TP n°2 : Inférence et problème de l'ancrage

UE Modélisation et Simulation Multi-Agents

LY Jean-Baptiste
Sorbonne Université
20 décembre 2020

En vous appuyant sur le cours et l'article de *Langley et al* fourni la semaine précédente, rappelez en un paragraphe ce qu'est le problème de l'ancrage puis quels sont les avantages et inconvénients des principales approches d'architectures cognitives existantes.

Le problème de l'ancrage consiste à l'attribution de symboles, de formes, d'ensembles de pixels, à des concepts ou des sens. Le système manipule des informations, mais qui sont dénués de sens pour lui. Ce n'est que l'utilisateur qui donne sens aux résultats du système.
Ainsi le problème de l'ancrage pose différentes problématiques, comme :
- Comment stocker les informations ?
- Comment choisir ce qui doit être conservé, oublié ?

Par exemple, DeepBlue et AlphaGo ne savent pas jouer. Leur objectif est de maximiser par exemple les gains, et/ou minimiser les pertes, et un chatbot ne comprend pas les mots qu'il utilise, il fait seulement des statistiques dans le but de choisir les bons mots. Il donne seulement l'impression à l'utilisateur qu'il comprend des choses, alors que cela n'est pas le cas.
Cet objectif du système, de minimisation et/ou de maximisation pose plusieurs problèmes. Par exemple, l'utilisateur humain donne pour objectif à un robot de maximiser sa distance parcourue et en minimisant ses coûts, en espérant ainsi que le robot aille le plus loin possible. Résultats : le robot tourne sur lui-même, ce qui valide les objectifs tels quels qu'on a donnés au robot : maximiser la distance parcourue et minimiser les coûts, alors qu'on ne s'attendait pas à de tels résultats, c'est un comportement émergeant. Cela est dû à nos biais causés par notre représentation humaine des choses.

Il existe plusieurs approches d'architectures cognitives top-down existantes :
- SOAR (State, Operator And Result) : elle utilise l'apprentissage par renforcement et ne fonctonne pas dans des environements non contrôlés. Ainsi ce système est lent à apprendre et reste bloquer sur le problème de l'ancrage, car il est compliqué de lier l'action et l'environnement. L'inférence est savoir où aller.
- CBRL : cette méthode est basée sur l'apprentissage par cas. On observe la situation et on fait une action en fonction du cas, ensuite on sauvegarde les conséquences. Au fur et à mesure, on accumule de plus en plus de cas. L'inférence est le choix de l'action à déclencher.

D'un autre côté, nous avons les approches bottom-up (subsomption et neuromimétiques) qui n'ont pas besoin d'une représentation explicite de l'environnement et laissent un grand choix de structures de données (pas de structure type). En revanche, elles ont dû mal avec les tâches de haut niveau d'abstraction.

De manière génréale, ces approches classiques sont efficaces lorsque nous définissons les règles à la main, mais des problèmes de représentation, de généralisation et d'abstraction se posent.

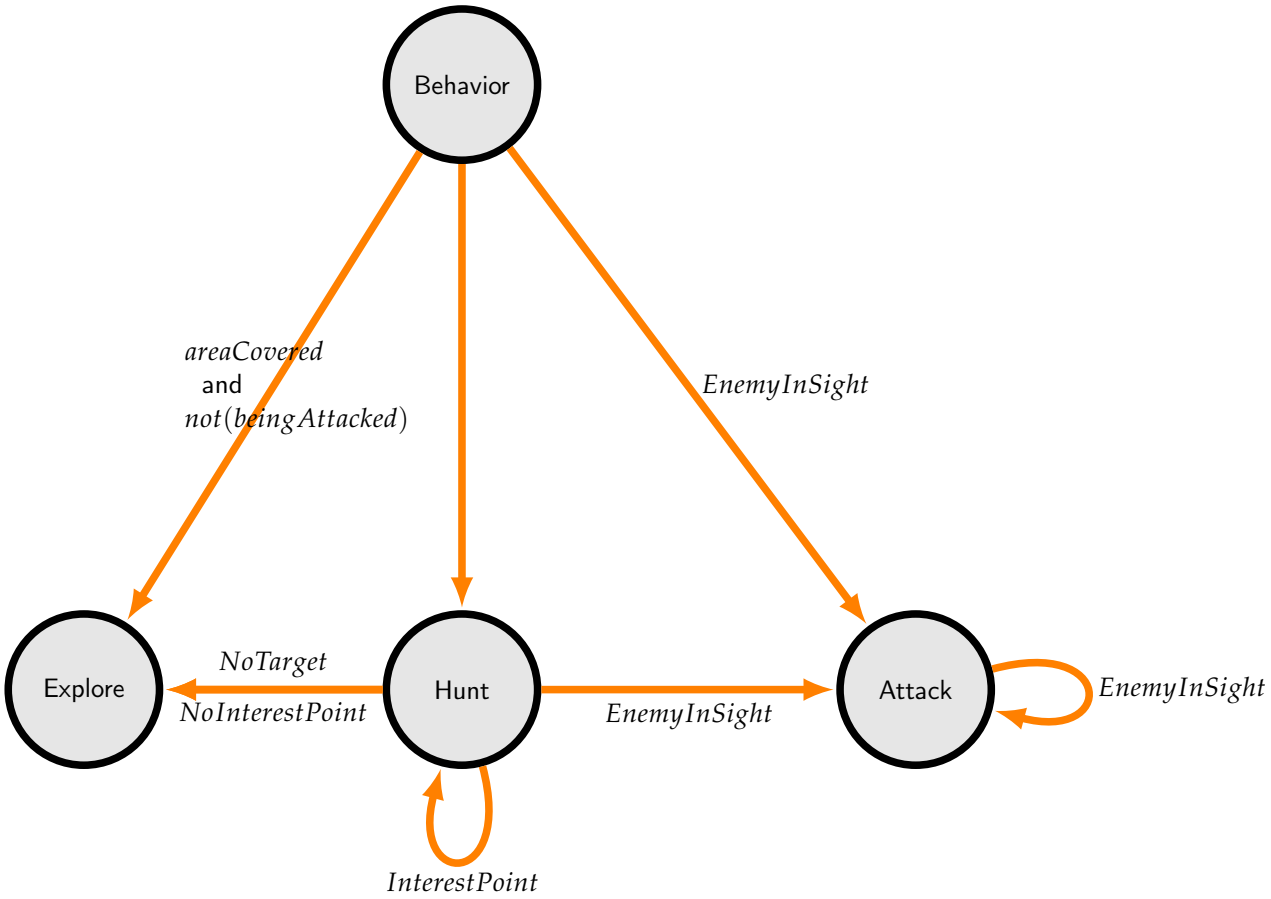
Etudiez et comprenez le raisonnement de l'agent Dummy et de l'agent dit "intelligent".

Quelle est la stratégie de l'agent Dummy, de combien de comportements est-elle constituée ?

La stratégie de l'agent Dummy est constituée de 2 comportements :
- L'attaque
- Le déplacement aléatoire

À chaque tick l'agent Dummy prend le premier adversaire visible pour lui et tire sur ce dernier. Ensuite il essaye de se déplacer aléatoirement : si il n'a pas changé de destination aléatoire depuis un certain temps, alors il change de destination aléatoire et se dirige vers celle-ci, sinon il continue à se diriger vers sa dernière destination aléatoire choisie.

Représentez la stratégie de l'agent dit "intelligent" sous la forme d'un automate.



À chaque tick, tout d'abord, on commence dans l'état *Behavior*.
Si le champ de vision de l'agent couvre moins de 60% de la carte et si il n'est pas attaqué, alors l'agent va explorer.
Si l'agent est en bonne santé, est attaqué, et ne voit pas d'ennemi, alors il commence à chasser, ou si l'agent n'est pas attaqué, son champ de vision offensif couvre plus de 60% de la carte, et de même pour son champ de vision défensif, alors il commence aussi à chasser.
Si l'agent voit un ennemi, alors il passe à l'attaque.
Si il est en train de chasser, et s'il voit un ennemi, alors il passe en mode attaque. Sinon si il trouve un point d'intérêt, alors il reste en mode chasse. Sinon dans le dernier cas, il retourne dans le behavior explore.
Si il est en mode attaque, tant qu'il voit l'ennemi, il reste en mode attaque.
Toutes les autres transitions mènent vers Behavior.

Générez un premier comportement pour votre agent. (Quelques commentaires sur le code)

En vous inspirant du comportement de l'agent "intelligent" existant, créer un nouvel agent dont le comportement le conduira à toujours privilégier lors de ses déplacements l'endroit le plus haut perçu dans son champ de vision, avec une probabilité de choisir une autre destination égale à 9%.

```
protected void onTick(){
        if (target == null && !setTarget()){
                randomMove();
                return;
        }

        if (agent.getCurrentPosition().distance(target)
            < AbstractAgent.NEIGHBORHOOD_DISTANCE){
                Vector3f nei;
                double rand = Math.random();
                if (rand <= 0.91) {
                        nei = findHighestNeighbor();
                }
                else {
                    nei = null;
                }
                if(nei != null){
                        target = nei;
                        agent.moveTo(target);
                }else{
                        addInterestPoint();
                        target = null;
                }
        }
}
```

Pour l'instant, j'ai préféré garder les anciens fichiers et je les ai adaptés au nouveau behavior créé : "HighestExploreBehavior", qui est une copie de "ExploreBehavior" mais avec sa méthode onTick() modifiée.
Néanmoins il est prévu de créér de nouveaux fichiers "Agent" en plus de celui d'agent de base, dans la suite du projet.

Modifiez le code de façon à sauvegarder la situation courante en cas de victoire/défaite dans le répertoire ressources/learningBase/(victory/defeat)

Voir réponse suivante.

Modifiez le code de façon à ce que votre agent sauvegarde en plus (dans un répertoire différent) la situation courante à chaque fois qu'il est touché ou perçoit son adversaire.

Pour plus de facilités, j'ai modifié le modifié le fichier NewEnv.java dans le répertoire src/env/jme car ces consignes n'ont pas d'impact sur le comportement de l'agent.