Université du Québec
à Chicoutimi

8INF846 - Intelligence Artificielle

Rapport d'implémentation du projet

Clément De Gaillande : DEGC30049809
Basile Nguyen : NGUB28099807

04/05/2021

I. Introduction du projet

Le but du projet est de faire s'affronter plusieurs algorithmes afin de présenter lequel est le plus efficace avec des résultats et des statistiques. Les deux algorithmes s'affrontent au travers d'un jeu de chiens de berger. Les chiens doivent parcourir un environnement pour ramasser les moutons qui s'y trouvent et les ramener à leur enclos respectif pour gagner des points. Ils peuvent porter plusieurs moutons à la fois jusqu'à un maximum défini. Le chien ayant ramené le plus de moutons remporte la partie. Nous avons choisi d'utiliser l'algorithme d'exploration informé Greedy Search et les algorithmes d'exploration adverse MiniMax et Minimax avec élagage AlphaBêta, au début de la partie l'utilisateur peut choisir les algorithmes d'exploration adverses et informés qui doivent s'affronter.

Dans ce rapport nous allons donc présenter l'environnement dans lequel évoluent les chiens puis l'implémentation des deux agents avec les différents algorithmes, les choix d'implémentation que nous avons fait, les résultats que nous avons obtenus et enfin la conclusion du projet.

II. Environnement

Les deux chiens de bergers évoluent sur un échiquier carré de n cases. Les chiens peuvent se déplacer dans les quatres directions cardinales mais ne peuvent pas aller sur une case contenant un chien adverse ou un enclos adverse. Les moutons sont répartis aléatoirement sur l'échiquier en nombre impair pour permettre à un chien d'avoir un score supérieur à celui de son adversaire.

L'environnement est :

- complètement observable, les deux agents peuvent obtenir toutes les connaissances nécessaire du plateau
- Déterministe, le prochain état de l'environnement est déterminé par l'état courant et par l'action de l'agent
- Séquentielle, la prochaine séquence de perception-action va dépendre des actions futurs pour empêcher son adversaire de gagner
- Statique, l'environnement ne change pas pendant que les agents délibèrent
- Discret, le nombre d'état est fini
- Multi-agents, un des agents essaye de prévoir les actions à effectuer pour mettre l'autre en difficultés

III. Implémentation

Dans notre implémentation, les chiens sont représentés par la classe *Dog* et les agents par les classes *AgentGreedy* et *AgentMiniMax / AgentAlphaBeta*, les deux agents sont gérés par la classe *Agents* qui permets d'organiser les tours de jeu, lancer les séquence perception-action de chaque agents et mettre fin à la partie. Les classes *Capteur* et *Effecteur* permettent aux agents de prendre connaissances et d'agir sur l'environnement. (cf. annexe diagramme de classes)

A. Greedy Search

L'algorithme Greedy Search ou algorithme glouton est un algorithme assez simple. On va déterminer le coût de chaque action menant à un nœud, et choisir l'action ayant le coût le plus faible à chaque fois en espérant obtenir une solution optimale.

Dans notre problème, l'agent va commencer par sélectionner son objectif. Cet objectif peut être soit le mouton le plus proche, soit l'enclos associé au chien.

Il va d'abord analyser l'environnement avec l'objet *Capteur* et la méthode *GetNearestObjective* pour sélectionner un nouvel objectif. Pour cela, il va calculer la distance de manhattan entre chaque mouton et le chien, et choisir le mouton le plus proche. S'il ne reste aucun mouton sur le plateau, il va automatiquement fixer l'objectif du chien sur son enclos.

Dans le cas où notre objectif est un mouton, on doit vérifier à chaque résolution qu'il est toujours atteignable et qu'il n'a pas déjà été capturé par l'autre chien pendant son tour de jeu. S'il est toujours atteignable, l'agent va simplement conserver cet objectif et calculer sa prochaine action. Sinon, il va analyser l'environnement à nouveau pour déterminer son nouvel objectif.

L'agent Greedy Search va ensuite choisir l'action à appliquer. Si le chien est arrivé à son objectif (enclos ou mouton) alors il choisit *release* ou *catch.* Par contre, si le chien n'est pas sur son objectif, l'agent va parcourir les choix possibles d'action pour le chien (haut, bas, gauche, droite) et sélectionner la direction qui réduit le plus possible la distance de manhattan avec l'objectif. Pour terminer, l'agent va appliquer l'action sélectionnée avec l'objet *Effecteur.*

B. Exploration adverse

a. Algorithme Minimax

L'algorithme MiniMax est un algorithme récursif, il considère l'ensemble des possibilités de chaque agent à chaque tour pour déterminer les choix les plus optimaux pour battre son adversaire.

L'agent génère tout d'abord le nœud initial (représenté par l'objet *Node*) en passant par le constructeur *Node(Environnement environnement, Dog activeDog, int depth).* Ce constructeur va faire appel à la méthode *CloneNode()*, qui va s'occuper de faire une copie de l'environnement complet au moment de la création du nœud.

Ensuite la méthode *TourMax* va lister les actions possibles pour le chien Max et va créer un par un des objets *node* pour chaque action en appelant la méthode *GenerateNextNode*, qui va permettre de calculer un nouveau noeud à partir d'un noeud en particulier auquel on appliquera une action. L'utilité de ce noeud est alors calculée de la manière suivante :

utility = $(Carried_{max} + 2 \times Score_{max}) - (Carried_{min} + 2 \times Score_{min}) \times 100 \pm n_{depth}$.

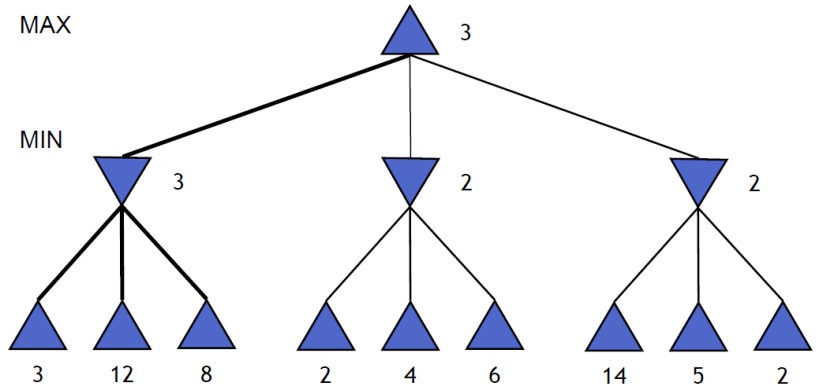
Carried = nombre de moutons portés par le chien
Score = nombre de moutons amenés à l'enclos
n_{depth} = profondeur de l'arbre

Ce calcul d'utilité permet de donner plus de poids au moutons qui ont été amenés à l'enclos plutôt que ceux qui ont été simplement ramassés, on prends également en compte la profondeur de l'arbre, de ce fait l'algorithme va privilégier les chemins les courts, nécessitant donc peu d'action pour atteindre un état final favorable.

Une fois qu'un nœud a été correctement généré, on appelle TourMin dessus. La méthode *TourMin* va, elle, lister les actions possibles pour le chien Min et va créer un objet *node* pour chaque action comme expliqué précédemment.

Ces appels récursifs vont se poursuivre jusqu'à atteindre un nœud final. Nous avons défini un état but comme étant un état dans lequel il ne reste aucun mouton sur le plateau, et qu'aucun chien ne porte encore de moutons sur lui. Cela peut correspondre au fait que les deux chiens ont ramené tous leurs moutons dans leur enclos, mais également au fait qu'un chien soit rentré à son enclos pour poser ses moutons pendant que l'autre se trouve encore dans la prairie sans pour autant porter de mouton sur lui.

On va alors progressivement faire remonter l'utilité des noeuds finaux en suivant ce schéma :



Pour chaque nœud, l'un des joueurs (min et max de l'algo) vont choisir l'action qui les satisfait le plus. Max va choisir les actions qui maximisent l'utilité, tandis que Min va choisir les actions qui la minimisent.

En plus de la détection d'un état but, nous avons dû fixer une profondeur maximum pour l'arbre. En effet, dans notre problème, il est possible pour les chiens de tourner en rond sans jamais atteindre un état but, générant ainsi un nombre infini de noeux.

b. Élagage Alpha Beta

L'élagage Alpha-Bêta est une méthode d'accélération de MiniMax. Il permet de diminuer le nombre de nœuds explorés par MiniMax en éliminant les nœuds qu'il est inutile de considérer.

Pour cela, on ajoute à la logique de Minimax un intervalle de valeurs acceptables qui va être maintenu et mis à jour pendant tout le processus de création et d'exploration des nœuds.

Dans MiniMax, Min et Max s'affrontent sur le choix des actions. Min va choisir des actions qui vont minimiser l'utilité, tandis que Max va choisir des actions qui vont la maximiser.

Le but d'Alpha-Bêta est de profiter de cette caractéristique pour réduire la recherche. En maintenant un intervalle de valeurs acceptables, on va pouvoir éliminer des nœuds avant même de les explorer.

En effet, si pour la délibération d'un joueur Max à une profondeur P, on a un intervalle allant de 10 à 15, et qu'on trouve pour l'un des noeuds à une profondeur P+1 une utilité de 16, on sait qu'il est inutile d'explorer les noeuds suivants, puisque Max va forcément choisir une valeur >= 16.

Or notre intervalle nous indique qu'on ne va accepter que des valeurs <15, probablement parce que le Min de profondeur P-1 à jusqu'alors le choix de prendre un nœud ayant une utilité de 15. Il ne choisira donc jamais la valeur 16. Par conséquent, il est inutile pour Max de continuer l'exploration des nœuds fils de P.

Réciproquement, on va pouvoir élaguer les fils d'un nœud associé à Min qui auront au moins un frère <= à la valeur d'alpha.

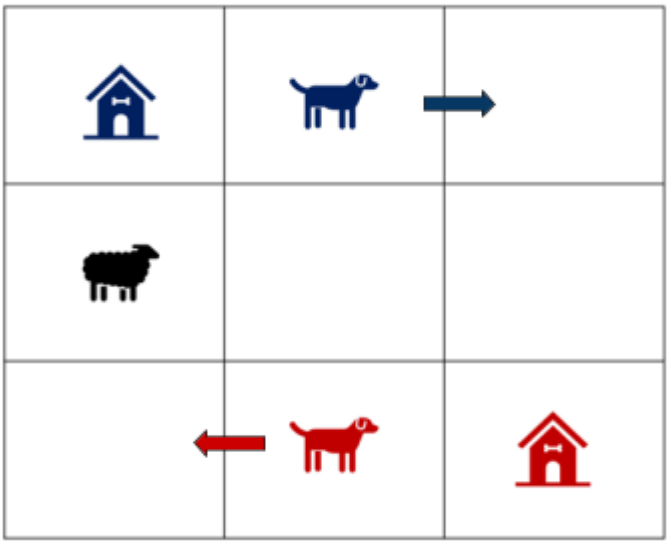
IV. Performances

Note : Le chien bleu utilise l'agent avec une recherche heuristique et le chien rouge utilise l'exploration adverse.

Nous avons réalisé des tests dans différentes configuration afin d'essayer de comparer les approches, mais également de voir comment le paramétrage de nos fonctions influe sur le comportement des agents.

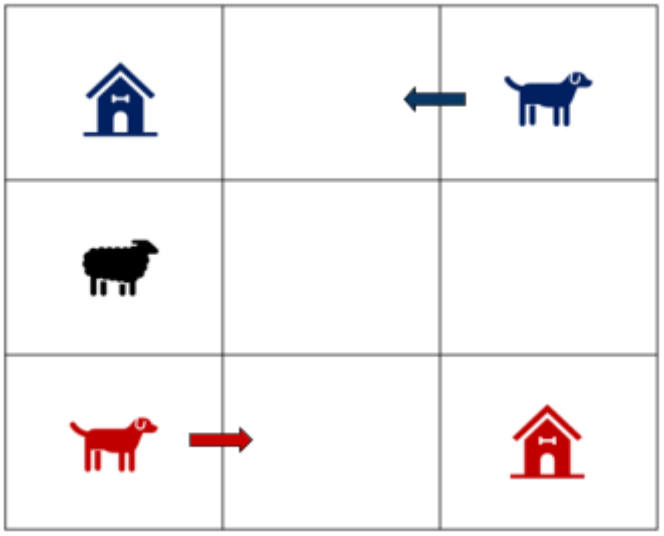
Nous l'avons évoqué précédemment, il est possible dans notre problème de ne jamais atteindre d'état but.

Nous allons illustrer ce propos :



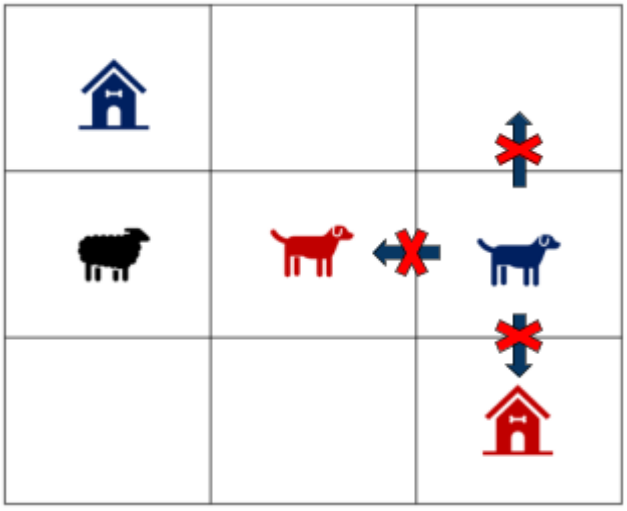
Dans cette situation, le chien rouge pourrait décider de se diriger à gauche, et le chien bleu à droite.

On se trouverait alors dans cet état là :



Une fois dans cet état, rien n'empêche le chien rouge de revenir sur la case précédente, et rien n'empêche le chien bleu de faire de même. La recherche de solution serait alors bloquée dans un cycle.

Pour empêcher cela, nous avons pensé à maintenir une liste d'actions précédentes empêchant de revenir sur ces pas, mais on entrait alors dans une contradiction qui nous bloquait dans une autre impasse. En effet, un chien ne peut se diriger sur une case contenant un chien ou un enclos adverse. Voyons alors la situation suivante :



Le chien bleu ne peut ni aller à gauche, ni aller en bas à cause du chien et de l'enclos adverse. Il ne lui reste que la case du haut. Or, c'est de cette même case qu'il vient, il lui est donc également interdit de revenir sur ses pas.

Pour régler cette situation, nous avons été obligés d'autoriser le chien à sauter une action à n'importe quel moment. Seulement, après introduction de cette règle, l'agent utilisant l'exploration adverse passait son temps à passer son tour et restait dans sa niche.

En corrigeant ce problème, d'autres sont survenus par la suite. Nous avons donc décidé de revenir sur nos pas et d'autoriser les cycles, mais de limiter la profondeur de l'arbre (lorsque l'agent atteint la profondeur max, il retourne l'utilité).

Ce choix a permis de mettre en lumière un comportement assez amusant de notre agent : nous avons introduit le fait que notre agent est “mauvais joueur” !

Pour expliquer ce terme, nous allons étudier différents cas de figure.

- Selon certaines configuration, lorsqu'il ne devrait pas avoir d'autres options que d'accepter sa défaite et de rentrer à l'enclos, le fait que l'exploration de notre agent soit coupée et retourne le noeud “coupé” produit une meilleure utilité pour un cycle que pour le fait d'aller déposer ses moutons à l'enclos. Il va donc choisir constamment des actions contraires et rester coincé sur les mêmes cases.
  Seulement, la plupart du temps, ce cas se produit lorsque le chien bleu est proche, et nous venons d'expliquer qu'un chien ne peut se diriger sur la case de l'enclos adverse ou sur celle d'un chien adverse. Le chien bleu est donc forcé d'éviter ces cases là et de s'éloigner, mais le chien rouge réplique en le suivant. Ce petit manège à pour effet d'empêcher le chien bleu de rentrer déposer ses moutons à l'enclos, ce qui par extension empêche l'atteinte d'un état but. Vous l'aurez compris : cette IA est une mauvaise perdante qui utilise l'anti-jeu pour ne pas perdre !

- Il arrive également que le chien rouge parvienne à forcer le chien bleu à entrer dans une position de victoire du chien rouge. Dans certains cas, il arrive que le chien rouge prenne son temps à faire des allers retours sur des cases pour empêcher le chien bleu d'accéder à un mouton, et comme pour le narguer, lorsque le chien bleu se rapproche enfin du mouton, le chien rouge le ramasse et le ramène à l'enclos pour sceller sa victoire. c'est également un mauvais gagnant !

Il est cependant parfois difficile de reproduire ces comportements, puisqu'ils dépendent de beaucoup de paramètres : la profondeur maximale autorisée pour l'exploration, la pondération de l'utilité (par exemple enlever le x2 sur le score du chien ou bien diminuer la différence entre le score-carried et la profondeur), le nombre de cases du plateau, le nombre de moutons … La composante aléatoire de l'apparition des moutons affecte également cela.

Pour ce qui est des performances, nous avons observé différents phénomènes :

- Tout d'abord, évidemment, obtenir des résultats dans des temps d'attente raisonnables n'est possible que pour des tailles de grille réduites, un nombre de moutons et une profondeur d'exploration très peu satisfaisants à notre goût, à cause de la présence des cycles et de l'infinité d'états qui en découle;

- L'utilisation de l'élagage diminue drastiquement le temps de calcul, et permet d'essayer des problèmes avec des grilles plus complexes, en augmentant la taille de la grille, le nombre de moutons et la profondeur max de calcul ;

- Le ratio de victoires dépend également de la disposition initiale des moutons. Pour la même grille avec le même nombre de moutons, la disposition de ces derniers peut changer l'issue d'un match du tout au tout (victoire totale du rouge / victoire totale du bleu / victoire serrée de l'un des deux)

V. Conclusion

Ce que l'on retire avant tout de ce projet, c'est que décider de travailler sur une modélisation agent en particulier avant de conceptualiser un problème n'est pas une bonne pratique.

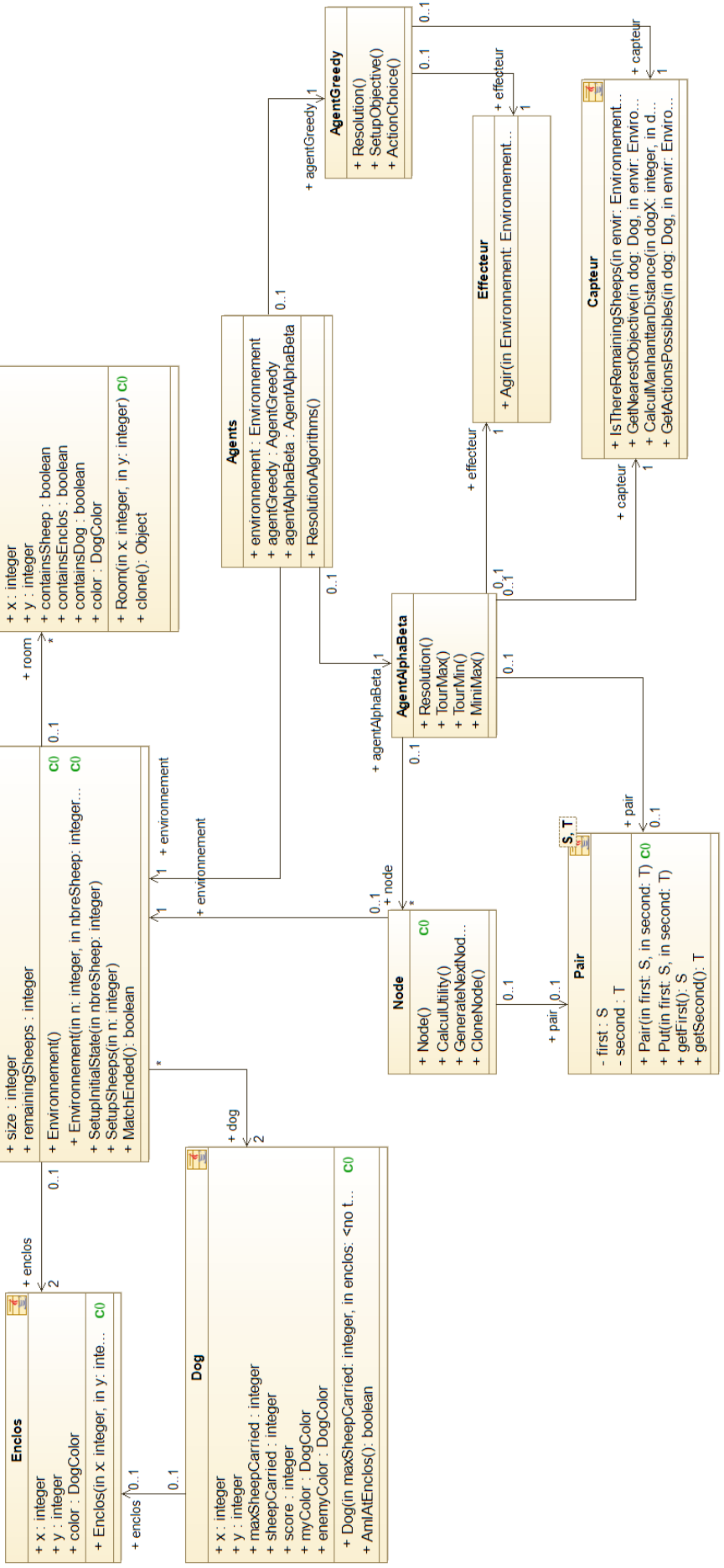
Comme on a pu le voir dans notre cas, nous avions décidé de travailler sur de l'exploration adverse avant de chercher un problème auquel l'appliquer. Ceci a eu pour effet de grandement nous compliquer la tâche quand il a fallu implémenter nos agents. MiniMax n'est pas adapté à la résolution de ce problème car il est possible d'entrer dans un cycle infini, et nous ne nous sommes rendus compte que trop tard de cette évidence. Cela a entraîné beaucoup de frustration durant le développement, et encore plus lorsque nous nous sommes rendus compte de notre erreur.

Cependant, il nous a quand même permis de voir à quel point l'élagage Alpha-Bêta pouvait booster MiniMax, en permettant des explorations plus poussées (que ce soit en termes de profondeur de recherche, de la taille du plateau ou du nombre de moutons présents).

Il nous a également permis de nous rendre compte que MiniMax pouvait avoir l'avantage dans un système de tour par tour, mais que dans un système en temps réel, il mettait beaucoup trop de temps à se décider comparé à GreedySearch, qui aurait alors le temps de rafler les moutons avant que MiniMax ne puisse calculer ses déplacements.

Malgré les difficultés rencontrées, il a tout de même été très intéressant de travailler sur ce projet, nous avons pu développer nos connaissances sur les systèmes multi-agents et l'exploration adverse

Annexe.


Room
+ x : integer
+ y : integer
+ containsSheep : boolean
+ containsEnclos : boolean
+ containsDog : boolean
+ color : DogColor
+ Room(in x: integer, in y: integer) C0
+ clone(): Object
Agents
+ environnement : Environnement
+ agentGreedy : AgentGreedy
+ agentAlphaBeta : AgentAlphaBeta
+ ResolutionAlgorithms()
AgentGreedy
+ Resolution()
+ SetupObjective()
+ ActionChoice()
Effecteur
+ Agir(in Environnement: Environnement...
Capteur
+ IsThereRemainingSheeps(in envir: Environnement...
+ GetNearestObjective(in dog: Dog, in envir: Enviro...
+ CalculManhanttanDistance(in dogX: integer, in d...
+ GetActionsPossibles(in dog: Dog, in envir: Enviro...
Environnement
+ size : integer
+ remainingSheeps : integer
+ Environnement() C0
+ Environnement(in n: integer, in nbreSheep: integer... C0
+ SetupInitialState(in nbreSheep: integer)
+ SetupSheeps(in n: integer)
+ MatchEnded(): boolean
AgentAlphaBeta
+ Resolution()
+ TourMax()
+ TourMin()
+ MiniMax()
Node
+ Node() C0
+ CalculUtility()
+ GenerateNextNod...
+ CloneNode()
S, T
Pair
- first : S
- second : T
+ Pair(in first: S, in second: T) C0
+ Put(in first: S, in second: T)
+ getFirst(): S
+ getSecond(): T
Enclos
+ x : integer
+ y : integer
+ color : DogColor
+ Enclos(in x: integer, in y: inte... C0
Dog
+ x : integer
+ y : integer
+ maxSheepCarried : integer
+ sheepCarried : integer
+ score : integer
+ myColor : DogColor
+ enemyColor : DogColor
+ Dog(in maxSheepCarried: integer, in enclos: <no t... C0
+ AmIAtEnclos(): boolean
+ agentGreedy
+ agentAlphaBeta
+ effecteur
+ capteur
+ room
+ environnement
+ node
+ pair
+ enclos
+ dog