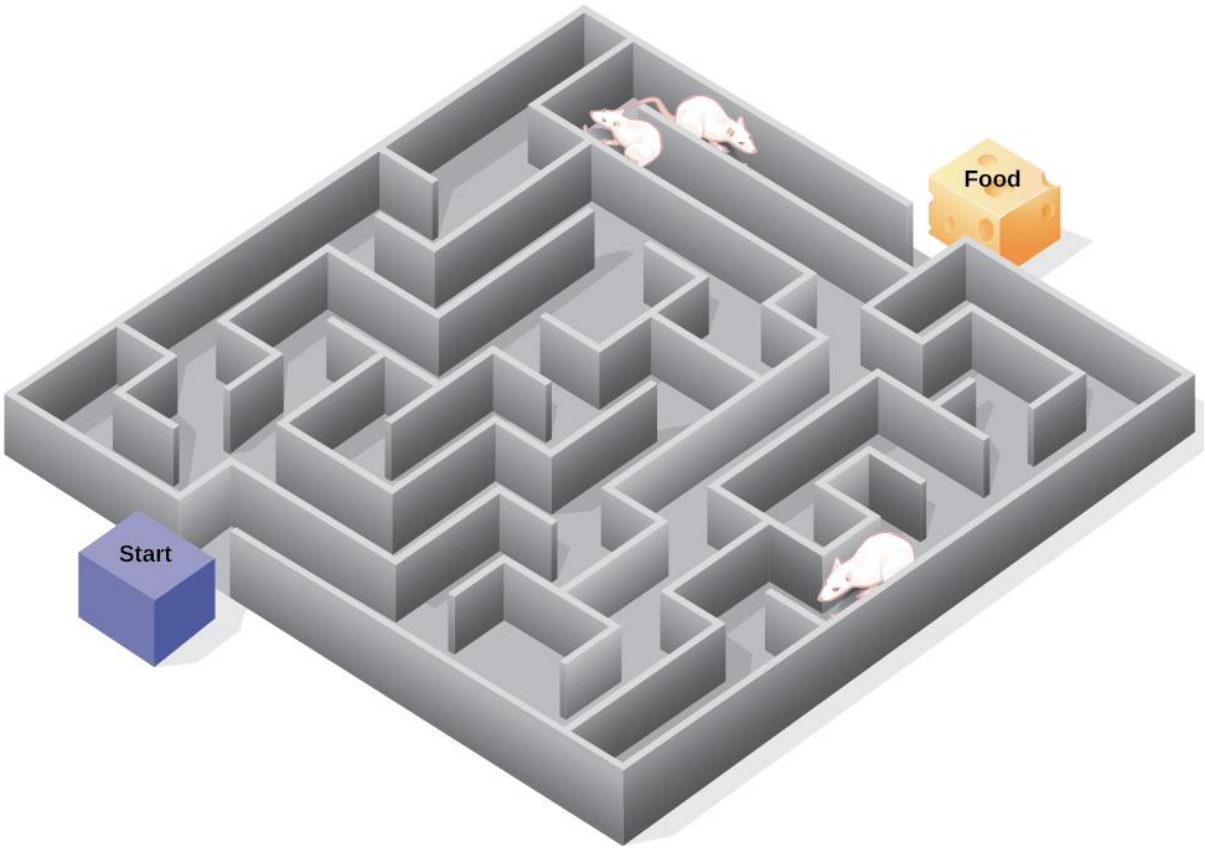
Projet IN104

Résoudre un labyrinthe par de l'Apprentissage par Renforcement



Introduction

En intelligence artificielle, l'apprentissage par renforcement fait référence à une classe de problèmes d'apprentissage automatique, dont le but est d'apprendre, à partir d'expériences, ce qu'il convient de faire en différentes situations, de façon à optimiser une récompense quantitative au cours du temps.
Un paradigme classique pour présenter les problèmes d'apprentissage par renforcement consiste à considérer un agent autonome, plongé au sein d'un environnement, et qui doit prendre des 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.

Sommaire

I. Q-Learning :

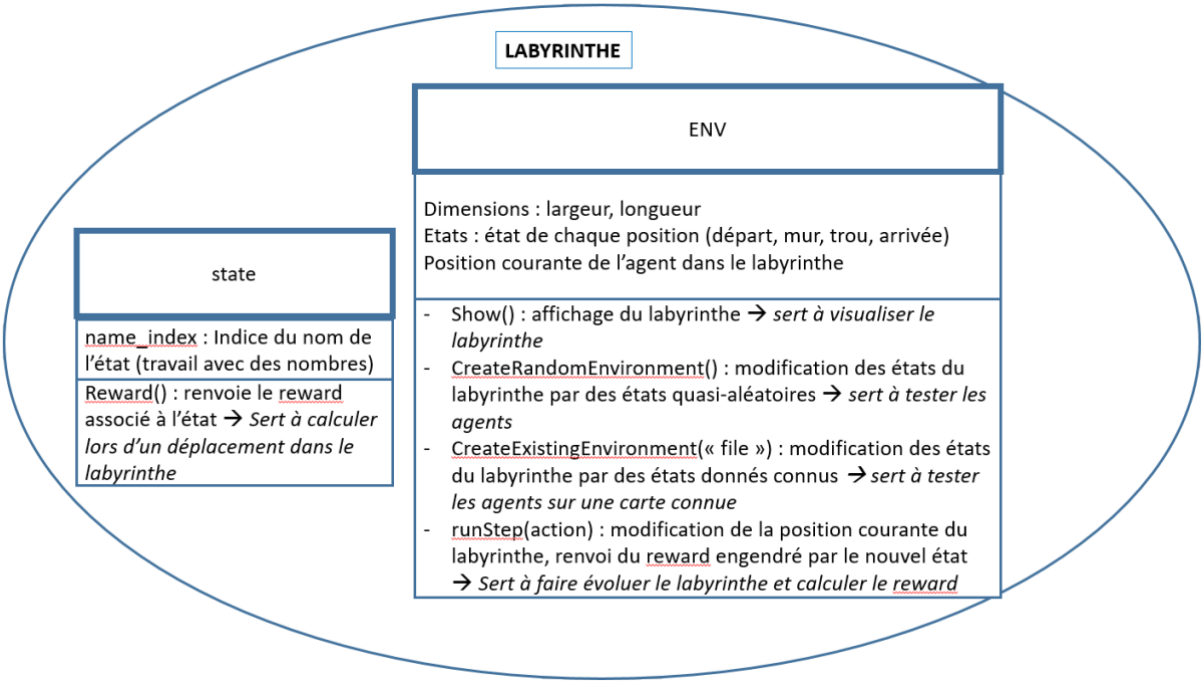
Cette méthode d'apprentissage peut être appliquée pour trouver une suite d'actions associées à des états (politique) d'un processus de décision markovien (fini) quelconque. Cela fonctionne par l'apprentissage d'une fonction de valeur d'état qui permet de déterminer le potentiel bienfait (récompense) de prendre une certaine action dans un certain état en suivant une politique optimale.
La politique est la règle de sélection des actions successives d'un agent dans l'état actuel du système. Lorsque cette fonction de valeur d'action-état est connue, la politique optimale peut être construite en sélectionnant l'action à valeur maximale pour chaque état.

Un des points forts du Q-learning est qu'il permet de comparer les récompenses probables de prendre les actions accessibles sans avoir de connaissance initiale de l'environnement. Cette notion d'apprentissage par récompense a été introduite à l'origine en 1989.
Par la suite, il a été prouvé que le Q-learning converge vers une politique optimale, à comprendre dans le sens de maximiser la récompense totale des étapes successives.

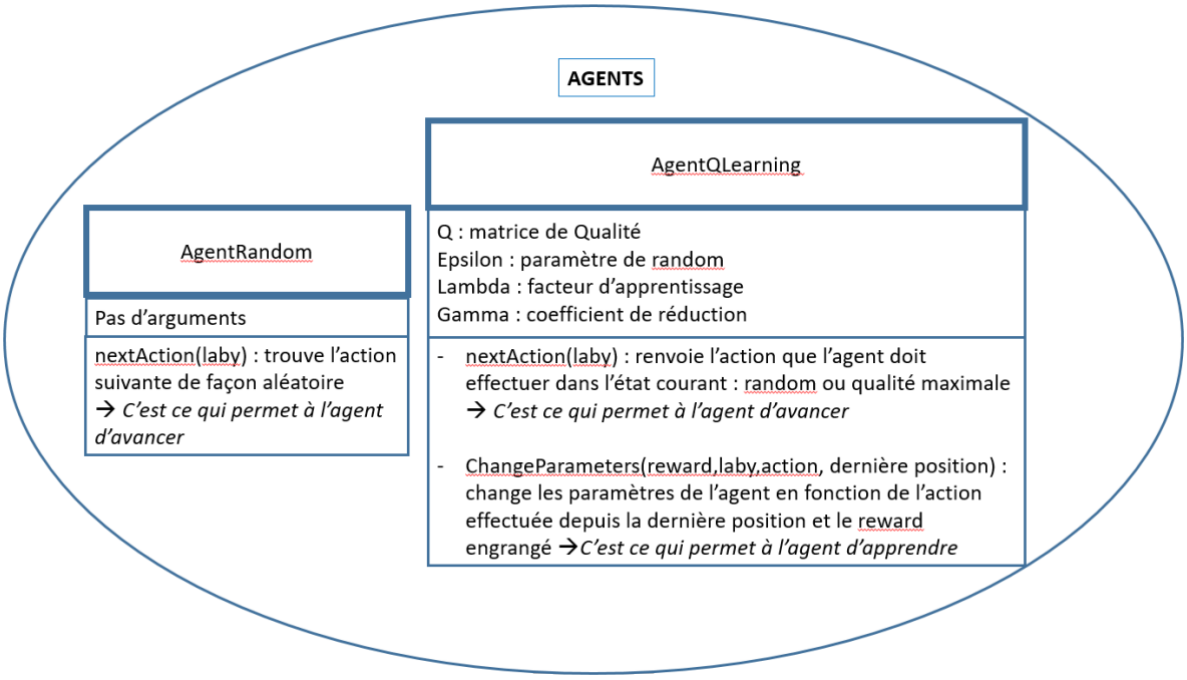
II. Architecture

Diagramme de classes

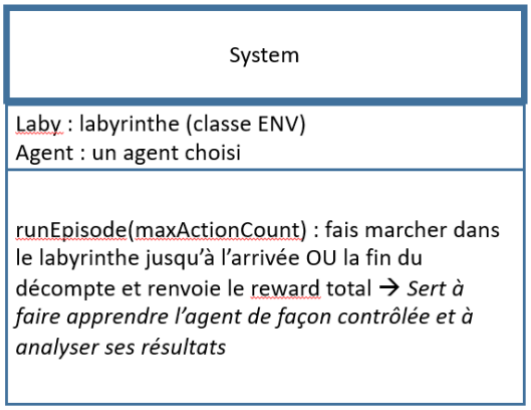
L'environnement est d'abord créé.



Puis on construit les agents qui vont explorer l'environnement.



Enfin le système, composé d'un agent et d'un environnement fixés, est prêt à fonctionner.



Classe Environnement

La bibliothèque numpy a été utilisée afin d'optimiser la puissance de calcul.
Le labyrinthe a été modélisé par une matrice numpy de taille fixe, où les différents états (« state ») des cases sont représentés par des entiers numérotés de 0 à 3, les correspondances étant décrites ci-dessous.

« State »	0	1	2	3
Signification	Case disponible	Mur	Départ	Arrivée

Cette matrice d'états « states » caractérise donc le labyrinthe mais ne possède pas la position de l'agent à chaque étape. Cette dernière est toutefois présente dans la classe par le biais de l'attribut *current_position*.

III. Vie du projet

Activité du 17/04 au 24/04 :
Début de la programmation de l'environnement.
Création de la classe « state » pour caractériser l'état d'une case dans le labyrinthe. Le labyrinthe sera modélisé par une matrice numpy de taille fixe, où les différents états (« state ») des cases sont représentés par des entiers numérotés de 0 à 3, les correspondances étant décrites ci-dessous :

« State »	0	1	2	3
Signification	Case disponible	Mur	Départ	Arrivée

Importation et adaptation d'une classe AgentRandom en prévision de futurs tests simples de notre environnement.

Activité du 28/04 au 01/05 :

Ajout de méthodes dans la classe « state » afin de créer aléatoirement un état pour une case en vue de créer un labyrinthe totalement aléatoire. Certaines précautions (murs sur les bordures sauf départ et arrivée) sont prévues en cas de case en bordure du labyrinthe ou à l'intérieur du labyrinthe.
Ajout de méthodes pour déterminer et aléatoirement des positions possibles de départ sur la bordure puis en sélectionne un au hasard pour le départ et un autre pour l'arrivée. D'autres méthodes permettent de tester si la position est le départ ou l'arrivée.

On réalise que toute la suite de notre projet ne peut avancer sans une méthode « possibleActions » essentielle. On commence alors à implémenter cette méthode et on modélise les actions (Nord, Est, Sud et Ouest) par un couple (i,j) à ajouter à la position courante.

Implémentation de la méthode « show » afin de visualiser le labyrinthe. Détermination de la position de l'agent à l'aide de « current_position » et on affiche un A pour cet agent à la place.

Ajout de l'architecture de runStep() et des récompenses dans l'environnement.

Activité du 02/05 au 09/05 :

Prise de recul sur le projet afin de réaliser sur papier l'architecture des différentes méthodes restantes à implémenter dans l'environnement ainsi que l'interaction avec l'agent.

Activité du 10/05 au 16/05 :

Finalisation de create_random_environment et implémentation du caractère aléatoire et avec 25% de chances d'avoir un mur sur une case à l'intérieur du labyrinthe.
Finalisation du premier Agent Random et tests sur l'environnement aléatoire.
Debugging et tests de l'environnement et des récompenses.

Activité du 17/05 au 23/05 :

Création de l'architecture de l'agent Q-Learning. Le précédent système de récompense n'était pas optimal ni adapté, on a donc revu le système en y

ajoutant une matrice de récompense représentant les récompenses de chaque position.
Implémentation des différentes méthodes de l'agent grâce à l'algorithme fourni dans les documents.

Activité du 24/05 au 30/05 :
Construction d'une classe Agent en général. Révision de la structure de la matrice de qualité, il s'agit désormais d'une matrice numpy à 3 dimensions afin d'utiliser les performances de calculs de numpy.

Ajout des principales méthodes afin d'implémenter la classe AgentQLearning, puis réalisation de quelques tests unitaires de ces méthodes pour vérifier leur validité.

Correction de quelques bugs apparus lors des tests unitaires et tests globaux de la classe ainsi que de la classe environnement. Initialisation des qualités initiales des murs à 10^{-6}. Commencement des tests globaux sur des labyrinthes aléatoires ainsi qu'un labyrinthe fourni en .txt, en implémentant une fonction qui permet de lire le fichier.

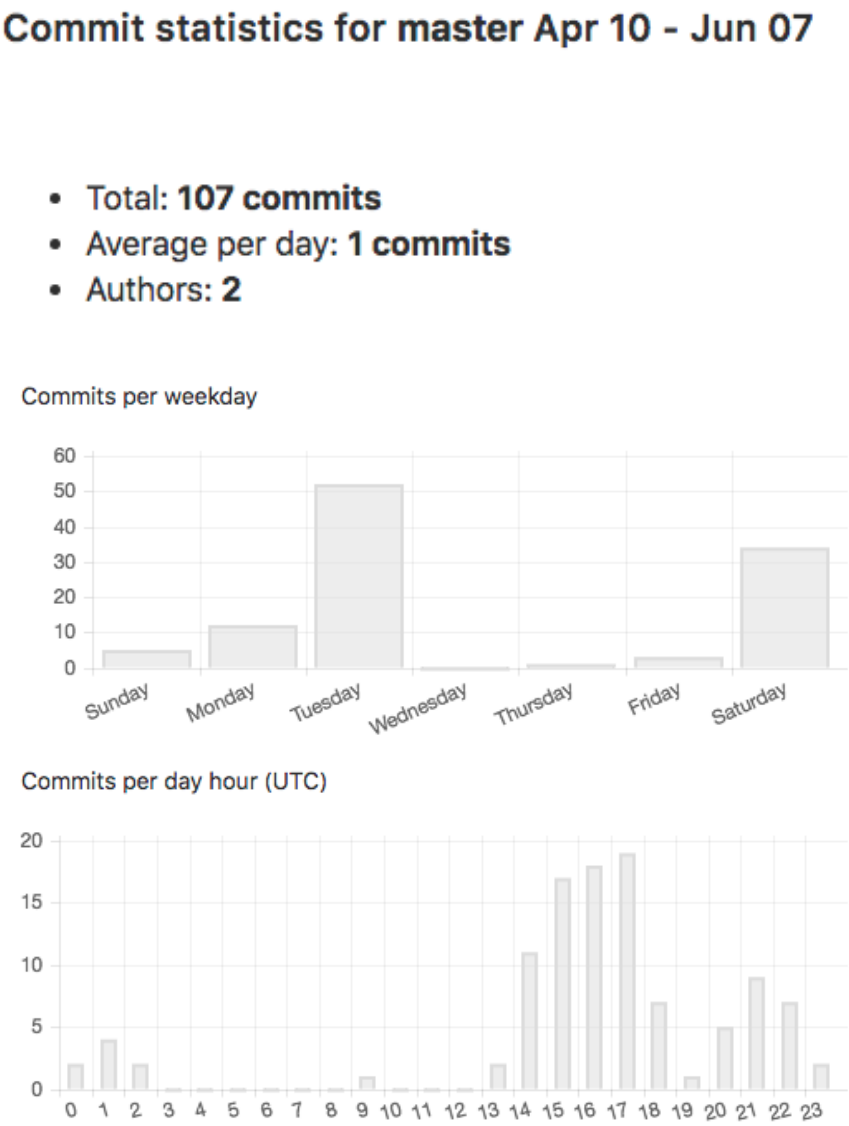
Activité du 01/06 au 07/06 :
Derniers tests globaux ainsi que des tests de performances de l'Agent Q-Learning et de l'influence des paramètres sur le ratio de victoires pour un nombre d'épisodes donné. Retour sur la classe d'Agent en général et correction de différents problèmes rencontrés. Avancement majeur sur le rapport final.

Activité du 08/06 au 12/06 :
Finalisation du rapport et des différents tests de performances. Préparation des supports pour la soutenance.

Expérience sur GIT

Afin d'optimiser l'expérience git, nous avons décidé d'installer Atom ainsi que son Package « git-plus » qui permet d'utiliser git au sein même du logiciel à travers des touches « git pull », « git add + commit », « git add + commit + push » ou alors de régler très facilement les différents conflits rencontrés à l'aide de l'interface graphique. L'environnement fut un peu complexe à prendre en main au départ mais finalement tout s'est bien passé, nous avons pu faire de nombreux commits afin d'interagir entre nos codes.

En reprenant les statistiques du GitLab on obtient les graphes suivants :



Nous remarquons bien entendu que le plus gros nombre de commits est effectué lors des heures de cours d'IN104 le mardi de 14h à 17h mais le nombre de commits est également important en soirée ou en weekend ce qui montre une certaine productivité et efficacité également lorsque le binôme n'est pas nécessairement ensemble. Il s'agit notamment du principal but de git, de pouvoir faire interagir et rassembler la programmation de deux (ou plus) personnes même lorsque ces dernières ne sont pas géographiquement proches.

Les avantages de Git que nous avons exploités sont donc :

- On peut voir toutes les modifications effectuées sur le projet.
- On peut travailler en parallèle, sur le même fichier ou non, en fusionnant les modifications effectuées.

- Toutes les fonctionnalités de Git et Gitlab ont été utilisées de façon très simple, notamment la fusion ou merging, via l'utilisation d'Atom. Cela nous a évité des problèmes que d'autres groupes ont eu avec des commandes directement écrites sur le terminal.

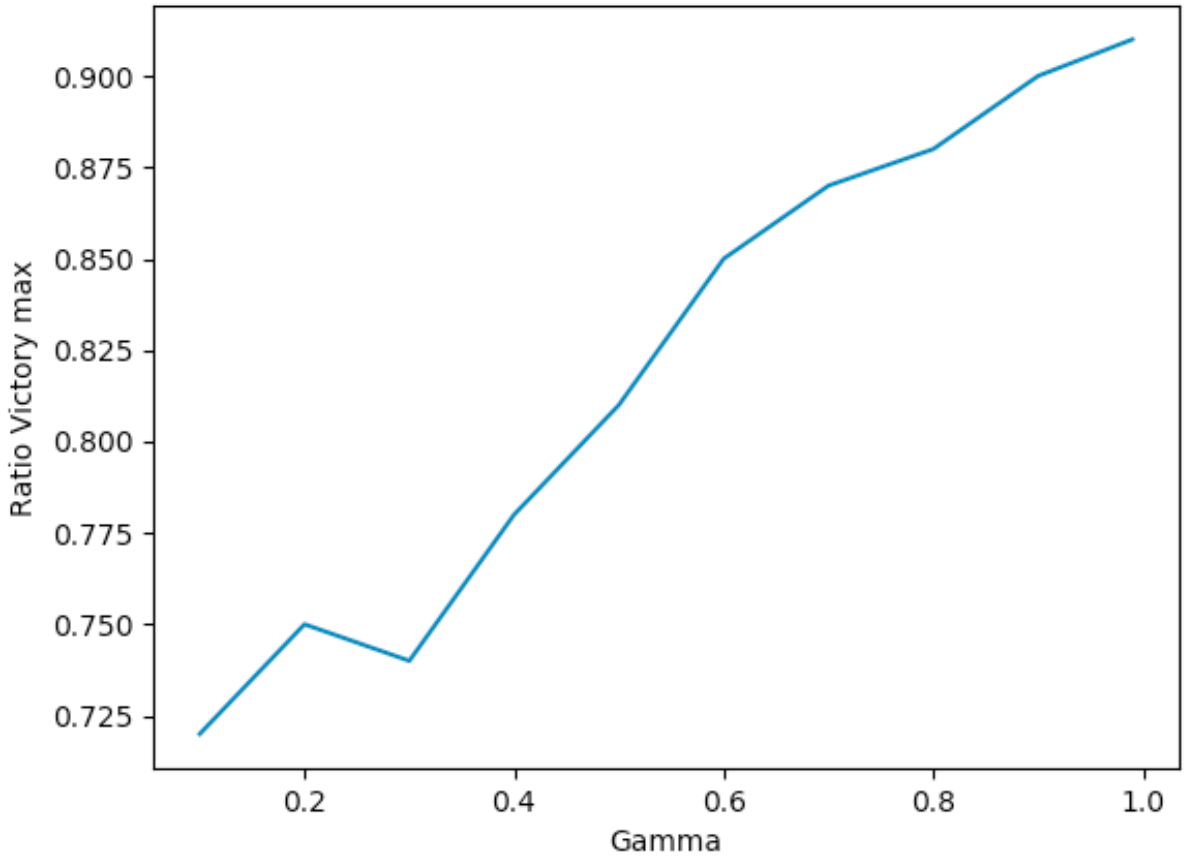
En somme, Git nous a été utile, et surtout on sent qu'il devient de plus en plus nécessaire à mesure que le projet grandit.
On commence à entrevoir son utilité dans de grands projets d'entreprises, où plusieurs branches d'un projet avancent en parallèle, avec des deadlines, etc.

IV. Etude des Performances

A) Un quantificateur du succès de l'algorithme : Ratio de victoires par nombre d'épisodes en fonction de Gamma

Conditions : Tests pour un labyrinthe 10x10 : 1000 Episodes et 1000 Actions max

Afin d'obtenir des résultats pertinents, nous avons effectué ces tests sur un même labyrinthe puis nous avons collecté 10 valeurs de Ratio pour chaque valeur de Gamma avant d'en prendre une moyenne. Cela permet d'évaluer avec plus de précision et de pertinence que de prendre les résultats uniquement sur une valeur particulière.

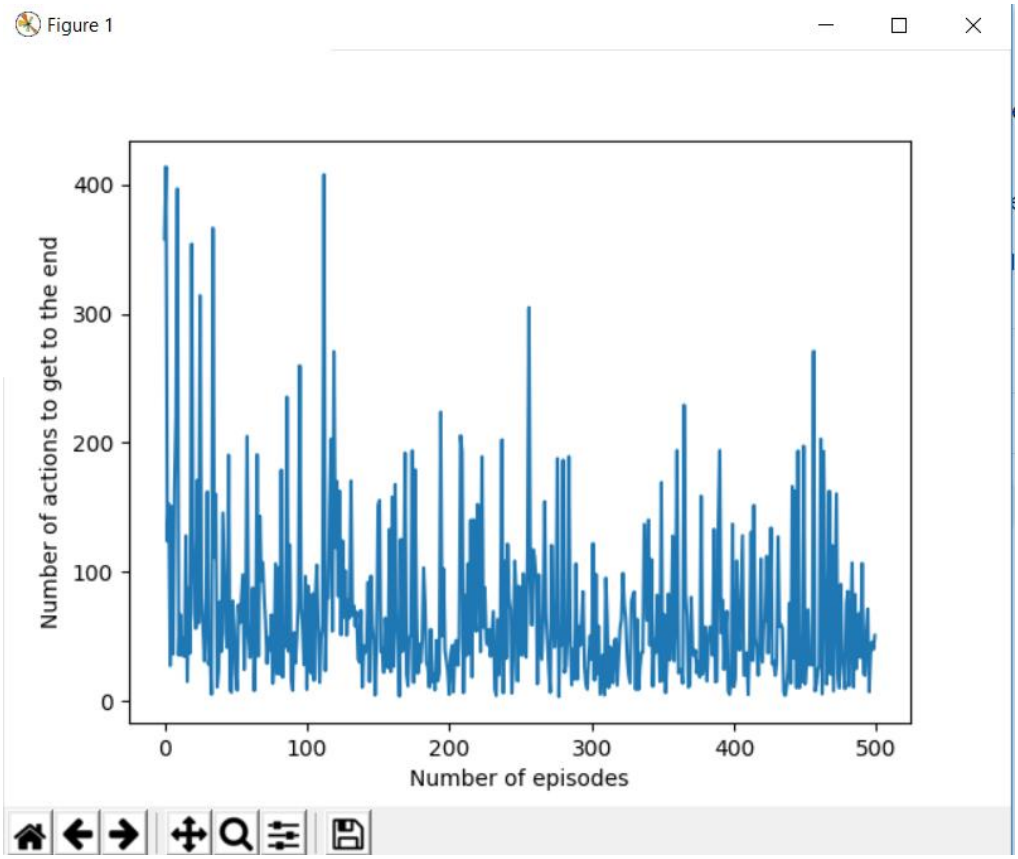


Nous remarquons notamment une importante influence du paramètre Gamma, le facteur d'apprentissage. En effet, plus Gamma est élevé plus la matrice de Qualité change rapidement et s'adapte au labyrinthe qui lui est confié. C'est pourquoi l'agent agit au mieux et trouve donc plus vite la sortie. Cela explique notamment un plus grand ratio de victoire.

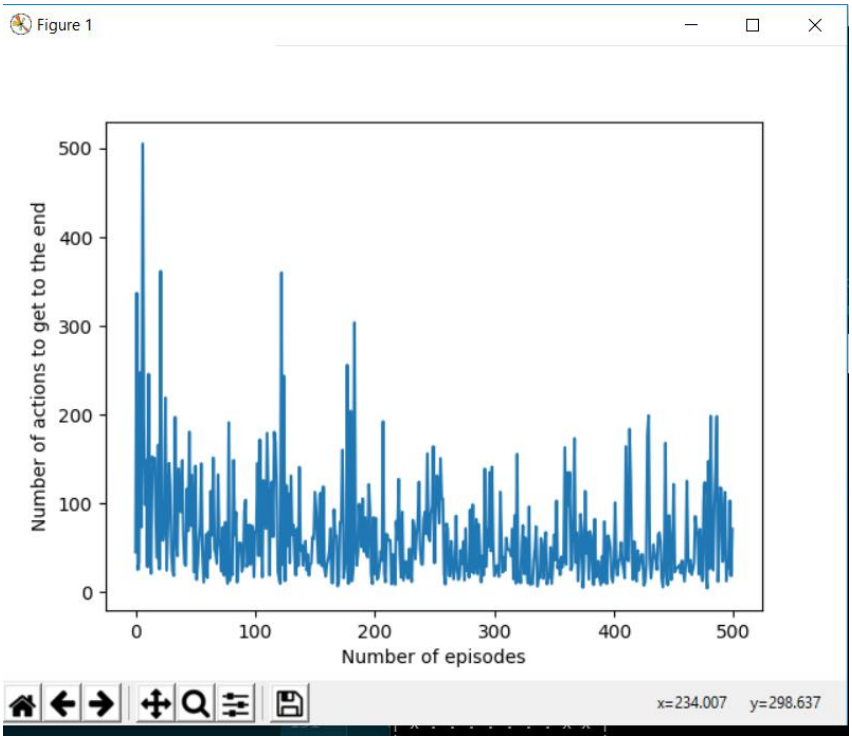
B) Un autre quantificateur : Nombre de déplacements pour sortir du labyrinthe, en fonction du numéro de l'épisode

On affiche le nombre de déplacements dans un épisode, c'est-à-dire le nombre de déplacements pour atteindre la fin du labyrinthe.

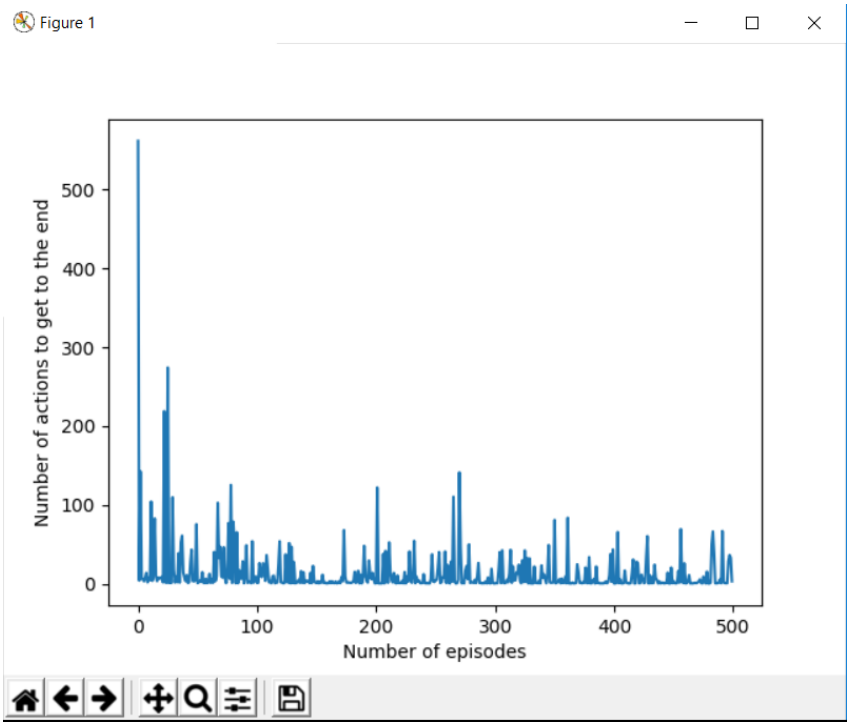
Conditions : SIZE=10, Gamma=0.1



Conditions : SIZE=10, Gamma=0.4



Conditions : SIZE=10, Gamma=0.8



<u>Interprétations :</u>

- Gamma est le facteur d'apprentissage. Plus il est grand, plus l'apprentissage (convergence de la matrice de qualité vers une qualité optimale) sera rapide.
- Il ne faut pas toujours qu'il soit trop rapide cependant. Mais comme on fait décroître Epsilon de 1% à chaque itération, il faut obtenir une bonne qualité *avant* qu'epsilon soit trop petit, sinon on risque de ne plus découvrir de nouvelles pistes et la matrice de qualité n'évolue plus !
- Donc dans ce cas précis, augmenter gamma permet un apprentissage plus rapide ce qui est mieux car epsilon varie (200 itérations donnent epsilon = 0.13 déjà…)
- Les « pics » apparaissant, même après un grand nombre d'épisodes, montrent que l'aléatoire est toujours présent : en effet, c'est parce qu'on réinitialise Epsilon à 1 à chaque épisode.

C) Ouverture sur des améliorations possibles

- Architecture de l'environnement

Les calculs et graphes ont été effectués sur des labyrinthes de taille faible, ne dépassant que rarement une taille de 20x20.

En effet, le temps de calcul se rallonge considérablement dès lors que l'on impose un nombre maximal d'itérations et un nombre d'épisodes grands.
Certains groupes montrent des calculs sur des labyrinthes de taille plus grande : cela dépend de l'environnement qu'ils ont choisi. Nous avons choisi de créer notre propre environnement d'une certaine manière et il n'est bien entendu pas optimal.

- Algorithme en lui-même

La taille de l'environnement a un autre défaut, c'est que la sortie est loin de l'entrée : un grand nombre d'actions (passage d'un état à un autre) sont nécessaires pour trouver la sortie.

Ainsi l'algorithme de QLearning utilisé ici ne modifie la matrice de qualité que dans un « rayon » 2 actions effectuées à partir de la position courante. Une amélioration possible serait de modifier cette matrice dans un rayon plus grand. Par exemple, Q(a,s) =f(Q(a',s'), Q(a'',s'')) où s'' est un état obtenu après deux actions successives depuis s ; a'' est une action amenant s'' à s''' nouvel état.

L'algorithme ainsi créé pourrait alors mieux convenir à la taille des labyrinthes, et donc des grands labyrinthes.

Conclusion

La résolution d'un labyrinthe par apprentissage par renforcement a été l'occasion de tenir et construite un réel projet, assez conséquent, à partir de très peu de choses.
Il a été nécessaire de faire des recherches et de s'informer de manière autonome sur le sujet et de s'adapter aux codes et aux informations fournies par l'encadrant ici, mais qui sera plus tard nos supérieurs ou une équipe en charge d'un projet similaire.
On se rend rapidement compte de l'importance de l'organisation du projet, la répartition des tâches, la clarté de tout cela et du code.

Cela a été également l'occasion d'expérimenter réellement git et d'apprivoiser ce système. Git est en effet essentiel dans la vie de tout programmeur ou ingénieur « Software », et avoir un Git devient de plus en plus un avantage qui s'apparenterait à avoir un CV. Posséder un projet Git d'une grande envergure est un point non négligeable.
Au-delà de Git, il nous apparaît important d'avoir une façon de travailler et de coder similaire dans un même groupe : organisation, syntaxe, clarté, explications…
Tout cela est rendu possible par des règles générales qui sont appliquées au quotidien par les (bons) professionnels mais aussi par des règles communes à un même groupe de travail. Ces dernières règles sont établies et modulés à volonté par le biais, par exemple, de logiciels de codage tels que ceux que nous avons utilisé.