Rapport de projet

SCHOTTEN TOTTEN

Benjamin STACH & Elian BELMONTE | IA41 | 20 décembre 2020

Sommaire

Introduction

Pour réaliser ce projet d'IA41, nous avons choisi le sujet du jeu Schotten Totten. Le but du projet est de développer le jeu choisi et ensuite d'implémenter une intelligence artificielle qui soit capable de remplacer un joueur. Lors du choix du sujet, nous avons décidé de sélectionner un sujet difficile car nous avions envie de découvrir comment implémenter une IA de plusieurs manières différentes et notamment via un algorithme de « minmax » avec élagage « alpha-beta ».

À l'origine, notre objectif était d'implémenter le jeu dans son entièreté en incluant le mode expert et c'est pourquoi le jeu a été codé de sorte que le mode expert puisse être rajouté plus tard. Cependant, par manque de temps, nous n'avions pas pu traiter ce cas-là qui aurait également rajouter une difficulté supplémentaire pour ce qui est de l'I.A. Nous voulions développer un jeu opérationnel et facile à utiliser via une interface graphique travaillée autour d'une architecture logicielle cohérente et structurée, d'y implémenter une I.A. qui permette d'avoir un adversaire jouant de manière « stratégique » et cohérente comme le ferait un vrai joueur. Ainsi, nous espérions pouvoir bien perfectionner nos connaissances du langage Python avec lequel nous débutions quasiment tous les deux ce semestre ainsi qu'apprendre à utiliser un outil d'interface graphique issu d'un set de modules Python.

En première partie de ce rapport, nous présenterons le jeu et la manière dont il a été créé en discutant de nos choix d'implémentation et de leur pertinence. Nous verrons ensuite comment nous avons abordé l'interface graphique. Nous aborderons ensuite la partie la plus importante du rapport qui traitera de l'implémentation de l'I.A. où ferons une analyse de cas du Schotten Totten et les différent problèmes à résoudre. Nous discuterons également des choix de représentation d'état, des algorithmes de résolutions adoptés, ses avantages et inconvénients ainsi que des potentielles pistes d'amélioration. Nous terminerons enfin par un conclusion.

Implémentation du jeu

Afin de reproduire le plus efficacement le jeu du Schotten Totten en version numérique, nous avons choisi de travailler avec le langage de programmation Python qui dispose de nombreuses bibliothèques très pratiques comme par exemple « pygame ».

Grâce à Python, nous nous sommes également dirigés vers de la programmation orienté objet. La programmation orienté objet est plutôt utile lorsqu'il s'agit de représenter quelque chose de concret sous forme algorithmique, c'est pourquoi nous avons pu nous en servir dans le cas d'un jeu de société comme le Schotten Totten dont les principaux éléments sont des cartes. Afin d'implémenter toutes les classes nécessaires, nous avons dans un premier temps fait l'inventaire des différents éléments qui composent le jeu comme les cartes normales, les cartes tactiques, les frontières, la pioche etc. Concernant l'ordre d'implémentation des classes, nous avons réfléchi sur la composition de chaque objet et nous avons opté pour une approche de type « bottom-up » en commençant l'élément à la base du jeu, la carte.

Etant donné que notre objectif de base était d'implémenter le jeu en entier (mode expert compris), il était important de gérer les types de cartes. Nous avons donc implémenté la classe « Card » comme étant une classe abstraite donc le seul attribut est son type (clan, effet sur frontière…). Différentes autres classes de cartes ont hérités de cette classe. La plus importante étant la classe « Clan » représentant les cartes de clan qui composent le mode normale (54 cartes au total). Les attributs de cette classe sont simplement la valeur de la carte (de 1 à 9) et sa couleur.

La classe « Card_Draw », modélisant la pioche, a ensuite été implémenté. Cette classe se devaient d'assurer certaines fonctionnalités comme se remplir contenir des objets de type « Card », pouvoir les mélanger, et simuler une pioche de la part d'un des joueurs. Elle a donc comme attribut une liste d'objet « Card », sa taille et son type car elle peut être composé de carte clan ou bien de cartes tactiques dans le mode expert. Une fonction lui permet de s'initialiser en ajoutant toutes les cartes au paquet tandis qu'une autre change l'ordre des cartes de la liste en modifiant aléatoirement leur index.

La classe « Hand » modélise la main d'un joueur. Elle a pour attributs une liste d'objet « Card » qui ne pourra pas excéder une certaine taille, la taille en question (6 pour le mode normal, 7 pour le mode expert) et un entier qui désignera la carte actuellement sélectionnée par le joueur. Une fonction permet de jouer une carte de la main du joueur en la retournant et en la supprimant de la liste. Une autre lui permet de piocher à partir d'un objet « Card_Draw » passé en argument.

Ensuite, pour modéliser le plateau de jeu, il fallait prendre en compte le fait que celui-ci est composé de 9 frontières sur lesquelles les joueur peuvent poser jusqu'à 3 cartes sur leur coté respectif. Nous avons donc implémenté les classes « Frontier » et « Side ».

« Side » comprend dans ses attributs une liste d'objet « Card », un entier pour sa taille, un entier pour la taille maximale, un entier pour représenter la force relative de la combinaison de ses cartes, et un entier pour la somme de valeurs des cartes. Cette classe se devait de pouvoir ajouter une carte à sa liste lorsque celle-ci n'est pas pleine mais aussi de pouvoir calculer la puissance des cartes qui la compose. Les fonctions « isBrelan », « isSuite » et « isColor », extérieur à la classe, permettent de répondre à cette problématique. Afin de définir les puissances de chaque combinaisons selon les règles du jeu, nous avons créé les constantes « SUITE_COLOR », « BRELAN », « COLOR », « SUITE », respectivement égalent à 4, 3, 2 et 1. Une absence de combinaison équivaut à une puissance de 0.

La classe « Frontier », quant à elle, a été une des classes les plus importantes car elle devait assurer plusieurs des principales fonctionnalités propre aux mécaniques du jeu, à savoir le fait d'être revendiqué et de pouvoir déterminer s'il y a un vainqueur lorsque toutes les cartes ne sont pas encore posées. Cette classe a pour attributs un entier qui détermine le statut de la frontière (-1 : Non revendiqué, 0 : Revendiqué par le premier joueur, 1 : Revendiqué par le deuxième) et deux objets « Side » contenu dans une liste pour pouvoir les sélectionner par rapport à l'index associé à chaque joueur. Afin que les joueurs puissent revendiquer une frontière, nous avons dû implémenter la fonction « claim » pour laquelle nous pouvions ne retrouver dans deux cas de figure. Un joueur peut revendiquer une frontière que lorsqu'il a posé 3 cartes dessus et que la combinaison adverse n'est pas plus forte ou bien qu'il ne puisse pas avoir mieux avec les cartes restantes dans le cas où moins de 3 cartes sont posées du côté adverse. Dans le premier cas, il suffisait de comparer les puissances des deux côtés et les sommes (en cas d'égalité) grâce à la fonction « isStronger ». En revanche, pour le second cas, l'objectif était de prouver qu'il existait au moins une combinaison adverse plus forte. Nous avons donc implémenté la fonction « cantBeStrongerThan » qui procède de la manière

suivante ; on rassemble dans une liste toutes les cartes qui n'ont pas encore été révélé (cartes de la pioche et des mains des deux joueurs), on teste toutes les combinaisons possibles, on renvoie « False » si une meilleur combinaison est trouvée, on renvoie « True » dans le cas contraire.

Afin de modéliser le plateau de jeu, nous avons implémenter la classe « Gameboard » qui regroupe les classes précédentes. Elle a pour attributs une variable de type booléen qui indique si la partie se déroule en mode expert, une liste de frontières composée de 9 objets « Frontier », un objet « Card_Draw », et deux objets « Hand » contenus dans une liste pour qu'ils puissent être sélectionnés par rapport à leur index. Cette classe a l'avantage de regrouper toutes les informations nécessaires au pilotage de la partie. Elle est aussi très pratique pour que les fonctions de l'interface graphique puissent afficher la partie en temps réel.

La classe « Player » a pour attributs un entier que défini le numéro associé au joueur (0 ou 1), une chaîne de caractère pour le pseudo son nom, un objet « Gameboard » et un objet « Hand ». Cela permet à la classe « Player » de faire la relation entre sa main et le plateau pour qu'il puisse jouer une carte ou bien revendiquer une frontière.

Enfin, nous avons implémenter la classe « Game » qui devait s'occuper du déroulement de la partie en gérant les tours de chaque joueur, les actions des joueurs et en faisant jouer une I.A. pour le mode « Player vs I.A. ». Game a en attributs un booléen qui définit la présence ou non de l'I.A., un objet « Gameboard », deux objets « Player » pour chaque joueur, un entier qui fait référence aux nombres de tours déjà passés, et un booléen qui définit si le tour du joueur est fini ou non. Cette classe possède deux fonctions essentielles au projet, à savoir « play » et « playIA » qui effectue les actions du joueur ou de l'I.A. Nous verrons plus en détails le contenu de la fonction « playIA » dans la partie « Implémentation de l'I.A. ».

Implémentation du GUI

Les classes de notre projet dans lesquelles ont été développées l'interface graphique sont les classes MainMenu, Game et Gameboard. La fenêtre principale à l'ouverture du jeu ainsi que la fenêtre qui permet de modifier les noms des joueurs sont gérées par la classe MainMenu. La fenêtre principale de jeu est gérée par la classe Game qui a pour rôle de gérer les différents évènements du jeu comme les actions des joueurs. Cette classe inclut aussi toutes les fonctions nécessaires pour rafraichir la fenêtre et gérer les modifications de l'interface à faire en fonction de l'avancée du jeu. La classe Gameboard a pour rôle de gérer les éléments du jeu comme les différentes cartes, les frontières ou les mains des joueurs. C'est donc cette même classe qui crée les visuels de ces éléments et qui contient les fonctions permettant de les afficher.

Classe MainMenu

La classe « MainMenu » a pour attributs deux booléens et deux chaines de caractères. Les deux booléens servent à indiquer si le mode expert est activé ou non et si l'on souhaite jouer contre l'IA ou non. Les deux chaines de caractères correspondent aux noms des deux joueurs choisis. Ces attributs seront passés en paramètres lors de la création de l'instance de la classe Game.

La fonction displayMainMenu() permet de créer la fenêtre d'accueil du jeu en plaçant les ressources images nécessaires et en les plaçant sur la fenêtre. Cette fonction gère aussi les évènements comme les clics sur les boutons et gère les actions à réaliser en conséquence.

La deuxième fonction displayNameSelect() permet d'afficher le sélecteur de noms de joueurs et au moment de sa fermeture de modifier les deux attributs « player_name » de la classe. Cette fonction intègre donc une gestion de l'entrée clavier qui n'a pas été facile à réaliser car aucun « text input box » n'existe déjà dans la bibliothèque pygame. Ainsi, il nous a fallu créer complètement un « text input box » avec la bibliothèque pygame. Pour réalise cela, nous avons utilisé la lecture des évènements du clavier et de la souris afin de choisir le nom à modifier avec la souris, changer le nom de couleur pour montrer qu'il est sélectionné. De plus, il fallait bien sûr que le texte se modifie quand nous pressions des touches du clavier quand le nom était bien sélectionné. Cette fonction nous a donc finalement pris un certain temps à créer bien que cela pouvait paraître facile à développer au premier abord.

La classe « Game » possède de nombreux attributs dont notamment une instance de la classe Gameboard, deux instances de la classe Player pour les deux joueurs ainsi que des éléments dynamiques (boutons, booléens, message à afficher dans la fenêtre…) indispensables au bon déroulement du jeu que les fonctions de cette classe modifient et doivent se partager tout au long de la partie. Cette classe récupère aussi les attributs de la classe MainMenu qui lui sont transmis en paramètres lors de son instanciation.

La première fonction de cette classe est la fonction displayGame(). Elle permet de créer la fenêtre principale de jeu et de gérer les évènements de clics sur les éléments de jeu (frontières et cartes de joueurs). Pour faire fonctionner une interface graphique avec pygame, il faut que le gestionnaire d'évènements ainsi qu'un ensemble d'instructions servant à modifier la fenêtre si des éléments du jeu ont changé soient présents dans une boucle « while game_running : » avec « game_running » un booléen qui est faux lorsque l'on souhaite fermer la fenêtre. Pour lire les évènements de type « clic » et pour savoir si le clic a bien été exécuté sur une carte ou une frontière, pygame appelle une fonction pygame.event.get() qui renvoie tous les évènements qui ont été passé en liste d'attente. Ensuite, cette liste est traitée dans une boucle qui traite chacun des évènements listés. A l'intérieur de cette boucle, nous pouvons alors traiter les évènements qui nous intéressent comme par exemple ici les évènements de type « MOUSEBUTTONDOWN » qui correspond au clic de la souris :

```
if event.type == pygame.MOUSEBUTTONDOWN:
```

Il est alors possible de traiter les surfaces du jeu et vérifier si le clic a été exécuté sur cette surface ou non via la fonction « pygame.Rect.collidepoint(event.pos)» avec « Rect » la surface à tester et le paramètre « event.pos » la position du clic. Si cette fonction renvoie « True », nous modifions alors l'une de ces deux variables d'instance (dans le cas de la fonction displayGame()) selon la surface qui a été cliquée en lui affectant le numéro de la carte dans la main du joueur ou de la frontière choisie :

- selectedCardNumber
- selectedFrontierNumber

Après avoir géré les évènements de jeu, notre fonction va vérifier si le jeu est remporté par l'un des deux joueurs via la fonction isGameover() de la classe Gameboard et si cette fonction revoie « True », le jeu va afficher le nom du joueur gagnant dans la boite de messages du bas de la fenêtre. Sinon, c'est que le jeu est toujours en cours, alors les fonction de notre classe updateScene() et playTurn() sont appelées.

La fonction updateScene() va permettre le rafraichissement de la fenêtre de jeu par superposition des éléments sur une surface vierge. Elle va s'exécuter à chaque itération de la boucle de jeu principale de la fonction précédente. Elle va d'abord recouvrir la fenêtre de la couleur choisie pour le fond, puis afficher le message contenu dans la variable d'instance si la chaine de caractère n'est pas vide en appelant la fonction displayMessage(). Ensuite, la fonction va afficher tous les éléments visuels nécessaires (boutons, cartes posées, mains des joueurs, frontières…) via les fonctions d'affichage des éléments définies dans la classe Gameboard. Enfin, la fonction « pygame.display.flip() » est appelée pour afficher la nouvelle surface créée sur la fenêtre c'est-à-dire de mettre à jour notre variable d'instance « scene ». La fonction displayMessage() permet créer une boite réservée aux messages de jeu pour avertir les joueurs des actions à effectuer. Elle va donc créer une surface blanche avec le texte de la variable d'instance « message » superposé dessus avant d'ajouter cette surface à la scène principale de jeu.

La fonction playTurn() a pour but d'appeler la fonction play() ou playAI() si c'est au tour de l'AI de jouer et de gérer le passage des tours de jeu. Ainsi, si c'est le tour du joueur 1 et que le joueur a choisi une frontière ainsi qu'une carte à jouer alors la fonction play() est appelé avec en paramètre l'instance de la classe Player assignée au joueur 1. Pour savoir si une carte et une frontière sont sélectionnées et ainsi appeler la fonction play(), la fonction vérifie les variables selectedCardNumber et frontierCardNumber pour voir si leur valeur est différente de -1 (qui correspond à l'état où aucun élément n'est sélectionné). A la fin du tour, la variable d'instance « Turn » est incrémentée de 1 et les valeurs des variables selectedCardNumber et selectedFrontierNumber sont remises à -1. La fonction play() et playAI() permettent au joueur passé en paramètre de jouer son tour et de modifier les données de jeu du joueur. Ils permettent aussi au joueur de revendiquer une ou plusieurs frontières.

La classe « Gameboard » contient tous les éléments nécessaires à la gestion des données de jeu. Au niveau de l'interface graphique, cette classe contient aussi tous les visuels des éléments de jeu (cartes et frontières).

Les visuels des cartes sont créés par la fonction initializeCards(). Pour cela, la fonction renvoie un dictionnaire classé par numéro et couleur avec tous les visuels de chaque carte du jeu. Les visuels des cartes sont des objets pygame.Surface. La fonction initializeHandCardsRect() permet de générer deux listes qui donnent la position des surfaces occupées par les visuels des cartes des mains des deux joueurs. Cela servira lors de la gestion des évènements clics et pour placer les visuels des cartes au bon endroit. La fonction initializeSideCardsRect() est similaire à la fonction précédente et renvoie la liste des positions occupées par les surfaces des

visuels des cartes jouées. La fonction initializeFrontiers() créée les visuels des frontières et génère les positions à prendre pour chaque frontière selon son état (non revendiqué, revendiqué par le joueur 1, revendiqué par le joueur 2).

Les trois dernières fonctions de cette classe intervenant dans l'interface graphique sont les fonctions displayFrontiers(), displayHands() et displaySides(). Ce sont ces fonctions qui vont être appelées à chaque fois que la scène principale de jeu va être rafraichie avec la fonction updateScene() de la classe Game. Le but de ces fonctions est de placer les différents visuels des mains des joueurs, des frontières et des cartes jouées en fonction des données du jeu qui sont les attributs de la classe Gameboard. La surface principale du jeu sera donc fournie en paramètre de ces fonctions lors de leur appel et les différents visuels seront rajoutées par-dessus. Pour superposer une surface sur une autre, nous utilisons la fonction « pygame.blit() » qui prend en paramètre le visuel et sa position de la surface à afficher.

Implémentation de l'I.A.

1. Étude de cas

Après avoir étudié le cas du Schotten Totten dans une démarche d'implémentation d'une I.A., nous en avons tiré plusieurs caractéristiques.

Le Schotten Totten est un jeu :

- Asynchrone : c'est un jeu au tour à tour, cela implique que le joueur à connaissance de l'état actuel des choses lors de sa prise de décision et cela n'évoluera pas avant qu'il agisse.
- À somme nulle : le gain d'un joueur entraine inévitablement une perte pour l'autre.
- À informations incomplètes : le joueur n'a pas connaissance des cartes que possède son adversaire.

Dans un premier temps, la première caractéristique implique que notre I.A. n'aura besoin que de connaître l'état dans lequel elle se trouve pour décider de son prochain coup. Ensuite, la deuxième caractéristique nous indique que notre I.A. devra toujours chercher à maximiser ses gains et minimiser les gains de l'adversaire. Enfin, la troisième implique que l'I.A. devra émettre des hypothèses sur le contenu de la main de son adversaire pour générer les états suivant son coup.

2. Représentation des états

Au début, nous avions songé à créer des copies de notre objet « Gameboard » pour aller plus vite en ce qui concerne l'implémentation et afin d'éviter de devoir adapter certaines fonctions déjà implémentés. Nous aurions utilisé la fonction « deepcopy » de la librairie « copy » que propose Python pour créer des copies profondes de nos objets. Cependant, après réflexion, nous en avons conclu que cela n'était pas du tout optimal dans le cadre d'une I.A. car notre objet contiendrait trop d'informations inutiles et qu'il serait préférable de minimiser le plus possible la représentation de notre état.

Nous avons donc codé notre état de la manière suivante. Nous avons représenté les cartes comme étant un tuple de deux entiers (un entier pour sa valeur et un autre pour sa couleur). Etant donné que les états prochains d'un état initial dépendent des cartes que le joueur possède en mains, la représentation de notre état est une association de la représentation simplifié de la main du joueur et du plateau de jeu. La forme simplifié de la main est simplement une liste de tuples qui représentent les cartes. La forme simplifié du plateau est une liste de 9 autres listes représentant les frontières. Chaque représentation de frontière comporte en tête son statut (-1,0 ou 1 en fonction des revendications), puis trois tuples pour les cartes posés d'un côté ainsi que 3 autres tuples pour les cartes du côté adverse. Une absence de carte sera comblée par le tuple « (0,0) ».

Voici ci-dessous un exemple de représentation d'état :

[[(4,6), (5,2), (2,1), (6,3), (4,3), (7,4)], ← main du joueur
[[-1, (0,0), (0,0), (0,0), (0,0), (0,0), (0,0)], ← frontière 1
[-1, (8,4), (0,0), (0,0), (7,6), (8,6), (0,0)], ← frontière 2
[-1, (0,0), (0,0), (0,0), (0,0), (0,0), (0,0)], ← frontière 3
[1, (4,4), (5,1), (6,1), (1,3), (1,2), (1,4)], ← frontière 4
[1, (7,5), (8,1), (9,3), (2,2), (2,4), (2,1)], ← frontière 5
[-1, (0,0), (0,0), (0,0), (0,0), (0,0), (0,0)], ← frontière 6
[-1, (8,2), (0,0), (0,0), (0,0), (0,0), (0,0)], ← frontière 7
[-1, (0,0), (0,0), (0,0), (0,0), (0,0), (0,0)], ← frontière 8
[-1, (0,0), (0,0), (0,0), (0,0), (0,0), (0,0)]]] ← frontière 9

3. Evaluation d'un état

Afin d'évaluer le gain potentiel pour l'I.A. du passage d'un état à un autre, il était nécessaire d'implémenter une fonction d'évaluation capable d'associer un score à un état en fonction de la position de l'I.A. dans la partie. Dans le cas d'un jeu comme le Schotten Totten qui peut paraître assez compliqué au premier abord, il est judicieux de diviser la fonction d'évaluation en plusieurs sous-fonctions d'évaluation. Nous avons donc implémenter 4 sous-fonctions d'évaluations qui traitent chacune d'un thème en particulier. Nous leur avons également attribué des poids de valeurs proportionnelles à leur importance dans la partie (selon notre analyse du jeu). La somme totale des poids est égale à 1. Voici ci-dessous la fonction « eval » :

$$eval(x) = W1 \times eval1(x) + W2 \times eval2(x) + W3 \times eval3(x) + W4 \times eval4(x)$$

Etant donné que chaque sous-fonctions d'évaluation renvoie une valeur comprise entre -1 et 1, il en va de même pour la fonction d'évaluation globale.

a. 1ère fonction d'évaluation

La première fonction évalue la position d'un joueur par rapport à son nombre de frontières revendiquées. On prend également en compte la nombre de frontières revendiquées du côté adverse. Pour renvoyer une valeur comprise entre -1 et 1, on renvoie la différence de frontières revendiqués sur le nombre maximale possible à savoir 4 (car si les 5 frontières sont revendiquées, c'est un état terminal). Voici ci-dessous la fonction « eval1 » :

$$eval1(x) = \frac{nbFrontière(x) - nbFrontière(y)}{maxFrontière}$$

Grâce à cette fonction, l'I.A. privilégiera les coups qui permettent la revendication d'une frontière.

b. 2ème fonction d'évaluation

La deuxième fonction évalue la position d'un joueur par rapport à son nombre de possibilité de gagner, c'est-à-dire le nombre de combinaisons possibles de 3 frontières adjacentes. On prend également en compte la nombre de combinaisons possibles du côté adverse. Pour renvoyer une valeur comprise entre -1 et 1, on

renvoie la différence de combinaisons possibles sur le nombre maximale possible à savoir 9. Le simple fait de revendiqué une frontière peut supprimer jusqu'à 3 combinaisons à son adversaire. Voici ci-dessous la fonction « eval2 » :

$$eval2(x) = \frac{nbCombiPossible(x) - nbCombiPossible(y)}{maxCombiPossible}$$

Grâce à cette fonction, l'I.A. privilégiera les coups qui permettent de supprimer des possibilités de victoire à son adversaire.

c. 3ème fonction d'évaluation

La troisième fonction évalue la position d'un joueur par rapport à son nombre de combinaisons de frontières adjacentes, c'est-à-dire le nombre de séries de 3 frontières où il ne manque qu'une frontière à revendiquer pour gagner la partie. On prend également en compte la nombre de combinaisons gagnantes du côté adverse. Pour renvoyer une valeur comprise entre -1 et 1, on renvoie la différence de combinaisons sur le nombre maximale possible à savoir 7. Voici ci-dessous la fonction « eval3 » :

$$eval3(x) = \frac{nbCombiGagnante(x) - nbCombiGagnante(y)}{maxCombiGagnante}$$

Grâce à cette fonction, l'I.A. privilégiera les coups qui permettront de gagner par 3 frontières adjacentes et essayera d'empêcher l'adversaire de gagner de cette façon-là.

d. 4ème fonction d'évaluation

La quatrième et dernière fonction évalue la position d'un joueur par rapport à la domination de ses cartes sur chaque frontière, c'est-à-dire le nombre de frontière pour lesquelles la combinaisons de cartes et plus forte que celle de son adversaire. Si les joueurs n'ont pas posé toutes leurs cartes, on la combinaison encore réalisable avec les cartes non-révélées la plus forte. On prend également en compte la nombre de frontières dominées par côté adverse. Pour renvoyer une valeur comprise entre -1 et 1, on renvoie la différence de frontières dominées sur le nombre maximale possible à savoir 9. Voici ci-dessous la fonction « eval4 » :

$$eval4(x) = \frac{nbFrontièreDominée(x) - nbFrontièreDominée(y)}{maxFrontièreDominée}$$

Grâce à cette fonction, l'I.A. privilégiera les coups qui assureront plus tard la revendication d'une frontière.

4. Détermination de l'arbre de jeu

Dans cette partie, afin de rendre notre I.A. la plus efficace possible, notre objectif aura été de créer un arbre de jeu ayant un facteur de branchement le plus faible possible dans but de maximiser sa profondeur de recherche. Il nous semblait indispensable d'atteindre une profondeur minimale de 6 pour qu'on puisse prévoir au moins la revendication d'une frontière pour laquelle aucune carte n'a encore été posée.

Afin de pouvoir déterminer notre arbre de jeu, nous avons dû implémenter une fonction générant les états successeurs d'un état initial, la fonction « Next State ». Si l'on voulait prendre en compte tous les états successeurs possibles dans le cas du joueur connaissant sa main, nous n'aurions qu'à créer un état par cartes posées sur chaque frontières encore en jeu et cela pour chaque cartes de la main du joueur. Pour un état ayant connaissance de la main complète du joueur, à savoir les 6 cartes et où les 9 frontières seraient libres, nous aurions à traiter 54 états successeurs. Cela paraît déjà assez lourd. Nous avions envisagé de garder cette solution dans un premier temps, nous nous sommes rendu compte que, dans certains cas et notamment au tout début d'une partie, lorsque le plateau est vide, cela ne paraît pas pertinent de se préoccuper de toutes les frontières et il serait surement plus judicieux de cibler de frontières stratégiques. Nous avions donc eu l'idée d'évaluer la force des cartes de la main du joueur dans le but de cibler les frontières à attaquer. Par exemple, si la main du joueur est évaluée comme étant plutôt faible l'I.A. posera ses cartes en priorité sur les côtés étant donné qu'elles ont moins d'importance car elles offrent moins de possibilités de victoire par frontières adjacentes. En revanche, si la main est évaluée comme étant forte, l'I.A aura tendance à cibler les frontières du centre qui sont plus importantes.

Sachant que le Schotten Totten est à informations incomplètes puisque le joueur ne connaît pas les cartes de son adversaire, l'I.A. n'est pas censé les connaître non-plus. C'est pourquoi nous avions dû adapter la fonction « NextState » pour qu'elle génère des coups relatifs à l'adversaire. Nous avons donc implémenté la fonction « NextStateOpponent ». Il était impensable de générer des états en tenant compte de toutes les cartes dont l'I.A. n'a pas connaissance car cela entraînerait un facteur

de branchement bien trop important. Nous avons donc dû procéder autrement. Cette fonction génére les états successeurs d'un état du tour adverse en partant du principe que celui-ci possède les meilleurs cartes (en termes de valeurs) parmi celles qui n'ont pas encore été révélées. Cela à l'avantage de traiter les pires cas pour que l'I.A agisse en conséquence. Cependant, cela implique également que l'I.A. soit moins performante en début de partie mais s'améliore sur la durée.

Pour ce qui est de la génération de l'arbre, nous avons opté pour un algorithme de type « MinMax » avec élagage « AlphaBeta ». L'élagage semblait indispensable étant donné que la difficulté de ce projet réside principalement dans la gestion du facteur de branchement qui rend la tâche plus compliquée.

5. Pistes d'amélioration

Dans un premier temps, nous avions pensé que l'implémentation d'un algorithme de type « MTD » (Memory-enhanced Test Driver) pourrait être judicieux car celui-ci tiendrait compte des états générés plusieurs fois dans l'arbre. Cela réduirait fortement le facteur de branchement et l'on pourrait construire des arbres de profondeurs plus importante.

Dans un second temps, nous avions eu l'idée d'une 5ème fonction d'évaluation prenant en compte le fait que les frontières soient dominées avec une écart de puissance de combinaisons minimale. Cela aurait pour effet de rendre l'I.A. plus attentive au fait qu'il ne vaut mieux pas poser une combinaison de cartes largement plus puissante face à une combinaison déjà très faible.

Enfin, concernant les poids attribués aux fonctions d'évaluations, étant donné que nous les avons choisis selon notre estimation des importances de chaque fonctions, cela ne se rapproche pas forcement de la réalité et il serait suremement préférable de les déterminer par le biais d'un algorithme, chose que nous n'avons pas pu faire par manque de temps. On pourrait imaginer se faire s'affronter deux I.A. avec des poids de fonctions d'évaluation générer initialement aléatoirement sur plusieurs parties. L'I.A qui remporterait le plus de partie conserverait ces poids tandis que l'autre en générerait de nouveaux. On répèterait cette opération un très grand nombre de fois pour déterminer avec quelles poids l'I.A est la plus performante.

Conclusion

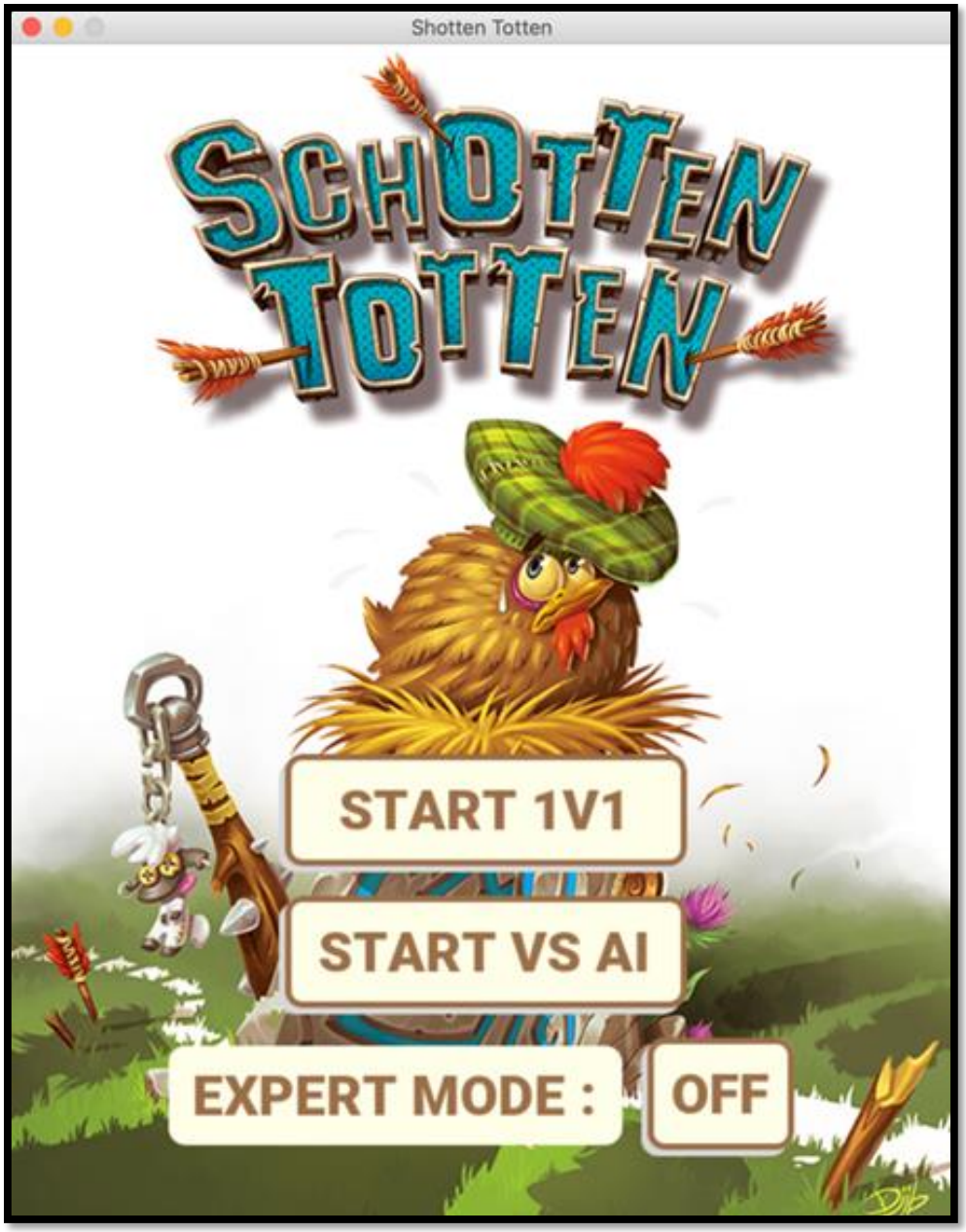
Ce projet nous a demandé un très grand investissement personnel auquel nous étions préparés en choisissant un sujet complexe alors que nous n'étions seulement un groupe de deux. Notre objectif d'implémenter le mode expert du jeu était plutôt ambitieux et nous nous sommes rendu compte que plus tard que cela serait plus compliqué que prévu. Nous sommes donc assez déçus de ne pas avoir pu traiter ce cas. La partie de développement la plus longue a été celle du développement du jeu, de son architecture ainsi que de son interface graphique. Le fait de travailler à plusieurs sur un même projet implique une très bonne communication pour que chacun puisse avancer de son côté tout en tenant compte du travail réalisé par l'autre. Par exemple, lors de la réalisation du jeu, l'un d'entre nous a majoritairement développé le jeu et l'autre a réalisé l'interface graphique qui s'appuie dessus. Ainsi, il fallait donc que celui qui a réalisé l'interface ait une vision très précise des choix d'implémentations de l'autre pour pouvoir récupérer les données du jeu en conséquence et pour que la structure logicielle de l'interface colle à celle du jeu.

Nous sommes satisfaits du rendu visuel et opérationnel du jeu bien que certains points auraient mérité un peu plus de temps pour remplir les objectifs que nous nous étions fixés. Nous avions pour but de créer une IA jouant comme un vrai joueur. Nous avons rencontré quelques problèmes lors de sa conception car la génération des états engendrait un facteur de branchement trop important lorsque nous voulions augmenter la profondeur de recherche de l'algorithme, ce qui nous a contraint à devoir nous restreindre à une faible profondeur rendant ainsi notre IA moins performante. Malgré tout, ce projet nous a permis de maitriser davantage le langage Python et le développement d'une interface graphique d'une bibliothèque Python. Cela aura été très enrichissant de s'initier à la conception entière d'une intelligence artificielle et nous avons tirés de bonnes leçons de certaines de nos erreurs qui paraissait évitable.

Annexe

Présentation de l'interface

Voici la fenêtre principale au lancement du jeu ainsi que la fenêtre de sélection des noms des deux joueurs qui se lance dès qu'on appuie sur un des deux boutons START :



On peut alors modifier les noms des deux joueurs et appuyer sur le bouton START pour lancer le jeu. La fenêtre de jeu s'ouvre alors :

Pour commencer à jouer, le joueur 1 doit sélectionner une carte de son jeu ainsi qu'une frontière où placer sa carte en cliquant sur les éléments voulus. Une fois une frontière et une carte sélectionnées, la carte se place automatiquement sous la frontière voulue et il est demandé au joueur de cliquer sur une frontière s'il désire revendiquer une frontière. Il a alors la possibilité de revendiquer autant de frontière qu'il veut avant de passer son tour. Si la frontière peut être revendiquée, elle se place automatiquement du côté du joueur qu'il l'a revendiqué et un message lui indique que la frontière a bien été revendiquée, sinon, le jeu lui indique que la frontière ne peut pas l'être. Une fois le tour du joueur 1 passé, ses cartes se retournent pour que son adversaire ne les voit pas et ce de manière identique pour l'autre joueur.

Voici une fenêtre de jeu lorsque plusieurs cartes sont posées et que deux frontières ont été revendiquées par le joueur 1 :

Une fois le jeu terminé, lorsque la dernière frontière a été revendiqué et qu'un joueur en possède assez pour remporter la partie, un message apparaît notifiant le joueur gagnant de sa victoire et tous les éléments du jeu disparaissent.