



SORBONNE UNIVERSITÉ

2I013 : PROJET (APPLICATION)

Sujet :  
**IA Football**

*Fangzhou Ye*  
*Angelo Ortiz*

# Table des matières

<b>Introduction</b>	<b>2</b>
<b>I Architecture logicielle</b>	<b>3</b>
1 Module de base	3
2 Additions chemin faisant	4
3 Librairies externes	5
<b>II Méthodes d'optimisation</b>	<b>6</b>
4 Recherche en grille	6
5 Algorithmes génétiques	6
6 Apprentissage automatique	7
<b>III Stratégies</b>	<b>10</b>
7 Un joueur	10
8 Deux joueurs	11
9 Quatre joueurs	13
<b>Conclusion</b>	<b>15</b>

## Table des figures

1	Un aperçu du jeu . . . . .	2
2	La structure du module . . . . .	3
3	Les stratégies de base . . . . .	4
4	Les stratégies pour la recherche en grille . . . . .	4
5	Les classes pour l'algorithme génétique . . . . .	4
6	Les stratégies pour l'arbre de décision . . . . .	5
7	Les classes pour l'apprentissage automatique . . . . .	5
8	Un exemple d'arbre de décision . . . . .	8

# Introduction

## Présentation

L'unité d'enseignement 2I013 consiste en un projet d'application des connaissances en programmation et de découverte de nouveaux algorithmes. La thématique du projet du groupe 1 a été le football.

Le jeu comporte un terrain, deux équipes de joueurs et une balle. Étant donné qu'une grille rectangulaire sous forme matricielle, c'est-à-dire discrète, n'était pas envisageable, un repère orthonormé a été choisi comme représentation du terrain. Pour ce faire, il y a deux droites en plus des axes qui délimitent la surface du terrain.

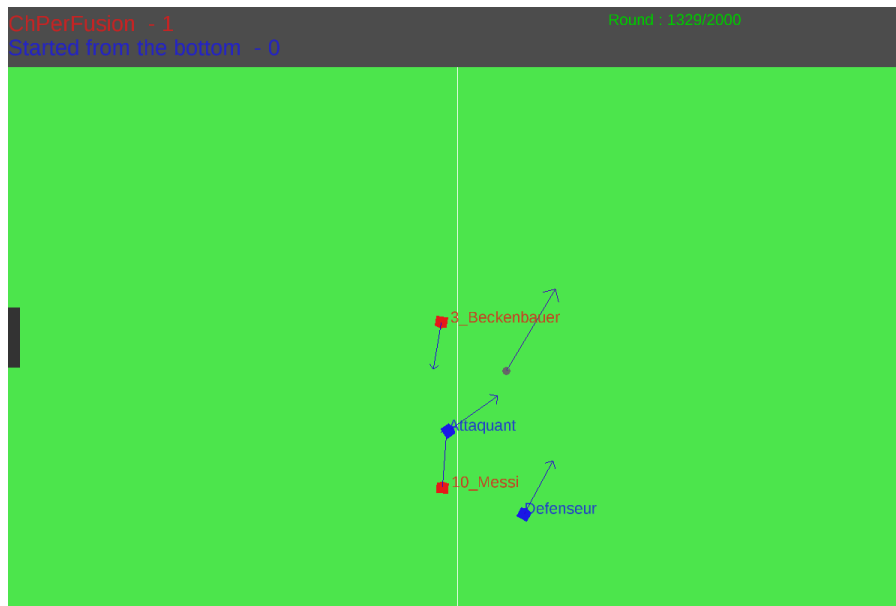


FIGURE 1 – Voici un aperçu du simulateur de football lors d'une partie de notre équipe *ChPerFusion* en rouge face à l'équipe d'un de nos camarades de classe.

Les règles du football ont été légèrement simplifiées dans le cadre du projet. En effet, il n'existe pas de notion de touche, mais de rebond : la balle ne peut pas sortir du terrain et elle choque élastiquement dans les bordures. Par ailleurs, les objets physiques du jeu, à savoir les joueurs et la balle, sont intangibles. Ceci veut dire que la balle peut passer à travers les joueurs sans modifier sa direction. C'est le cas notamment lorsque le joueur décide de ne pas frapper.

Pour rendre le jeu plus réaliste, la notion de frottement a été prise en compte, ce qui facilite le contrôle de la balle aux joueurs. Le moteur du jeu gérant toutes ces règles et les interactions entre les objets a été fourni.

## Objectifs

L'objectif du module a été d'apprendre à bien mener un projet long sur un nouvel environnement de développement, à savoir Python. Sachant que travail à été fait en binôme, on a eu besoin d'un outil collaboratif, en l'occurrence **git** qui est un logiciel de gestion de versions décentralisé. On a ainsi appris à gérer l'avancement d'un projet à travers la méthode par fonctionnalités.

Le travail a consisté à développer des intelligences artificielles dirigeant la prise de décisions des joueurs pour réussir au mieux les matches dans les catégories à un, deux et quatre joueurs.

## Première partie

# Architecture logicielle

Dans cette première partie, on vous expliquera les choix faits concernant l'architecture logicielle.

Le projet a été organisé en un module **ia** contenant les fichiers sources de nos joueurs, un répertoire regroupant les différentes versions des paramètres et en divers fichiers exécutables permettant de lancer les algorithmes d'optimisation.

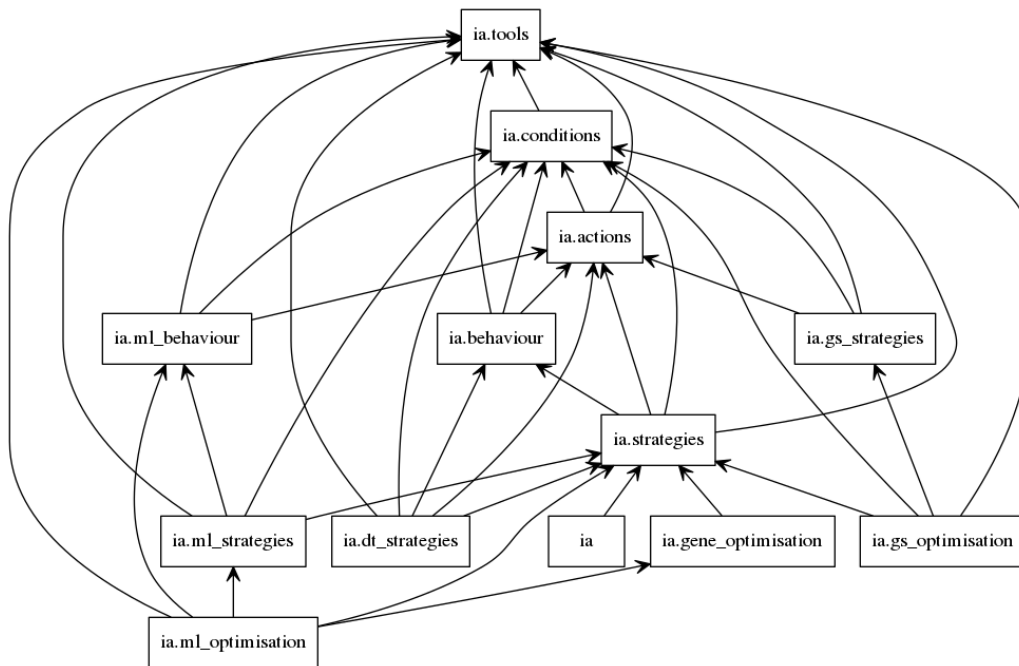


FIGURE 2 – La structure de notre module **ia** développé tout au long du semestre

## 1 Module de base

Au tout début du projet, on a réalisé une analyse approfondie des fonctionnalités à implémenter et de la structure du simulateur fourni. On a conclu qu'il était impératif de séparer les fonctions et classes en plusieurs fichiers pour mieux les repérer. On comptait ainsi de base quatre fichiers.

Dans le fichier *tools*, on a regroupé une classe enveloppe facilitant l'accès aux informations d'un état de jeu donné et des fonctions utilitaires, par exemple, des implémentations de formules mathématiques.

Puis on retrouve le fichier *conditions* qui contient toutes les fonctions booléennes dirigeant la prise de décisions. Autrement dit, le comportement d'un joueur varie selon les valeurs de vérité de ses conditions associées.

On passe ensuite au cœur du module, à savoir le fichier *actions*. On y retrouve les actions de base qu'un joueur peut effectuer comme frapper, faire une passe ou revenir vers la cage.

On a vu ces fichiers grossir au fur et à mesure que le projet avançait, notamment *actions*. On s'est alors vu contraints d'ajouter un autre fichier pour améliorer la structure de base. C'est ainsi que le fichier *behaviour* est apparu. Il contient en effet des super-actions, i.e. le comportement d'un joueur pour une phase du jeu donnée, par exemple, la situation d'attaque avec la possession du ballon.

Par ailleurs, on communique la prise de décisions de nos joueurs au simulateur du jeu au moyen des classes appelées stratégies. Celles-ci sont incluses dans le fichier de même nom. Ces stratégies correspondent alors au comportement d'un joueur pendant plusieurs phases du jeu, c'est-à-dire tout au long d'un match.

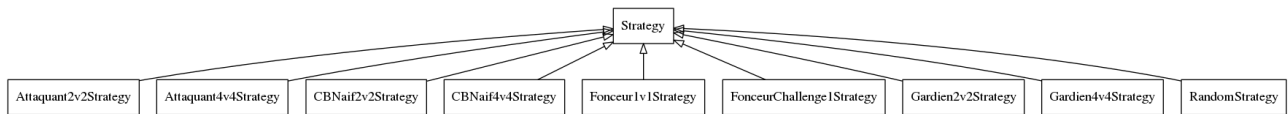


FIGURE 3 – Les différentes stratégies pour les équipes à un, deux ou quatre joueurs, une stratégie complètement aléatoire et une autre pour l'unique challenge proposé

## 2 Additions chemin faisant

Les stratégies conçues ont évolué pendant tout le semestre. On a vu quatre méthodes d'optimisation en cours pour rendre plus intelligentes ces stratégies de base.

Pour chaque méthode d'optimisation, on a eu besoin d'une classe réalisant l'optimisation sur la base des données recueillies soit par elle-même, soit par des nouvelles stratégies reposant sur celles déjà codées.

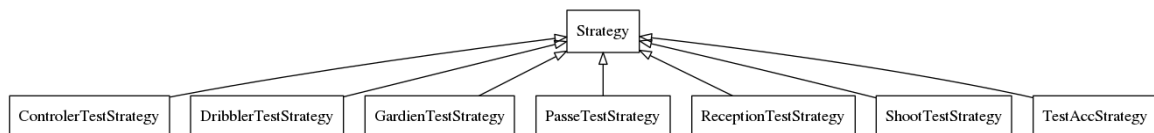


FIGURE 4 – Les stratégies à améliorer à l'aide de la recherche en grille

La figure 4 contient le diagramme de classes des stratégies spécialement conçues pour la recherche en grille de paramètres. Ces stratégies ont été utilisées par des *observateurs* des matches pour obtenir les valeurs optimales cherchées. Les fichiers concernés par cette méthode sont repérés par le préfixe « gs », abrégé du terme en anglais *grid search*.

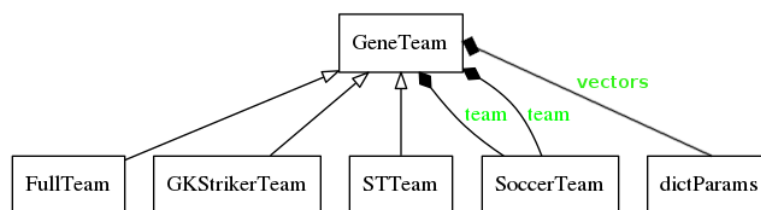


FIGURE 5 – Le diagramme de classes pour la mise en œuvre d'un algorithme génétique

Pour la méthode par un algorithme génétique, on a développé deux classes génériques pour traiter le bilan des résultats des stratégies de base dans plusieurs matches. On voit dans la figure 5 le diagramme de classes résultant. Le fichier créé pour cette méthode est repéré par le préfixe « gene ».

La méthode d'optimisation par arbres de décision est différente des autres en ce qui concerne la conception. En effet, la classe l'implémentant étant déjà fournie, il a suffi de développer les stratégies briques qui allaient former une stratégie intelligente. Vous pouvez les apercevoir dans la figure 6. Le fichier contenant ces stratégies est préfixé par « dt », abrégé du terme en anglais *decision tree*.

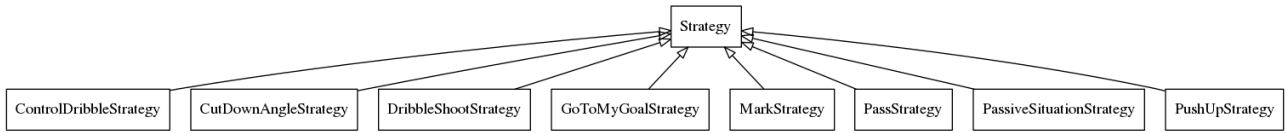


FIGURE 6 – Les stratégies utilisées pour l'application de la méthode par arbres de décision

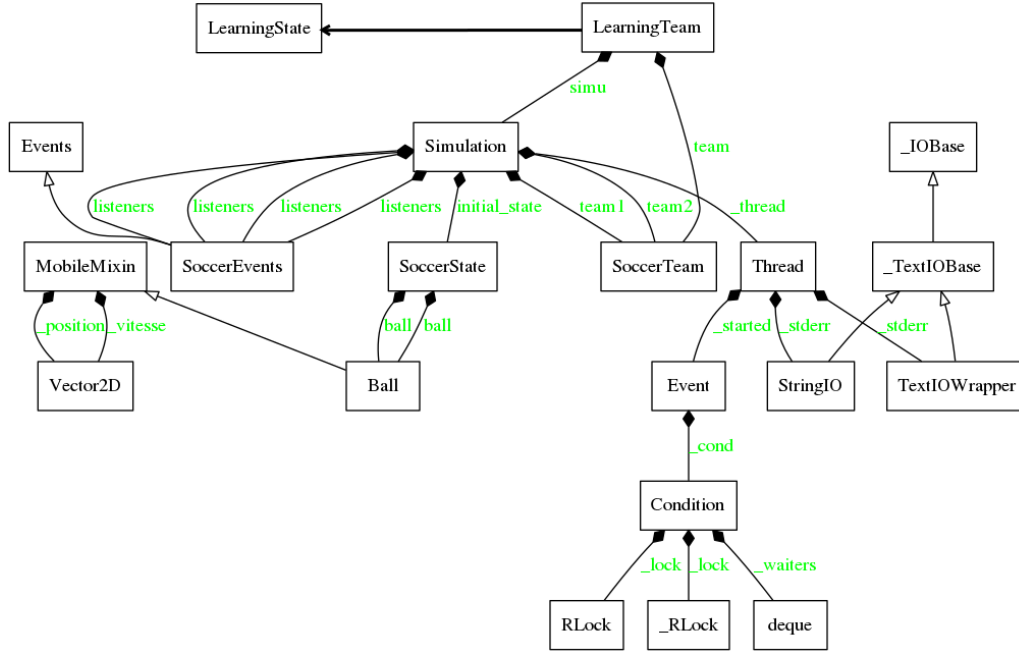


FIGURE 7 – Le diagramme de dépendances des classes `LearningState` et `LearningTeam` pour l'implémentation de l'algorithme Q-learning

La dernière méthode d'optimisation était facultative, mais on a voulu suivre le module jusqu'au bout. On a dû coder tout l'algorithme et les briques nécessaires pour sa mise en place. Tout comme la méthode précédente, celle-ci a utilisé des stratégies briques. Par ailleurs, on a implémenté l'algorithme à l'aide de deux classes dites d'*apprentissage*. Le préfixe utilisé pour cette méthode a été « ml », abrégé du terme en anglais *machine learning*.

Les fichiers exécutables utilisés pour tester nos algorithmes d'optimisation se trouvent au même niveau que le module `ia` préfixés par « exe ».

### 3 Bibliothèques externes

Il est important de rappeler que l'on compte à présent des nombreuses bibliothèques disponibles et très souvent performantes qui ont été le fruit du travail de beaucoup d'informaticiens avant nous. Ceci dit, on doit remarquer que l'on en a utilisé quelques-unes durant le développement du projet. On peut citer notamment *math*, puisque la physique du jeu comporte un gros composant mathématique ; *random*, en raison de notre engouement pour l'aléatoire ; *pickle* pour la sérialisation des données ; *os.path* pour l'accès au système de fichiers de l'ordinateur ; et *numpy* pour les opérations simplifiées sur les tableaux.

## Deuxième partie

# Méthodes d'optimisation

Dans cette deuxième partie, on vous présente les différents algorithmes appris tout au long du semestre pour améliorer les stratégies proposées.

### 4 Recherche en grille

Il s'agit de la première méthode d'optimisation présentée en cours. Elle a été utilisée pour l'optimisation d'une action.

Tout d'abord, on doit définir l'action à optimiser et le critère d'optimalité, et repérer les  $n$  paramètres associés. Ces paramètres réels sont dénotés  $x_i$ , avec  $i = 1, \dots, n$ . Le critère d'optimalité correspond alors à la fonction  $f : \mathbb{R}^n \rightarrow \{0, 1\}$ .

Puis on obtient l'ensemble  $\mathbb{E}_i$  de valeurs possibles pour chaque paramètre. Pour les paramètres discrets, il suffit d'utiliser l'ensemble fini des valeurs qu'il peut prendre. Quant aux paramètres continus, on compte un intervalle fermé  $I_i$  de valeurs possibles pour chaque  $x_i$ . Il est alors nécessaire de faire en amont une discrétisation, i.e. on divise l'intervalle selon un pas de précision. La fonction se transforme alors en  $f : \mathbb{E} \rightarrow \{0, 1\}$ , où  $\mathbb{E} = \mathbb{E}_1 \times \dots \times \mathbb{E}_n$ .

On fait ensuite une recherche exhaustive sur tout l'espace de points  $\mathbb{E}$  : on teste chaque vecteur  $x \in \mathbb{E}$  sous différentes conditions environnementales et on moyenne les évaluations par  $f(x)$  dans lesdites conditions. Finalement, on prend la valeur optimale et son vecteur  $x$  associé.

Il est important de remarquer que la plupart de nos paramètres sont continus. Pour obtenir des valeurs plus précises et un meilleur comportement, on a besoin d'un pas de discrétisation très petit. De ce fait,  $|\mathbb{E}| \gg 1$ . À cause des longs temps d'exécution, on a décidé que l'algorithme mis en place ne convenait pas à moyen terme.

### 5 Algorithmes génétiques

Dans un deuxième temps, on est passés à une méthode heuristique au lieu d'une recherche exhaustive de l'espace de représentation. En effet, une solution approchée est suffisante dans un certain intervalle de confiance. Ceci a donné lieu à la mise en place d'une implémentation d'algorithme génétique pour optimiser toutes nos équipes.

Cette classe d'algorithmes se base sur la théorie de l'évolution. En effet, la notion de sélection naturelle est le mécanisme permettant de choisir des solutions potentielles de plus en plus meilleures.

La toute première chose à faire est la définition de la fonction  $f : \mathbb{R}^n \rightarrow \mathbb{E}$  à optimiser, où  $n$  est le nombre de paramètres concernés. Un élément  $x = (x_1, \dots, x_n) \in \mathbb{R}^n$  est appelé *candidat* ou *chromosome*, et chacune de ses coordonnées  $x_i$  avec  $i = 1, \dots, n$ , appelée *propriété* ou encore *gène*, correspond à un paramètre à optimiser. Elles forment le *génotype*. Autrement dit, un candidat correspond à une équipe définie par les valeurs de ses propriétés.

Dans le cadre du projet, on a défini  $f(x) = (V, N, D, P, C)$  comme le bilan après avoir disputé plusieurs matches avec les mêmes valeurs des paramètres données par  $x$ , i.e. le tuple composé du nombre de victoires, matches nuls, défaites, buts marqués et buts encaissés. La relation d'ordre suit les règles du classement d'un tournoi de football. De plus, l'ensemble  $\mathbb{R}^n$  a été réduit à  $I_1 \times \dots \times I_n$ , avec  $I_i$  l'intervalle de valeurs optimales pour le paramètre  $x_i$  pour chaque  $i = 1, \dots, n$ .

Cette classe d'algorithmes comportent quatre étapes.

1. Initialisation : Tout d'abord, il faut générer un ensemble de candidats  $\{x_1, \dots, x_m\}$ . Le but est d'obtenir une population initiale la plus diverse possible de sorte à pouvoir atteindre le plus de maxima relatifs. C'est pourquoi cette première génération est obtenue de manière aléatoire.
2. Évaluation : Ensuite, on évalue la fonction en chacun des candidats, i.e. on calcule  $\{f(x_1), \dots, f(x_m)\}$ .
3. Sélection : Suivant la notion de sélection naturelle, on ne conserve que les candidats avec les meilleurs résultats. Autrement dit, on dresse un classement des équipes associées aux candidats et n'en garde qu'une partie.
4. Reproduction : Finalement, on attribue les places libres aux candidats nés du brassage génétique des candidats les plus performants. Pour ce faire, on compte sur deux méthodes, à savoir le croisement et la mutation. Étant donné deux parents, on commence par les cloner en deux enfants. Puis on choisit une ligne de coupe divisant le génotype en deux parties. Ensuite, les enfants s'échangent une partie de leur génotype. Le croisement s'arrête à ce stade-là, tandis que la mutation ajoute du bruit aléatoirement dans un gène pour chacun des nouveaux chromosomes. On a décidé de faire une petite modification à cette étape-là lors de l'implémentation pour avoir encore plus de diversification : une minorité des places libres est attribuée à des candidats générés aléatoirement.

Cet algorithme étant itératif, on recommence à la deuxième phase avec la nouvelle génération. Au bout d'un nombre raisonnable d'itérations, on se retrouve avec une génération composée de chromosomes très proches des maxima relatifs. On finit donc l'algorithme par prendre le candidat optimal de la dernière population.

Cette méthode a permis d'améliorer nos équipes progressivement et les valeurs de tous les paramètres mis en place ont été définies au moyen de nombreuses exécutions de cette méthode.

## 6 Apprentissage automatique

Dans la dernière partie du semestre, on s'est concentrés sur l'apprentissage automatique. Les enseignants ont présenté les deux grandes familles d'algorithmes d'apprentissage : supervisés et non supervisés, ainsi que l'apprentissage par renforcement. Deux méthodes ont été implémentées lors du projet.

### Arbres de décision (de classification)

Il s'agit d'une technique d'apprentissage supervisé. Pour l'appliquer, on a besoin d'un *espace de représentation*  $\mathcal{X}$ , d'un ensemble d'*étiquettes* ou *classes*  $Y$  et d'une liste de  $m$  exemples d'apprentissage  $(x^i, y^i)$ , où  $x^i \in \mathcal{X}$ ,  $y^i \in Y$ ,  $i = 1, \dots, m$ , appelée *ensemble d'apprentissage*. Le but de cette technique est de trouver une fonction  $f : \mathcal{X} \rightarrow Y$  telle que l'on puisse prédire à quelle classe appartient une future variable  $x \in \mathcal{X}$ .

Concrètement, on veut associer une certaine action à chaque état du jeu. Comme les informations contenues dans les états du jeu ne sont pas généralisables dans l'état, on utilisera une représentation par  $n$  caractéristiques généralisées qui précisent la configuration du terrain. On a ainsi que  $\mathcal{X} = \mathbb{R}^n$ , avec  $n$  la dimension de l'espace de représentation. De plus, l'ensemble d'étiquettes devient l'ensemble d'actions possibles  $A$ . On veut donc trouver une fonction  $f : \mathbb{R}^n \rightarrow A$  qui fournit la meilleure action possible pour tout état du jeu.

Le problème auquel on est confrontés se réduit alors à partitionner l'espace de représentation en  $p = |A|$  parties  $\mathcal{X}_1, \mathcal{X}_2, \dots, \mathcal{X}_p$  de sorte que  $f([\mathcal{X}_i]) = \{a_i\} \subset A$ ,  $i = 1, \dots, p$ . On utilise les arbres de décisions pour représenter ce partitionnement. Ceci implique que chaque nœud interne



correspond à un test sur une des  $n$  dimensions de  $\mathcal{X}$  et que chaque feuille correspond à l'action à entreprendre pour l'état donné par le chemin suivi depuis la racine.

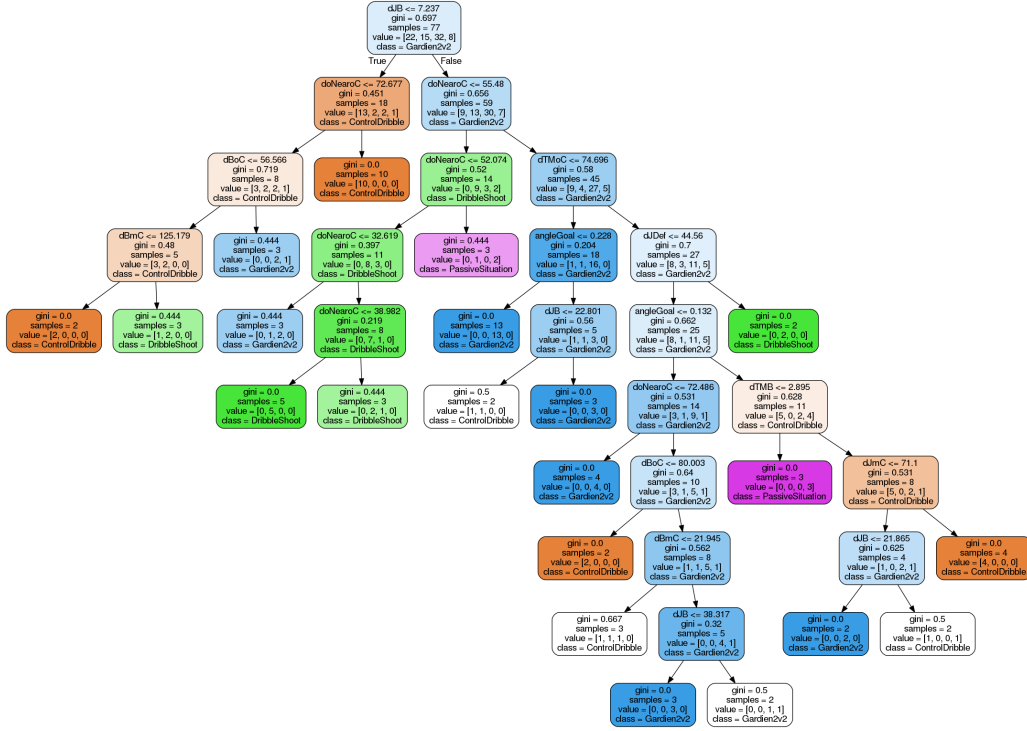


FIGURE 8 – Voici un exemple d'arbre de décision obtenu à l'issue de l'exécution de l'algorithme proposé

On nous a fourni une stratégie spéciale **KeyboardStrategy** comportant les outils de base nécessaires à la mise en place de cette méthode, allant de la génération de l'ensemble d'apprentissage au partitionnement de l'espace de représentation, en passant par l'application de la fonction lors des matches. Elle permet d'associer des touches clavier aux différentes stratégies de sorte que l'on puisse préciser la stratégie préconisée pour les différentes phases du jeu. Ceci conforme ce que l'on appelle l'ensemble d'apprentissage. Cet ensemble est traité à l'aide des  $n$  caractéristiques généralisées définies auparavant et, à l'issue de ceci, on compte un partitionnement de  $\mathbb{R}^n$ . Cette stratégie utilise alors la fonction  $f$  lors des matches suivants, ce qui donne un comportement plus intelligent à nos joueurs.

Cette méthode n'a pas été utilisée pour améliorer la prise de décisions sur les actions parce que l'on a trouvé difficile l'interaction à travers le clavier pour plus de quatre actions.

## Q-learning

Cet algorithme d'apprentissage par renforcement repose sur un ensemble d'états  $S$ , un ensemble d'actions  $A$ , et une fonction  $Q : S \times A \rightarrow \mathbb{R}$ , appelée *politique*, pondérant une paire formée par un *état* donné et une *action* à entreprendre pour cet état-ci. Cette pondération est nommée *récompense*. La notion d'état correspond à celle donnée dans la section des arbres binaires de décision. Il utilise aussi la notion d'*épisode* que l'on associe à un match dans le cadre du projet. Son principe est le suivant : on apprend une politique comportementale qui optimise l'espérance des récompenses.

La phase d'initialisation consiste à attribuer à  $Q$  une valeur fixe arbitraire, ici 0. La phase itérative concerne un épisode, ou encore un match, et est répétée pour les  $k$  épisodes. Pour un état  $s_t$  donné, on choisit une action  $a_t$  selon la politique  $Q$  et observe la récompense  $r_t$  associée à cette paire. Cette action génère un nouvel état  $s_{t+1}$ . Il est à noter qu'il n'y a pas de choix

concernant l'état initial  $s_0$ , puisqu'il est toujours le même pour un match de football : ceci correspond au coup d'envoi. Ensuite, on met à jour la politique selon la formule ci-dessous.

$$Q(s_t, a_t) \leftarrow (1 - \alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{valeur actuelle}} + \alpha(k) \cdot \underbrace{\left( r_t + \gamma \cdot \max_a(Q_{t+1}, a) \right)}_{\text{valeur apprise}}$$

Ici,  $\alpha$  représente la vitesse d'apprentissage et  $\gamma$  le facteur d'actualisation. D'un côté, la vitesse d'apprentissage détermine l'équilibre entre exploration et exploitation. En effet, une valeur de 0 implique l'utilisation exclusive des choix actuels, alors qu'une valeur 1 ne ferait que considérer le dernier résultat. Comme notre agent est censé être plus intelligent au fur et à mesure des épisodes, il doit apprendre de moins en moins des itérations restantes. C'est pourquoi on a choisi une vitesse d'apprentissage polynômiale, i.e.  $\alpha(k) = \frac{1}{k^\omega}$ , avec  $\omega \in ]0, 5; 1[$  fixé.

D'un autre côté, le facteur d'actualisation détermine l'importance des récompenses futures. Une valeur de 0 ne prend en considération que les récompenses courantes, tandis qu'une valeur de 1 met en valeur les récompenses plus lointaines. On avait précisé qu'un épisode correspondait à un match ; par conséquent, la réussite d'une succession d'actions dépend du résultat final du match. De ce fait, on a intérêt à retarder au maximum les récompenses. Ceci se traduit par le choix d'un facteur  $\gamma$  proche de 1.

Si le nouvel état  $s_{t+1}$  est terminal, on passe à l'épisode suivant. Sinon on choisit une action pour cet état et refait la mise à jour de la politique.

À la fin de l'algorithme, on se retrouve avec une politique qui nous rend l'action la plus adéquate pour tout état du jeu. Étant donné que les caractéristiques utilisées pour définir un état sont continues, on a défini des intervalles d'équivalence. Cela n'empêche que le nombre total d'états est énorme et fait donc exploser la mémoire. Bien qu'implémenté, on ne s'est pas servi de cet algorithme pour les versions complètement fonctionnelles de nos équipes de football.

## Troisième partie

# Stratégies

Dans cette dernière partie, on vous détaille le comportement de nos différents joueurs. Compte tenu de la taille du terrain de jeu, le nombre maximum de joueurs par équipe est limité à quatre. Ainsi, il y a trois catégories de matches selon le nombre de joueurs par équipe : un, deux ou quatre.

## 7 Un joueur

Pour l'équipe composée uniquement d'un joueur, on a développé un seul joueur que l'on appelle *Fonceur*.

### Fonceur

Il s'agit du point de départ du projet. Le but des premières semaines étant de se familiariser avec le code source fourni, on a créé une stratégie très simple et naïve. Ainsi, il faisait tout au maximum : il devait se rapprocher de la balle à toute vitesse lorsqu'il n'en avait pas la possession et il frappait avec toute sa puissance dans le cas contraire. Compte tenu que ce comportement ressemblait beaucoup aux autres fonceurs de la classe à une paramétrisation près et que les matches finissaient en nul, on a ajouté les fonctionnalités suivantes : l'avancée balle au pied et le dribble. Pour ce faire, on a utilisé la recherche en grille des paramètres. En effet, les valeurs optimales pour la puissance du contrôle de la balle et le dribble étaient inconnues. Comme on ne connaissait même pas quel était le bon sous-intervalle des valeurs, on a commencé par discrétiser tout l'intervalle de valeurs possibles  $[0; 6]$  avec un pas de 0,5. Puis on a répété le même procédé pour le sous-intervalle avec les meilleurs résultats avec des pas de discrétisation de plus en plus petits.

Par ailleurs, on savait que la frappe était plus réaliste avec une variation sur la précision selon les vitesses du joueur et de la balle, l'angle entre ces deux vecteurs et la puissance de la frappe, en plus d'un paramètre aléatoire. C'est pourquoi on ne pouvait plus maintenir une puissance de frappe constante. On a donc décidé de la faire varier seulement dans le cas d'une frappe vers le but adverse. En effet, on a pris en considération la distance du joueur à la cage adverse et l'angle du vecteur allant de la position du joueur vers le centre de la cage adverse par rapport à l'horizontale. Ceci a donné lieu à deux paramètres optimisés aussi grâce à la recherche en grille. On a utilisé d'ailleurs le même procédé que pour les paramètres précédents.

Cela a rapporté de bons résultats contre les équipes restant naïves, mais n'a pas suffi pour battre les équipes plus développées. En effet, il est très difficile de devancer un adversaire qui fait une pression de près sur le ballon de sorte que les tentatives de dribbles du fonceur échouent. De plus, ses essais de dribble dans sa propre surface de réparation étaient souvent trop faibles pour échapper au danger que signifie l'adversaire venant le presser. Il dégage à présent la balle le plus loin possible pour pallier cette difficulté.

On a ensuite trouvé les valeurs optimales pour tous les paramètres en même temps grâce à notre algorithme génétique, plus précisément au travers de la classe `STTeam`. Il n'empêche que cette stratégie est restée assez basique à l'égard de celles de nos camarades de classe vu les résultats des tournois hebdomadaires. Autrement dit, le dribble est très peu réussi et les contre-attaques arrivent lorsque le joueur est dans sa course vers l'avant, ce qui le surprend et fait que l'adversaire se trouve facilement seul face à notre cage.

Finalement, on a voulu tester la performance de notre algorithme d'apprentissage par renforcement. Pour ce faire, on a développé plusieurs stratégies simples sur la base du comportement

présent à ce moment-là. On l'a déroulé pour le cas particulier de l'équipe à un seul joueur. On a bien vérifié que l'algorithme nécessite de beaucoup d'espace mémoire pour offrir des bons résultats. En effet, on compte un fichier d'un peu plus de 400MB qui montre des indices d'un comportement plus intelligent. Cette optimisation n'ainsi pas été prise en compte pour la version fonctionnelle de l'équipe à un joueur.

## Cas particulier : challenge

Au début du semestre, les enseignants nous ont présenté un challenge, à savoir marquer le plus de buts en 2000 pas de temps. On a réutilisé notre fonceur pour ce cas particulier. À ce moment-là, le fonceur ne comptait que la frappe naïve.

Puisqu'il est très peu probable que la balle rentre dans la cage adverse d'un seul coup, le fonceur fait son travail en deux phases. Tout d'abord, il frappe doucement la balle de sorte qu'elle reste dans une position axiale. Puis il fait une frappe plus forte pour mettre la balle au fond. On a aussi utilisé la recherche en grille pour optimiser cette stratégie.

## 8 Deux joueurs

Cette catégorie a été celle sur laquelle on a consacré la plupart du projet. On compte trois joueurs fonctionnels que l'on appelle *Attaquant*, *Gardien* et *Défenseur*.

### Attaquant

Il a une vocation plutôt offensive. Son comportement initial était celui d'un attaquant classique. Effectivement, il l'a hérité du fonceur, étant jusqu'à alors optimisé seulement par la recherche en grille.

Sa conduite sans la balle dépendait de la zone dans laquelle se trouvait le porteur de la balle : il le pressait si l'adversaire était éloigné de sa surface et se décalait vers les côtés pour faciliter les contre-attaques dans le cas contraire. Quant à son style de jeu avec la balle au pied, il avançait de petites distances sur le terrain et frappait la balle lorsqu'il se trouvait dans la surface de réparation adverse. Tout comme pour le fonceur, il perdait la balle trop facilement. C'est pourquoi on a ajouté le dribble aléatoire, ce qui l'a rendu imprévisible face aux tentatives adverses de lui arracher le ballon. C'est avec cette modification que l'on a utilisé pour la toute première fois notre implémentation d'algorithme génétique pour optimiser la stratégie au complet. On a donc développé les classes `dictParams`, correspondant au dictionnaire des paramètres, et `GeneTeam`, une classe générique permettant de lancer l'algorithme. Ce dictionnaire gérait aussi bien les valeurs des paramètres que leurs intervalles optimaux. Pour les paramètres déjà optimisés par la recherche en grille, on a juste repris leurs intervalles optimaux. Quant aux autres, on a d'abord encadré les valeurs de manière analytique et cohérente avec le rendu attendu. Puis on a utilisé de nouveau la méthode essai et erreur pour trouver des intervalles plus petits et appropriés. Pour le cas particulier de l'équipe à deux joueurs dont l'attaquant fait partie, on a développé la classe `GKStrikerTeam` héritant de `GeneTeam`.

Lorsque le contrôle de la balle avait lieu en sa zone défensive et qu'un adversaire était si proche de lui qu'il pouvait aussi la frapper, la puissance de son dribble était dépassée par celle de la frappe. On a ajouté le dégagement vers un point quelconque tel que le tir soit le moins risqué possible pour éviter ce genre de situations.

Puis on s'est rendu compte que dans la plupart des matches il se trouvait en infériorité numérique face à l'équipe adverse. Cette contrainte a motivé l'idée de faire monter le coéquipier dans le terrain et, par conséquent, le développement de l'action de passe de la balle. Ce

comportement s'est révélé très efficace en situation d'attaque vu que les autres équipes utilisent seulement un attaquant, mais les contre-attaques s'avéraient très dangereuses. Celles-ci arrivaient comme résultat des dribbles échoués. Par ailleurs, on a remarqué que les adversaires restaient fixés dans l'axe tout en laissant dans les côtés des voies libres. Tout ceci nous a amené à mettre en place la toute dernière modification : le dribble latéral. Dans le milieu de terrain, le joueur essaie systématiquement de dribbler le joueur lui bloquant la trajectoire vers les côtés. Le comportement final est encore plus dangereux en attaque et les possibilités de contre-attaques sont à présent très mitigées.

## Gardien

La prise de décisions initiale du gardien avait pour vocation de défendre la cage à tous les coups. En effet, il se positionnait dans le point central sur la ligne de sa cage et n'en sortait que pour dégager la balle lorsqu'un adversaire se rapprochait avec elle. Ce dégagement consistait en fait à faire une pseudopasse vers son coéquipier, c'est-à-dire qu'il l'adressait vers l'un des deux points axialement symétriques fixés à l'avance selon si le coéquipier se trouvait en la partie haute ou basse du terrain. Le point faible de cette conduite était que certains attaquants adverses tiraient vers notre cage depuis des longues distances et on essayait ainsi d'attraper la balle avec des trajectoires circulaires. On a donc décidé d'améliorer la trajectoire d'interception de la balle avec une certaine prévision de sorte à y arriver en ligne de droite et plus rapidement. De plus, sa position de base passerait à une certaine distance de sa cage alignée avec une potentielle trajectoire de la balle vers la cage. Ces modifications ont été ajoutées au dictionnaire de paramètres pour les optimiser à l'aide de la classe `GKStrikerTeam`.

On n'est jamais parvenus à perfectionner l'interception de la balle et on compte ainsi jusqu'à présent un taux non négligeable d'échec. De plus, comme dit précédemment, une attaque à un n'offrait pas d'avantage à l'égard des équipes de nos camarades. Ceci a impulsé le développement de la déjà mentionnée montée du gardien. Lorsque son coéquipier possède la balle, il monte dans le terrain pour proposer des possibilités de passe. Son comportement avec la balle a fait aussi objet de modifications : c'est exactement le même que celui de l'attaquant. On peut donc résumer le comportement de notre équipe à deux joueurs en l'expression footballistique « la meilleure défense, c'est l'attaque. »

## Optimisations non réussies

La méthode d'optimisation par arbres de décision a aussi été testée pour l'équipe à deux joueurs. On a voulu comparer ses résultats avec ceux obtenus grâce à l'algorithme génétique. On a ainsi défini plusieurs caractéristiques généralisables permettant de représenter au mieux l'espace d'états du jeu. On a notamment utilisé les distances entre les joueurs et la balle, ainsi que la distance entre deux coéquipiers. Comme mentionné précédemment, on n'a pas trouvé de moyens de faire les instantanés de manière automatisée. Ceci nous a donc demandé d'avoir une très bonne coordination pour préciser les stratégies préconisées pour les différentes phases du jeu, étant donné que l'on avait défini plus de quatre stratégies de base pour cette méthode. Malgré les efforts mis en place, les captures n'ont pas été suffisamment précises, ce qui s'est traduit par un arbre de décision qui ne correspondait pas tout à fait à nos exigences. On a décidé en conséquence de ne l'utiliser pour la version fonctionnelle d'aucune de nos équipes.

De la même manière que pour le fonceur, on a aussi testé le fonctionnement de notre implémentation de l'algorithme `Q-learning` pour l'équipe à deux joueurs. Malheureusement pour nous, ce cas précis demandait beaucoup plus d'espace mémoire que pour le fonceur. En effet, pour le même espace mémoire du fichier, le comportement restait trop naïf : ils se déplaçaient à peine et ne faisaient pas de frappe.

## Défenseur

Il est à noter que ce joueur ne fait pas partie de l'équipe préconisée. Il correspond à la dernière stratégie implémentée dans cette catégorie, à savoir celle d'un défenseur, qui n'a par contre qu'une vocation défensive. En effet, s'il a la balle, il s'en défait le plus rapidement possible soit à l'aide d'une passe vers son coéquipier si démarqué, soit via le dégagement le moins risqué possible. Quant à son effort défensif, il demeure à une certaine distance radiale de sa cage et s'apprête à intercepter la balle s'il en est suffisamment proche.

Cette stratégie n'a fait l'objet d'aucune optimisation. Elle a été conçue pour faire des tests et a ainsi repris les valeurs de paramètres déjà définies pour ses pairs de la catégorie.

## 9 Quatre joueurs

Il s'agit de la catégorie que l'on a trouvée la plus intéressante en raison de la difficulté de la gestion des espaces et la répartition des fonctions. Tout comme pour la catégorie précédente, on compte trois joueurs fonctionnels que l'on repère d'ailleurs par les mêmes noms.

Le développement des stratégies suivantes a commencé lorsque leurs pairs de la catégorie précédente étaient proches de leurs versions finales. En effet, il n'y avait que le dribble latéral qui n'était pas encore mis en place lors du début de leur conception. Cette modification dans le dribble a été ajoutée à ces stratégies au même temps que pour leurs pairs.

### Attaquant

Il y a deux différences en son comportement par rapport à son pair de la catégorie ci-dessus et celles-ci concernent toutes les deux la situation où il se trouve dans sa surface de réparation. La première fait référence à la récupération de la balle : il ne se précipite plus et fait de nouveau le contrôle, le dribble ou la passe nécessaire de la même manière que dans le milieu de terrain. La deuxième correspond à l'effort de marquer les adversaires sans marquage. En effet, les avancées adverses comportent plus de participants et il se révèle crucial de marquer les adversaires susceptibles de recevoir une passe pour intercepter la balle et lancer des contre-attaques rapides et efficaces.

### Gardien

Ce joueur combine le comportement de son pair de la catégorie ci-dessus et de l'attaquant de cette catégorie à des valeurs des paramètres près. En effet, il suit les mêmes indications que l'attaquant lorsqu'il contrôle la balle et réalise les mêmes mouvements sans ballon que le gardien de l'équipe à deux joueurs.

### Défenseur

Ce défenseur prend les mêmes décisions que celui de l'équipe à deux joueurs, à l'exception de sa prise de risque lors d'une interception. Effectivement, si un coéquipier se trouve dans une meilleure position pour intercepter la balle, i.e. il en est plus proche que le défenseur, celui-ci continue de positionner radialement à une certaine distance de sa cage. Cette modification est le produit de la constatation que l'effort défensif des autres joueurs de cette catégorie n'égale pas celui du défenseur et que l'abandon de sa position défensive entraîne des terribles risques au niveau du score.

## Optimisation finale

Vue la longue attente nécessaire pour que l'algorithme génétique offre des résultats précis dans le cas de l'équipe à deux joueurs, on avait prévu dans un premier temps de n'utiliser que les valeurs de paramètres définies pour ladite équipe. Puis on a changé d'avis et on a développé la classe `FullTeam` héritant de `GeneTeam` pour appliquer l'algorithme génétique à cette équipe. Les résultats se sont révélés plus efficaces, malgré les peu nombreuses itérations faites.

# Conclusion

TODO