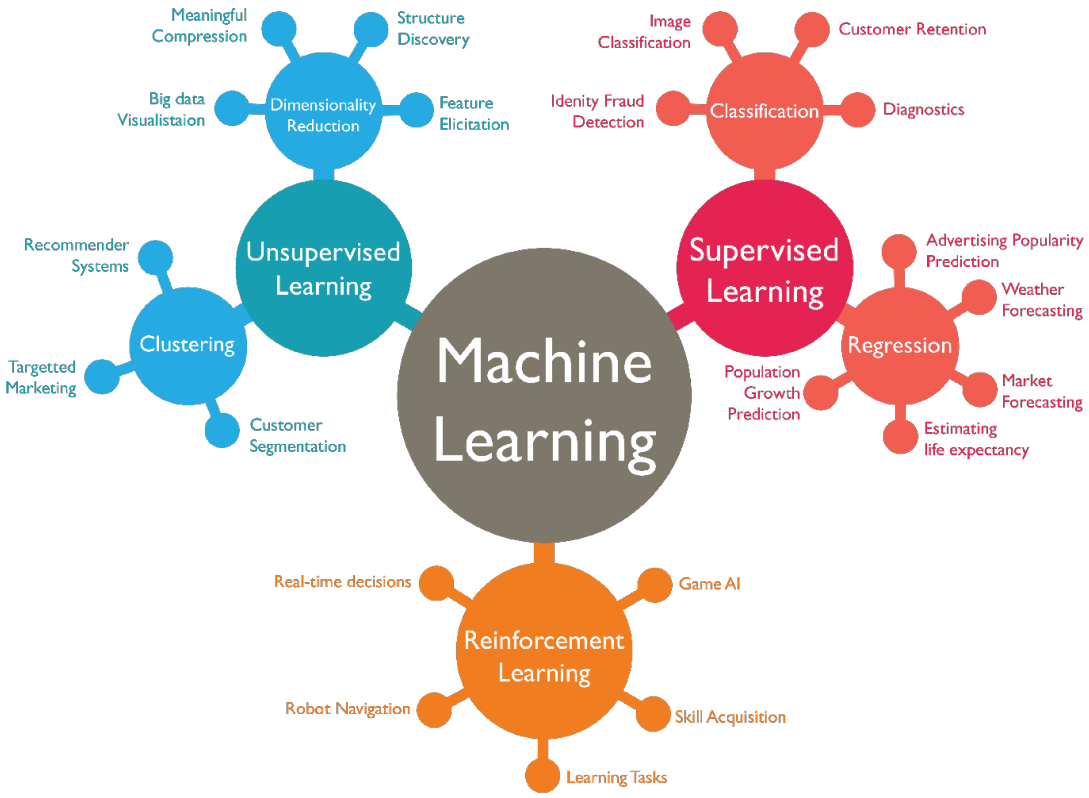
1. Conclusion globale

Le projet nous a permis de mieux nous familiariser avec les algorithmes solveurs de jeux mais aussi plus généralement avec l'intelligence artificielle. Il nous a aussi permis de découvrir un nouveau langage : le python.

Suite au projet en java de l'année dernière sur l'algorithme de recherche de sortie de labyrinthe, nous nous sommes intéressés à l'intelligence artificielle. Grâce à notre projet nous avons mieux compris les différents types d'algorithmes dans ce domaine et leurs cas d'utilisations, plus particulièrement dans le reinforcement learning.



Hiérarchie des différents concepts dans l'intelligence artificielle avec exemples d'applications



Les différents concepts dans le machine learning avec des exemples d'applications

Le python nous a appris une manière différente de coder, sans parenthésage mais avec des tabulations. Ce langage offre plusieurs particularités telles que le retour de plusieurs variables à une méthode, différentes manières de manipuler des structures de données complexes (matrices, listes, dictionnaires).

Et avec l'intégration des librairies et du langage orienté objet, le python facilite la visualisation de la donnée[xiv] et la construction en API.

2. Conclusions personnelles

2.1. Aurélien

Le projet m'a permis de mieux me familiariser avec les algorithmes d'intelligence artificielle et j'ai pris beaucoup de plaisir à découvrir le python qui est parmi mes langages favoris.

Le python par rapport aux langages étudiés en cours, paraissait au premier abord difficile à lire, n'ayant ni de types pour les variables ni de parenthésage. Mais au fur et à mesure j'ai aimé ce langage qui permet de coder rapidement.

Le langage possède aussi quelques particularités tel que le fait de pouvoir retourner plusieurs variables à une fonction ou bien avoir des fonctions à arguments multiples (*args et kwargs) que j'ai pu essayer lors du projet.

Tout comme le php le fait de ne pas avoir de type est vraiment pratique quand on code seul mais par contre, oblige d'avoir une documentation rigoureuse pour tout utilisateur extérieur. Enfin par rapport au java, je trouve que les programmes en python sont plus simple à écrire car le langage ne force pas l'écriture d'un code purement objet ou impératif mais offre les deux possibilités tout en permettant de les faire cohabiter.

2.2. Léa

Ce projet m'a tout d'abord donné l'occasion d'apprendre un nouveau langage : le python. Il est soumis à moins de règles que d'autres langages comme le Java ou le C, ce qui peut sembler être un point positif au premier abord. Mais je me suis vite aperçue qu'il était plus difficile de s'y retrouver si nous ne produisions pas un code lisible et bien documenté. Cependant, certaines structures de données telles que les listes et les dictionnaires se sont avérées très utiles pour la représentation du plateau de jeu et des différents coups jouables.

J'ai également pu mettre à profit ce qui a été vu dans d'autres cours. Nous pensions tout d'abord employer l'algorithme de Pledge pour la résolution du labyrinthe avant de nous rappeler de celui du parcours en profondeur des graphes que nous avions abordé en maths et qui s'est finalement montré moins compliqué et plus efficace. Nous avons aussi travaillé en méthode agile et notamment en pair programming, ce qui nous a permis de mieux avancer sur le jeu du Quoridor. La plus grande difficulté a été l'implémentation de l'API, qui nécessitait je pense plus de communication de notre part.

2.3. Kévin

Le projet m'a permis de redécouvrir sous un autre angle le langage python, grâce à la programmation orientée objet. Le langage python étant un langage non typé avec très peu de syntaxes, cela implique une documentation des classes et méthodes utilisées pour une meilleure compréhension du code pour des personnes autres que l'auteur.

Par ailleurs, j'ai trouvé intriguant le fait que le langage accorde autant d'importance à l'indentation du code, je pense que c'est une manière d'inculquer des habitudes à prendre lors de l'apprentissage du langage.

Durant la réalisation du projet, j'ai pu mettre en application les notions apprises dans les différentes matières : APL, ACDA, Math et la gestion de projet. Particulièrement en gestion de projet, car nous avons mis en place des méthodes agiles tel que le "pair programming". Je pense qu'on que nous aurions pu être plus efficace dans la résolution de problèmes en faisant plus de tests unitaires.

Par ailleurs, j'ai pu découvrir de nouveaux algorithmes tel que Minimax, Q-learning et SARSA, ainsi que leurs utilisations dans le domaine de l'intelligence artificielle.

2.4. Nicolas

Ce projet m'a apporté de nombreuses satisfactions, notamment en me replongeant dans le langage python, que j'avais abordé en spécialité ISN au lycée (même si son utilisation n'était pas orienté objet). En effet de par sa syntaxe, on peut penser que son utilisation est plus complexe alors que, en s'y habituant, c'est le contraire. Python est facile à apprendre comparer au C ou au Java par ses méthodes et le non-typage des variables, possède de nombreuses librairies, est multiplateforme et open source.

J'ai pu apprendre des autres, approfondir mes connaissances en python et découvrir le monde de l'intelligence artificielle de part plusieurs algorithme. J'ai trouvé la cohésion de groupe très positive, personne n'est resté en retrait, la communication au sein de l'équipe était bonne sans être excellente, nous étions d'accord sur les horaires de présence, et l'échange de nos idées était fructueuse.

En plus d'apprendre, j'ai su appliquer des notions apprises en cours à l'Iut comme les notions de gestion de projet informatique, l'algorithmique avancée, la programmation orientée objet, et même appliquer des notions vues en cours de communication tel que la communication professionnelle.

VI. Sources

Tictactoe :

- https://medium.com/@ODSC/how-300-matchboxes-learned-to-play-tic-tac-toe-using-menace-35e0e4c29fc
- https://medium.com/@carsten.friedrich/teaching-a-computer-to-play-tic-tac-toe-88feb838b5e3

Quoridor :

- https://en.wikipedia.org/wiki/Quoridor
- https://www.youtube.com/watch?v=yC39QskYAVE

Reinforcement learning :

- https://www.youtube.com/watch?v=nSxaG_Kjw_w
- https://cdancette.fr/2017/08/18/reinforcement-learning-part1/

Parcours de graphe :

- https://www.python.org/doc/essays/graphs/

Formule Q :

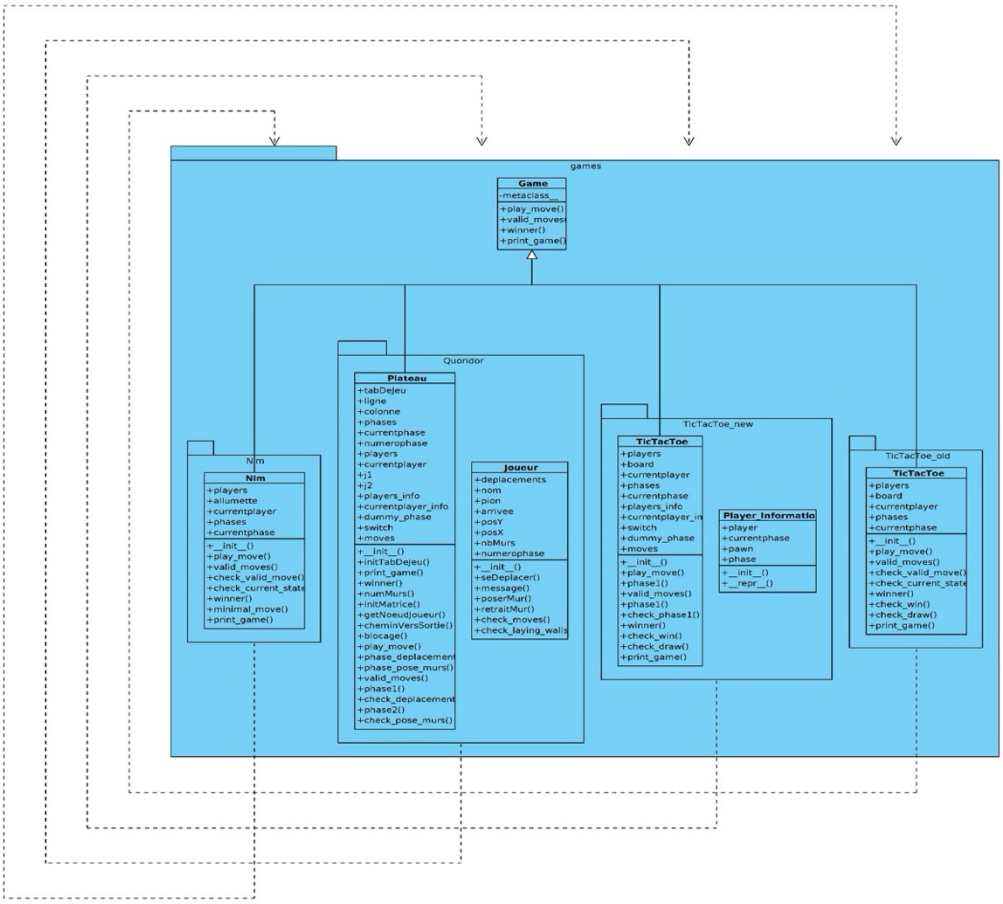
- https://devopedia.org/reinforcement-learning

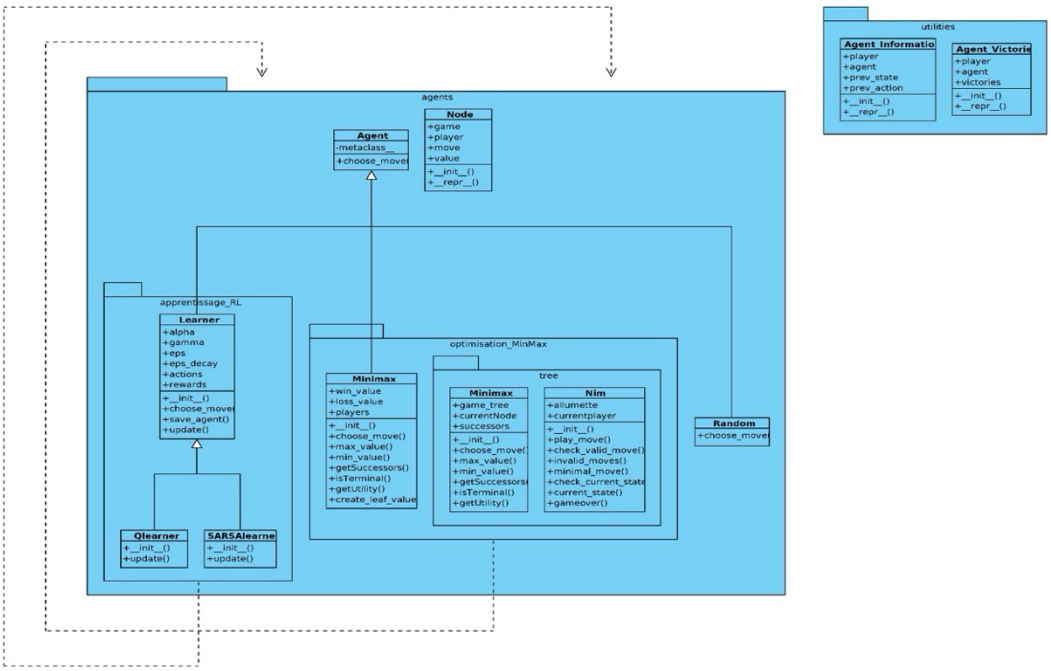
Schémas de conclusions :

- https://www.researchgate.net/publication/328106221_Deep_learning_and_virtual_drug_screening
- https://www.linkedin.com/pulse/business-intelligence-its-relationship-big-data-geekstyle/

VII. Annexes

1. Diagrammes de packages





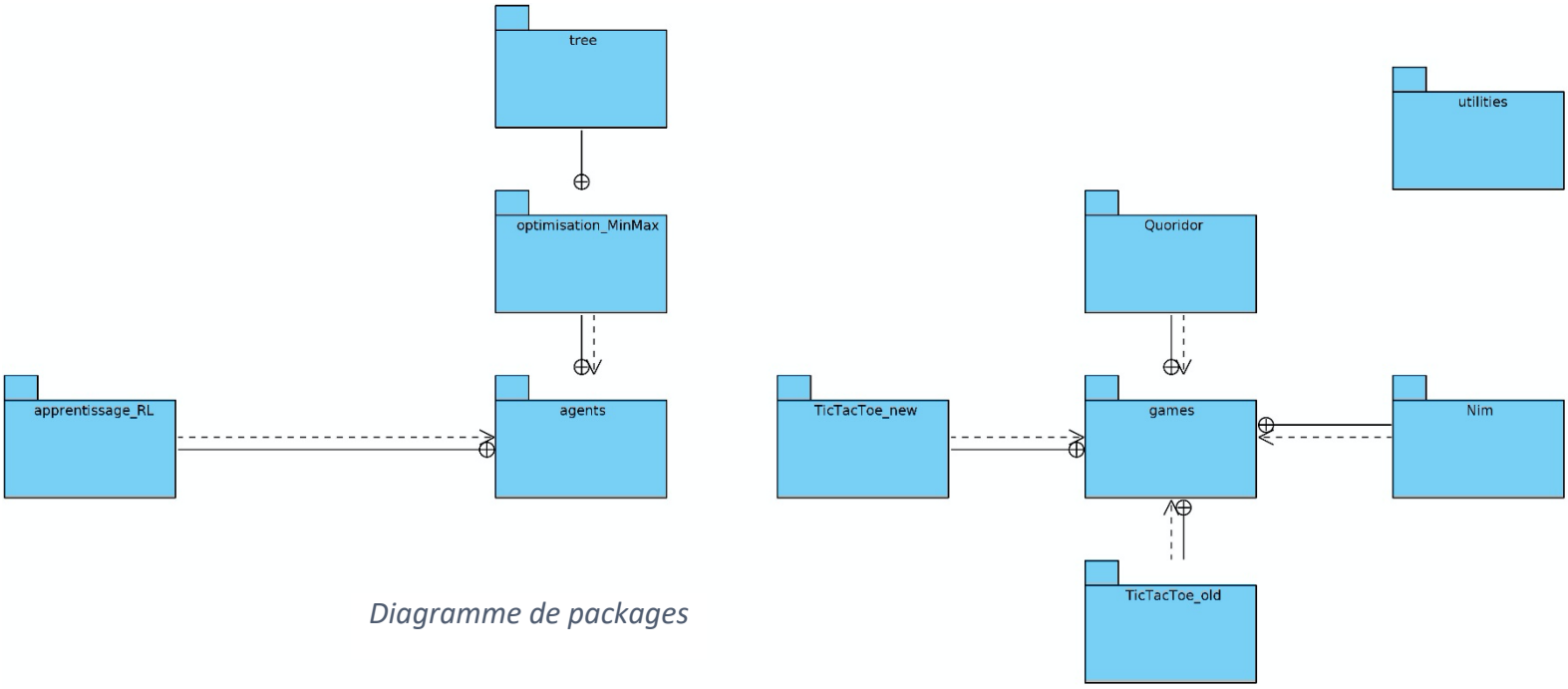


Diagramme de packages

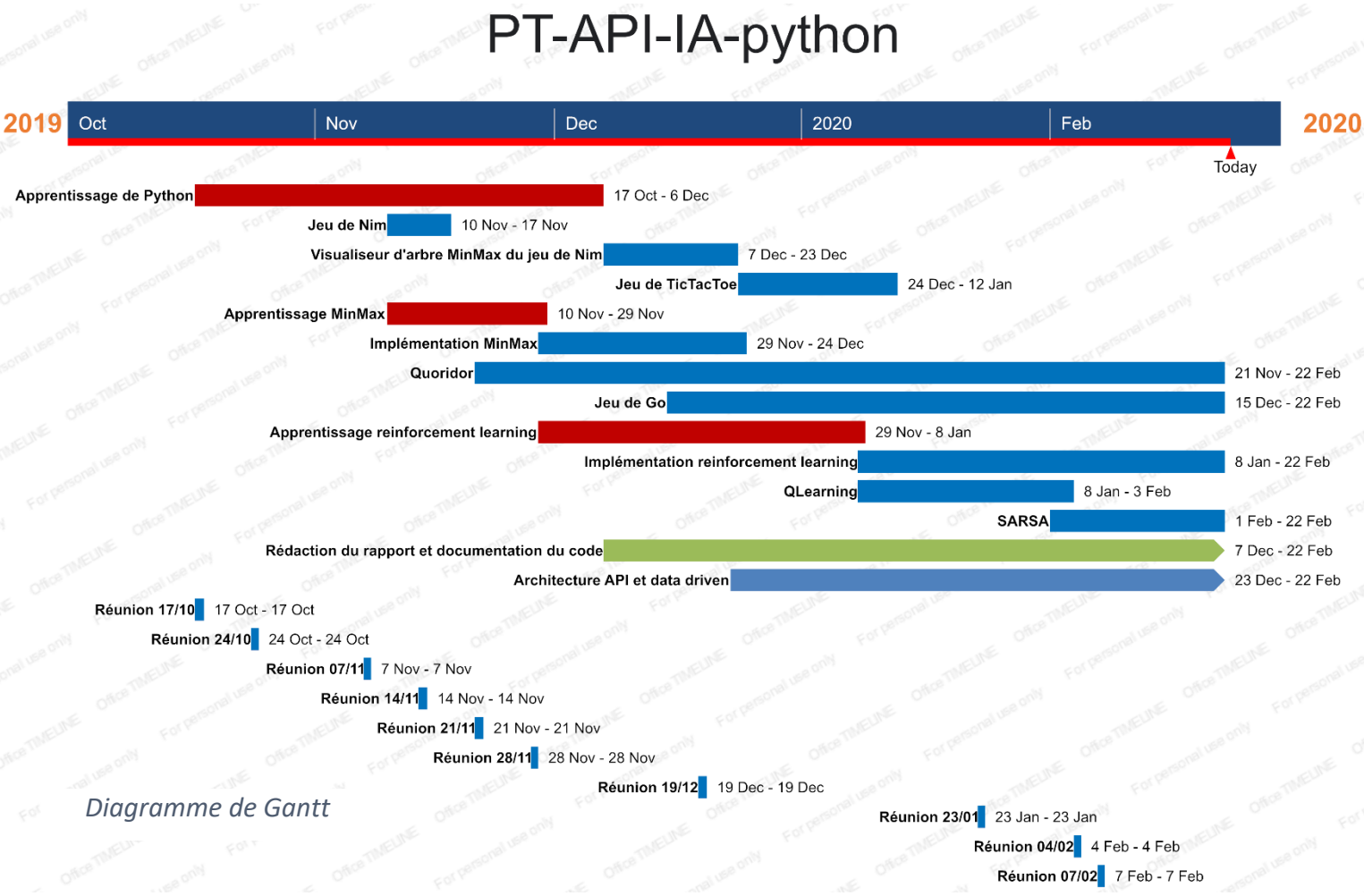


Diagramme de Gantt

VIII. Glossaire

[i] L'algorithme minimax ou minimax est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle consistant à minimiser la perte maximum.

[ii] En intelligence artificielle, plus précisément en apprentissage automatique, le Q-learning est une technique d'apprentissage par renforcement. Cette technique ne nécessite aucun modèle initial de l'environnement.

[iii] Traduit de l'anglais-Etat – action – récompense – état – action est un algorithme d'apprentissage d'une stratégie de processus de décision de Markov, utilisé dans le domaine de l'apprentissage par renforcement de l'apprentissage automatique.

[iv] L'apprentissage par renforcement consiste, pour un agent autonome (robot, etc.), à apprendre les actions, à partir d'expériences, de façon à optimiser une récompense quantitative au cours du temps. L'agent est plongé au sein d'un environnement, et prend ses décisions en fonction de son état courant. En retour, l'environnement procure à l'agent une récompense, qui peut être positive ou négative. L'agent cherche, au travers d'expériences itérées, un comportement décisionnel (appelé stratégie ou politique, et qui est une fonction associant à l'état courant l'action à exécuter) optimal, en ce sens qu'il maximise la somme des récompenses au cours du temps.

[v] API est l'acronyme d'Application Programming Interface, que l'on traduit en français par interface de programmation applicative. L'API peut être résumée à une solution informatique qui permet à des applications de communiquer entre elles et de s'échanger mutuellement des services ou des données. Il s'agit d'un ensemble de fonctions qui facilitent, via un langage de programmation, l'accès aux services d'une application.

[vi] Les méthodes agiles impliquent au maximum le demandeur (client) et permettent une grande réactivité à ses demandes. Elles reposent sur un cycle de développement itératif, incrémental et adaptatif.

[vii] L'architecture dirigée par les données (en anglais Data Driven Architecture, DDA) est un modèle d'architecture informatique qui insiste sur la structuration des données. L'application est dirigée par les données qui lui sont entrées.

[viii] Agent : Le résolveur de problèmes, peut effectuer certaines actions.

[ix] Environnement : Un agent réside ici. Un environnement fournit des réponses à un agent en fonction des actions qu'il effectue.

[x] Les algorithmes d'optimisation sont un groupe d'algorithmes mathématiques pour trouver le meilleur choix possible dans certaines contraintes données.

[xi] État : L'action d'un agent peut le faire entrer dans un état qui est un instantané de l'environnement. (Comme échec et mat (état) sur un échiquier (environnement))

[xii] Récompense : Lorsqu'un agent effectue une action dans un environnement, il y a une récompense associée ; les récompenses peuvent être positives, négatives (punition) ou nulles.

[xiii] Politique : Définit le comportement d'un agent, peut répondre à des questions comme : quelle action doit être effectuée dans cet état ?

[xiv] La visualisation des données (ou data visualisation en anglais) allie des fonctionnalités simples et esthétisme, offrant un gain de temps conséquent dans la recherche et l'analyse des données. C'est aussi un outil de communication.