Gestionnaire d'unités *ranged* pour Command Center

par Raphaël Royer-Rivard, Benjamin Ross, Antoine Théberge

Introduction:

Nous avons choisi CommandCenter comme base pour le bot puisque c'était le seul disponible lors du début du projet. Puisque nous étions en "compétition" avec l'équipe de Deep Learning, qui elle se concentrait sur des mini-jeux ne gérant que le combat avec des unités "ranged", nous avons fait de même. Le code ainsi que la documentation pour Command Center est disponible ici: https://github.com/davechurchill/commandcenter

Le travail principal a été effectué dans les classes *RangedManager* et *Micro*. Nous avons changé la façon de savoir quelle action doit être effectuée par l'utilisation de *Behavior Trees* (BT) et de *Finite State Machines* (FSM). Plus de documentation sur ces concepts se trouve dans le livre "*Game AI Pro*" disponible au Planiart, particulièrement les chapitres 4 "*Behavior Selection Algorithms: An Overview*" et chapitres 6 "*The Behavior Tree Starter Kit!*". Le livre est aussi disponible sur le web si vous cherchez bien. Une tentative d'implémenter une version de l'algorithme *Alpha Beta Pruning*, inspiré par *SparCraft*, le gestionnaire de combat de *UAlbertaBot*, le prédécesseur de Command Center pour *Starcraft: Broodwar*, a aussi été implémentée dans le *RangedManager*.

Behavior Tree:

La classe RangedManager, particulièrement la méthode *assignTarget* contiens le gros du travail. Pour chaque unité "ranged" du joueur, un BT ayant la structure de la figure 1 est bâti. On détecte d'abord si une cible est proche de l'unité. Si oui, on l'attaque. Pour savoir si on emploie le *kiting* ou le *focus fire*, on détermine si la cible est "ranged" ou "melee". Puis on invoque la bonne action, implémentée dans la classe *Micro*. Si aucun ennemi est visible, on vérifie si un *mineral shard* est proche. Si oui, on va vers celui-ci, sinon on continue vers l'objectif. Aller vers le *mineral shard* s'il est disponible sert à remplir l'objectif du mini-jeu *CollectMineralShards* de DeepMind, qui était aussi un objectif mineur de l'équipe de Deep Learning. Aller vers l'objectif est géré par Command Center et hors de la portée du travail accompli ici.

Nous avons choisi un Behavior Tree pour gérer le choix de l'action à prendre puisqu'une fois implanté, il serait facile de non seulement modifier la structure existante pour changer le comportement des unités "ranged", mais aussi d'implémenter un BT pour les unités "melee".

Finite State Machines:

Deux FSMs ont été implémentés pour ce projet décrivant deux stratégies de micro-gestion employées par les joueurs de Starcraft aguerris: le *kiting* et le *focus fire*. Le *kiting* consiste à déplacer vers

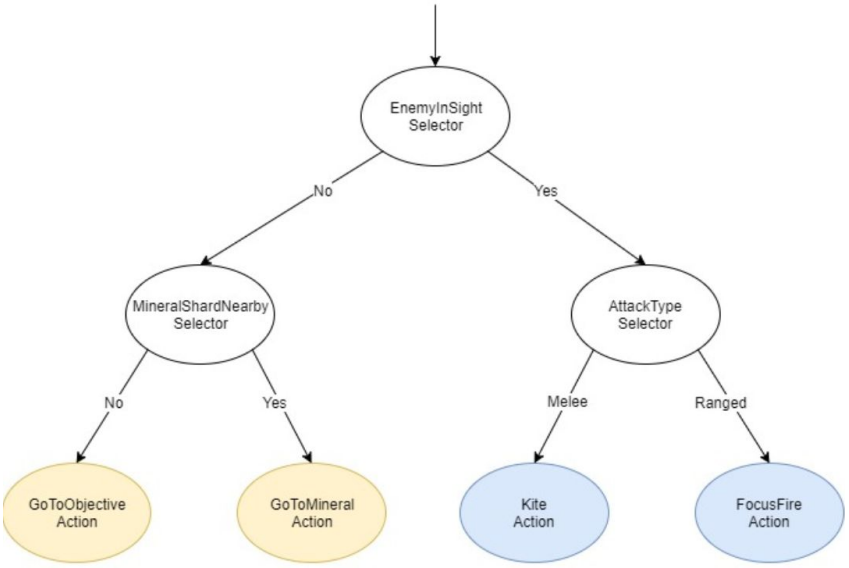

Figure 1: Le Behavior Tree du Ranged Manager. Les actions en bleu sont des FSMs, les actions en jaune ne sont que des simples ordres.

l'arrière des unités ranged lorsqu'une cible "melee" s'approche d'elle, évitant que celle-ci prenne des dégats de la part de la cible. Lorsque l'unité est suffisamment loin, elle peut continuer d'attaquer l'unité "melee", puis profiter de son *cooldown*, son temps de rechargement, pour s'éloigner. Le *focus fire,* quant-à-lui, consiste à attaquer avec le plus d'unités possible la même cible afin qu'elle meurt le plus rapidement possible et donc qu'elle ne puisse plus attaquer nos propres unités. Nous avons aussi intégré la stratégie de *dancing*, où les unités ciblées par les ennemis sont éloignées momentanément afin qu'elles ne soient plus ciblées. C'est la façon de contrer le potentiel *focus fire* de la part de l'ennemi. La figure 2 représente visuellement l'implémentation de ces deux FSMs.

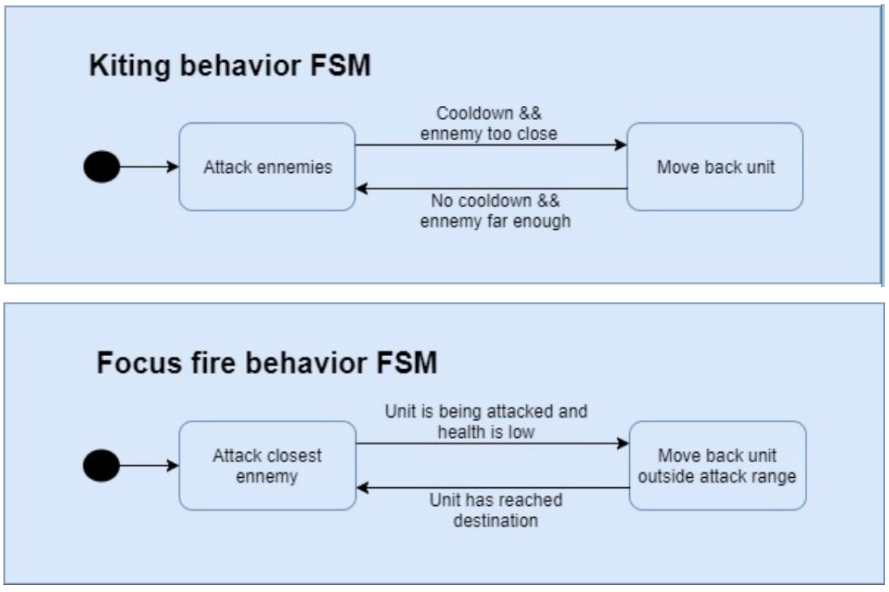

Figure 2: Les FSMs implémentés. Les rectangles représentent les états alors que les flèches représentent les transitions. Aucune transition n'existe ni n'est nécessaire entre les deux FSMs.

Ces deux FSMs sont implémentés dans la classe Micro, dans les méthodes *SmartKiteTarget* et *SmartFocusFire.* Puisque l'état doit être préservé entre chaque *game step,* mais que les unités (et donc leur potentiel état) sont rechargées entre chaque *step*, la classe Micro, qui n'est chargée qu'une fois au début de la partie, s'occupe de garder en mémoire les états des unités.

Alpha Beta Considering Durations:

Suite au succès relatif du BT + FSMs, nous avons essayé une autre approche cette fois-ci inspirée d'*UAlbertaBot*, plus particulièrement son module de combat *SparCraft. SparCraft* utilise une variétés d'algorithmes de parcours de graphe afin de déterminer quelles actions sont à prendre pour les unités en jeu et les cibles ennemies. L'algorithme que nous avons choisi d'implanter, en s'inspirant fortement de *SparCraft,* se nomme *Alpha Beta Considering Durations* (ABCD). Son fonctionnement est décrit ici:
https://skatgame.net/mburo/ps/aiide12-combat.pdf

Le code source de *SparCraft* est disponible ici:
https://github.com/davechurchill/ualbertabot/tree/master/SparCraft
(voir le wiki de la page Github pour plus de documentation)

Pour résumer, ABCD s'inspire fortement de l'algorithme très connu en intelligence artificielle *Alpha Beta Pruning* pour générer les actions possibles pour les unités du joueur. Par contre, contrairement à un jeu comme le tic-tac-toe où le nombre d'actions possibles à chaque tours est très limité, un RTS comme *StarCraft 2* possède un nombre quasi-illimité d'actions possibles à chaque moment dans le jeu. De plus, alors qu'une action dans le jeu de tic-tac-toe se produit instantanément, les actions d'une unité dans *StarCraft* peut durer très longtemps. Par exemple, le déplacement d'une unité d'un bout de la carte vers un autre, la durée d'attaque d'une cible et même le *cooldown* de l'arme de cette unité. Il faut donc prendre en compte ces paramètres lors de la générations des actions du prochain joueur et de nous-même.

Pour palier au problème du nombre d'actions important, le nombre d'actions possibles par unités a été réduit. Plutôt qu'une unité puisse aller à toutes les positions possibles *x,y* à chaque tour, certains mouvement ont été prédéfinis pour chaque unité. Dans notre cas, il s'agit soit d'avancer vers une unité ennemi pour être en portée d'attaque, soit de s'éloigner pour avoir le temps de recharger son arme. Une stratégie d'attaque a aussi été mise en place: Plutôt que pouvoir attaquer n'importe quelle unité ennemie, seulement la plus proche est considérée. Nous passons donc d'une quasi-infinité de mouvements * la possibilités d'attaquer toutes les cibles ennemies à trois actions possibles par unités. Dans le but d'optimiser encore plus, plus d'une action est considérée par *tour*, et ces actions sont regroupées dans des *moves.* Ici, un *move* est composé d'une action aléatoirement choisies par toutes celles que qu'une unité peut faire par unités. Un *move* contient donc autant d'actions qu'il y a d'unités.

Pour palier au problème de la durée de temps, les temps requis pour faire ces actions sont enregistrées dans celles-ci et leur durée restante est mise à jour à chaque génération d'état enfant. Ceci évite qu'une unité puisse attaquer sans arrêt en théorie alors qu'elle doit attendre un certain temps en pratique. Dans l'implémentation de *SparCraft,* il est aussi mentionné que les *tours* des joueurs ont une

durée variable et qu'il peut arriver que deux joueurs puissent jouer en même temps. Ceci n'a pas été considéré ici par manque de temps.

L'implémentation (partielle) de cet algorithme se trouve aussi dans *RangedManager,* dans la méthode *assignTargets.* La profondeur maximale doit être spécifiée afin que la recherche ne continue pas à l'infini, ainsi qu'une limite de temps par recherche, qui n'a pas été mise en vigueur par manque de temps. Les unités doivent aussi être "construites" à l'extérieur de la classe de recherche afin de ne pas mettre trop de logique de *StarCraft 2* dans l'algorithme. De plus, leur précédente action est sauvegardée dans la classe *Micro* afin que la durée d'une action soit conservée entre chaque *step* et donc réellement plus grande qu'une *game step.*

Pour la suite:

Puisque nous ne sommes pas des experts de Starcraft 2, il est possible qu'il y ait des stratégies de microgestion que nous n'avons pas considéré pour les unités "ranged" qui seraient intéressantes à implémenter et qui donneraient de bons résultats.

De plus, il serait probablement intéressant d'implémenter la stratégie de BT + FSM pour les unités "melee", afin que le bot puisse mieux gérer les différentes unités des différentes races (les *zerglings* ou les *zealots*, par exemple).

CommandCenter dans la version que nous utilisons est très stupide aussi. Sa stratégie pour gagner une partie complète, pour les *Terrans,* consiste en *spammer* des marines vers la base ennemie jusqu'à la fin de la partie. Il serait probablement intéressant d'implémenter un planificateur pour donner au bot une stratégie décente.

L'implémentation d'ABCD présente n'est pas complète et a beaucoup de place à l'amélioration. Beaucoup de *TODOs* ont été laissés dans le code spécifiant ce qu'il reste à faire. Il serait probablement intéressant d'essayer d'autres algorithmes de recherche.