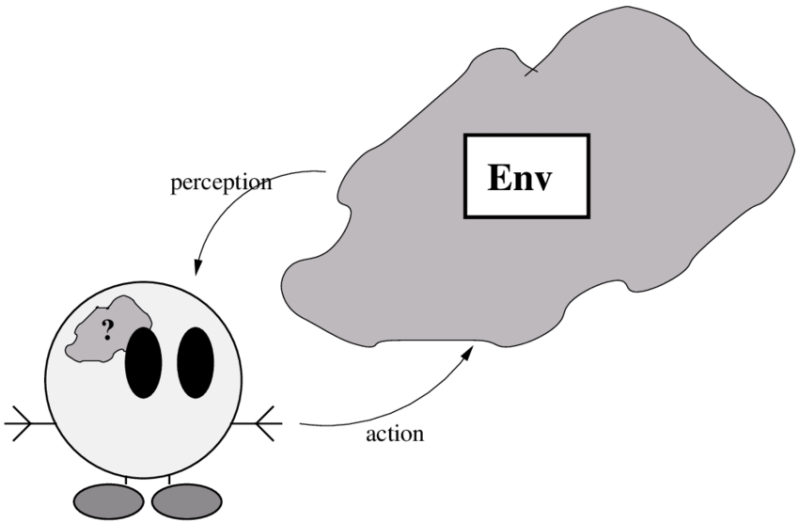
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Projet Java
GM4-2018/2019

Rapport de projet semestriel

Janvier 2019



Etudiants : David Albert, Thibaud Clavel et Henri Durozay

à l'attention de : M.Vercouter

Table des matières

Partie 1

Introduction

1.1 Le cadre du projet

Ce projet a été réalisé dans le cadre de la formation GM. Tout au long de ce projet nous avons poursuivit différents objectifs. Un premier objectif a été d'étudier différentes méthodes d'apprentissage par renforcement pour le contrôle d'un agent robotisé. Ensuite nous avons du définir précisément le type d'environnement simulé et les tâches que devait réaliser le robot. Après avoir choisi l'environnement global de simulation, nous avons finalement implémenté deux algorithmes permettant au robot d'accomplir le plus efficacement possible la tâche qui lui est allouée. Pour l'implémentation de l'environnement de simulation nous avons utilisé la librairie JavaFX, qui, comme nous avions pu le constater lors de précédents projets est mieux adapté pour les animations graphiques.

1.1.1 Choix de l'environnement d'évolution de l'agent

Dans un premier temps nous avons donc spécifié les caractéristiques propres à l'environnement d'évolution de l'agent. L'environnement dans lequel celui-ci évolue est une carte composée d'un certain nombre d'objectifs. Dans un premier temps il n'y a pas d'obstacles. Puis, si les résultats sont satisfaisant, des obstacles seront ajoutés (3 au minimum). Le robot doit rejoindre les objectifs tout en évitant les obstacles.

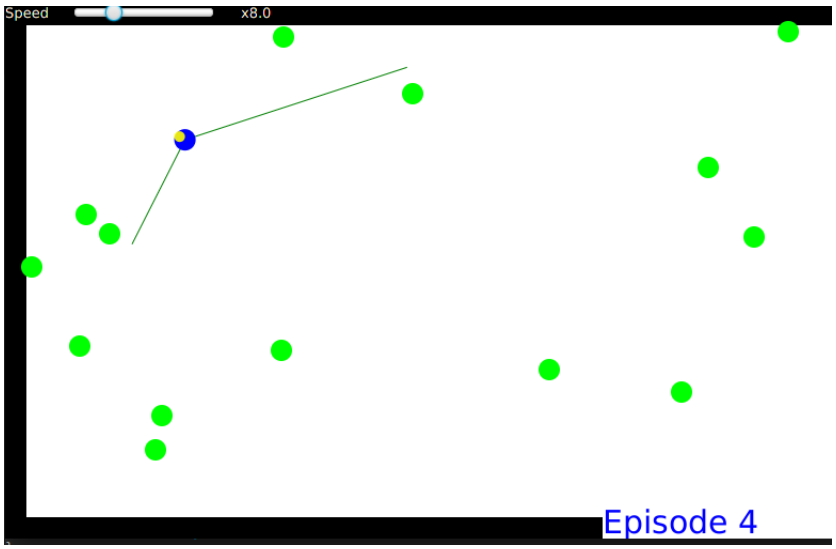


FIGURE 1.1 – Environnement sans obstacles

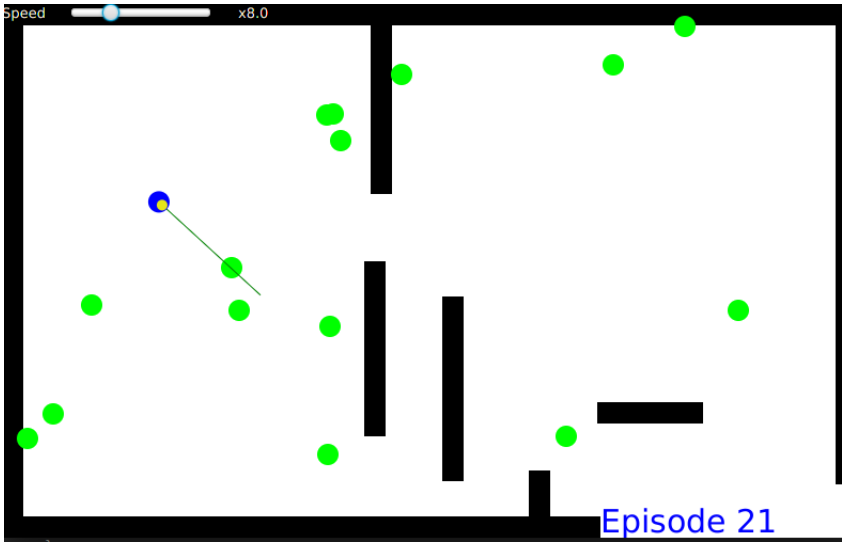


FIGURE 1.2 – Environnement avec 5 obstacles

1.1.2 Choix de l'agent

L'agent est un robot composé de deux types de capteurs et pouvant effectuer trois actions. Le nombre de capteurs et leur position dépend du modèle et sera détaillé dans la suite du rapport.

Les capteurs : Il y a deux types de capteurs ; d'obstacles et d'objectifs. Les capteurs sont positionnés sur l'agent avec des angles différents. Chaque capteur renvoie la distance séparant le centre de l'agent à l'objet détecté dans la direction spécifiée. Dans la réalité on pourrait imaginer utiliser des capteurs à ultrasons pour déterminer la distance entre le robot et un objet dans une direction spécifique, couplé d'une caméra réalisant du traitement d'image pour détecter le type d'objet. Il est évident que cette solution étant très coûteuse, elle ne semble pas être réalisable en réalité.

Les actions : L'agent peut choisir entre trois actions : avancer, pivoter vers la droite et pivoter vers la gauche.

Partie 2

Algorithme génétique

Au début du projet nous avons passé un long moment sur choix du modèle. Dans un un premier temps, après différentes recherches et pour convenir à tout le monde nous avons décidé d'utiliser un algorithme génétique pour réaliser la tâche désirée.

2.1 Cadre Théorique

2.1.1 Origine et principe

Les algorithmes génétiques font leur apparition au 19ème siècle et sont inspirés de la théorie de l'évolution de Darwin. Selon lui, les espèces animales et végétales ont du changer pour survivre, en s'adaptant aux variations de leur environnement. Et seuls ceux qui survivent ont des descendances, c'est ce qu'on appelle la sélection naturelle.

Les algorithmes de génétique utilisent ce principe de sélection naturelle. Le principe est de simuler l'évolution d'une population initiale, générée aléatoirement, dans un milieu. Cette population doit faire face à un problème. On attribut donc un score à chaque individu, qui correspond à son adaptation au problème. Ensuite, on effectue une sélection au sein de cette population en fonction du score des individus. Puis par croisement de ces individus, on passe à la génération suivante. Ce processus est donc constitué de 4 phases majeures :

- **Évaluation :** Un score est attribué à chaque individu.
- **Sélection :** En fonction des scores obtenus, on prends les individus qui peuvent permettre d'obtenir les meilleurs résultats, une sélection est donc opérée.
- **Les croisements :** On croise les gènes de deux individus (parents) pour créer de nouveaux gènes qui serviront a créer les nouveaux individus (enfants) de la nouvelle génération.
- **Les mutations :** les gènes des nouveaux individus (enfants), peuvent être modifiés de manière aléatoire. Avec une probabilité relativement faible, afin de conserver le principe de sélection et d'évolution.

Ces différentes étapes sont donc répétées jusqu'à l'obtention d'une convergence. C'est à dire, jusqu'à ce que des individus de la générations obtiennent des scores "satisfaisants". En d'autre terme, jusqu'à ce que leur ADN (ensemble de leurs gènes) leur permette de survivre dans l'environnement.

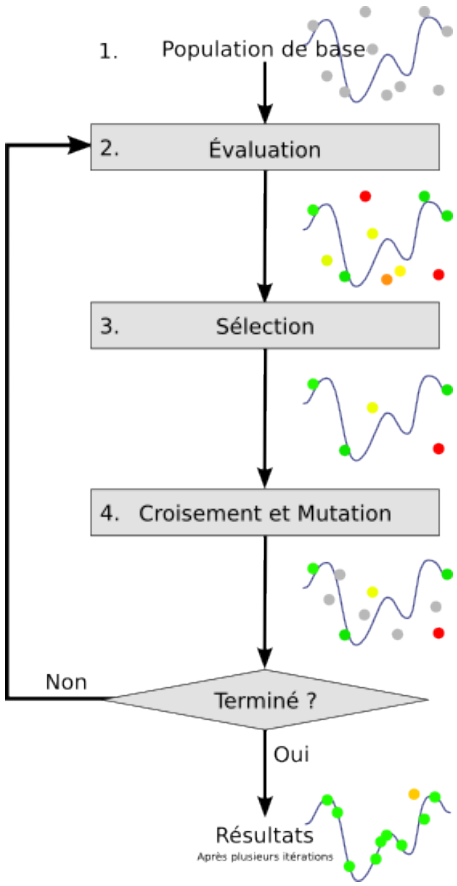


FIGURE 2.1 – Algorithme de génétique

2.2 Application à notre problème

Dans le cadre de notre problème. On a choisi de générer aléatoirement une population de $N = 1\,000$ individus. A chaque fin de génération, on garde un certain pourcentage de la population correspondant aux individus qui ont obtenu les meilleurs score. Ainsi, on effectue les croisements sur ces individus afin d'obtenir la nouvelle génération. Le score attribué à chaque individu est établi en fonction du nombres d'objectifs atteints, du temps et des obstacles. Plus précisément, le score d'un individu augmente lorsqu'un objectif est atteint en fonction du temps. Donc plus vite un objectif est atteint, plus il rapportera de points. Si l'individu atteint un obstacle, le score diminue.

2.2.1 Paramétrage

Afin d'obtenir de meilleurs résultats, on peut jouer sur différents paramètres :
— *nb_cap*, le nombre de capteurs du robot.
— *duree_simul*, la durée d'une simulation pour une génération.
— Le nombre d'objectifs.
— *R()*, la fonction récompense. En modifiant le score attribué dans la fonction récompense (ex : augmenter le score quand un objectif est atteint).
— la fonction *crossover()*, La façon de choisir les individus lors de l'étape du croisement.
— *t_mut*, le taux de mutation. On choisit ce taux 0,001 et 0,01. Il doit rester assez faible pour ne pas rendre l'expérience trop aléatoire. Mais il est important pour conserver le caractère évolutif de l'algorithme.
— le nombre d'obstacles. Qui sera de 0 dans un premier temps.

Il faut donc ajuster au mieux ces paramètres pour obtenir une convergence.

Les capteurs

Pour l'ensemble de nos simulations, nous avons fait le choix de prendre *nb_cap*=11 (cf. schéma ci dessous). Il y a donc 11 capteur dont 3 pour les obstacles(en vert) et 8 pour les objectifs(en rouge).

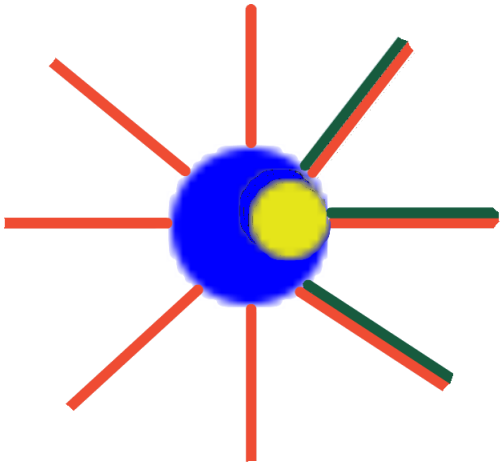


FIGURE 2.2 – Schéma de l'agent et ses capteurs

Les capteurs d'objectifs sont donc répartis équitablement autour de l'agent(à angle réguliers). Et les capteurs d'obstacle sont juste reparties sur la partie avant du robot, car il est presque inutile d'en mettre de l'autre coté du robot. Cela augmenterait inutilement les coûts de calcul et donc le temps d'exécution.

A noter que, les paramètres modifiable peuvent être changés directement dans la classe paramètre.

Fonction de récompense

Dans le cas où un objectif est atteint, on augmente le score de l'agent pour avoir trouvé l'objectif. Celui-ci se voit accorder la récompense suivante en fonction du temps actuel t et de la durée de la simulation T. :

$$r_t = (T - t)$$

Dans le cas où l'agent rencontre un obstacle, on lui attribut une pénalité.

$$p_t = (T - t)$$

A chaque frame, le score de l'agent est donc mis à jour tel que suit :

$$score = score + \begin{cases} r_t - p_t \ \ si\ collisionObstacle = true\ et\ collisionObjectif = true \\ r_t \ \ si\ collisionObstacle = true \\ -p_t \ \ si\ collisionObjectif = true \\ 0 \ \ si\ robotMort\ ou\ pasDeCollision \end{cases}$$

Le fait de prendre en compte le temps de la simulation et le temps écoulé depuis le début de la simulation a pour objectif de faire comprendre à l'agent qu'il doit trouver les objectifs rapidement et qu'il ne doit pas rentrer en collision avec des obstacles. Il peut être intéressant de multiplier cette quantité par un facteur différents dans les deux cas. Par exemple, dans le cas où l'agent rentre en collision avec un obstacle, si on multiplie cette quantité par 10 cela pourrait permettre de faire comprendre à l'agent qu'il ne doit surtout par rencontrer un obstacle.

Croisement

A chaque étape de l'algorithme, on conserve un certain pourcentage de la population (3% par ex.). Ces 3% de la population correspondent au individus ayant obtenu les meilleurs scores et seront les parents de la prochaine génération. Pour la nouvelle génération, on conserve d'abord les 3% d'individus sélectionnés à la génération précédente. Quant aux 97% restant, chaque individu est obtenu par croisement de gènes de deux individus sélectionnés

de manière aléatoire parmi les parents. On pourrait donc ajuster ce croisement en modifiant la façon aléatoire de choisir les parents. On pourrait par exemple attribuer une probabilité à chaque parent d'être sélectionné, en adéquation avec son score.

2.2.2 Le choix de l'ADN

Dans notre modèle, nous avons utilisé un réseau de neurone dit "fully-connected" ou standard en guise d'ADN pour chaque individu.

Réseau de neurones

Un réseau de neurones reprend le principe d'un cerveau. Il est constitué de neurones organisés en différentes couches connectées grâce à des liaisons neuronales. Chaque liaison est coefficientée et relie un neurone de la n^{ieme} couche à un autre de la $(n+1)^{ieme}$. On peut modéliser un tel réseau sous la forme d'un graphe (cf. schéma ci dessous).

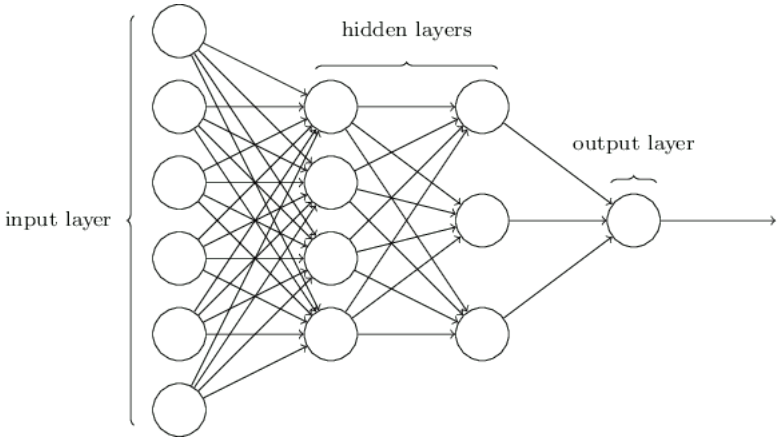


FIGURE 2.3 – Schéma d'un réseau de neurone

Plus formellement, un réseau de neurones modélise une fonction $f_\theta : R^n \to R^p$, où n est le nombre d'entrées du réseau (= nombre de neurone de la couche d'entrée) et p est le nombre de sorties (=nombre de neurones de la couche de sortie) et $\theta = (\theta_1, ..., \theta_n)$ représente les coefficients des liaisons. Le but de tout algorithme d'apprentissage automatique est d'ajuster les paramètres θ de tel sorte que pour une entrée donnée, la sortie soit adéquate au problème. Il existe différentes façons d'optimiser un réseau de neurones. Une des plus connue est l'algorithme de descente du gradient stochastique. Dans notre cas nous utilisons un algorithme génétique pour ajuster les paramètres du réseau (voir plus haut).

Le choix d'architecture

L'architecture de l'ADN est donc composé des paramètre θ d'un réseau de neurones entièrement connecté. L'entrée du réseau est composé des n distances perçues par les capteurs du robot (ici 8 d'objectifs et 3 d'obstacles). La couche d'entrée comprend donc ici 11 neurones. La sortie du réseau est composée de p valeurs qui correspondent aux p actions possibles (ici $p = 3$). La couche de sortie comprend donc ici 3 neurones. L'action prise par le robot est celle dont la valeur de sortie est la plus grande.

Fonction d'activation : Entre chaque couche, on applique une fonction d'activation, dans ce projet nous avons utilisé la fonction d'activation $sigmoid(x) = \frac{1}{1-e^{-x}}$. D'autres fonctions d'activations peuvent être appliquées telles que $tan(x)$ ou $max(x, 0)$.

Nombre de couches et tailles : Nous avons vu que la couche d'entrée comprend 11 neurones et la couche de sortie en comprend 3. Concernant les couches cachées, nous avons testé différentes tailles (3 couches de 3 neurones, 2 couches de 6 et 4 neurones, 1 couche de 7 neurones, etc). Finalement, les différents choix n'impactant pas grandement les résultats nous avons choisi d'utiliser 2 couches cachées avec respectivement 6 et 4 neurones.

2.3 Résultats et axes d'amélioration

Afin de visualiser nos résultats, nous avons implémenté une méthode dans l'algorithme de la simulation génétique (appelée à chaque fin de génération) qui enregistre sur un fichier externe le nombre de robots ayant touchés 1,2,...,n objectifs.

Toutes les expériences ne seront pas visible dans ce rapport, seulement les plus pertinentes y seront affichées.

2.3.1 Première expérience :

Paramètres
-Population : 1000 robots
-Pourcentage de parents : 3%
-Objectifs : 15
-Vitesse : 3
-Temps génération : 90s
-Génération : 400

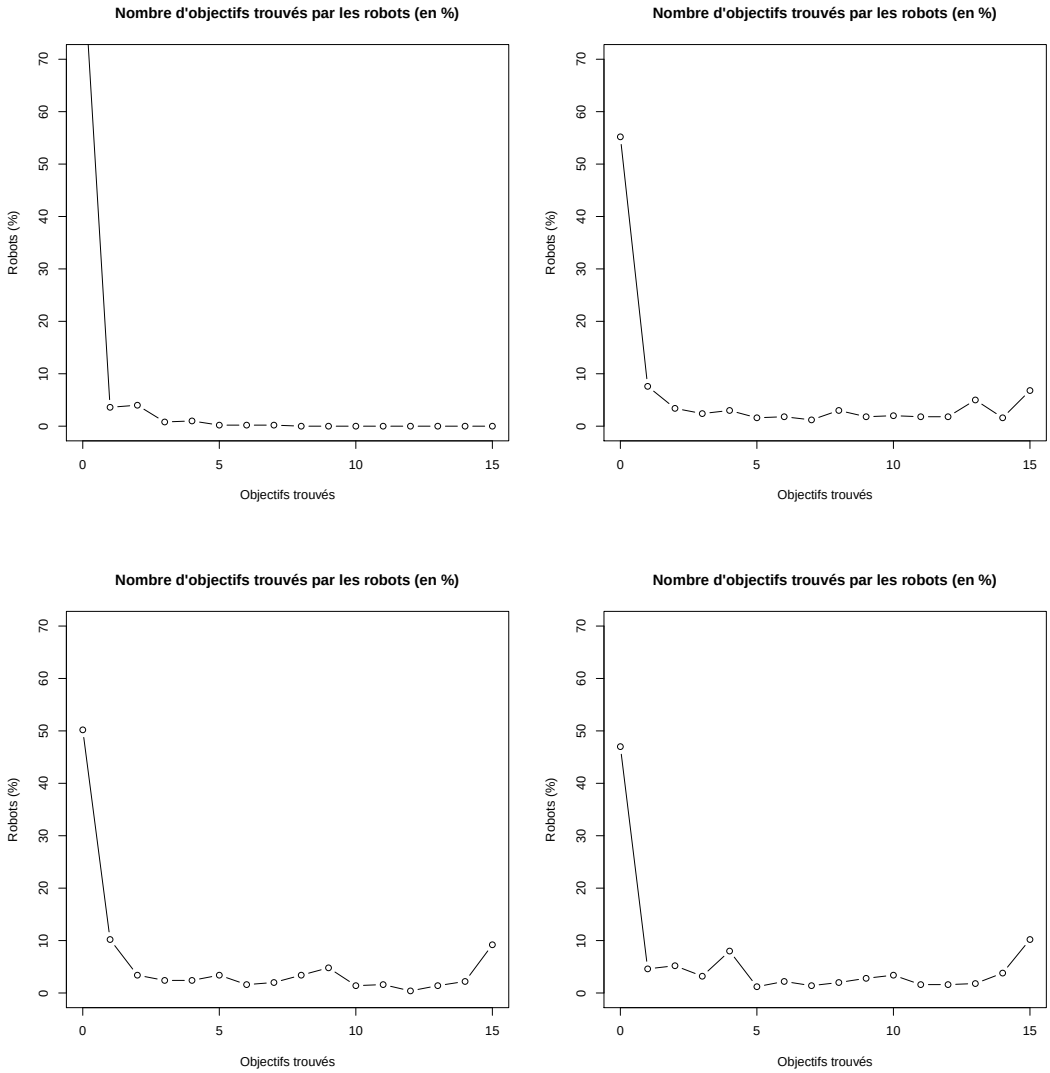


FIGURE 2.4 – Objectifs trouvés génération 1,40,300,375

Nous pouvons voir avec ces graphiques que les robots s'améliorent rapidement pour atteindre, à la 40e génération, 8% de réussite totale (obtenir tous les objectifs du plateau). A partie de ce stade, la progression est beaucoup plus lente et atteint un point de stagnation autour de 10% de réussite.

Par ailleurs, la meilleure IA de cette simulation est assez satisfaisante puisque celle-ci est capable de trouver la totalité des objectifs. Cependant, sa méthode de recherche n'est pas optimale. En effet, le robot a tendance à tourner toujours dans le même sens et a chercher les objectifs qui sont devant lui.

2.3.2 Deuxième expérience :

Comme dit précédemment, l'IA de la première simulation n'est absolument pas optimisée. Ainsi, cette simulation a eu pour but d'améliorer cette aspect du robot.

Les paramètres ci-dessous ont donc été choisis pour remédier à ce problème. Nous avons augmenté le pourcentage de parents afin d'augmenter la diversité des IA de la simulation.

Ensuite, le nombre d'objectif a été légèrement augmenté pour que l'IA comprennent rapidement l'utilité de trouver ces derniers.

Enfin, le temps de chaque génération a été fortement diminué pour privilégier les IA allant rapidement sur les objectifs.

Paramètres
-Population : 500 robots
-Pourcentage de parents : 10%
-Objectifs : 20
-Vitesse : 3
-Temps génération : 30s
-Génération : 700

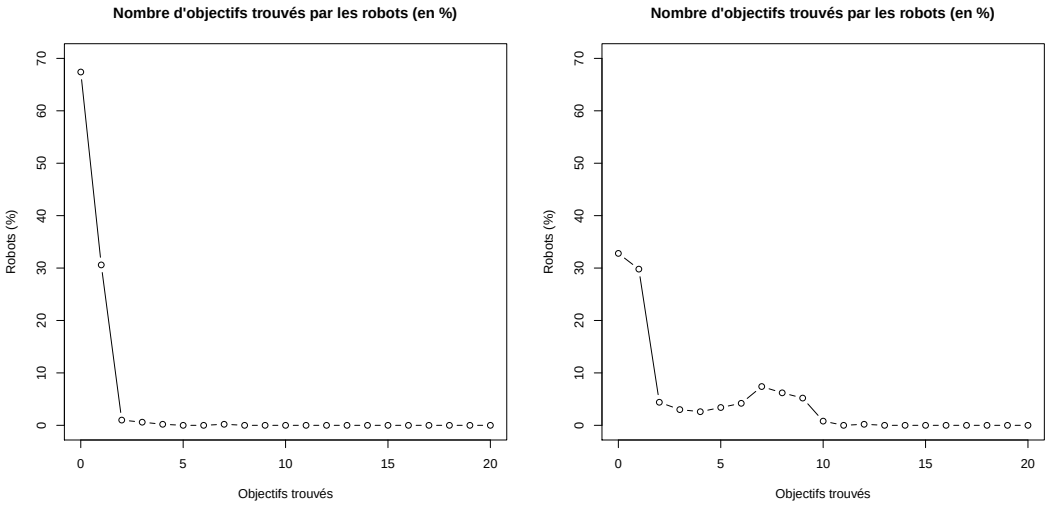


FIGURE 2.5 – Objectifs trouvés génération 1,25

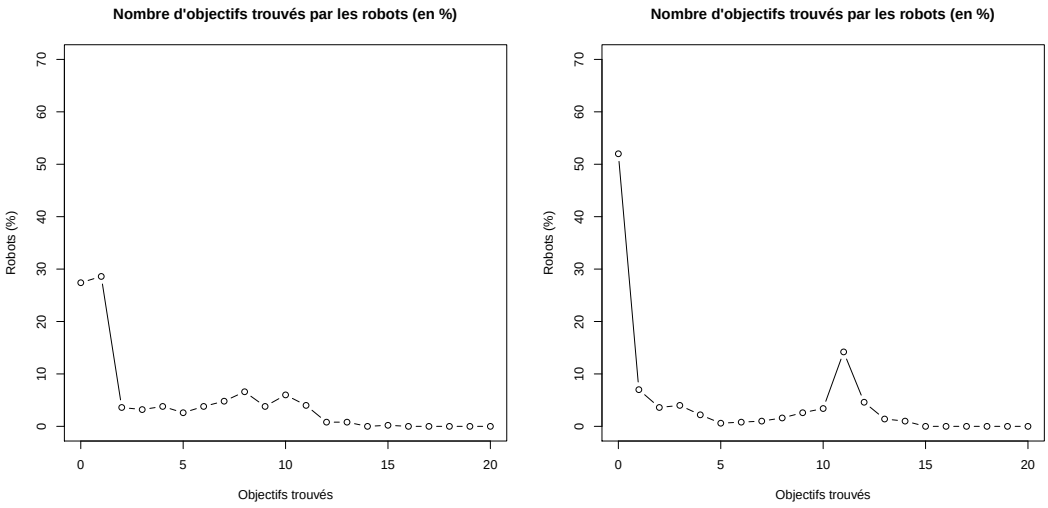


FIGURE 2.6 – Objectifs trouvés génération 50,100

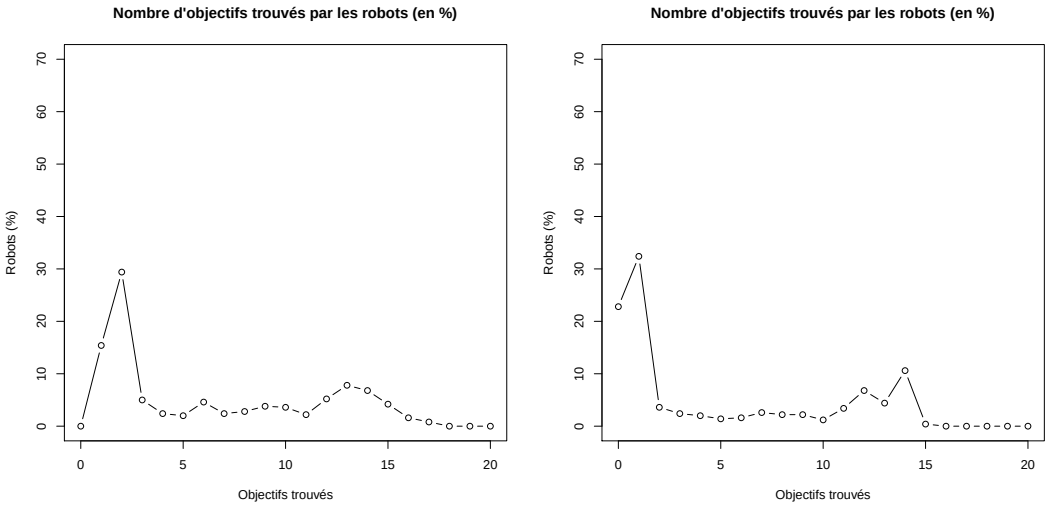


FIGURE 2.7 – Objectifs trouvés génération 300,700

Encore une fois, nous pouvons remarquer avec ces graphiques que les robots s'améliorent rapidement et que 15% d'entre eux atteignent à la 100e génération au moins 11 objectifs sur les 20, ce qui reste satisfaisant puisque le temps d'une génération a été divisé par 3 . A partir de ce stade, la progression est beaucoup plus lente comme précédemment et atteint un point de stagnation où 10% des robots trouvent environ 15 objectifs

Ainsi, la meilleure IA de cette simulation est plus efficace que l'IA précédente. En effet, cette IA est beaucoup plus rapide et récupère tous les objectifs si on la place dans une simulation identiques à la première expérience. Au final, l'IA générée par cette simulation est plutôt satisfaisante mais reste néanmoins imparfaite.

Partie 3

Algorithme du Q-Learning

Dans un second temps, nous nous sommes penché sur l'implémentation d'une méthode d'apprentissage par renforcement, appelée Q-learning.

3.1 Cadre Théorique

3.1.1 Processus décisionnels de Markov

Les processus décisionnels de Markov (PDM) sont définis comme des processus stochastique contrôlés qui satisfont la propriété de Markov et qui assignent des récompenses au transitions d'états.

On définit les PDMs par un quintuplet (S, A, R, p, r) où :
— S est l'espace d'état dans lequel évolue le processus
— A est l'espace d'actions qui contrôlent la dynamique de l'état
— T est l'espace des temps, ou axe temporel
— $p()$ sont les probabilités de transition en états
— $r()$ est la fonction de récompense

Ce type de processus permet de modéliser la dynamique de l'état d'un système soumis au contrôle d'un agent au sein d'un environnement stochastique. Les méthodes de planification de ces processus visent à chercher une politique d'action optimale pour l'agent. C'est-à-dire, grâce à différents types d'algorithmes de résolution des processus décisionnels de Markov, tel que l'algorithme d'itération sur les valeurs ou l'algorithme d'itération sur les politiques, l'agent sera capable de prendre l'action adéquate quelque soit sont état $s \in S$.

3.1.2 Apprentissage par renforcement

Les méthodes d'apprentissage par renforcement se distinguent des PDMs par le fait qu'elles permettent de traiter les situations dans lesquelles la fonction de transition p et la fonction de récompense r ne sont pas connues. Le système est donc régit par (S, A, T). Une certaine fonction de récompense r existe, cependant elle n'attribue pas une récompense propre à chaque état, ou couple état/action.

Ce type d'apprentissage est dit "non-supervisé". En effet, bien que l'agent reçoit des récompenses, lorsqu'il les reçoit il ne sait pas si l'action qu'il a effectué est la meilleure possible ; il doit essayer d'autres actions pour déterminer s'il peut recevoir une meilleure récompense. Dans le cas d'un apprentissage supervisé, on aurait spécifié à l'agent l'action étant la meilleure à prendre. Un point primordial de l'apprentissage par renforcement est que tout choix d'action a des conséquences sur plus ou moins long terme et la récompense immédiate n'est rien sans les autres données. On est souvent confronté à des récompenses retardées et qui dépendent d'un grand nombres d'états/actions. C'est pourquoi la particularité de ce type d'apprentissage est de mélanger $exploration$ et $exploitation$, et ceci dans les bonnes proportions.

Compromis exploration/exploitation

L'exploration : Qualifie le fait que l'agent doit explorer son environnement pour déterminer dans quelles circonstances il est puni ou récompensé et quelles sont les séquences d'action qui lui permettent d'atteindre les récompenses plutôt que les punitions.

L'exploitation : Désigne l'exploitation de la connaissance de l'agent. Cela consiste pour l'agent à refaire les actions dont il connaît déjà la récompense.

Le compromis entre exploration et exploitation doit être bien choisi pour éviter que l'agent n'ait un comportement sous-optimal. Plusieurs méthodes existent pour choisir entre exploration et exploitation :

— Méthode $\epsilon - greedy$: suit la meilleure politique connue avec la probabilité $1 - \epsilon$ ou tire uniformément a_n dans A avec la probabilité ϵ, $\epsilon \in [0, 1]$.

— Méthode $softmax$: tire a_n dans A selon la distribution de Boltzmann. La probabilité associée à l'action a est :

$$p_T(a) = \frac{e^{-\frac{Q_n(s_n,a)}{T}}}{\sum_{a'} e^{-\frac{Q_n(s_n,a')}{T}}}$$

où $lim_{n\to\infty} T = 0$ et $Q_n(s, a)$ représente la fonction de valeur (voir si après).

Dans notre cas, nous avons utilisé la méthode $\epsilon - greedy$.

3.1.3 Q-learning

La plupart des algorithmes d'apprentissage par renforcement ne travaillent pas directement sur l'élaboration d'une politique mais passent par l'approximation itérative d'une fonction de valeur, issue de la théorie des processus décisionnels de markov. Dans l'algorithme Q-learning, cette fonction de valeur est appelée Q (pour Quality). L'algorithme Q-learning est une simplification de l'algorithme de $Sarsa$ qui lui-même est inspiré de la l'algorithme d'itération sur les valeurs des PDMs. La différence avec l'algorithme de $Sarsa$ est qu'il n'est plus nécessaire, à un instant n, de connaître a_{n+1}, l'action réalisée à l'intant $n + 1$.

La mise à jour de la matrice Q se fait de la façon suivant :

$$Q(s_n, a_n) \leftarrow Q(s_n, a_n) + \alpha[r_n + \gamma \max_a Q(s_{n+1}, a) - Q(s_n, a_n)]$$

α : le taux d'apprentissage γ : facteur de pondération s_n : l'état à l'instant n a_n : l'action à l'instant n

Le principe de cet algorithme est de mettre à jour itérativement, à la suite de chaque transition (s_n, a_n, s_{n+1}, r_n), la fonction de valeur courante Q_n pour le couple (s_n, a_n).

3.2 Application à notre problème

A ce stade, le but était d'implémenter cette nouvelle méthode en réutilisant ce qui avait été fait auparavant. Pour cela nous avons choisi un modèle réutilisant le même système de capteur. Toutefois, les algorithmes de type processus de décision markovien nécessitent que l'agent soit dans un nombre fini d'états. Nous avons donc dû discrétiser l'ensemble d'état (jusqu'ici continue), en choisissant d'associer à chaque capteur trois états : proche (s_0), moyennement éloigné (s_1) et éloigné (s_2). Ainsi l'agent peut se situer dans 3^n états différents avec n, le nombre de capteurs.

Exemple d'un agent à 2 capteurs (1 d'obstacle et 1 d'objectif) et 3 actions : L'agent peut se situer dans $3^2 = 9$ états distincts et effectuer 3 actions. On a :
— $S = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_0), (s_1, s_1), (s_1, s_2), (s_2, s_0), (s_2, s_1), (s_2, s_2)\}$.
— $A = \{a_0, a_1, a_2\}$

3.2.1 Paramétrage

Dans notre cas, nous avons gardé les $P = 3$ actions de base et utilisé 3 capteurs d'obstacles(en rouge) et 7 d'objectifs (en vert) soit un total de $N = 3^{10} = 59\,049$ états possibles et $Q \in M_{N,P}$. Le nombre de capteurs doit être limité pour ce modèle car l'espace mémoire alloué pour la matrice Q dépend de façon exponentiel du nombre de capteur. pour cette raison il nous a paru plus judicieux de privilégier les capteurs dirigés vers l'avant du robot.

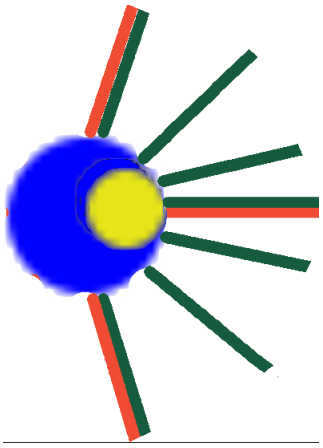


FIGURE 3.1 – Schéma de l'agent et ses capteurs (vert=objectif/ rouge=obstacle)

De plus, nous avons utilisé la fonction de récompense suivante :

$$r_t = \tau \times (l_{plateau} - d_{obj})^2 - (1-\tau) \times (l_{plateau} - d_{obs})^2 + \begin{cases} (-l^3_{plateau} \times \frac{t}{T}) & si\ collisionObstacle = true \\ l^2_{plateau} \times \frac{t}{T} & si\ collisionObjectif = true \\ 0 & sinon \end{cases}$$

avec
- $l_{plateau}$: la longueur du plateau (distance par défault)
- d_{obj} : distance perçu par le capteur d'objectif d'angle 0
- d_{obs} : distance perçu par le capteur d'obstacle d'angle 0
- $\tau \in [0, 1]$: l'importance de la détection d'obstacle et d'objectif
- $t \in [0, T]$: le temps depuis le début de la simulation

Quant aux paramètres liées à l'ajustement de la matrice Q, nous avons utilisé un taux d'apprentissage constant $\alpha = 0.2$ et un coefficient γ grand, $\gamma = 0.999$. En effet, il s'est avéré nécessaire pour notre modèle d'utiliser un coefficient γ très très proche de 1 car notre agent prend des actions toutes les 0.5 secondes, pour simuler la prise d'action continue ; la conséquence est que la probabilité de changer d'état en effectuant une action est faible. Ainsi, on prend $\gamma > 0.999$ pour prendre en compte la récompense espérée sur du suffisamment long terme.

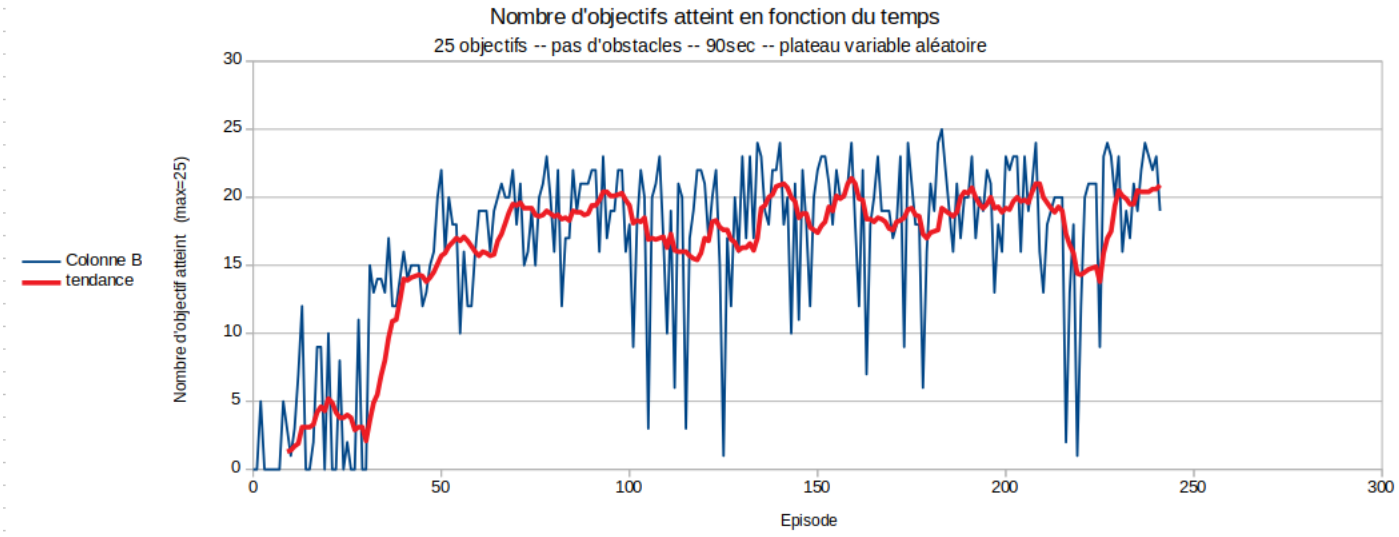
A noter que, les paramètres modifiable peuvent être changés directement dans la classe paramètre.

3.3 Résultats et axes d'amélioration

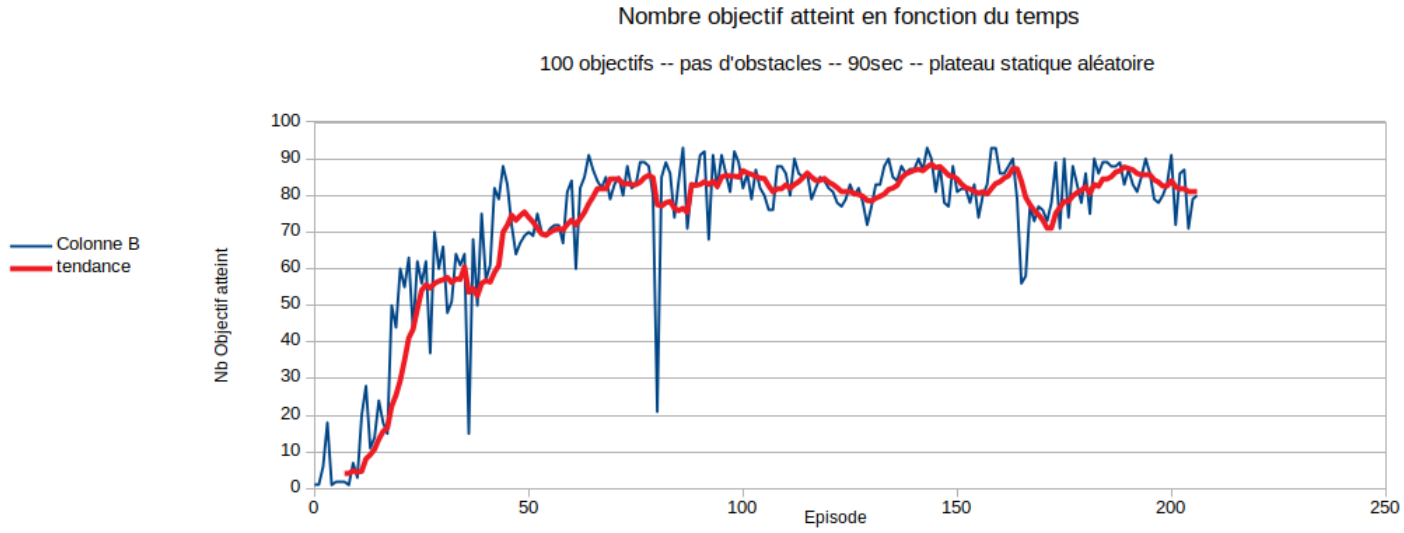
Nous avons testé l'algorithme de Q-learning avec différents paramètres concernant le robot(voir si dessus) mais également différents type d'environnements dans lequel évolue l'agent. Une fois notre robot fixé, nous avons donc changé les paramètres liés à l'environnement d'évolution du robot. Pour choisir ces paramètres il a fallu se poser plusieurs questions :
- Combien d'objectif sur le plateau ?
- Le nombre d'objectif est-il fixe ?
- Présence ou non d'obstacles ?
- Est-ce que le plateau est le même ou change régulièrement ?
- Plateau aléatoire ou non ?
- Combien de temps a le robot pour atteindre les objectifs ?

3.3.1 Résultats de tests

Dans un premier test, nous avons regarder comment notre robot apprend à évolué dans un environnement constitué de 25 objectifs positionnés aléatoirement sur le plateau. Le robot apprend vite à récupérer



Dans un second test, nous avons regarder comment évolue dans un environnement constitué de 100 objectifs positionnés toujours à la même position sur le plateau.



Dans un dernier test, nous avons regarder comment évolue dans un circuit "en zig-zag" constitué de 34 objectifs positionnés toujours de la même façon sur le plateau et de longs murs.

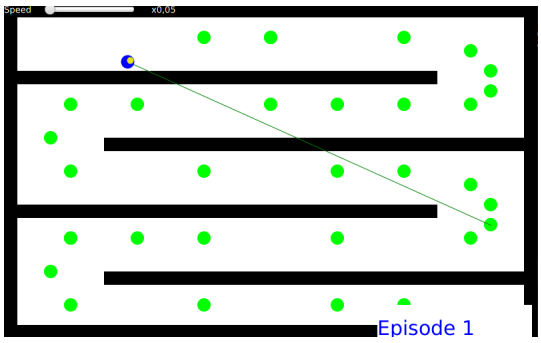
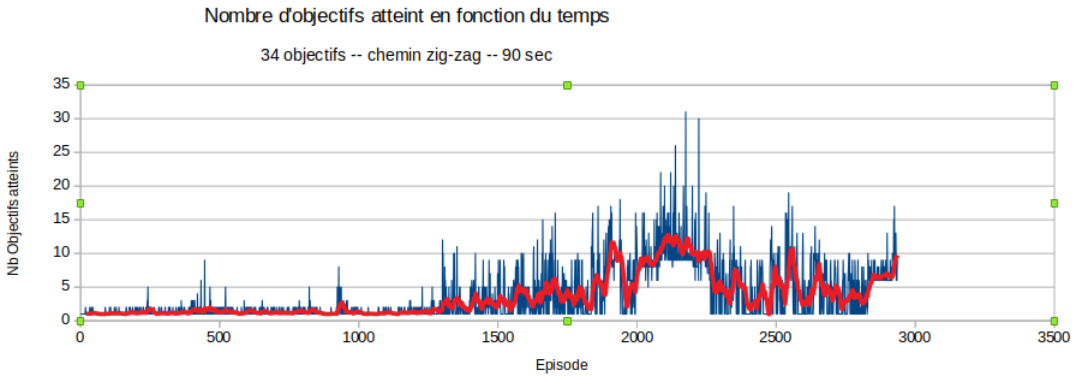


FIGURE 3.2 – Circuit "en zig-zag"



Dans cette dernière configuration, et contrairement aux deux premiers tests, le robot ne parvient pas à élaborer une politique d'action optimale.

3.3.2 Axes d'amélioration - l'approche Deep Q-learning

Le modèle choisi pour récupérer les objectifs est adéquate dans le cas où les obstacles ne sont pas trop nombreux et le nombre d'objectifs suffisamment grand. Dans le cas où le robot doit apprendre à contourner un obstacle d'une taille aléatoire, il n'y arrive pas.

Une possible raison est que le robot n'a qu'un nombre fini d'état et donc le fait d'associer à chacun de ces états une action, n'est pas suffisant à la compréhension d'un environnement plus complexe et donc à l'élaboration d'une fonction plus complexe de prise de décision. Nous aurions pu, pour tenter de comblé ce manque, utiliser un réseau de neurone exactement comme celui de la première partie sur l'algorithme génétique pour estimer la fonction Q. Ainsi le robot n'admettant pas un nombre fini d'états, il aurait peut-être pu apprendre plus de caractéristiques et en conséquent il se serait mieux adapté aux situations comportant des obstacles. Ce type d'algorithme, proche du Q-learning et utilisant un réseau de neurone pour approcher la fonction Q, s'appelle Deep Q-learning. L'appellation "Deep" vient surtout du fait que le réseau de neurone Q utilisé en général est un réseau de neurones profond avec des couches de convolutions. Ces couches permettent de reconnaître des caractéristiques dans les images en entrée, et ainsi de faire le lien entre ces caractéristiques et la récompense perçue par l'agent. L'agent apprendra donc à connaître et reconnaître le résultat de ses actions. La méthode utilisée peut être qualifiée d'apprentissage "bout-à-bout" car l'agent apprend à la fois à reconnaître l'état dans lequel il se trouve ET à choisir une action en adéquate à cet état. La mise à jour des paramètres θ du réseau de neurone Q_θ se fait dans ce cas de la façon suivante :

$$\Delta\theta \leftarrow \alpha[r_n + \gamma \max_a Q_\theta(s_{n+1}, a) - Q_\theta(s_n, a_n)] \times \nabla_\theta Q_\theta(s_n, a_n)$$

α : le taux d'apprentissage
γ : facteur de pondération
$\nabla_\theta Q_\theta$: le gradient de Q_θ

Partie 4

Logiciel

4.1 Explication du logiciel

Le logiciel implémenté permet 3 types de simulations. Deux servent à entraîner des IA et la dernière ne sert qu'à essayer ces dernières sauvegardées préalablement.

Le fonctionnement de l'application est simple. Lors de son lancement, le menu général s'affiche dans une fenêtre. Vous avez alors 3 choix qui s'offrent à vous. Le premier correspond à l'algorithme de Q-Learning, le second à l'algorithme génétique et enfin le dernier à l'algorithme de test d'IA.

Avant chaque simulation d'entraînement, une fenêtre s'affiche vous demandant le nom de la simulation. Ceci servira lors de la sauvegarde des données et de l'IA tout au long de la simulation.

4.2 Diagramme UML

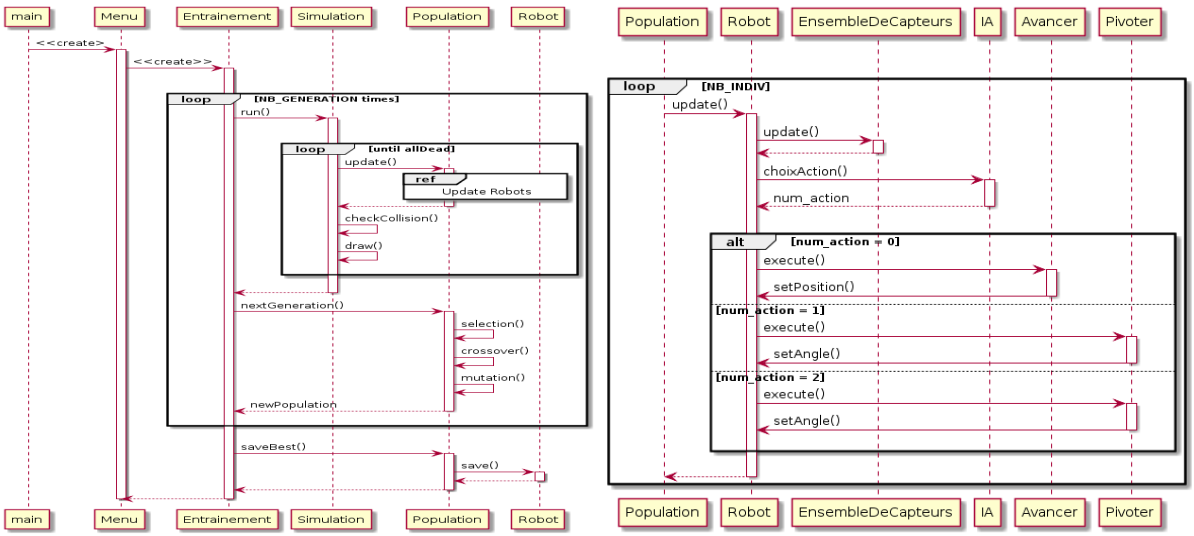


FIGURE 4.1 – Diagrammes de séquence

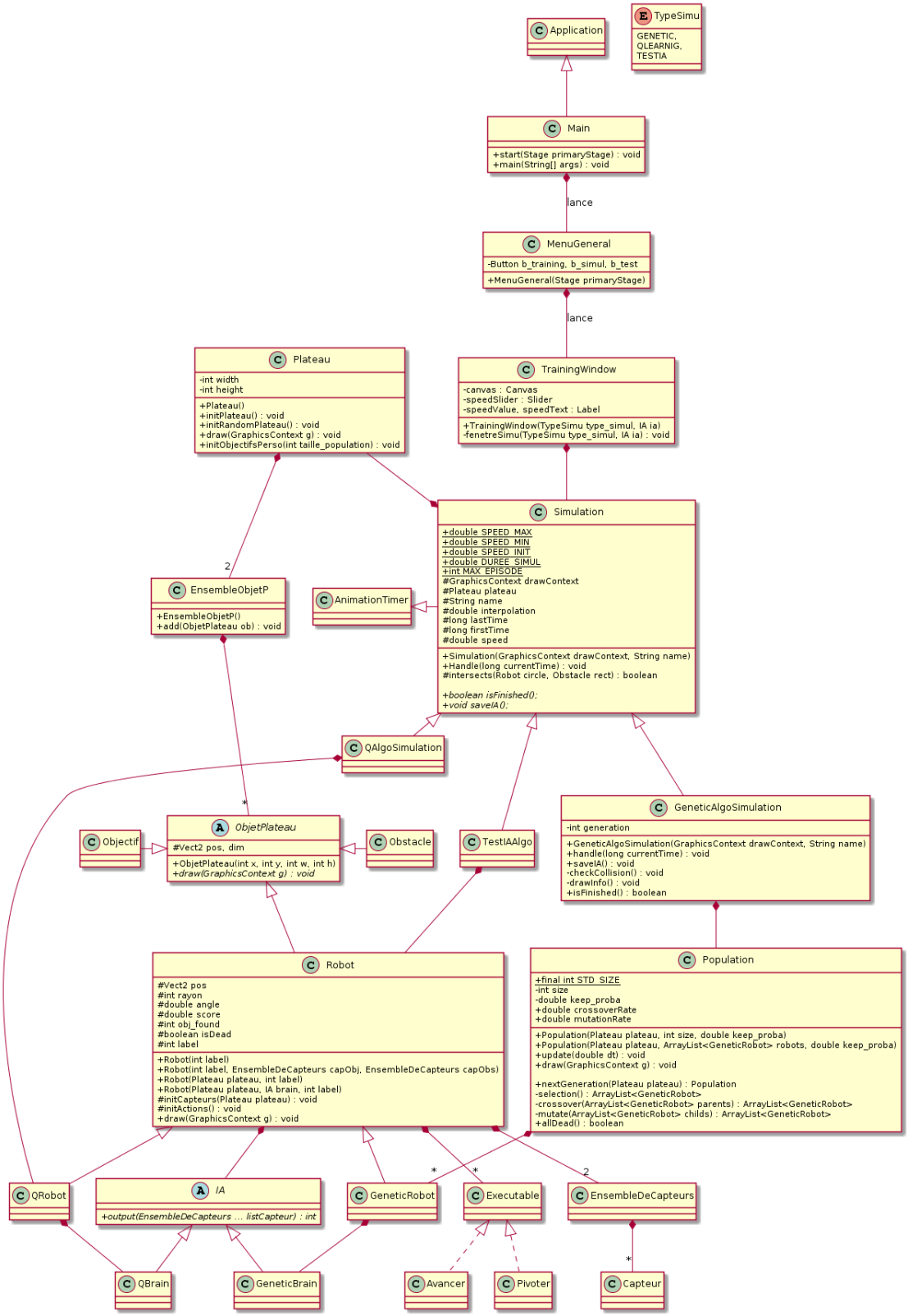


FIGURE 4.2 – Diagramme UML

Partie 5

Conclusion

Ce projet semestriel a été très enrichissant d'un point de vu recherche, dans la mesure où au début du projet, nous ne connaissions pas vraiment les outils que nous allions manipuler et nous ne savions pas si ces outils nous permettraient de répondre à notre problématique.

Ce projet nous a permis, d'une part, d'améliorer nos compétences en programmation objet. Mais il nous a surtout permit de développer de nouvelles compétences dans des domaines qui nous étaient inconnus. En effet, nous avons pu voir ce qu'était un algorithme d'apprentissage automatique à l'aide des algorithmes, de génétique et de Qlearning. Ainsi, nous avons eu l'occasion de manipuler des concepts tels que les réseaux de neurones ou l'intelligence Artificielle. D'autant plus que ces concepts sont de plus en plus manipulés dans le domaine de l'informatique.

Les résultats obtenus sont plutôt positifs, car pour les deux algorithmes nous avons réussi à obtenir des résultats satisfaisants. En effet, dans les deux cas on obtient des agents qui atteignent la quasi totalité des objectifs.

Concernant le choix de l'algorithme. les deux répondent au cahier des charges en terme de résultats. Cependant, on peut voir les limites de l'algorithme de génétique. En effet, on observe rapidement une stagnation des résultats. De plus, le fait d'entraîner une grande population rend les coûts de calculs plus important et l'utilisation des capteurs dans le Qlearning multiplie de façon exponentielle l'espace de stockage. Pour faire face à ces deux problèmes, nous avons du faire des choix de paramètres adéquates. Il s'est avéré que l'algorithme de Qlearning ne nécessitait pas plus de 10 capteurs pour converger vers une solution satisfaisante(bien que "non optimale"). Donc finalement, les résultats obtenus par le Qlearning semblent meilleurs.

D'autre part, pour les deux algorithmes, on peut observer une certaine fluctuation des résultats. En effet, d'un test à l'autre, les résultats différent. C'est pour cela que le choix des paramètres est très important.

Enfin, nous n'avons pas réussi à obtenir de résultats suffisamment satisfaisant lorsque nous ajoutons des obstacles sur la carte. Sauf dans certain cas, où le nombre d'obstacles est très faible, les obstacles sont fixes et le nombre d'objectifs élevé. C'est donc cette partie nous aimerions améliorer par la suite. Notamment, en utilisant un nouvel algorithme tel que celui du Deep Q-Learning.