2I013 Projet
Groupe 1 : Football et stratégie

Ariana Carnielli et Parth Shah

Table des matières

1 Introduction

Ce projet s'intéresse à la programmation des stratégies pour l'automatisation des joueurs dans le jeu de « football » SoccerSimulator (disponible sur `https://github.com/baskiotisn/SoccerSimulator-2017`). Il s'agit d'un jeu codé en Python où les joueurs peuvent s'affronter dans plusieurs types de matchs : en 1 contre 1, en 2 contre 2 ou encore en 4 contre 4. Les joueurs et la balle peuvent se déplacer partout sur un terrain de jeu dont la taille est fixée par deux constantes et est constitué de deux cages placés aux bords verticaux. Si la balle touche les bords du terrain, elle rebondit, par contre les joueurs ne rebondissent pas et perdent toute leurs vitesses. Les joueurs ne sont pas considérés comme des obstacles à la balle, celle-ci peut les croiser sans rebondir. Chaque match est joué en temps discret, donné aussi par une constante. À chaque pas de temps le jeu actualise la position de chaque objet mobile sur le terrain et vérifie si un but a été marqué. Dans ce cas, les positions des joueurs et de la balle sont réinitialisées. Le jeu se joue de manière autonome, chaque joueur choisissant une action à chaque pas de temps en accord à la stratégie qui lui a été passée.

L'objectif est d'implémenter des stratégies permettant à nos joueurs de gagner la plus grande quantité de matchs contre les stratégies adverses codés par d'autres binômes. Dans ce rapport, nous présentons la démarche mise au point lors de la conception des stratégies tout en abordant les problèmes rencontrés. Dans un second temps nous exposons des stratégies que nous avons écrites et l'implémentation de l'algorithme génétique pour leur optimisation ainsi que l'implémentation des arbres de décision. Une conclusion récapitule les principaux points de ce rapport et les compétences acquises.

2 Démarche utilisée et problèmes rencontrés

Pour construire des stratégies, plusieurs classes étaient à notre disposition, notamment `SoccerAction`, `Vector2D` et `SoccerState`. `Vector2D` contient deux coordonnées `x` et `y` ainsi que de méthodes pour, par exemple, calculer la norme d'un vecteur donné et son angle par rapport aux axes des coordonnés. Quant à `SoccerAction`, celui-ci est composé de deux `Vector2D` : un qui donne la direction du mouvement du joueur et un autre qui donne la direction du shoot. `SoccerState` contient toutes les informations nécessaires pour connaitre l'état du jeu à un moment donné y compris la position et vitesse de la balle et de tous les joueurs.

Il existe déjà une classe `Strategy` donnant la base pour la construction d'une stratégie : Toute classe de stratégie que nous avons codé hérite de celle-ci. Une stratégie doit implémenter une méthode `compute_strategy` qui prend en argument l'état du jeu à travers un `SoccerState` et l'identification d'un joueur et retourne un `SoccerAction` à chaque pas de temps. Au départ nos stratégies étaient des classes indépendantes

regroupés dans un fichier appelé `strategy.py`.

Comme nous n'avions aucune expérience sur la conception de stratégies, nous avons tout d'abord essayé de coder une petite stratégie telle qu'un fonceur, en reprenant l'exemple vu en cours, qui court vers la balle et tire au but. Notre intention était de l'utiliser principalement en 1 contre 1. Nous nous sommes rendu compte après quelques essais qu'il serait intéressant de ne pas encombrer la stratégie avec des tests pour voir si le joueur a la balle, etc., car à chaque fois que nous voulions changer un paramètre ou les choix faits par la stratégie, il fallait changer tout le code. Nous avons ainsi crée une classe appelée `ToolBox` avec des méthodes qui retournaient les résultats des tests fréquents qu'on utilisait, dans un fichier nommé `toolbox.py`.

Au fur et à mesure du projet nous étions amenés à développer de nombreuses stratégies et nous avons remarqué que plusieurs d'entre elles partageaient des morceaux de code suffisamment complexes pour ne pas être dans `ToolBox`. Nous avons alors rajouté de couches entre nos stratégies et `ToolBox` en créant d'autres classes. Cela a permis également de rendre le code plus lisible et modulaire. Nous avons mis au point deux fichiers en plus de `toolbox.py` : `action.py` et `comportement.py`. Ainsi, en cas de problème à l'exécution, nous pouvions le trouver plus facilement et modifier seulement la partie de la fonction ou méthode concernée, plutôt que l'intégralité du code.

Nous avons décidé de laisser `ToolBox` avec toutes les méthodes retournant des booléens, des `Vector2D` et des listes de `Vector2D`. Les fichiers `action.py` et `comportement.py` ont chacun une classe homonyme : `Action` contient des méthodes qui retournent des `SoccerAction` alors que `Comportement` utilise les méthodes dans `ToolBox` et `Action` pour bien choisir le `SoccerAction` qu'une stratégie doit retourner à chaque pas de temps. Nous avons choisi de séparer les comportements des stratégies afin d'être capable d'appeler plusieurs comportements dans une stratégie quelconque. À la fin, nous avons trouvé cela un peu redondant car nos stratégies appellent uniquement un comportement même si le fait de séparer stratégie et comportement a été utile dans une stratégie décrite plus en détail dans la section suivant.

Tous nos fichiers ont été mis dans un même module contenant un fichier principal `__init__.py`, où la fonction de création de joueurs en appelant nos stratégies a été placée. Pour ce faire, nous avons eu besoin de la commande `import` de Python pour l'utilisation d'une classe ou bien d'une fonction écrite dans un autre fichier. Au début, lorsque nous lancions notre fichier exécutable, quelques erreurs apparaissaient à cause des `import` non aboutis. En effet, la syntaxe de `import` en Python change selon la version du langage et si l'importation est absolue ou relative. Nous avons donc utilisé exclusivement Python 3, qui utilise des import relatifs explicites, pour régler les soucis.

Nous nous sommes mis à développer également d'autres stratégies plus intéressantes qu'un fonceur, notamment des défenseurs et des dribleurs. Le développement continu de ces stratégies était indispensable pour contrer les améliorations apportées

par les autres binômes à leurs stratégies, afin de permettre ainsi une bonne cohésion entre nos joueurs et un équilibre entre la quantité de buts marqués et encaissés pour enfin obtenir un bon rang dans les classements. La section suivante présentera plus en détails nos principales stratégies, leur fonctionnement et les raisonnements derrière nos choix.

Après quelques séances nous nous sommes rendu compte que la meilleure façon de battre tous nos adversaires était d'étudier plus en détail chacune de leurs stratégies et d'en exploiter les faiblesses afin d'avoir l'avantage. C'est pourquoi nous avons commencé à télécharger les modules des autres binômes depuis leurs dossiers en ligne sur *Github*. Pour chaque groupe, nous essayions d'améliorer et de réajuster notre code en fonction des résultats obtenus en simulant des matchs pour enfin arriver à une moyenne de victoires convenable.

En revanche, il n'était pas toujours facile de trouver une riposte contre les stratégies des autres groupes. Certaines stratégies étaient si bien codées qu'il fallait du temps pour d'abord mettre au point de nouvelles stratégies pour contrer une attaque ou intercepter la balle par exemple, d'autant plus que ces tests devaient être réalisés contre l'ensemble des autres binômes. En outre, certaines étaient mises en ligne quelques instants avant les résultats finaux, nous empêchant ainsi de pouvoir travailler sur les stratégies récentes. En utilisant la recherche exhaustive et l'algorithme génétique nous avons enfin automatisé une partie du processus d'amélioration de nos stratégies, dont la démarche sera présenté dans la Section 4.

3 Présentation des stratégies

Cette section présente les principales « familles » de stratégies que nous avons codées à la main pour ce projet : Fonceur, Dribbleur, Défenseur et Attaquant. Cette division suit l'évolution de notre code dans le sens où le Fonceur et ses améliorations ont été les premières stratégies codées, suivies par le Dribleur et le Défenseur qui nous ont permis de passer aux matchs 2 contre 2 et enfin l'Attaquant qui a été crée pour les matchs 4 contre 4.

3.1 Fonceur

Comme nous avons dit précédemment, la première famille de stratégies codée a été le Fonceur qui peut être représenté par l'arbre de décision de la Figure 1.

Dans un premier temps cette stratégie a été code sans aucune paramètre mais nous avons ensuite identifié quelques uns ayant une influence sur le comportement et l'efficacité de cette stratégie : l'accélération du joueur quand il court, l'accélération de son tir et la quantité de pas de temps en avance qu'il essaie de prévoir la position du ballon.

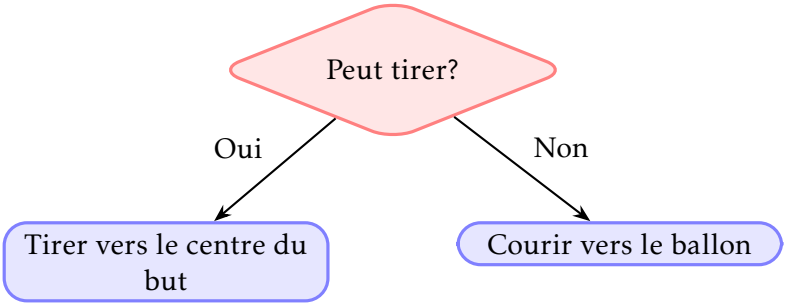

Figure 1 – Arbre de décision du Fonceur

Lorsque ce fonceur joue contre d'autres stratégies de fonceur identiques ou très similaires, il arrive assez souvent qu'on tombe sur une situation de « blocage », où les deux joueurs courent ensemble vers le ballon, arrivant et tirant au même moment en directions opposées, auquel cas le ballon part dans un direction essentiellement aléatoire et aucun des deux joueurs n'arrive à marquer un but. Ce blocage peut être partiellement résolu en changeant le nombre de pas de temps de prévision de la position du ballon, mais cette solution ne donne pas en général des résultats satisfactoires, surtout car une quantité de pas de temps peut être très bonne pour certaines variantes du fonceur, mais pas pour d'autres. Cela nous a motivés à créer une stratégie plus élaborée, le Dribbleur, décrit dans la prochaine partie.

3.2 Dribbleur

La famille de stratégies qui forme la pièce maitresse de notre jeu est le Dribbleur dont l'arbre de décision représentatif est donné sur la Figure 2.

En comparant les Figures 1 et 2 on remarque que le Dribbleur a le même comportement que le Fonceur s'il n'a pas le ballon, s'il est très proche de la cage adverse ou s'il n'existe pas d'adversaire devant lui. Cependant il a un comportement plus élaboré dans d'autres situations. Lorsqu'il arrive au ballon au même temps qu'un adversaire, il essaie de contrer le tir adverse en faisant un tir d'accélération maximale vers le champ adverse, sans viser la cage. Dans le cas où l'équipe adverse a un gardien, c'est-à-dire un joueur au voisinage immédiat de la cage, le Dribbleur frappe en visant le coin de la cage le plus proche de lui. Enfin, s'il n'est pas proche de la cage et il existe un joueur adverse devant lui qui n'est pas trop loin, il dribble selon le schéma de la Figure 3.

Pour le dribble, le joueur détermine les vecteurs entre lui-même et le centre du but adverse et entre lui-même et le joueur adverse le plus proche devant lui, calculant ensuite l'angle θ entre ces vecteurs. Si θ est trop grand, il considère que l'autre joueur ne bloque pas son chemin au but et ne fait pas de dribble. Sinon, il fait un dribble : si l'adversaire est situé en bas par rapport au joueur, il dribble vers le haut, et au contraire si l'adversaire est situé en haut par rapport au joueur, il dribble vers le bas.

Un souci observé avec cette implémentation de dribble est que, dans certaines

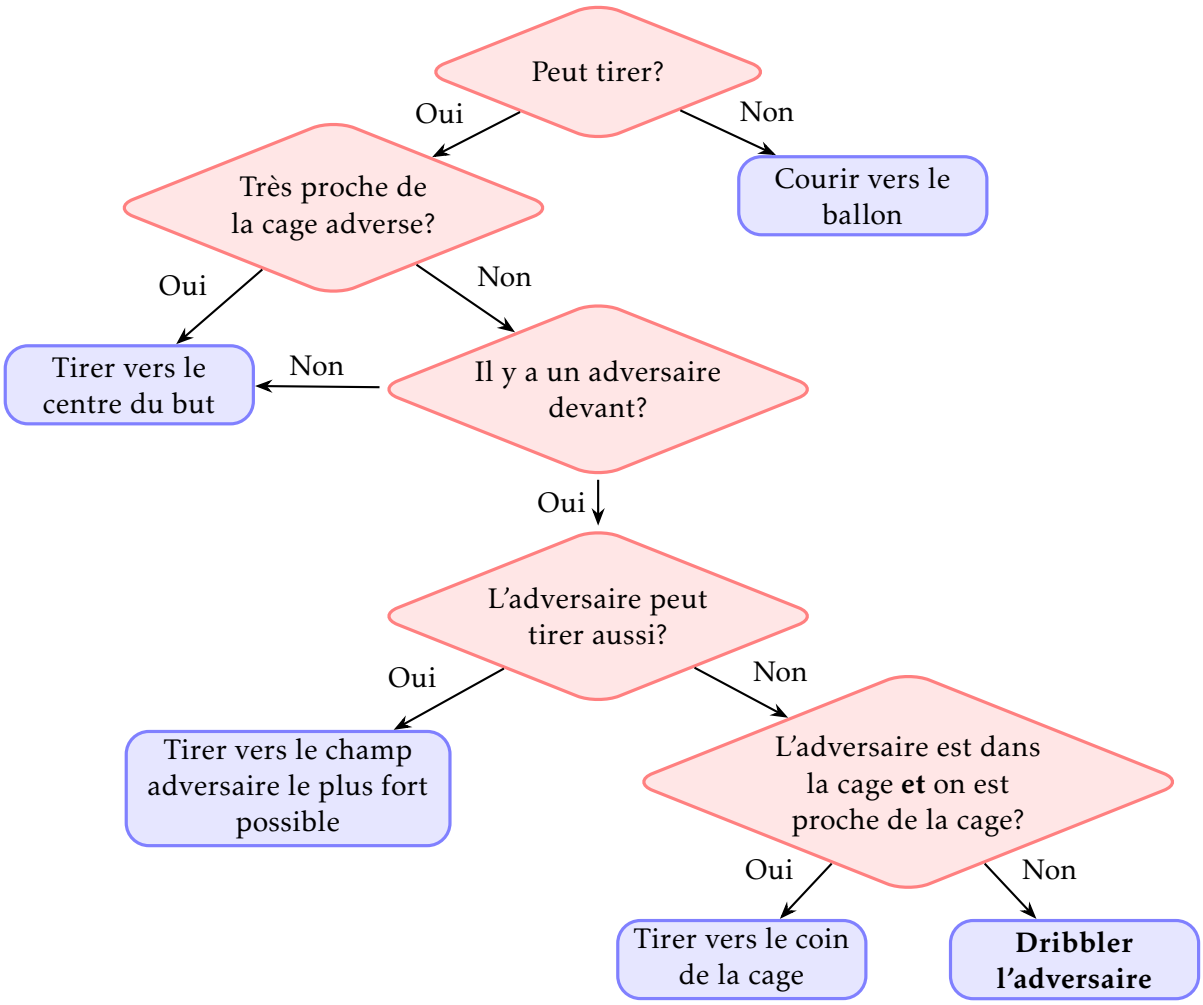

FIGURE 2 – Arbre de décision du Dribbleur

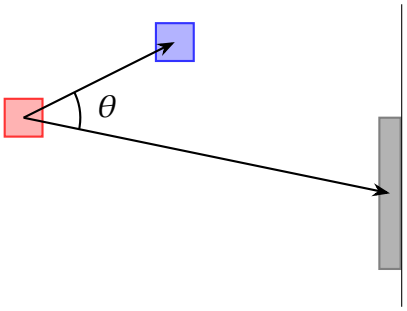

FIGURE 3 – Schéma d'un dribble : Dribbleur en rouge, adversaire en bleu, cage en gris

situations où le joueur, son adversaire et le centre du but sont presque alignés, le joueur peut, dans un premier moment, trouver que l'adversaire est en bas et dribbler vers le haut, pour tout de suite après trouver que l'adversaire est en haut et dribbler vers le bas. Cette hésitation lui fait souvent perdre le contrôle du ballon. Pour éviter ce problème, nous avons implémenté une mémoire du dernier dribble : si l'angle θ est trop petit, il dribble dans la même direction que son dernier dribble, en changeant de di-

rection uniquement si θ est plus grande qu'une valeur fixée. Pour créer cette mémoire, nous avons utilisé le fait que `Comportement` et `Strategy` sont deux classes séparées : la mémoire est implémentée par une variable dans la `Strategy` du Dribbleur qui est mise à jour à chaque pas de temps dans l'appel de la fonction `compute_strategy` (un nouveau `Comportement` est crée à chaque appel à `compute_strategy`, ce pourquoi la mémoire doit forcément être dans `Strategy`).

Pour être capable de faire une optimisation de ce Dribbleur, nous avons introduit une dizaine de paramètres, entre eux l'accélération pour faire un tir au but mais aussi une autre pour dribbler, ainsi que les angles définissant le comportement de dribble. Il est nécessaire que le joueur dribble avec une accélération optimale pour qu'il soit assez rapide afin de se déplacer sur l'ensemble du terrain tout en contournant ses adversaires et en gardant le contrôle du ballon. Le Dribbleur est utile car il favorise la liaison entre défense et attaque et il est autonome c'est-à-dire qu'il ne dépend pas de ses coéquipiers mais uniquement de ses adversaires, c'est pourquoi nous pouvons l'utiliser dans tous les types de matchs : en 1 contre 1, en 2 contre 2 voire en 4 contre 4. Pour les matchs en 1 contre 1, nous avons légèrement modifié le Dribbleur pour enrichir son arbre de décision afin d'être plus performant.

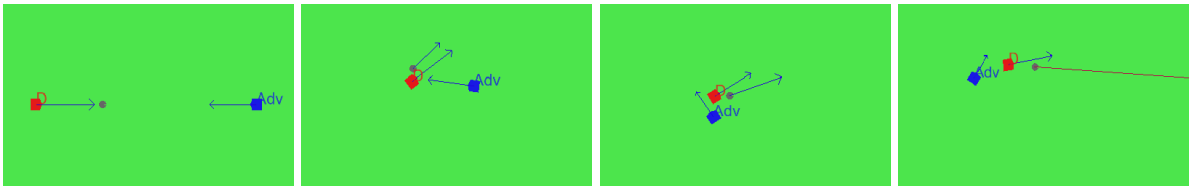

Figure 4 – Example de dribble dans un match, Dribbleur en rouge

3.3 Défenseur

La famille de stratégies de Défenseur a été fondamentale dans les matchs à plusieurs joueurs. Son arbre de décision représentatif est donné sur la Figure 5.

Notre but était de nous assurer que lorsqu'un attaquant de l'équipe adverse s'attaque à notre Défenseur, ce dernier puisse le confronter et adopter un certain comportement défensif empêchant ainsi aux ennemis de marquer un but. Pour ce faire, nous avons implémenté quelques versions de Défenseur, qui diffèrent essentiellement par leur position de défense et par des simplifications de la branche « Oui » du test « Peut tirer? ». Dans un premier moment, nous avons codé une version simplifiée de Défenseur que nous avons appelé Gardien puisque sa position de défense par défaut était le centre de notre but. Dans cette version, lorsqu'il pouvait tirer, il faisait un tir aveugle vers l'avant.

Cette stratégie a bien marché pendant les premières semaines après son implémentation, mais les autres binômes ont réussi à trouver des stratégies d'attaque exploitant

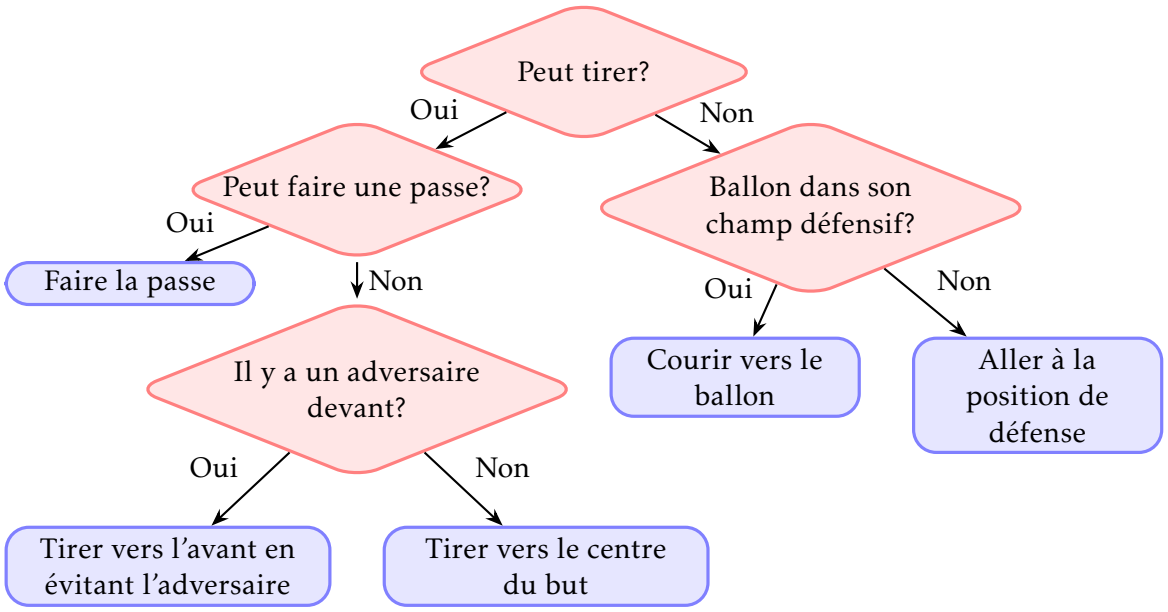

FIGURE 5 – Arbre de décision du Défenseur

des failles de notre gardien, notamment le fait qu'il ne pouvait pas attraper le ballon si sa vitesse était trop élevée. Pour améliorer notre Gardien, nous avons codé une nouvelle version de Défenseur dont la position de défense par défaut n'était pas la cage mais avait un degré de liberté supplémentaire : nous avons choisi une position horizontale `frac_p`, représentée comme une fraction de la longueur du champ défensif et passé en argument au Défenseur, de telle sorte que la position de défense avait la position horizontale voulue et était toujours entre le ballon et le centre du but, comme sur la Figure 6. Notre intention avec cette modification était de faciliter l'interception du ballon. Son but consistait principalement à empêcher le jeu d'attaque de l'équipe adverse.

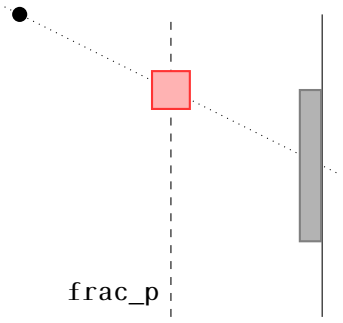

FIGURE 6 – Position de défense par défaut: position du Dribbleur en rouge, ballon en noir, cage en gris

Cette modification de la position de défense a amélioré notre Défenseur mais n'était cependant pas satisfaisante, et nous l'avons modifiée dans la version finale, en augmentant encore un degré de liberté. Pour cela, nous avons choisi un paramètre $\alpha \in [0,1]$ et défini la position défensive par défaut comme la position sur le segment reliant le bal-

lon au centre de la cage, à une proportion α du centre de la cage et $1 - \alpha$ du ballon. Ainsi, il n'est pas à une position horizontale fixée, mais peut avancer — même sur le champ d'attaque si le ballon est proche de la cage adverse — ce qui peut être utile pour récupérer le ballon plus tôt. Remarquons que, si le ballon est trop proche de la cage de défense, il court vers le ballon comme indique sur la Figure 5.

C'est également à ce moment que nous avons codé le mécanisme de passe indiqué sur la Figure 5. Nous avions auparavant codé une stratégie de passe entre les joueurs, qui prenait en paramètre deux accélérations différentes : une pour tirer et une autre pour faire la passe. L'accélération de passe différente était importante pour que la passe ne soit ni trop longue ni trop courte pour favoriser l'avancement progressif de la balle entre les joueurs jusqu'aux attaquants.

Pour notre stratégie de passe, lorsque le joueur peut tirer, il calcule la position du joueur qui est le plus proche de lui — car il est inutile de faire une passe à un joueur qui est loin — et détermine si ce joueur n'est pas trop proche ni trop loin et si un joueur adversaire n'est pas proche de lui. Une fois la position calculée et les conditions de passe remplies, il fait la passe à ce joueur. Comme nous n'avions pas utilisé cette stratégie de passe de façon isolée, nous l'avons incorporée à notre Défenseur.

Enfin, lorsque le Défenseur a le ballon et ne peut pas faire une passe, plutôt que de tirer aveuglement vers l'avant, il cherche à déterminer si un joueur adverse pourrait intercepter le ballon. Pour ce faire, il détermine les vecteurs entre lui-même et le centre du but adverse et entre lui-même et le joueur adverse le plus proche devant lui, calculant ensuite l'angle entre ces vecteurs. Comme dans le cas du dribble, il choisira, en fonction de la taille de cet angle et de la position de l'adversaire s'il tirera vers le centre du but ou s'il déviera ce tir vers le haut ou le bas.

3.4 Attaquant

4 Optimisation des stratégies

4.1 Recherche exhaustive

Pour le chalenge consistant à faire le nombre maximal de buts en 10000 pas de temps (sans joueur adverse), nous avons utilisé ce Fonceur sans prévision de la position du ballon et avec accélération de course maximale mais avec optimisation de l'accélération de tir. Cette optimisation a été faite par une recherche exhaustive en discrétisant l'intervalle d'accélérations possibles et cherchant à minimiser le temps nécessaire pour marquer 10 buts. Comme celle-ci a été fait avant la partie du cours sur la recherche

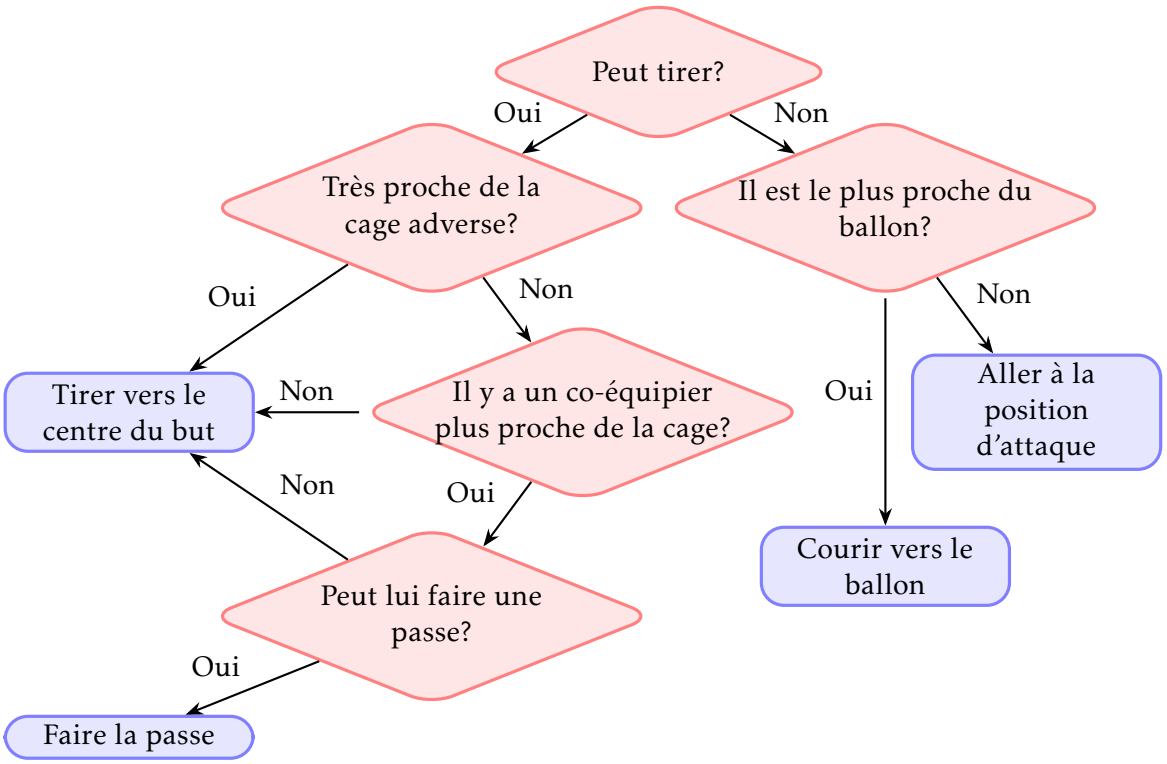

Figure 7 – Arbre de décision de l'Attaquant

exhaustive, nous avons dû coder par nous mêmes tous les fonctions nécessaires. Le résultat de cette optimisation est présenté dans la Figure 8.

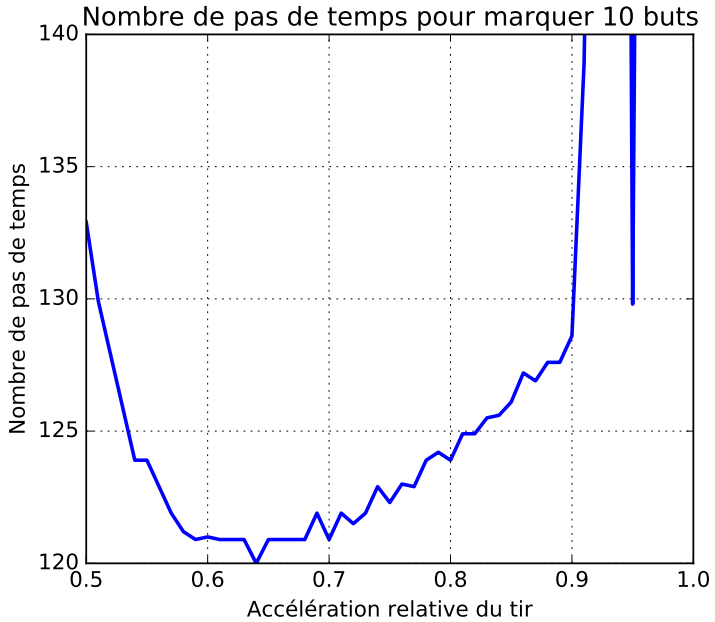

Figure 8 – Recherche exhaustive de l'accélération de tir optimale

L'accélération est donné en fraction de l'accélération maximale d'un tir. Nous pou-

vons voir sur la figure que sa valeur optimale est 0.64. En réalisant de simulations nous avons pu constater que, avec cette accélération, 2 tirs sont nécessaires pour marquer un but. Nous avons donc codé une recherche exhaustive sur 2 paramètres (correspondant chacun à l'accélération d'un tir) mais le résultat était que 2 tirs de la même intensité étaient l'optimum. Par contre, la accélération optimale pour cette chalenge n'est pas bonne pour les matchs.

4.2 Algorithme génétique

4.3 Arbres de décision

5 Conclusion

Ce projet a été l'occasion de s'intéresser au développement de stratégies en Python, et à comprendre certains principes sur le Football. Je pense que c'est un thème qu'il faut étudier, et que très peu de personnes connaissent. La réalisation des expériences nous a permis de comprendre les difficultés du projet en soi, notamment avec la prise en main de Github, à repenser aux stratégies qu'il faut utiliser... Mais au-delà d'un grand apport de connaissances, le projet a été l'opportunité de chercher seules des solutions à nos problèmes. Le travail d'équipe a, je pense aussi nouer des liens entre nous. Nous avons fait d'énormes progrès en informatique, en particulier avec la programmation en Python, à travers les stratégies.