Architecture Cognitive pour le Transfert de Connaissances

Laura Greige
Pablo Petit

Enseignants :
Cédric Herpson
Vincent Corruble

2016

Table des matières

Introduction **1**

1 Exploration de l'environnement **1**
1.1 Amélioration de l'environnement 1
1.2 Exploration 2

2 Mécanisme d'inférence **2**
2.1 Représentation des connaissances 2
2.1.1 Points d'intérêts 2
2.1.2 Situation 3
2.2 Raisonnement 4
2.3 Raisonnement en Prolog 5

3 Généralisation et apprentissage **6**

4 Conclusion et amélioration **6**

Conclusion et amélioration **6**

Table des figures

1 Visualisation des points d'intérêts dans l'environnement 2
2 Algorithme Golden Section Spiral (itérations $i = 7, 23, 58, 106$ et 208) 3

Introduction

Dans ce projet, nous considérons des agents cognitifs qui sont capable d'évoluer au sein d'un environnement partiellement inconnus ou dynamiques. Notre agent sera face à un autre agent piloté par une IA simpliste et l'objectif de chacun d'eux sera de remporter le duel, c'est à dire, éliminer leur opposant. Pour garantir l'atteinte d'un objectif donné, il est nécessaire de pouvoir faire évoluer les comportements des agents au cours du temps, ce qui est réalisable en développant les connaissances des agents. Les agents seront donc doté de différents comportements qui leurs permettront d'explorer l'environnement, mettre à jour leurs connaissances, et chasser, suivre et attaquer l'ennemi. Dans un premier temps, les connaissances seront mise-à-jour « manuellement ». Ensuite, les agents seront dotés de capacités d'adaptation par le biais de processus de raisonnement et de mécanisme d'apprentissage.

Pour l'implémentation du code, on utilisera la plateforme de programmation multi-agent *JADE* implémentée en Java, le langage de programmation logique *Prolog* comme mécanisme d'inférence, la suite de logiciels d'apprentissage automatique *Weka* écrite en Java pour les algorithmes d'apprentissage et enfin, *jMonkeyEngine* comme moteur graphique et support de l'environnement. Dans la suite, nous présenterons les différents comportements et stratégies implémentés.

1 Exploration de l'environnement

1.1 Amélioration de l'environnement

Suite à la découverte d'un certain nombre d'incohérences qui posaient des problèmes de compilation ou qui diminuait le réalisme de la simulation, nous avons pris la liberté de réécrire les classes `Environement`, `Situation` ainsi qu'`AbstractAgent`.

Nous avons remarqué que dans la fonction d'observation, l'observation du terrain se faisait en tirant des rayons parallèles au terrain, en suivant la direction de la caméra. Cela a pour effet de limiter très fortement l'espace de vision de l'agent qui ne correspond alors plus qu'à un rectangle de largeur correspondante à la largeur du cadre de la caméra. L'agent ne peut donc pas voir sur les côtés, ni au dessus ou en dessous de lui-même. Aussi, cela rend le débuggage compliqué car la vision affiché ne correspond pas à la vision réelle de l'agent. Pour pallier à ce problème, nous avons écrit une fonction de `RayCasting` sphérique (sphereCast) qui vas projeter des rayons dans toutes les directions en partant de l'agent, puis récupérer les points d'impact et les filtrer en fonction de la distance et de l'angle maximum fournit en paramètre. La méthode de génération des rayons est basé sur la méthode de la spirale d'or qui permet d'obtenir un nombre N de points équirépartis sur une sphère, ce qui nous permet de régler la précision du sphereCast. D'autre part, nous avons aussi changé la probabilité de faire mouche lors d'un tir, qui ne considérait aucun paramètre initialement. Celle-ci est maintenant dépendante de la distance de tir et de la différence de hauteur entre le tireur et la cible (un tir fait d'au dessus est plus facile).

La `Situation` renvoyée a elle aussi été modifiée de manière à être plus riche, plus claire, et plus adaptée à nos besoins. Nous y avons notamment ajouté des informations sur la base de connaissances de l'agent.

Enfin, dans la classe `AbstractAgent`, a été simplement réécrite pour servir de façade adaptée au nouvel environnement, et pour fournir certaines fonctions de confort.

1.2 Exploration

L'environnement dans lequel se déplacent les agents est constitué de montagnes d'altitudes différentes. Dans le projet, on suppose que les agents ont une connaissance partielle du terrain. Au déploiement, notre agent est tout d'abord placé aléatoirement dans l'environnement et possède deux attributs, `depth-of-field` et `angle-of-view`, qui représentent respectivement le rayon de vision et l'angle de vision. Afin d'explorer l'environnement, l'agent est doté d'un comportement `exploration` qui, comme son nom l'indique, permettra à l'agent de se déplacer et d'explorer l'environnement afin de trouver des « points d'intérêts ». Ceux-ci se déclinent sous deux formes, offensifs et défensifs et lui serviront de position d'attaque ou de lieu de refuge.

FIGURE 1 – Visualisation des points d'intérêts dans l'environnement

Le comportement par défaut de l'agent est l'exploration. À partir de sa position courante, il observe autour de lui afin de trouver un point d'intérêt à ajouter à sa base de données. Ensuite, il se déplace jusqu'à ce point, l'évalue, puis l'enregistre. Pour évaluer le point d'intérêt, l'agent tire un sphereCast depuis ce point et somme les distances parcourues par les rayons. Plus cette valeur sera grande, plus ce point offre un champ de vision dégagé. Un point offensif sera d'autant mieux valué que cette valeur est grande, à l'inverse d'un point défensif, pour lequel on cherche à être le moins visible possible. Le choix entre point d'intérêt défensif ou offensif est fait au préalable, et ce choix sera expliqué dans la section suivante. Pour une meilleure visualisation des points d'intérêts dans l'environnement, les points offensifs seront représentés en rouge et les points défensifs en bleu.

2 Mécanisme d'inférence

2.1 Représentation des connaissances

2.1.1 Points d'intérêts

Chaque agent sera équipé de deux attributs, une liste de points d'intérêts offensifs `offPoints` et une liste de points d'intérêts défensifs `defPoints`. Comme précisé dans la partie précédente, lorsqu'un agent adopte le comportement `exploration`, il ajoute les points intéressants dans la liste de points correspondante. Ces deux listes seront utilisées par la suite lorsque l'agent adoptera l'un des comportements `attaque` ou `retrait`.

Pour trouver un point d'intérêt, nous avons implémenté une méthode appelée `goldenSphereCast`. Elle est basée sur l'algorithme **Golden Section Spiral** qui permet d'obtenir des points équitables appartenant à la sphère.

Algorithme 1 : Golden Section Spiral

pour i *de* 1 *à* N **faire**
  $points = []$
  $inc = \pi \times 3 - \sqrt{5}$
  $s = \frac{2}{N}$
  pour $k \in \{0, ..., N-1\}$ **faire**
    $y = k \times s - 1 + \frac{s}{2}$
    $r = \sqrt{1 - y^2}$
    $\phi = k \times inc$
    $points.append(\cos(\phi) \times r, y, \sin(\phi) \times r)$
  fin
  retourner *points*
fin

Une fois qu'on a tous nos points sur la sphère, on les filtre pour obtenir seul les points qui sont bien dans le rayon et l'angle de vision de l'agent. Les méthodes `getHighest` et `getLowest` nous permettent ensuite de trouver le meilleur point offensif parmi les points filtrés ainsi que le meilleur point défensif. Voici une illustration de la procédure Golden Section Spiral après N itérations.

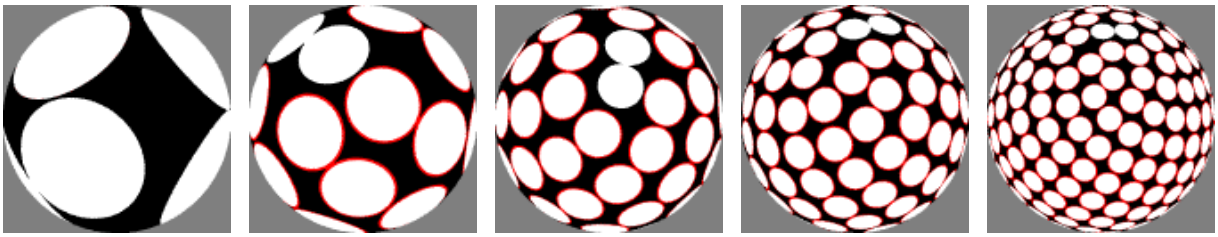

FIGURE 2 – Algorithme Golden Section Spiral (itérations $i = 7, 23, 58, 106$ et 208)

On affecte à la liste des points offensifs, ainsi qu'à celle des points défensifs, une valuation e qu'on utilisera par la suite afin de déterminer si le comportement d'exploration et de recherche de points d'intérêts est toujours nécessaire.

2.1.2 Situation

Nous représentons la situation actuelle de l'agent par la classe `Situation`. Elle prend différents attributs qui nous donne plus d'informations sur la situation actuelle de l'agent par rapport à :

- sa base de connaissance : la taille de ses listes `offPoints` et `defPoints`, leurs valuations et valeurs de dispersion respectives ainsi que la valeur du champ de vision

- sa position courante : son altitude, l'altitude moyenne, l'altitude minimale et l'altitude maximale
- autres paramètres : sa dernière action exécutée, sa vie restante, la durée écoulée depuis le dernier instant auquel il s'est fait attaqué, un booléen indiquant si l'agent repère un ennemi et enfin la probabilité d'impact de tir

Pour chaque situation différente, un raisonnement différent sera suivi. Selon la situation actuelle de l'agent, une action parmi les actions définies dans la suite sera donc exécutée.

2.2 Raisonnement

Pour le processus de raisonnement, nous utiliserons le langage de programmation logique *Prolog* comme mécanisme d'inférence. On alimente donc une base de connaissances de faits et de règles afin de faire des requêtes à cette base et choisir l'action à exécuter selon notre situation courante. Les actions seront implémentées en Java. Deux différents cas se présentent, le cas où un agent se fait attaquer, et le cas sans attaque.

En cas d'attaque

Si l'agent se fait attaquer, l'action intuitif à exécuter serait de se défendre (battre en retraite), ou d'attaquer. Ainsi, quatre comportements ont été implémentés.

— `retreatBehavior` est le comportement de retrait. L'agent exécute ce comportement lorsqu'il se fait attaquer et que sa vie restante n'est pas assez importante. Il va donc chercher dans sa liste de points défensifs la meilleure position pour se protéger d'éventuels attaques supplémentaires.

— `huntBehavior` est le comportement de chasse. Lorsqu'un agent se fait attaquer par un ennemi qui n'est pas dans son champ visuel, et qu'il a une vie restante importante, ce comportement sera exécuté. Il permet à l'agent de se déplacer de point d'intérêt en point d'intérêt afin d'observer autour de lui et trouver ennemi pour le suivre ou le détruire. Pour trouver le prochain point d'intérêt à visiter on calcule la valuation v des points contenues dans notre liste tel que :

$$v(\texttt{point}) = \texttt{MapWidth} - \texttt{distance} + 5 \times \max(\frac{\texttt{time} - \texttt{point.lastVisit}}{1000}, 0) \tag{1}$$

Lorsque l'agent repère un ennemi, il lance le comportement `attackBehavior`.

— `attackBehavior` est le comportement de suivi et d'attaque. Lorsque ce comportement est exécuté, si un ennemi est dans son champ visuel, que sa vie restante est grande et que sa probabilité d'impact de tir est importante, il tire. Sinon, il le suit. La probabilité d'impact de tire dépend de la distance séparant l'agent de l'ennemi et de la différence d'altitude entre sa position et celle de sa cible (tirer d'une altitude plus haute que sa cible aura un plus grand impact que tirer d'une altitude plus basse).

Cas sans attaque

Si aucune attaque n'a lieu, l'agent explore l'environnement pour, soit chercher des points d'intérêts, soit se déplacer afin de trouver un ennemi dans l'environnement.

— `explorerBehavior` est le comportement d'exploration de points d'intérêts. Il permet à l'agent d'exécuter sa méthode d'exploration en cherchant les points d'intérêts. Le type de points à rechercher (offensif ou défensif) est décidé par Prolog par la clause `explore_points`.

— `huntBehavior` est le comportement de chasse décrit précédemment. En revanche, s'il n'y a pas d'attaque, ce comportement sera exécuté lorsque l'agent a une connaissance suffisante de l'environnement (en fonction des valuations des listes `offPoints` et `defPoints`, expliqué dans le paragraphe suivant). Continuer l'exploration de la carte serait inutile dans ce cas, l'agent se déplace donc de point d'intérêt en point d'intérêt jusqu'à trouver une cible.

2.3 Raisonnement en Prolog

Quelques fonctions implémentées en Prolog qu'on utilisera pour vérifier les clauses décrites par la suite :

— `explore_points(X,Y)` : dans l'idéal, on voudrait avoir deux fois plus de points d'intérêts offensifs que de points d'intérêts défensifs. Cette fonction nous permet donc de savoir quel type de points d'intérêts l'agent devrait chercher. Elle compare donc la taille des listes `offPoints` et `defPoints` de l'agents et retourne `true` si `length(offPoints)` $\geq 2 \times$ `length(defPoints)`.

— `being_attacked(T)` : cette fonction prend en paramètre la durée écoulée depuis le dernier instant auquel il s'est fait attaqué et retourne `true` si `T` $< \Delta t$, `false` sinon.

— `areaCovered(Radius, Size, MapWidth)` : dans le cas où on voudrait savoir s'il est toujours nécessaire d'explorer la carte ou non, on calcule la surface couverte par les points d'intérêts. En effet, l'agent possède un rayon de vision `depth-of-field`, ce qui signifie qu'à partir d'un point p, on peut supposer que l'agent a une connaissance sur toute la surface $s(p) = 2\pi \times$ `depth-of-field`. Sachant que les points d'intérêts sont choisis tel qu'il n'y ait pas d'intersection entre les surfaces $s(p)$ pour tout $p \in$ liste de points d'intérêts, on peut calculer la surface totale couverte par ces points.

— `inGoodHealth(life)` : cette fonction retourne `true` si l'agent est en bonne santé, `false` sinon.

— `shotImpact(P)` : cette fonction indique si la probabilité d'impact de tir `P` est supérieure à un certain δ, et renvoie `true` dans le cas échéant.

Pour l'exploration et la recherche de points d'intérêts, on évalue d'abord l'état de notre base de connaissance. Si nos points d'intérêts couvre une surface satisfaisante de l'environnement, continuer la recherche de points serait inutile. Dans le cas où nos points ne couvre pas l'ensemble du terrain, on voudrait savoir quel type de points il faudrait rechercher. De plus, l'exploration de points est exécutée lorsque l'agent ne se fait pas attaquer par un ennemi. On exécute alors l'exploration de points, lorsque la clause suivante est vérifiée :

$$\neg\texttt{being\_attacked(T)} \wedge \texttt{areaCovered(R, OSize, W)} < 20$$

Ensuite, pour savoir quel type de points rechercher on appelle la fonction `explore_points`.

Pour chasser un ennemi, on a deux situations possibles. Soit, l'agent ne se fait pas attaquer et qu'il a une connaissance suffisante de l'environnement, soit, il a une vie restante importante, il se fait attaqué et

qu'aucun ennemi ne se trouve dans son champ de vision.

$$(\neg\texttt{being\_attacked(T)} \wedge \neg(\texttt{areaCovered(R, OSize, W)} < 20) \wedge \neg(\texttt{areaCovered(R, DSize, W)} < 20))$$
$$\vee \ (\texttt{inGoodHealth(Life)} \wedge \texttt{being\_attacked(T)} \wedge \neg\texttt{EnemyInSight})$$

Pour le suivre, l'ennemi devrait être dans le rayon et l'angle de vision de l'agent, c'est-à-dire si `EnemyInSight` est vrai, nous pouvons appeler le comportement `attackBehavior`. Ensuite pour tirer, on vérifie que la probabilité d'impact de tir de l'agent est supérieure à δ et que l'ennemi est toujours dans le rayon et l'angle de vision de l'agent. Celle-ci devrait donc vérifier la clause :

$$\neg\texttt{shotImpact(P)} \wedge \texttt{EnemyInSight}$$

Et enfin, lorsque la vie de l'agent est basse, l'agent va battre en retraite. Donc, si l'agent vérifie la clause suivante il exécutera le comportement `retreatBehavior`.

$$\neg\texttt{inGoodHealth(Life)} \wedge \texttt{being\_attacked(T)}$$

3 Généralisation et apprentissage

Nous avons jusque là, construit « à la main » nos situations intéressantes. Pour chaque situation, une différente action qui permettrait à l'agent d'atteindre son objectif (survivre et battre son adversaire) peut être exécutée. Dans cette partie, nous allons concevoir un agent adaptatif. Un agent adaptatif est un agent qui est en mesure de réagir face à une situation nouvelle et un environnement qui change, mais aussi de choisir par lui-même le comportement le plus approprié. L'agent devrait donc être capable de percevoir son environnement et d'évaluer sa situation courante, de représenter et de mettre à jour ses connaissances et enfin, d'agir et de prendre des décisions.

Malheureusement, par contrainte de temps, nous n'avons pas pu effectuer les tests nécessaire pour réaliser un arbre de décision sous `Weka`. Ces tests seront réalisés avant le jour de la soutenance.

4 Conclusion et amélioration

Plusieurs améliorations sont possibles, autant du point de vue de l'environnement et des interactions que peuvent avoir les agents avec lui, que du point de vue du comportement des agents.

D'abord du côté de l'environnement, si nous avons changé la probabilité de faire mouche lors d'un tir, pour qu'elle dépende de plusieurs paramètres, il serait aussi intéressant de faire de même vis-à-vis des dégâts infligés lors d'un tir. La combinaison de ces deux facteurs donnerait un avantage au combat à un agent bien placé et donc un fort intérêt à rechercher des positions de tirs intéressantes avant de passer à l'attaque.

Il serait aussi pertinent de permettre aux agents de récupérer leurs vies après avoir essuyé des tirs, par exemple en s'immobilisant ou en attendant un certain moment. Les agents auraient alors la possibilité de battre en retraite s'ils se sentent en position de faiblesse pour ensuite reprendre le combat dans de meilleures conditions. Cela ajouterait aussi de l'intérêt à l'exploration de la carte grâce à la nécessité de trouver des points de refuges. Dans le même esprit, ont pourrait imaginer que nos agents puissent connaître la direction dans laquelle regarde un ennemi quand ils en repèrent un. Ils pourraient alors chercher à l'attaquer par derrière pour pouvoir tirer un maximum de coups avant de se faire détecter.

Actuellement lorsque nos agents cherchent à ce rendre à un endroit, ils se déplacent en ligne droite. Or, ils ont tout intérêt à minimiser leur chance de se faire repérer pour ne pas se faire tirer dessus. Cela peut être réalisé en favorisant par exemple les vallées et crevasses et probablement en évitant le centre de la carte qui est plus fréquenté qu'ailleurs. La recherche de chemin fait donc parti des axes d'améliorations.

Si nos agents voient un ennemi, ils vont soit le suivre pour le détruire, soit fuir en s'éloignant au maximum. Dans les deux cas, seule la dernière position de l'agent ennemi est retenue. Il serait possible d'imaginer un mécanisme d'inférence permettant de faire des suppositions sur la localisation de l'ennemi en fonction du temps écoulé et des dernières positions connues afin de le retrouver plus facilement si nécessaire.

Vous pouvez trouver le code du projet ainsi que le rapport dans notre dépôt Git a l'adresse suivante : `http://github.com/PabloPetit/Duel`