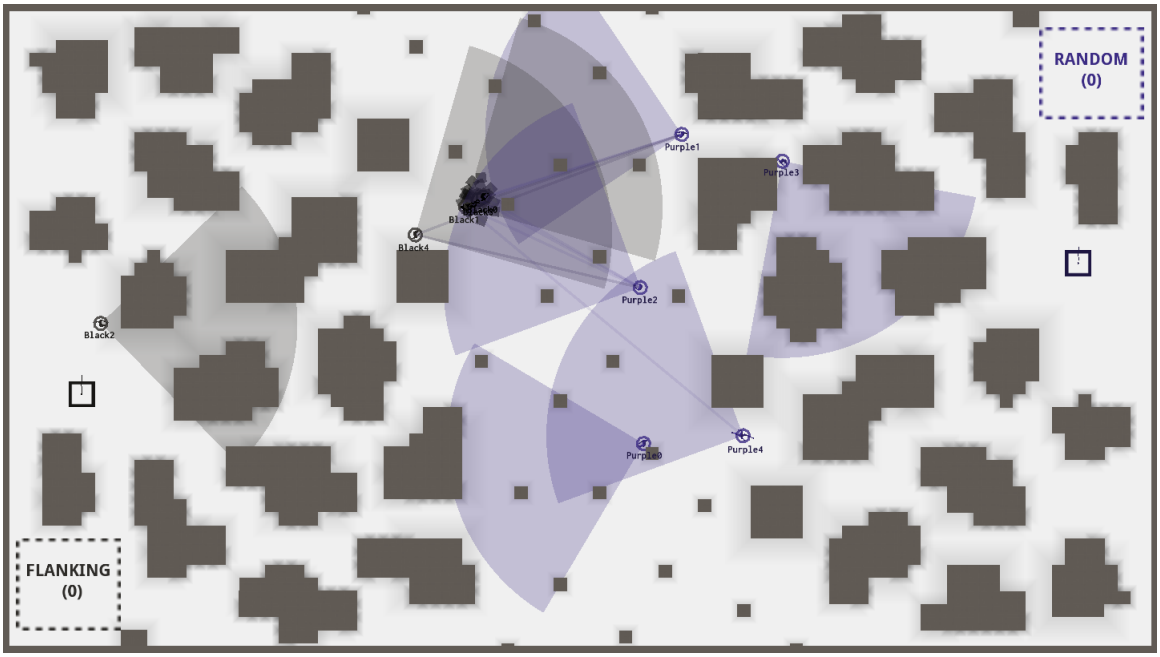
Université de bordeaux

Mémoire Projet de Programmation

IA pour un jeu de Capture the Flag temps réel en Python

Robin Navarro, Adrien Boitelle, Yann Blanchet
Alexis Perignon, Alexis Flazinska

6 avril 2020



openai challenge

Table des matières

1 Description du projet

Ce projet a pour but la réalisation d'un framework pour un jeu de capture de drapeaux, et la réalisation d'IA pour y jouer. Le jeu est un capture the flag à deux équipes, ayant chacune le même nombre de bots.

Chaque équipe évolue sur une carte dont elle a connaissance dès le départ. Tous les bots d'une même équipe commencent dans une zone commune, la zone de départ, et ont pour objectif de ramener le drapeau de couleur adverse dans leur zone de dépôt.

Chaque bot dispose d'un champ de vision, qui lui permet de détecter des éléments positionnés dans la carte (en particulier les bots adverses). Le champ de vision des bots d'une même équipe est partagé, c'est-à-dire qu'un bot peut voir tout ce que les autres bots de son équipe voient. Les bots peuvent attaquer les membres de l'équipe adverse afin de ralentir leur progression. Si un bot meurt avec le drapeau, il le pose à terre.

Les IA devront piloter ces bots en temps réel, sur une carte en 2D, en implémentant des stratégies d'équipe.

2 Analyse de l'existant

Il y a un existant codé en Java pour le moteur de jeu, mais la manière dont il est implémenté ne convient pas au client. En effet, ce moteur repose sur un modèle vectoriel, dans lequel il est plus compliqué d'ajouter des fonctionnalités et de modifier la physique du jeu. C'est pourquoi le client a recommencé l'écriture du moteur en python, en se basant cette fois sur un fonctionnement par "tuile". Mais ce nouveau moteur n'est pas terminé à ce jour.

La plupart des projets similaires sont des ajouts à deux jeux déjà existant. Par exemple, nous avons trouvé une IA de DeepMind[3] pour le jeu capture the flag, qui se repose sur le jeu Quake. Nous ne pouvons utiliser un jeu existant, car le framework doit pouvoir être facilement réutilisable par des étudiants pour un possible projet dans le cadre de leurs cours.

Suite à nos recherches, nous avons découvert l'existence d'un challenge en ligne datant de 2012 nommé iasandbox.com. Le but de ce challenge était de réaliser une Intelligence Artificielle distribuée pour un moteur de jeu déjà fourni et relié par un serveur.
Les participants devaient respecter une interface, et implémenter un ou plusieurs bots afin de les faire collaborer et affronter d'autres challengers. Malheureusement, ce site n'est plus accessible suite à un abandon. Nous avons pu trouver des archives web du site, mais le moteur de jeu n'était plus accessible au téléchargement.

3 Description des termes techniques

— Framework : Traduit littéralement, signifie "cadre de travail". Désigne un ensemble cohérent de composants éprouvés et réutilisables (bibliothèques, classes, helpers...) ainsi qu'un ensemble de préconisations pour la conception et le développement d'applications.

— IA : Intelligence artificielle, consiste à mettre en œuvre un certain nombre de techniques visant à permettre aux machines ou programmes d'imiter une forme d'intelligence réelle.

— Bot : Un programme autonome, qui peut interagir avec des systèmes ou utilisateurs. Très souvent, on parle d'un programme qui se comporte comme un humain dans les jeux vidéos.

— Alpha beta : Algorithme de recherche qui sert à réduire le nombre de noeuds calculés par l'algorithme Minimax dans les jeux. L'algorithme Minimax cherche à trouver la meilleure valeur possible d'un plateau de jeu tout en minimisant la meilleure valeur possible d'un plateau de jeu adverse.

— Moteur de jeu : collection de code sous forme de bibliothèques et de frameworks afin de réaliser un jeu.

— Librairie/Bibliothèque/API : un ensemble de fonctions utilitaires, regroupées et mises à disposition afin de pouvoir être utilisées sans avoir à les réécrire.

— Behavior Tree : Arbre (structure de données) permettant de contrôler le comportement d'une intelligence artificielle, chaque noeud représente une tâche, décision en réponse aux données fournies, ou une structure de contrôle. Permet de réaliser des comportements très complexes et cela en parcourant depuis la racine de l'arbre tout en suivant les branches adéquates en évaluant différentes conditions jusqu'à la décision finale à adopter.

— Pathfinding : désigne la recherche de chemin entre deux points , noeuds...

— A*(A star) : Algorithme de pathfinding réputé et peu coûteux permettant de rechercher un chemin entre une position initiale et une position finale. Les solutions trouvées par A* sont optimales.

— Frame : Une image affichée à l'écran.

— FPS(Frames per second) : Fréquence en hertz à laquelle l'affichage est mis à jour.

— Pygame : Bibliothèque libre multiplateforme qui permet de faciliter le développement de jeux vidéos en langage de programmation Python.

— Heuristique : Une heuristique est un raisonnement formalisé de résolution de problèmes (représentable par une computation connue) dont on tient pour plausible mais non pour certain qu'il conduira à la détermination d'une solution satisfaisante du problème.

— Complexité : Évaluation du temps, ressources, et stockage nécessaires à la réalisation d'un algorithme.

4 Description des besoins

4.1 Besoins fonctionnels

4.1.1 Besoins utilisateur

— Fournir une API pour que les IA communiquent avec le moteur de jeu.
Priorité : Essentiel.
L'API doit communiquer aux IA la position de chacun de leurs bots, ainsi que la direction dans laquelle ils regardent. Elle doit aussi communiquer les évènements du jeu, c'est à dire notifier une IA lorsqu'un de ses bots voit quelque chose (ennemi, drapeau, ...), prévenir quand un bot se fait tirer dessus.
En retour, l'IA doit communiquer la destination voulue pour chacun de ses bots, leur vitesse, et des évènements tels que le déclenchement du tir d'un de ses bots.

— Afficher une carte en deux dimensions.
Priorité : Essentiel.
  — Afficher les blocs de la carte. Il y a deux blocs de base. Les murs, qui sont solides et bloquent la vue des bots, et des murs transparents. Ils bloquent les bots, mais pas leur champ de vision.
  — Afficher les zones de chaque équipe.
  — Afficher les objets dynamiques (bots, drapeau, ...). Les bots sont des cercles.
  — Afficher le champ de vision des bots. C'est un cône ayant pour base chaque bot, avec une taille variable.

— Fournir un mode de jeu sans affichage graphique.
Priorité : Conditionnel.
Seul les évènements principaux du jeux doivent être affichés (capture de drapeau, mort d'un bot, victoire d'un équipe, ...).
Cet affichage doit pouvoir servir dans le cadre de l'entraînement d'une intelligence artificielle ou pour l'évaluation automatique de projet étudiant.

— Proposer une interface pour qu'un humain puisse jouer contre une IA.
Priorité : Conditionnel.
L'interface doit permettre au joueur de sélectionner plusieurs bots, bouger les bots vers une position sélectionnée en cliquant.
Pour faciliter le jeux, les bots doivent tirer automatiquement sur les adversaires qu'ils rencontrent.
Pour tester une interface humaine, dans un premier temps nous vérifierons que dès qu'un bot détecte un bot ennemi dans son champ de vision celui-ci attaque bien automatiquement : un scénario simple serait de mettre un bot contrôlé par l'interface humaine dans le champ de vision d'un autre bot, et bien vérifier qu'il y a attaque.
Il faudra aussi tester le déplacement des bots en cliquant à des positions aléatoires et en vérifiant que les bots ont bien reçu l'ordre de déplacement.

— Proposer une interface pour modifier au cours d'une partie l'arbre de comportement d'une équipe.
Priorité : Optionnel.
L'interface doit permettre à l'utilisateur de modifier en temps réel l'arbre de comportement

utilisé par son IA,
afin de voir rapidement les répercutions d'un changement de comportement.
Des noeuds doivent pouvoir être inter-changés, supprimés, ajoutés.

Pour tester, il faut modifier un arbre de comportement en passant par le contrôleur qu'utiliserait le joueur en condition normal,
et vérifier que l'arbre de comportement pointé par l'IA possède bien les modifications demandées par le contrôleur.

4.1.2 Besoins système

— Implémenter les règles du jeu.
Priorité : Essentiel.
Les IA doivent avoir à leur disposition 5 bots, se trouvant dans leur zone de spawn respective. Un drapeau se trouve près de chaque base. Un IA qui amène les deux drapeaux dans sa base gagne la partie.
Une contrainte de temps sera définie pour éviter que les IA restent inactives. Si aucune n'a réussi à ramener un drapeau dans sa propre base, alors il y a égalité.

— Charger une carte en mémoire depuis un fichier texte.
Priorité : Conditionnel.
  — Différentes tailles de carte possibles.
  — La carte doit être rectangulaire.
  — La carte ne doit pas permettre l'évasion des bots de la zone de jeu.
  — Le format doit permettre de préciser les zones de chaque équipe et les zones des drapeaux.

Au moins une carte par défaut doit être disponible.

— Générer des cartes aléatoires.
Priorité : Conditionnel.
Le moteur doit pouvoir générer des cartes correctes (correspondant aux besoins ci-dessus) de manière aléatoire. Les cartes doivent être jouables, c'est-à-dire qu'elles doivent permettre le déplacement des bots depuis leur zone d'apparition jusqu'à la zone du drapeau.
En guise de test, il faudra vérifier que la carte est de la dimension demandée et qu'il est possible de se déplacer des zones de spawn jusqu'aux drapeaux en utilisant A*.

— Implémenter des IA utilisant l'API du jeu.
Priorité : Essentiel.
  — Implémenter une première version basique permettant de montrer le fonctionnement du moteur de jeu. Elle doit effectuer des actions basiques, c'est à dire diriger ses 5 bots vers le drapeau, tirer sur les adversaires visibles, capturer le drapeau et le ramener à la base.
  — Ce comportement simple doit se reposer sur un behavior tree[2].
  — Les bots doivent pouvoir réaliser des actions communes, attaquer ensemble, défendre celui qui a le drapeau, défendre le drapeau ...

Pour tester cette IA basique, il faut la faire jouer dans une carte avec des ennemis placés aléatoirement, qui sont immobiles. L'IA doit gagner la partie.
Il faut ensuite tester l'IA sur une carte avec des ennemis placés aléatoirement sur la carte,

mais cette fois avec la possibilité de tirer dès qu'un bot se trouve dans son champ de vision. L'IA doit gagner.
Un test un peu plus élaboré consiste à faire jouer l'IA contre une équipe de bots faisant des "rondes" sur la carte, et tirant à vue. Encore une fois l'IA doit gagner pour passer le test.

Pour tester la capacité de l'IA à se défendre, on réutilise le test précédent mais en donnant le drapeau de l'équipe à un des bots effectuant des rondes. L'IA doit le récupérer et gagner pour réussir le test.
Une contrainte de temps doit être incorporée dans les tests pour pouvoir mesurer l'évolution des performances de l'IA.

— Gérer les collisions.
Priorité : Essentiel.
  — Entre les bots et les éléments de la carte.
  — Entre les projectiles des bots et les objets du jeu.

Les bots ne doivent pas pouvoir avancer dans une zone considérée comme solide (par exemple un mur). Un moyen pour tester est de lancer une partie avec les bots répartis sur la carte de manière homogène, et en faisant bouger les bots dans des directions aléatoires. Si un bot se retrouve sur une case étant pourtant considérée comme solide, le test échoue.

— Détecter les objets dans le champ de vision des bots.
Priorité : Essentiel.
  — Transmettre les informations sur les objets vus par les bots aux IA qui les gèrent.
  — Prendre en compte la transparence des objets de la carte. Par exemple, un bot ne peut pas en voir un autre à travers un mur opaque.

Pour tester le champ de vision des bots, il faut tout d'abord vérifier qu'en mettant un bot sur une carte de jeu, il ne détecte pas les objets n'étant pas dans son champ de vision. Si le moteur détecte une collision, il y a une erreur.

Ensuite, il faut tester que le moteur détecte bien la présence d'objet dans le champ de vision d'un bot. On place un objet aléatoirement un bot dans le champ de vision du bot. Le moteur doit le détecter, sinon il y a erreur.
Ensuite, il faut placer le bot devant un obstacle opaque, avec derrière un autre bot. Les bots ne doivent pas se détecter entre eux.

— Déplacer les bots à l'aide d'un vecteur et d'une vitesse.
Priorité : Essentiel.
Le moteur doit pouvoir calculer la nouvelle position des bots à l'aide de deux informations : un vecteur et une vitesse.

— Proposer des files de calcul aux joueurs pour les traitements.
Priorité : Conditionnel.
Le moteur doit mettre à disposition des joueurs un mécanisme permettant de réaliser des calculs trop coûteux pour être effectués sur un tour de jeu. Un soin particulier doit être appliqué à la gestion de la priorité des calculs.

— Permettre aux bots de pouvoir tirer.
  Priorité : Essentiel.
  — Respecter une cadence de tir.
  — Détecter lorsque qu'un bot est touché.
  — Dire au bot qu'il s'est fait toucher.
  — Gérer les collision entre le projectile et les éléments de la zone de jeu (par exemple les murs).

— Gérer la santé des bots.
  Priorité : Conditionnel
  — Permettre aux bots d'encaisser des dégâts dans la limite de ses points de vie.
  — Faire réapparaître les bots lorsque leur santé tombe à zéro.
  — Faire apparaître une représentation visuelle indiquant le nombre de point de vie restant des bots.

— Permettre au bot de récupérer le drapeau.
  Priorité : Essentiel.
  — Donner au bot le drapeau.
  — Lâcher le drapeau en cas de mort du bot.
  — Lâcher automatiquement le drapeau quand le bot revient dans la zone de son équipe.

  Pour tester, il suffit de téléporter un bot sur le drapeau. Si le moteur donne le drapeau au bot, le test passe. Sinon, il échoue.

— Permettre au bot de récupérer des items spéciaux
  Priorité : Optionnel.
  — Différentes armes.
  — Des bonus tels que : vitesse, gains de vie , téléportation.

  Le test est le même que pour le besoin précédent. Il faut téléporter un bot sur les différents objets, et vérifier que le moteur donne bien l'objet au bot. Sinon le test échoue.

— Permettre d'ajouter différents types de bots
  Priorité : Optionnel.
  Par exemple, permettre l'ajout d'un bot ayant la capacité de détruire certains murs.

— Modification interactive de la carte
  Priorité : Optionnel.
  — Casser les murs.
  — Différents types de murs

4.2 Besoins non fonctionnels

4.2.1 Besoins utilisateur

— L'api doit être bien documentée car possiblement destinée à des étudiants.
Quantificateur : Doit être compréhensible par un étudiant qui découvre le projet.

— L'affichage doit être fluide.
Quantificateur : Le jeu doit pouvoir tourner à 30fps, en utilisant la bibliothèque pygame.

— Le projet doit être codé en utilisant python.
Quantificateur : Doit être utilisable dans un cours où les étudiants travaillent avec python.

4.2.2 Besoins système

— Le moteur de jeu doit être robuste.
Contrainte, difficulté technique : Le moteur doit continuer à tourner avec des entrées erronées.
Risques et parades : En cas de problème de robustesse, si le moteur est utilisé pour une évaluation automatique, les résultats seront faussés. Il faut donc une gestion des erreurs efficace.
Un moyen pour tester la robustesse du moteur est l'envoi de réponses aléatoires et n'ayant aucun sens au moteur.
Le moteur doit alors continuer à tourner correctement.

— Les IA doivent répondre rapidement.
Quantificateur : Les réponses doivent être apportées à chaque frame.
Contrainte, difficulté technique : Le code étant en python, il est compliqué d'avoir des performances élevées.
Risques et parades : Le risque est que l'IA ne réponde pas à temps. Dans ce cas, l'IA se fait disqualifier ou doit passer son tour.

— Améliorer l'algorithme de pathfinding.

— Le moteur de jeu doit être capable de traiter efficacement les réponses des IA.
Contrainte, difficulté technique : La mise à jour de l'état du jeu selon les réponses doit permettre un affichage à 30 fps, sur un ordinateur possédant un processeur i5 minimum 3 threads (donc 2 cores hyperthreadés ou 3 cores minimum), et 4Go de ram.
Risques et parades : L'affichage peut perdre de sa fluidité.
Pour tester, on peut donner au moteur des réponses aléatoires mais correctes. Le test doit durer longtemps pour vérifier que le moteur tient dans la durée, sans perte de performances.

4.3 Justifications

Pour l'algorithme de pathfinding, nous avons choisi A* car c'est un algorithme performant et très utilisé dans l'industrie du jeu vidéo, il possède une bonne complexité, $\mathcal{O}(n + m\ log(n))$, n étant le nombre de sommets et m le nombre d'arêtes du graphe représentant le terrain de jeu. De plus, on peut changer l'heuristique pour que les bots puissent s'adapter aux différentes situations (éviter des zones de danger, aller au plus vite vers le drapeau, etc.).

Une première version des bots doit utiliser les behavior trees, car c'est un moyen rapide de prendre des décisions. En effet, avec de nombreux bots sur le terrain et une contrainte de temps sévère, les bots n'ont pas le temps d'utiliser des algorithmes d'exploration d'arbre, tels que l'élagage alpha bêta, de manière efficace. Un autre avantage est de garder un contrôle strict sur le comportement de l'IA et donc de respecter au mieux les stratégies définies à l'avance.

Nous avons choisi d'élaborer nous même le moteur de jeu, d'une part en raison de la faible présence d'existant comme expliqué en introduction, d'autre part pour contrôler son évolution, l'ajout de nouvelles fonctionnalités et la maintenance de celui-ci.

5 Architecture

Notre architecture est composée de cinq packages qui permettent à deux logiques distinctes de se compléter : Le moteur de jeu et l'IA.

Le moteur de jeu est composé des packages Model, Ui, Service et Domain.
L'ia est composé principalement du package AI, mais peut utiliser les objets se trouvant dans les packages Domain et Service.

Le moteur repose en grande partie sur le pattern Model/View/Controller.

Nous avons voulu, en architecturant cette application, que le maximum de chose soient remplaçables. Par exemple, il est facile de changer la bibliothèque d'affichage, ici pygame, par n'importe quelle autre. Il est de même facile de changer l'implémentation des méthodes gérant la physique du jeu, si celles-ci doivent par exemple être implémentée dans un autre langage pour gagner en performances.

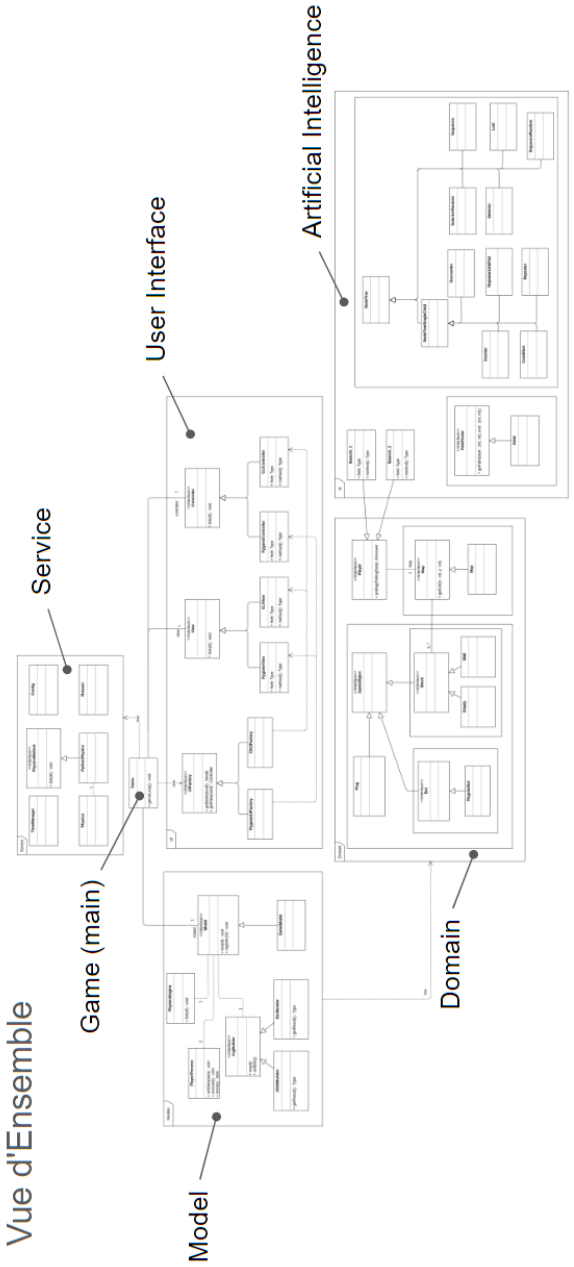


FIGURE 1 – Vue d'ensemble de l'architecture, chaque élément est précisé dans les points suivants

5.1 Service

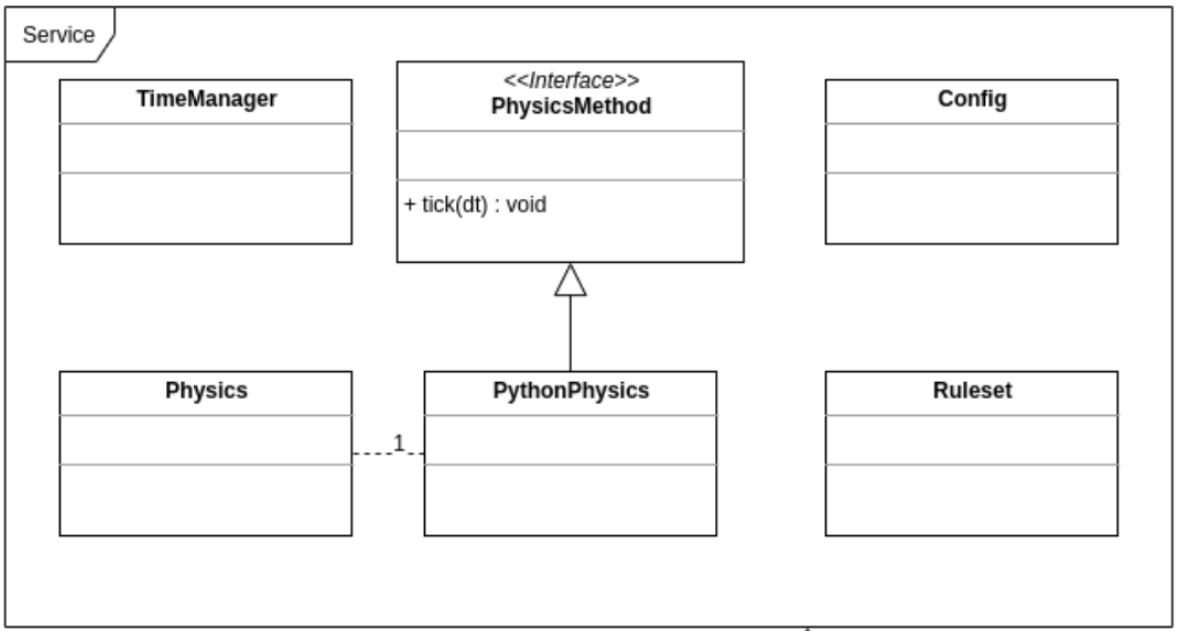


FIGURE 2 – Package Service

Service est un package indépendant dans lequel nous avons créé des utilitaires plus ou moins centrés sur le cas particulier du jeu. Ils sont accessible partout dans le code.

5.1.1 Config, Ruleset

Ces deux classes, très similaires, sont un exemple de classe centrée sur le jeu. Elles permettent respectivement de gérer un fichier de configuration pour des paramètres ou des règles de jeu. Leur utilité est d'offrir un paramétrage par défaut, la création d'un fichier de configuration automatique avec les valeurs par défaut si il n'existe pas ou bien la complétion de paramètres manquants, puis un accès aux paramètres en lecture et en écriture.

5.1.2 Physics, PhysicsMethod

A l'opposé en terme de particularité, ces classes servent à implémenter des méthodes calculatoires génériques tout en gardant la main sur la technologie utilisée. Par défaut, nous utilisons l'implémentation PythonPhysics.

5.1.3 TimeManager

Cette classe sert à mesurer ou manipuler des données de temps comme par exemple chronométrer ou passer le temps jusqu'à un prochain tour.

5.2 Partie Moteur de jeu

5.2.1 Domain

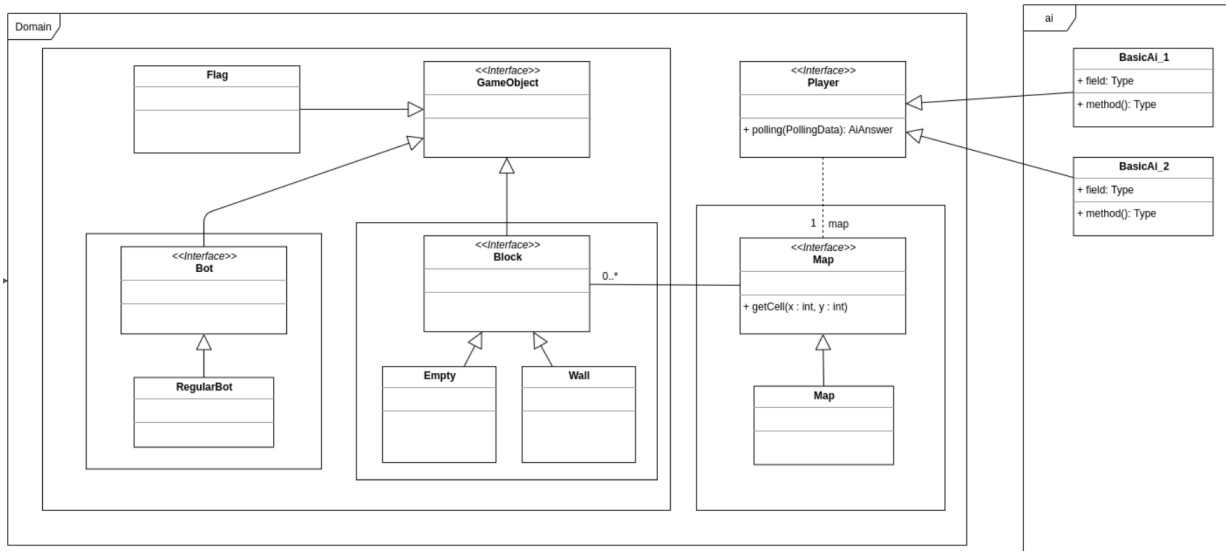


FIGURE 3 – Package Domain

Ce package contient des classes de description des données tel que la carte, ses blocks et ses objects, les différents Bots à contrôler ainsi que le joueur les contrôlant.

Elle est très utilisée par le modèle du moteur de jeu et sa définition est connue par les joueurs qui peuvent s'en servir pour former leur propre représentation s'ils le souhaitent, grâce aux données qui leur sont communiquées à chaque tour.

5.2.2 Model

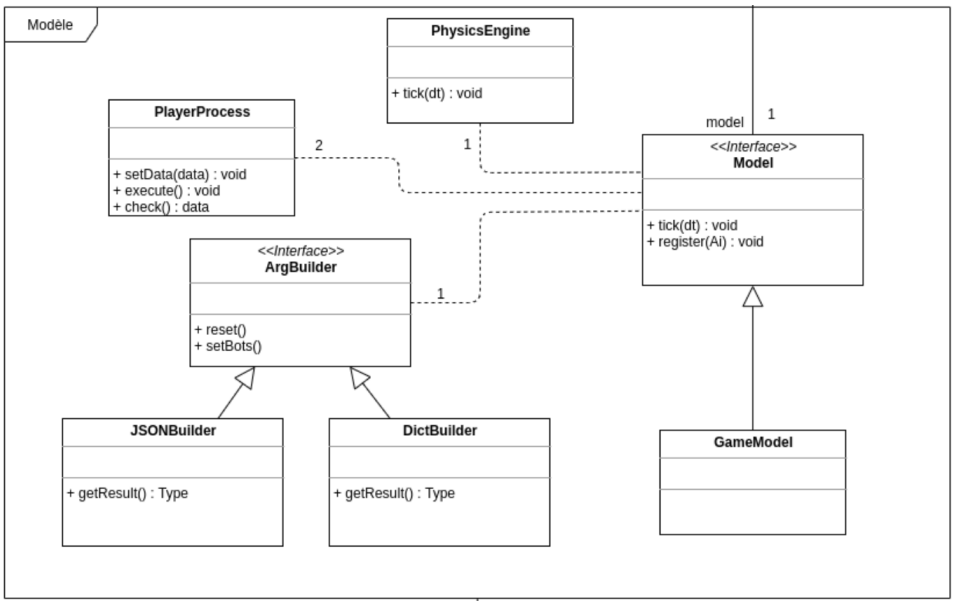


FIGURE 4 – Package Model

C'est le package sur laquelle le jeu se base pour gérer les tours. Il contient le modèle à qui on demande d'initialiser et d'entretenir tous les éléments présents sur la carte, y compris les joueurs.

Le modèle interagit avec le moteur physique qui l'aide à prendre les bonnes décisions en terme de déplacements, interactions physiques et autres calculs. Il contient et gère deux PlayerProcess qui seront l'environnement dans lequel évolueront les IA. Enfin, il utiliser un ArgBuilder afin de correctement formaliser les données qui seront envoyées aux PlayerProcess.

Pour entrer dans plus de détails concernant PlayerProcess, c'est une simple implémentation de Process qui lance la fonction de polling d'un Player lorsque le modèle lui envoie les données d'un nouveau tour dans une queue. Le modèle récupère le retour à la fin du temps imparti pour ce tour. De cette manière, les IA ne peuvent pas directement impacter la performance du moteur de jeu en utilisant tout le temps processeur : ce serait son propre temps processeur.

5.2.3 User Interface

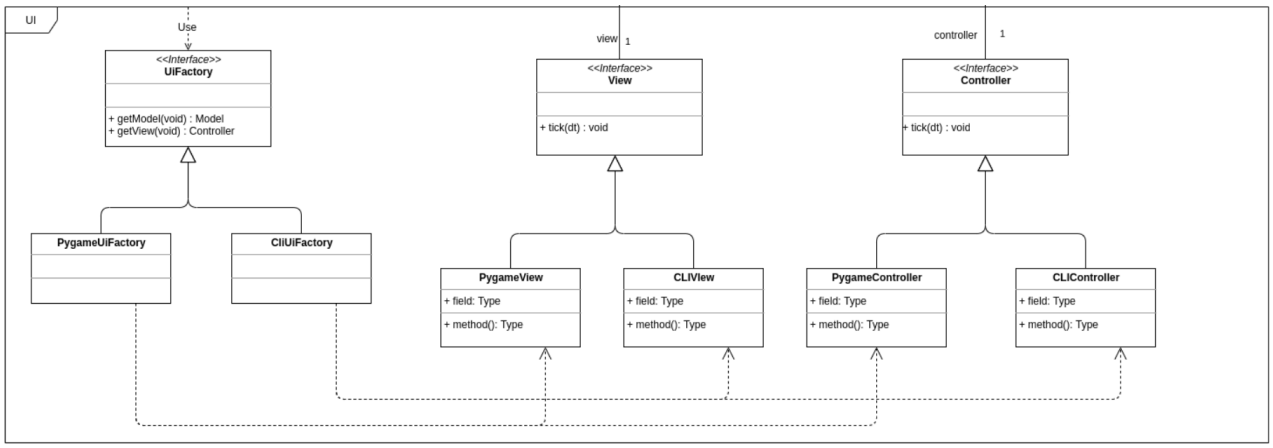


FIGURE 5 – Package User Interface

Ce package est une factory d'interface view/controller utilisée par le jeu. Ceux-ci sont simplement appelés à chaque tour de jeu et ont la possibilité d'intéragir avec le modèle.

Dans notre implémentation, on dispose de la vue Pygame (graphique avec des contrôles de debug) et de la vue en CLI (minimaliste mais permet de ne pas "gaspiller" de temps pour une interface).

5.3 Partie IA

Tout d'abord, l'architecture pour les codes relatifs à l'IA est décomposée en trois sous-parties étant :

— Un package PlayerTest où différentes IA sont implémentés.
— Un package BehaviorTree relatif aux arbres de comportements.
— Un package Pathfinding pour la recherche de chemin utilisées pour les déplacements.

5.3.1 Behavior Tree

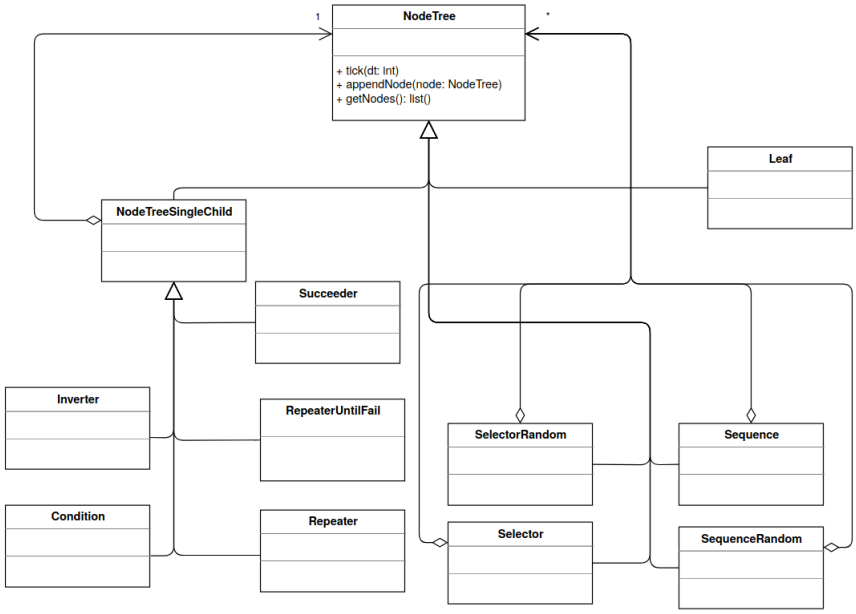


FIGURE 6 – Architecture Behavior Tree

L'ia va reposer sur un arbre de comportement, qui est un automate, mais avec une hiérarchisation des noeuds. Chaque noeud peut posséder ou non des enfants. Chaque type de noeud redéfini la méthode *tick()*, qui permet d'avancer dans l'arbre.
Chaque noeud doit renvoyer une valeur parmi ces trois : *SUCCESS*, *FAILURE* ou *RUNNING*.
Les noeuds possède une mémoire pour se souvenir du noeud en cours d'exécution, pour pouvoir reprendre son exécution lors du prochain tick.

Nous utilisons donc le pattern composite pour cette partie de l'architecture. *NodeTree* est donc l'interface qui permet la gestion des noeurs enfants. Trois types de noeuds implémente cette interface. *NodeTreeSingleChild* et ses dérivés sont des noeuds ne pouvant posséder qu'un seul enfant. En cas de tentative d'ajout, une exception est levée.
Il y a ensuite *Leaf*, qui est la feuille de l'arbre. Cette feuille prend dans son constructeur un pointeur vers la fonction qui sera appelée lors du *tick()*. Comme pour *NodeTreeSingleChild*, une tentative d'ajout d'un noeud lève un exception, une feuille ne pouvant avoir de fils.

Il y a ensuite tous les autre noeuds, qui n'ont rien de particulier dans la gestion des enfants, seul la fonction *tick()* est redéfinie.

5.3.2 Pathfinding

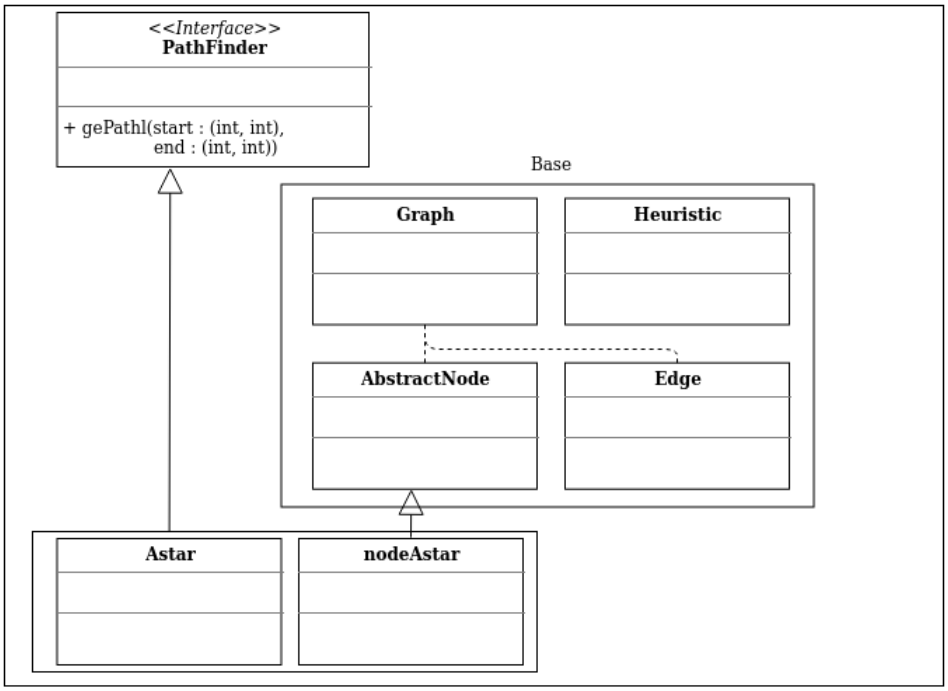


FIGURE 7 – Architecture Pathinding

Pour la package Pathfinding, nous avons organisé de manière à ce que n'importe quel algorithme de recherche de chemin puisse être implémenté et utilisé sans avoir à modifier le code des joueurs mise à part la sélection de l'algorithme.

Nous avons d'une part le package Base où tout outil nécessaire à la création de chemin a été généralisé, c'est à dire une classe pour les noeuds abstraite que chaque algorithme pourra hériter selon ses besoins, une classe pour arête et graphe ainsi qu'une interface pathfinder avec une seule méthode getPath pour récupérer le chemin calculé.

Chaque algorithme de pathfinding doit hérité et utilisé cette base, par exemple pour l'algorithme A*, nous avons la classe Astar qui implémente pathFinder ainsi que NodeAstar qui implémente la classe pour les noeuds (Abstract Node).
Tout ajout d'algorithme de pathfinding se fera par la création d'un package relatif à ce pathfinding, avec deux classes minimum, une implémentant la représentation abstraite des noeuds, et une implémentant l'interface pathfinder, c'est à dire une classe avec une fonction getPath permettant de renvoyer le chemin calculé (une liste de points).

Cette représentation permet d'ajouter facilement n'importe quel algorithme de recherche de chemin sans avoir à perturber le fonctionnement des autres instances tout en permettant de mettre tout le monde d'accord sur quelle est la représentation d'un noeud, graphe, arête...

Pour le package PlayerTest, toute nouvelle IA ou nouveau joueur devant être implémenté doit hériter de la classe Player du domaine, étant le contrat à respecter pour pouvoir communiquer avec le moteur de jeu, c'est à dire pouvoir recevoir des informations sur l'évolution de la partie et en retour pouvoir envoyer nos prochaines actions que le moteur de jeu doit effectuer. La classe MyLittlePlayer représente une IA avec arbre de comportement et se déplaçant à l'aide de l'algorithme A*.
La classe MyPlayerAstar représente une IA sans arbre de comportement se déplaçant avec A*.

6 Analyse et tests

6.1 Moteur

Pour le moteur, nous avons réalisé des tests unitaires. Ceux-ci se découpent en plusieurs parties :

— Les tests de la physique, vérifiant que nos formules retournent les résultats auxquels nous nous attendons lorsque nous les utilisons. Ceux-ci sont incomplets car nous n'utilisons plus certains formules dans le code suite à des modifications.

— Les tests du modèle, limités car les bugs sont en général surtout observables depuis l'interface graphique. Ils consistent à vérifier que l'initialisation du modèle et des processus de joueurs se déroule correctement, que les tours se déroulent comme attendu et que le chronomètre du jeu soit correctement interprété.

— Les tests de construction d'argument, utiles pour garder la cohérence dans les communications entre IA et moteur : critique pour que les parties se déroulent normalement et que l'implémentation des IA soit sans surprises.

Au niveau du profilage des performances, nous utilisons *cProfile*, qui nous permet de connaître quelle partie de code nous prend le plus de temps.
Par exemple, nous pouvons voir que *getImpactPoint* et *getShootedBot* prennent une part importante du temps dans la physique du jeu.

```
    ncalls tottime percall cumtime percall filename:lineno(function)
1229 0.022 0.000 26.205 0.021 GameModel.py:113(tick)
189 0.850 0.004 0.850 0.004 PhysicsEngine.py:233(getImpactPoint)
189 0.006 0.000 0.867 0.005 PhysicsEngine.py:277(getShootedBot)
```

Pour tester les cas extrêmes, nous avons modifié le terrain à l'ajout de nouvelles fonctionnalités afin de couvrir le plus de situations. Comme nous n'avons trouvé aucun cas de la sorte en nous mettant à la place des utilisateurs, nous pensons que notre gestion est suffisamment solide pour être équitable.

6.2 Intelligence Artificielle

Les tests de l'ia sont découpés en deux parties. Nous avons tout d'abord les tests du code utilisé par les ia, c'est à dire l'implémentation des behaviors trees, et ensuite les tests du comportement de l'ia. Nous avons commencé par les premiers, les seconds devant attendre que le moteur soit terminé pour être réalisés correctement.

Les tests des behaviors trees sont composé uniquement de tests unitaire sur chacun des noeuds, et des tests sur le bon déroulement de la méthode *tick()* sur des compositions de ces noeuds.
Après avoir testé unitairement les noeuds, nous avons créé un nouveau joueur, *myLittlePlayer*, pour tester l'intégration de ce système dans le reste de notre application. Nous mettons en place un comportement simple : L'ia trouve un chemin jusqu'au drapeau adverse, y envoie ses bots. Ensuite, elle cherche un chemin jusqu'à la zone de dépôt, et y envoie ses bots.
Nous n'avons pour le moment pas rencontré de bug avec l'utilisation de notre implémentation des arbres de comportement.

Pour les tests de comportement, nous mettons en place différents scénarios.
Pour le moment, seul des scénarios simples sont utilisé. Deux équipes sont présentes dans le jeu, le résultat attendu est que l'une des équipes gagne.

Deux problèmes ressortent de ce test : La gestion des chemins amène parfois les bots à se bloquer dans les murs, et la prise d'item ne se fait pas correctement. Les bots passent parfois juste à côté des drapeaux, sans passer dessus.
Ces problèmes viennent cependant de l'ia en elle même, et non de l'implémentation des behaviors trees ou du pathfinding. Nous n'avons donc pas passé beaucoup de temps dessus, nous avons préféré continué à ajouter des fonctionnalités.

7 Conclusion

Nous n'avons pas entièrement fini ce projet, il reste en effet plusieurs tâches à effectuer.

Du côté de l'intelligence artificielle, le plus gros du travail reste à faire. Il faut en effet utiliser les *behavior trees* pour créer des comportements plus complexes.
Tous les éléments permettant la réalisation de l'ia sont déjà réalisés. Il ne reste plus qu'à créer les méthodes pour les feuilles, et mettre en place un comportement avec les noeuds de structures existants.
Il faut également ajouter la gestion du temps à l'aide du paramètre dt dans les noeuds.
Plus de scénarios de tests doivent être réalisés.

Le moteur est suffisant pour accueillir des terrains variés, des parties entre IA et des évolutions futures. Voici ce que nous aurions cependant aimé terminer :

Tout d'abord l'intégration du joueur humain. Il faut ajouter dans le contrôleur les actions nécessaires au contrôle d'une équipe, et ensuite traduire ces actions en paramètres pour les envoyer au modèle.

Un fonctionnalité que nous pensions nécessaire était de représenter graphiquement le champ de vision en prenant en compte les obstacles. Après consultation avec le client, il s'est avéré que cela n'était pas important, nous avons donc basculé sur une gestion plus basique et moins gourmande qu'il nous a proposé en réunion.

Ensuite, il est possible de rajouter des bonus capables d'affecter les bots de multiple manières. Actuellement, il en existe trois : Un bonus de vitesse, un soin instantanée et une plate-forme de régénération progressive des points de vie. Ces bonus servent en partie à montrer quelles possibilités sont offertes par l'implémentation, avec un minimum de code.

Le client nous a précisé que l'ajout d'une mécanique de jeu "unique" et "originale" serait un grand plus, d'où ces bonus qui permettent d'en rajouter une. Nous n'avons cependant pas eu d'inspiration pour en trouver une à part un bonus qui agrandit le champ de vision en taille ou en diamètre. Cela nous semble original, mais difficile de savoir si cela est suffisament intéressant comparé au déséquilibrage et à la difficulté à gérer les IA.

Malgré ces fonctionnalités manquantes, nous sommes fiers de notre travail. Le client semble être convaincu et nous avons atteint les objectifs essentiels, ce qui semblait compromis notamment à cause des difficultés accrues à nous organiser dans cette période.

8 Bibliographie

[1] Joanna Joy Bryson. « Intelligence by Design : Principles of Modularity and Coordination for Engineering Complex Adaptative Agents. » In : (2001).

[2] Michele Colledanchise et Petter Ögren. « Behavior trees in robotics and AI : an introduction ». In : *arXiv preprint arXiv :1709.00084* (2017).

[3] Max Jaderberg et al. « Human-level performance in 3D multiplayer games with population-based reinforcement learning ». In : *Science* 364.6443 (2019), p. 859-865. issn : 0036-8075. doi : 10.1126/science.aau6249. eprint : https://science.sciencemag.org/content/364/6443/859.full.pdf. url : https://science.sciencemag.org/content/364/6443/859.

[4] Piers Williams. « Artificial intelligence in co-operative games with partial observability ». Thèse de doct. University of Essex, 2019.

[5] Qi Zhang et al. « Learning Behavior Trees for Autonomous Agents with Hybrid Constraints Evolution ». In : *Applied Sciences* 8.7 (juil. 2018), p. 1077. issn : 2076-3417. doi : 10.3390/app8071077. url : http://dx.doi.org/10.3390/app8071077.