Bataille d'Intelligence artificielle

Introduction

Ce document regroupe les différentes problématiques à l'intégration d'un arbre de décision dans l'intelligence artificielle du projet C++ Bataille d'armée.

Problématiques

1. Extracteurs

a. Étude du cas

Un extracteur permet de filtrer un ensemble d'unité de retourner une valeur. Cette valeur peut être de plusieurs types différents :

- Une primitive. Entier, flottant, …
- Un point 2D
- Une unité
- Un ensemble d'unité

Il doit être possible de combiner plusieurs filtres.

b. Propositions

Une classe IFilter template sur le type de retour de l'opérateur Fonctor. Cette classe mère contient en attribut l'unité courante de l'IA ainsi qu'une référence (Pointeur ou shared_ptr) sur les deux armées.

La problématique vient de l'effet en cascade des filtres. Il faut pouvoir combiner les filtres pour permettre un filtrage plus précis. Pour ce faire chaque Filtre aura un pointeur sur un filtre pouvant être lié à lui. Par exemple certain filtre se baseront sur une unité, ils ne pourront donc pas avoir de filtre en fils qui retourne un vecteur ou une value.

2. Arbre de décision

a. Étude du cas

L'arbre de décision est composé de deux sortes de nœud.

Nœud Action

Le nœud action renvoi l'action que l'unité devra exécuter. Elle paramètre l'action à effectuer et la retourne.

Nœud de décision

Le nœud de décision à deux fils, chaque fils peut être soit du type Nœud de décision soit de type Nœud action. Le nœud de décision est aussi composé d'un prédicat, il permet de choisir de récupérer la valeur du fils de gauche ou du fils de droite

b. Propositions

La class INode définit la méthode getValue qui retourne une Action (unique_ptr<Action>).

Chaque nœud Action implémente la méthode getValue et retourne une action spécifique et préalablement paramétré.

Chaque nœud de décision contient un fils gauche et un fils droit. Le prédicat peut être défini de deux façons. La première est de fournir un prédicat (std ::function<bool>) au nœud de décision, cette solution est plus modulaire car il n'y a pas besoin de créer des classes différentes pour chaque décision pouvant être prise. Seulement il faut alors bien définir les entrées/sortie que prendre ce prédicat. La seconde solution est de définir une méthode virtuel pur au nœud de décision, il suffira alors de créer des classes qui dériveront des nœuds de décision et définiront ce prédicat.

Les entités passées à l'IA sont passé aux différents nœuds en paramètre de fonction la méthode getValue. Il est possible sinon de paramétrer le nœud à l'aide d'une méthode setup qui settera des attributs interne à la classe avant d'appelé la méthode getValue.

L'arbre de décision peut être utilisé de deux façons différentes. La première est qu'un arbre correspond à une unité, chaque unité à son propre comportement. L'autre solution est le partage de l'arbre cette solution permet d'avoir le même comportement pour un groupe d'unité donnée (Groupe d'unité, chaque classe a son comportement, …)

3. Arbre ⇔ Code

a. Étude du cas

La construction de l'arbre se fait à l'aide d'une chaîne de caractère. Cette chaîne décrit chaque maillon qui compose l'arbre de décision.

b. Propositions

La construction des différents éléments de l'arbre se fera à l'aide de Factory. La Factory principale sera la Treefactory, elle permettra à partir d'une chaine de caractère de construire l'arbre correspondant. Chaque partie de l'arbre aura elle aussi une Factory.

Ces Factory ont aussi le traitement inverse, elles peuvent reconstruire la chaîne de caractère correspondant à un arbre donné.

4. Aléatoire

a. Étude du cas

Il doit être possible de générer un arbre de décision complet aléatoirement.

b. Propositions

Les Factory cité plus haut permettront de générer aléatoirement un arbre de décision. Avec cette solution l'utilisation et la gestion des arbres de décision sont déparés.

5. Sous-Ensemble

a. Étude du cas

Les extracteurs doivent pouvoir créer des sous-ensembles sans recopie.

b. Propositions

Modifier le code existant pour utiliser des shared_ptr au lieu des unique_ptr dans la class Army.

6. Croisements/Mutations

a. Étude du cas

Pour faire évoluer l'Intelligence artificiel nous devons mettre en place les transformations de l'arbre de décision.

b. Propositions

L'arbre de décision pourra être croisé avec un autre à l'aide de l'opérateur « * ». Le croisement génère un nouvel arbre dont chaque nœud sera un choix entre les nœuds de l'un ou de l'autre arbre de décision étant croisé.

L'arbre de décision pourra aussi muter. La mutation modifiera un nœud de l'arbre. Cette modification modifiera le prédicat d'un nœud.