AI Practice and Technos: Simulation

mémoire :

Le loup, le lapin et la carotte

Léa GREGOIRE, Antonin HIPPOLYTE, Aurélien JEANNEAU, Mbaye DIONGUE, Pierre-Emmanuel REBOURS

Lien URL de notre projet:
https://gitlab.emse.fr/ia-turtle-group/predatory-prey-simulator/-/tree/Add_smart_wolf_rabbit4

I. Description du problème

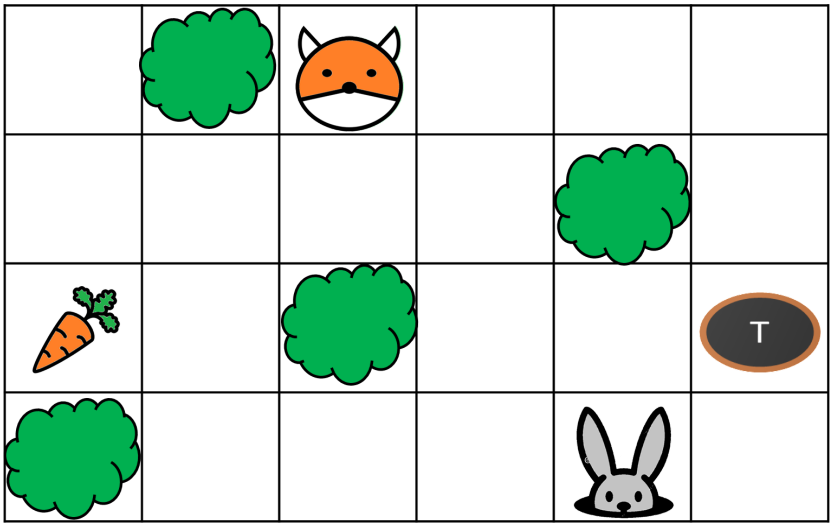
1) Le problème

Le projet consiste à modéliser la chasse d'un lapin par un loup, dans le but d'optimiser la survie du lapin. Cette situation classique d'une proie et d'un chasseur se représente mentalement très facilement avec des animaux mais s'adapte finalement à n'importe quelle situation où un ou plusieurs individus poursuivent un objectif tout en évitant un obstacle mobile.

Dans le modèle, l'objectif du lapin est sa nourriture : un point positionné aléatoirement sur la carte. Comme il s'agit de nourriture, nous avons décidé d'obliger le lapin à s'en procurer régulièrement à l'aide d'une jauge faim. Ainsi le lapin est obligé de se déplacer mais ses déplacements sont conditionnés d'une part à la distance par rapport au loup et d'autre part à la distance de sa nourriture. Pendant ce temps, le loup se déplace à sa recherche et peut donc apercevoir un lapin qui manque de prudence.

Le problème consiste donc à assurer, étant donné le déplacement du loup qui cherche le lapin, le temps de survie du lapin en quête de nourriture dans une grille donnée.

La simulation prend fin quand le loup a mangé le lapin ou lorsque le nombre de pas maximal est dépassé.



2) Les agents

Le lapin est un agent du problème. Le lapin a une certaine portée de vision. Il commence à se déplacer aléatoirement. Puis, en fonction de ce qu'il voit dans son champ de vision, le lapin maximise d'une part sa distance au loup et d'autre part minimise sa distance à la nourriture.

Le loup peut avoir plusieurs types d'intelligences. Dans un premier temps, nous avons pu choisir un loup "aveugle" qui se déplace aléatoirement. Puis nous avons octroyé à ce loup une "vision" qui lui permet d'apercevoir le lapin et de le prendre en chasse.

II. Démarche d'évaluation

1) Choix des scénarios

Les différents scénarios envisagés sont la possibilité d'avoir un loup plus ou moins "intelligent" avec une vision plus ou moins étendue de la grille.

On aurait également voulu un scénario où il pourrait y avoir plusieurs lapins qui peuvent communiquer et donc partager leur visibilité de la grille.

2) Indicateurs

Notre principal indicateur est la fonction "bien-être" du lapin. En effet, cette fonction qui prend en compte le niveau de faim du lapin et de sa distance au loup doit être maximisée. On mesure également le temps écoulé entre le début de la simulation et la fin de la simulation (mort du lapin ou nombre de pas max dépassé).

III. Résultat

Avant de tenir compte de la jauge de famine, le premier résultat que nous avons obtenu est que le lapin a si peur du loup qu'il n'ose pas aller vers la nourriture. Il reste donc bloqué à la même position tandis que le loup se déplace vers le lapin pour venir le manger.
Il finit inexorablement bloqué par peur du loup et puis mangé… Il n'arrive pas à prendre en compte la possibilité de s'approcher du loup pour ensuite s'en éloigner, ce qui serait au final plus utile.
Une solution pour rendre le lapin plus intelligent serait d'utiliser une fonction objective utilisant une heuristique analogue à l'algorithme *minimax.* A chaque étape, puisque le lapin sait que le loup cherche à minimiser la distance les séparant, il lui suffit de calculer quelques coups à l'avance (par exemple 10 pas de temps). Le meilleur déplacement est celui qui conduit à une situation où le compromis entre la distance au loup et la position par rapport à la nourriture est le plus intéressant.

IV. Choix d'implémentation

1. Travaux d'architecture

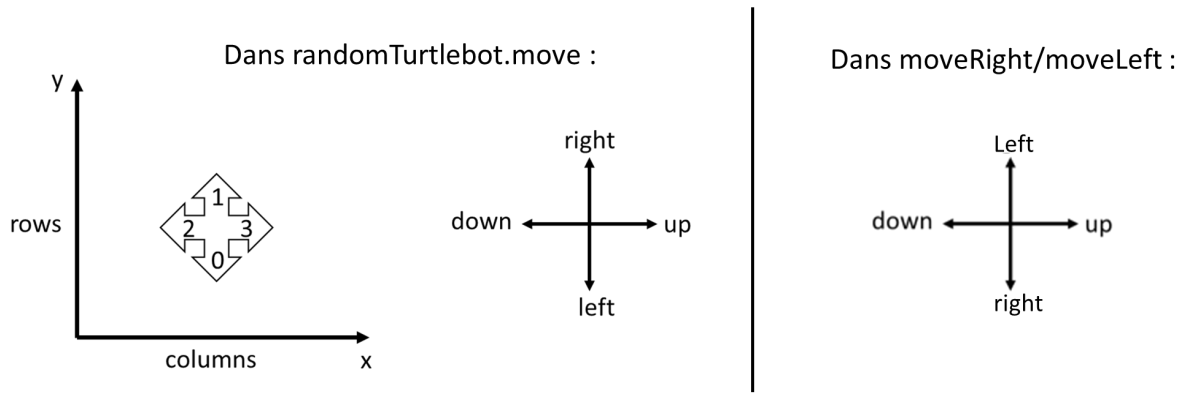
Il est nécessaire de s'adapter à l'architecture déjà présente pour faire les choix judicieux d'implémentations. Nous avons donc pris le temps nécessaire pour comprendre le code existant. Les variables relatives à la simulation sont définies dans le fichier "prop.ini". Ce fichier est ensuite lu par *TestAppli*. Ensuite *TestAppli* publie les informations sur plusieurs "topics" différents, plusieurs sujets. Les fichiers qui ont besoin de ces informations vont "souscrire" à certains sujets pour obtenir les informations nécessaires. L'architecture du projet s'inscrit donc totalement dans une architecture de type "broker". Pour implémenter le loup et le lapin, il est nécessaire que *TestAppli* reconnaisse le type de robot (RabbitTurtleBot et WolfTurtleBot). Nous avons donc ajouté des fonctionnalités dans *TestAppli*.

Cette approche a été suivie dans la branche master du gitlab du projet. Mais, comme il y a eu une mise à jour du projet de base après la première semaine de développement, certains membres de l'équipe n'ont pas pu faire la mise à jour, les branches avec la nouvelle architecture et l'ancienne ont été trop divergentes et nous n'avons pas pu fusionner les branches sur ce travail d'architecture. La version finale que nous vous livrons ne contiendra donc pas ce travail d'architecture. La version finale ne permettra pas de faire fonctionner les turtlebots, toujours à cause de ce souci de fusion de branche.

2. Robot

Dans le programme initial qui nous a été donné, les robots sont des composants : “robot”, les composants étant : robot, vide, but, obstacle, inconnu. Si l’on veut ajouter le composant wolfrobot ou rabbitrobot, cela demande trop de modifications dans le code, et entraîne des bugs. Il est donc nécessaire de s’y prendre autrement. Pour que les deux types de robots puissent communiquer ensemble, il est nécessaire qu’ils se reconnaissent. On a donc pensé qu’il serait plus opportun de faire plusieurs types de robots (robot loup et robot lapin), plutôt que plusieurs types de composants.

À propos du déplacement des robots on a pu constater une incohérence. Dans la classe randomTurtlebot, la méthode move considère que le côté droit est dans les y croissant et le côté gauche dans les y décroissants alors que moveRight et moveLeft inversent ces directions. On remarquera d’ailleurs que ce positionnement est plus cohérent lorsque l’on se positionne au-dessus de la grille.



Dans les prochaines parties, nous allons expliciter certaines classes du package “burger”.

3. Le loup

Notre première version du loup a été très inspirée de *RandomTurtleBot*. Le loup avance de trois cases, puis change de direction (il peut aller en face, à droite ou à gauche, en arrière). Ainsi le loup peut parcourir une large partie du plateau, et avoir plus de chance de croiser le lapin. Pour éviter que le loup ne retourne trop de fois sur ses pas, nous avons changé les probabilités d’orientation en rendant plus probable que le loup se déplace devant lui ou encore sur ses côtés mais qu’ils ne reviennent sur ses pas que très rarement. Ainsi, nous avons ajouté un attribut de classe nommé “changeOfOrientation” qui diminue à chaque action faite par le loup et qui, lorsqu’il est égal à 0, autorise le loup à revenir sur ses pas.

La seconde version du loup est plus intelligente (cf classe *SmartWolfTurtlebot2*). Ce dernier a une vision dont le champ est renseigné par une variable *field_wolf* présent dans la classe *TurtlebotFactory*. S’il repère le lapin, il prendra la direction qui minimise sa distance avec ce dernier et cherchera donc à le manger. C’est la fonction *wellBeing* et *locateRabbit* de la classe *SmartWolfTurtlebot2* qui permet cela.

4. Le lapin

Le lapin hérite de la classe *Turtlebot*, de nombreuses fonctionnalités ont été ajoutées. Le lapin est doté d'un champ de vision qui lui permet de détecter la présence soit du loup soit de la nourriture dans son voisinage proche défini par un carré centré sur le lapin. Ce carré est paramétré par la valeur du field qui donne la longueur du côté du champ de vision. Le lapin aura aussi une fonction Objectif appelée "WellBeing" dans la classe *RabbitTurtleBot*. Cette fonction est essentielle dans le code: en effet, il s'agit de la méthode qui va permettre au lapin de prendre sa décision pour maximiser sa fonction bien-être. Cette fonction prend en compte les positions de la nourriture et du loup si elles ont été localisées par le lapin. Si c'est le cas, alors le lapin va choisir le mouvement qui maximise sa distance au loup et minimise sa distance à la nourriture. Si le lapin n'a pas encore localisé ni la nourriture ni le loup alors il effectue une marche aléatoire.

Nous avons également envisagé la possibilité d'ajouter une capacité au loup à suivre la "trace" du lapin mais cela s'est vite révélé trop complexe. De plus, nous pouvions envisager un terrier dans lequel le lapin serait protégé du loup mais où il ne pourrait pas rester trop longtemps à cause de sa faim, ce qui s'est finalement révélé peu utile comparé à la complexité de sa mise en place.

5. La nourriture

Il existe également un troisième type de robot: la nourriture. Même si à première vue, la nourriture qui constitue le but du lapin ne partage aucune similarité avec un robot, nous avions besoin de pouvoir bouger la nourriture une fois que le lapin l'a atteinte pour simuler le fait qu'il a mangé et que la nourriture apparaît autre part. Nous avions besoin des méthodes setX et setY pour pouvoir positionner une nouvelle cible aléatoirement sur la grille.

6. Indicateur du temps de survie

Nous avons également ajouté dans le "main" un indicateur nous permettant de voir combien de temps notre programme a tourné. Cela nous donnera donc combien de temps le lapin a survécu dans la modélisation. L'objectif est d'avoir ce temps le plus long possible.

7. Fonctionnalité d'arrêt

La condition d'arrêt a également été modifiée. En effet, dans le code fourni, la simulation s'arrêtait lorsque tous les robots avaient atteint leur objectif. Dans notre cas, la simulation doit s'arrêter dès lors que le loup entre en contact avec le lapin. Ainsi, les classes *TurtlebotFactory*, *SmartWolfTurtlebot2* et *Goal* ont été adaptées en conséquence.