INF-781 Rapport Intelligence Artificielle

Auteurs : Sun-Lay Gagneux & Louis Pougis

Introduction:

Dans ce laboratoire, nous avons choisi de séparer le code selon le mode de vision du npc : le mode omniscient (le npc voit la map au complet) et le mode non-omniscient (le npc possède une capacité de vision limitée). Nous diviserons donc ce rapport selon ces 2 distinctions. Dans le premier cas d'une vision totale, une recherche de chemin au moyen d'un algorithme AStar sera employée. Pour le cas, d'une vision partielle, un Behaviour Tree sera employé pour modéliser le comportements du Npc. Ce choix de séparation entre les différents modes de vision est dû fait que nous n'avons pas encore eu le temps de l'appliquer au mode omniscient.
Tout d'abord, nous traiterons le cas de la vision omnisciente, puis nous présenterons la structure employée pour le mode non-omniscient.

I/ Mode omniscient:

Dans ce mode là, comme indiqués plus tôt, le npc voit la totalité de la map. Nous avons construit un graph de la map à partir de la disposition des tuiles sur la map. A partir de ce graph, nous avons pu stocker l'Id des tuiles d'attributs Targets. Enfin nous avons appliqué l'algorithme AStar aux npcs présents sur la carte pour qu'ils puissent se diriger vers la target qu'ils doivent rejoindre. Cet algorithme de recherche de chemin se comporte de la manière suivante:

Tout d'abord, nous créons une liste vide de noeud fermés (c'est-à-dire qui ne sont pas encore visités) puis une liste de noeuds ouverts (considéré comme étant déjà visités). Tant que la liste de noeuds ouverts n'est pas vide, les actions suivantes sont réalisées :

Nous choisissons le noeud ayant le coût le plus faible parmi ceux de la liste de noeuds ouverts (on en profite au passage pour le retirer de cette dernière). On vérifie si les coordonnées x et y de ce noeuds correspondent à celles de la target qu'il doit rejoindre.
Si c'est le cas, on reconstruit le chemin (en passant par le parent de current, puis par son parent,etc… et ce, jusqu'au noeud de départ) et indiquer le chemin au npc correspondant.
Sinon, pour chaque voisin v de current :

-si v est une tuile interdite on passe au voisin suivant.

-si v existe dans la liste de noeud ouverts (ou fermée) avec un coût inférieur on ne fait rien. Sinon, on met à jour le coût de v avec celui de current + 1 ainsi que son heuristique avec le coût de v + la distance de manhattan de v jusqu'à la tuile target.On place ensuite le voisin dans la liste de noeuds ouverts.

Enfin, le noeud "current" est mis dans la liste de noeuds fermés (on considère qu'il a été visité).

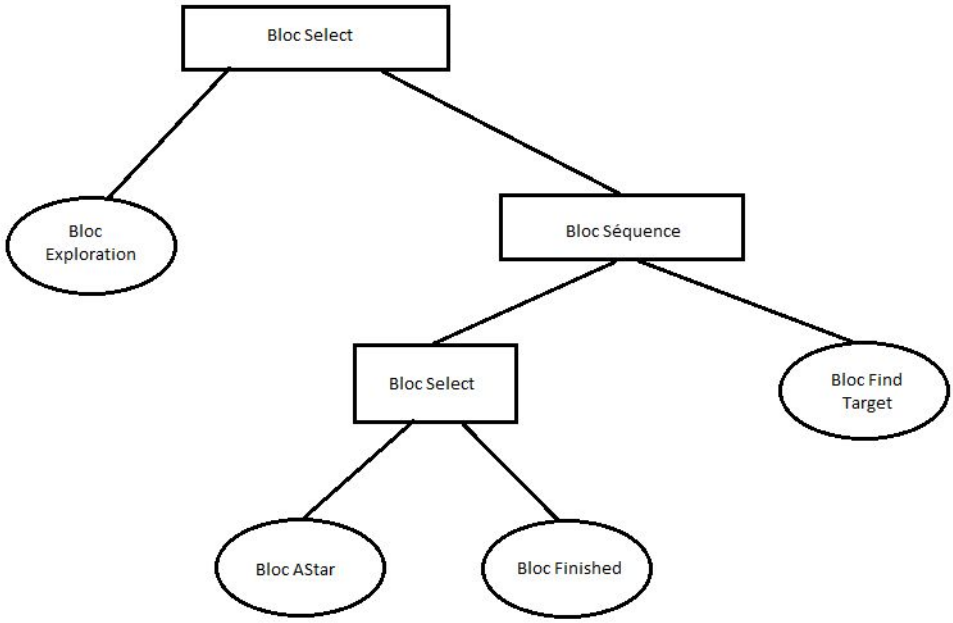
II/ Mode non omniscient

Afin de modéliser un bot ayant un comportement adaptatif en fonction des évènements possibles arrivant quand on ne connaît pas la map d'avance, nous avons choisi d'implémenter un "Behavior Tree" simple.
Pour créer ce Behavior Tree, nous sommes parti d'un objet abstrait (BaseBloc) dont hériteront tous nos bloc de Behavior Tree. Nous avons ensuite implémenté 3 blocs abstraits héritant de BaseBloc mais orientant ceux-ci vers des fonctions de blocs composés, blocs décorateurs ou blocs d'action. Dans ces 3 blocs, nous avons implémentés uniquement les méthodes liés à ces types de blocs mais pas leurs méthodes d'exécution. Nous avons ensuite créer des blocs Select, Sequence, Inverse, Fail, Success en adéquation avec ce que nous avons vu dans le cours. Par soucis de simplicité, nous avons aussi créé un bloc d'action général (nommé GeneralAction) qui est un template et qui prend un foncteur en paramètre de construction. Nous pouvons ainsi, grâce à celui-ci, créer n'importe quel bloc d'action à partir d'une lambda
et l'attacher à une suite de blocs. Les 2 avantages notables de cette architecture de code avec des lambdas est:

- Leur vitesse d'exécution (les foncteurs s'exécutent plus rapidements que les méthodes).
- La capacité qu'elles ont pour capturer des éléments par référence et travailler sur ceux-ci sans soucis.

C'est ce second avantage que nous avons mis à profit pour manipuler un "Blackboard" sans pour autant le faire devenir statique. Nous avons ainsi créé une classe "Blackboard" et l'avons passé en capture par référence aux lambdas ayant servi à créer nos blocs d'action. Dans ce blackboard, nous avons toutes les variables du jeu permettant de gérer les bots et leurs actions.
Ce blackboard est responsable de conserver et mettre à jour à chaque tour de jeu le TurnInfo et les informations du Graph d'exploration dont chaque Npc se sert pour avancer. Il est aussi responsable de remplir la liste d'actions à chaque tour pour chaque Npc.

Pour la création du graph, nous sommes parti de ce modèle pour chaque Npc:



Chaque Npc va d'abord se chercher une Target parmi son environnement.
Si l'action échoue (il n'y a pas de Target), le npc se choisira une tuile parmi ses voisins qui n'est pas "interdite"et qu'il a exploré le moins souvent (exploration). Si la recherche de Target réussit (une Target a bien été trouvée pour eux), il passera par un bloc select qui déterminera s'il a fini (arrivé sur sa Target). Si c'est le cas, le Bloc Finished retourne un "Succès" au bloc
Select, transmis au bloc séquence, puis au Bloc Select Root. Sinon, le bloc Select exécutera le bloc AStar. Si l'action du AStar réussit (il existe un chemin non nul allant à la cible), nous retournons un "Succès" (à son bloc parent Select), relayé par le bloc Séquence au bloc Select qui termine le Behaviour Tree du Npc considéré(notre Npc a donc un chemin valide vers une target). En revanche, si le AStar échoue (il n'y a pas de chemin vers la cible), le bloc d'action AStar retournera un Fail qui provoquera une nouvelle exploration pour notre Npc.

III/ Architecture générale du programme

Pour commencer, le centre de fonctionnement de notre programme se trouve être le Blackboard. Celui-ci est créé lors de l'initialisation de MyBotLogic à partir des informations contenues dans le LevelInfo. Durant cette étape, il se fabrique un graphe avec les informations du LevelInfo (nombre de lignes et nombre de colonnes).

Ensuite, à chaque tour, notre Blackboard se met à jour avec les informations fournies par le TurnInfo. Au cours de cette opération, il met à jour une copie du TurnInfo et, pour le mode non-omniscient, il met également à jour les informations de son graphe d'exploration.
Une fois ces mises à jour terminées, les algorithmes adéquates sont exécutés afin de donner aux npcs les actions qu'ils doivent faire.

IV/ Résultats :

Cette architecture nous a permis de valider toutes les maps demandées dans le cahier des charges, exceptée la TC_040.

NB: Pour en arriver à ce résultat, il nous a fallu une petite centaine d'heures.