INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Rapport de Conception

Etienne ALLAIN, Aymen BERRAJAA, Noha DOAIF,
Giorgi KUCHUKHIDZE, Aymane MENFAA

Encadrante: Laurence Rozé

Fevrier 2024

Table des matières

1 Introduction

Notre projet intitulé "Cœur Artificiel" se focalise sur la conception d'une application mobile dédiée au jeu des "Cœurs". Pour réaliser ce projet, il est nécessaire de développer deux parties :

— L'application qui propose aux utilisateurs une expérience de jeu contre des intelligences artificielles
— Les intelligences artificielles

La première partie du projet requiert la conception d'une interface utilisateur en Flutter permettant à un utilisateur de jouer au jeu des cœurs. Cette interface doit être capable de gérer diverses options de jeu tout en fournissant une expérience visuelle attrayante pour les utilisateurs.

La seconde partie du projet se penche sur le développement de deux intelligences artificielles en Python, hébergées sur un serveur Flask. Le choix de créer deux intelligences distinctes, plutôt qu'une seule, découle d'une volonté d'explorer diverses approches et de diversifier les solutions proposées.

Les deux intelligences artificielles envisagées reposent sur des algorithmes distincts, à savoir les réseaux de neurones et le Monte Carlo Tree Search (MCTS). Ces deux choix découlent de la pertinence des réseaux de neurones pour l'apprentissage à partir de données complexes et de la réputation des MCTS pour leur efficacité dans l'exploration des différentes possibilités dans les jeux de cartes.

Pour garantir le bon fonctionnement global du projet, il est impératif de connecter les intelligences artificielles à l'application Flutter. À cette fin, une communication HTTP sera établie entre le client et le serveur, permettant la transmission des décisions des intelligences artificielles et du serveur au client.

Dans la suite du rapport, nous examinerons de près la conception de l'interface utilisateur, le développement des intelligences artificielles basées sur les réseaux de neurones et le MCTS, ainsi que la mise en œuvre de la communication entre le client et le serveur.

2 L'application

Le projet du jeu des cœurs se décompose en plusieurs parties de conception distinctes. Nous allons aborder dans cette section la partie conception de l'application (interface utilisateur).

2.1 Fonctionnement global

Dans cette section, nous allons exposer les différentes fonctionnalités que propose notre application.

Tout d'abord, notre application permet de visualiser les règles du jeu afin que l'utilisateur puisse comprendre les objectifs du jeu.

Ensuite, nous avons mis en place une option qui permet à l'utilisateur de sélectionner les types d'intelligences artificielles contre lesquelles ils souhaitent jouer. Ces intelligences artificielles sont implémentées sur un serveur Python Flask. Les options comprennent une IA basée sur un algorithme MCTS (Monte Carlo Tree Search) et une IA reposant sur un réseau de neurones. Cette personnalisation permet aux utilisateurs de défier des adversaires d'intelligence artificielle de différents niveaux de difficulté et implémenter de façon différentes.

L'application est également dotée de la logique de jeu complète pour s'assurer que les parties se déroulent correctement, en respectant les règles du jeu des cœurs. Elle gère l'attribution des cartes, la distribution, les phases de plis, le suivi des points et bien évidemment la détermination du vainqueur à la fin de chaque partie.

Un aspect essentiel de notre application est sa capacité à communiquer en temps réel avec le serveur Python qui héberge les intelligences artificielles. Cette communication permet à l'application de transmettre les actions des utilisateurs aux IA, de recevoir leurs réponses, et enfin, d'afficher les cartes jouées par ces dernières à l'écran.

Pour résumer, notre application fournit une explication détaillée des règles, permet aux utilisateurs de sélectionner leurs adversaires parmi différentes intelligences artificielles, assure le bon déroulement du jeu, et établit une communication transparente avec le serveur Python pour interagir avec les intelligences artificielles.

2.2 Aperçu de l'application

Dans cette section, nous allons présenter une vue globale de l'interface utilisateur attendue. Voici un aperçu final de notre application :

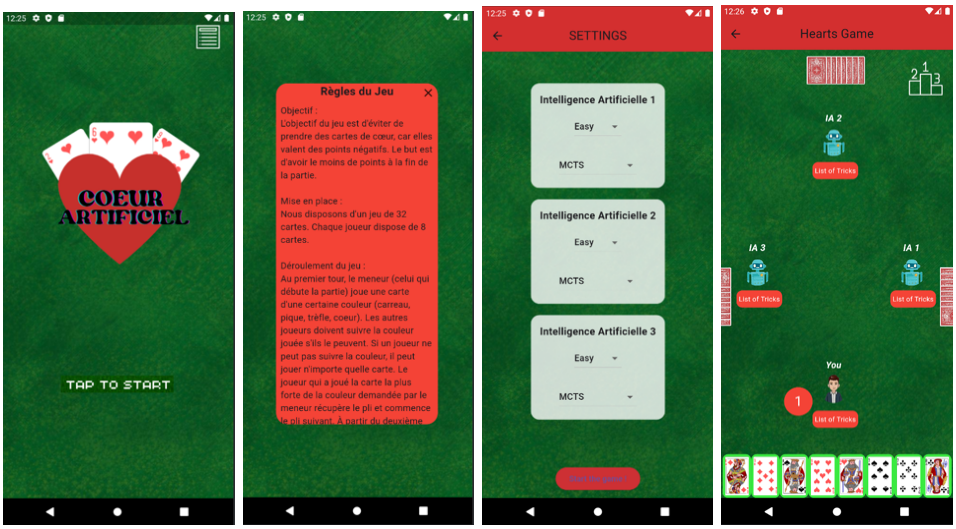

FIGURE 1 – Interface graphique attendue de l'application.

Nous pouvons voir les différentes vues de l'application : la page d'accueil, les règles du jeu, le choix des intelligences artificielles à affronter ainsi que leurs niveaux de difficulté et finalement la page principale du jeu.

A l'ouverture de l'application, l'utilisateur verra l'interface principale de notre jeu, qui lui indiquera qu'il faut cliquer sur l'écran pour lancer le jeu, comme illustré dans la première vue. Cette interface contient un boutton permettant de visualiser un menu défilant présentant les détails des règles du jeu comme le montre la deuxième vue.

En cliquant sur le boutton "TAP TO START", le joueur peut personnaliser sa partie en choisissant les différentes intelligences artificielles à affronter avec leurs niveaux de difficultés comme le montre la troisième vue de l'application.

Au lancement du jeu, l'utilisateur fera face à l'interface graphique du jeu des Cœurs, qui contient les cartes visibles du joueur en bas, les cartes non visibles en haut, à droite, et à gauche des joueurs qui l'affrontent. Cette interface contient également un boutton de retour tout en haut permettant au joueur de quitter la partie et un boutton à l'extrême droite permettant de visualiser le tableau de score des joueurs. La dernière vue permet de visualiser clairement celà.

2.3 Architecture de l'application

La Figure 2 permet de visualiser les différents frameworks et bibliothèques extérieurs utilisées pour accomplir les services de l'application. L'application (le client) est entièrement réalisée en Flutter, alors que le serveur est réalisé en Python, avec l'utilisation du framework Flask.

Le visuel est réalisé dans la classe MainWindow qui est associée à chacune des autres classes, chacune permettant une fonctionnalité bien précise de l'interface. Le diagramme de classes réalisé pour un joueur, est ainsi représenté en Figure 3.

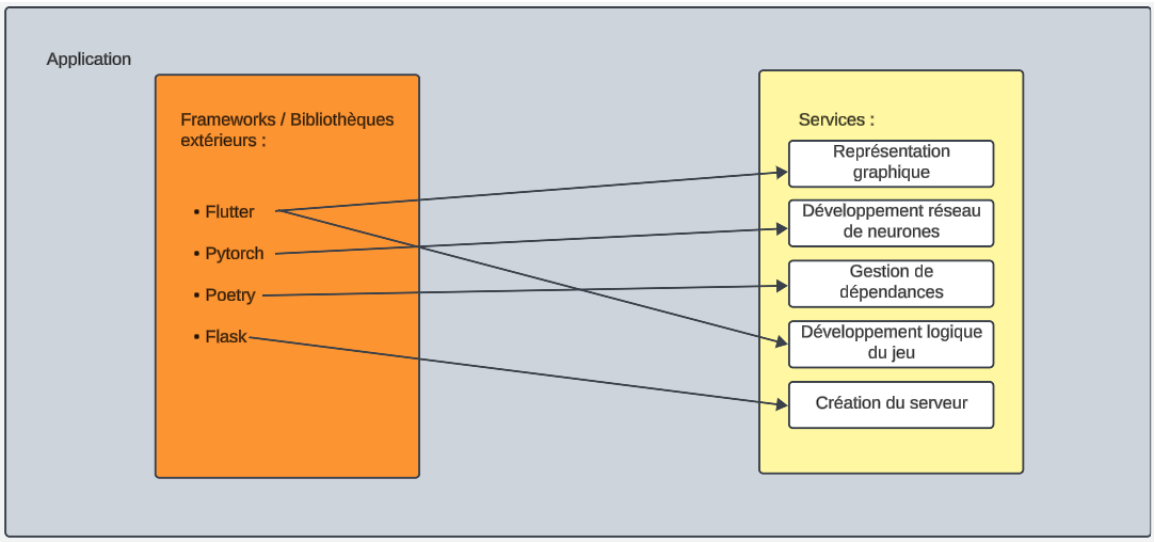

FIGURE 2 – Frameworks et bibliothèques de l'application.

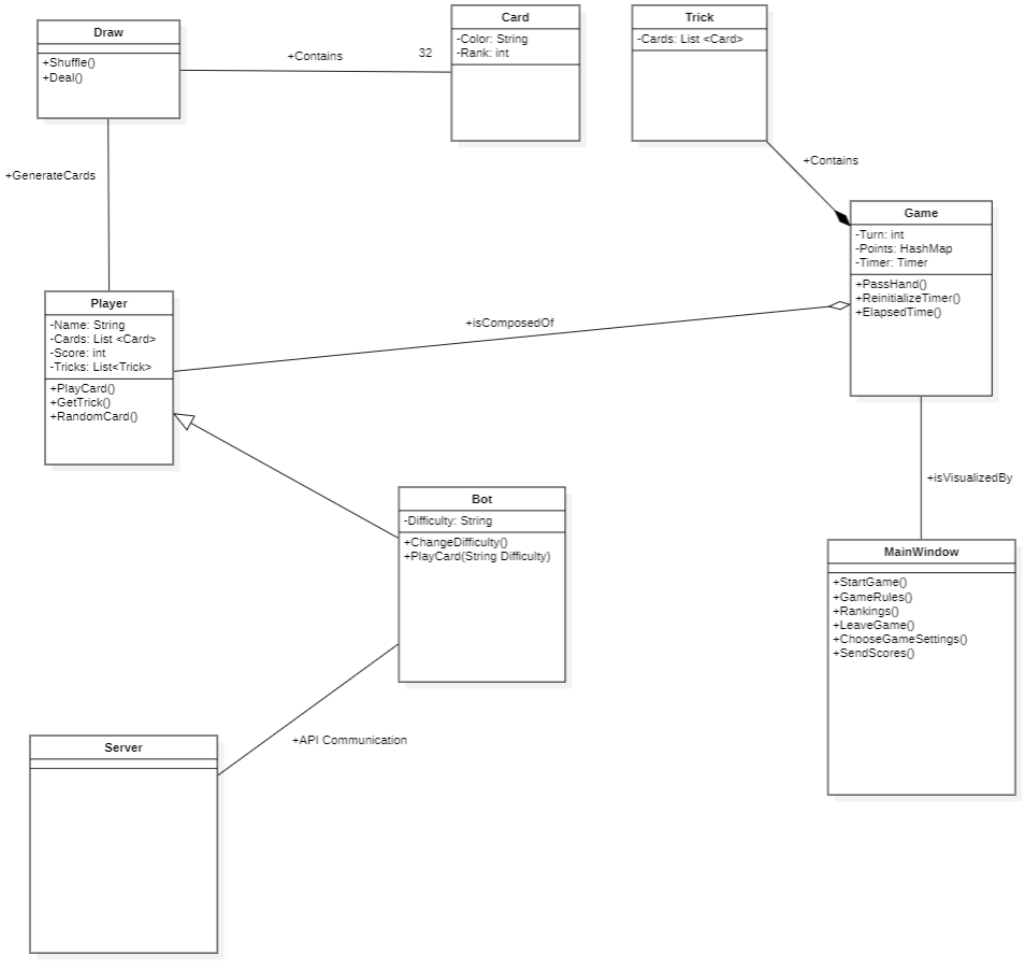

FIGURE 3 – Diagramme UML.

La classe principale **MainWindow** sert à gérer les interactions majeures avec le client à travers plusieurs fonctions :

— `StartGame ():` permet de lancer la partie.
— `GameRules ():` permet d'afficher les règles du jeu.
— `Rankings ():` permet d'afficher les classements des différents joueurs dans la partie en cours.
— `LeaveGame ():` permet de quitter la partie.
— `ChooseGameSettings ():` permet à l'utilisateur principal de personnaliser sa partie en choisissant les paramètres souhaités.
— `SendScores ():` permet d'envoyer au serveur les scores à la fin de la partie.

Ensuite, la classe **Game** permet de représenter une partie du jeu des cœurs. Elle est donc composée des attributs suivants :

— `Turn:` c'est un entier qui représente le tour du joueur dans la partie.

— `Points:` c'est une HashMap qui permet d'associer à chaque joueur ses points gagnés dans la partie. Elle contient comme clé le nom du joueur, et comme valeur les points de ce joueur.
— `Timer:` c'est un chronomètre associé à chaque joueur ayant le tour.

Cette classe permet également d'accomplir plusieurs fonctionnalités grâces à ses fonctions :

— `PassHand ():` permet de donner la main à un joueur dans la partie.
— `ReinitializeTimer ():` permet de réinitialiser le chronomètre pour chaque joueur.
— `ElapsedTime ():` permet de reconnaître le temps écoulé dans la partie.

La classe **Game** et **Player** sont complétées par la classe **Trick** qui contient comme attribut la liste des cartes du pli.

Ensuite, on a la classe **Player**, qui représente le joueur dans la partie. Cette classe contient les attributs suivants :

— `Name :` c'est une chaîne de caractères qui représente le nom du joueur.
— `Cards :` c'est une liste de cartes qui représente la main du joueur.
— `Score :` c'est un entier qui représente le score cumulé du joueur.
— `Tricks :` c'est une liste qui représente les plis remportés par le joueur.

Cette classe permet d'interagir avec la partie à travers trois fonctions principales :

— `PlayCard () :` permet de jouer une carte.
— `GetTrick () :` permet d'avoir les plis du joueur.
— `RandomCard () :` permet de sélectionner une carte aléatoire.

La classe **Bot** qui hérite de **Player** représente les intelligences artificielles et contient en plus les attributs suivants :

— `Difficulty :` c'est une chaîne de caractères qui spécifie le niveau de difficulté de l'intelligence Artificielle.

Cette classe accomplit en plus les fonctionnalités suivantes :

— `ChangeDifficulty () :` permet de changer le niveau de difficulté de l'intelligence artificielle.
— `PlayCard (String Difficulty) :` permet de jouer une carte selon le niveau de difficulté spécifié en paramètre.

La classe **Bot** communique avec la classe **Server** car la logique des intelligences artificielles est implémentée côté serveur.

La classe **Draw** permet de générer les cartes pour les joueurs à travers deux méthodes principales :

— `Shuffle () :` permet de mélanger les cartes avant de les distribuer.
— `Deal () :` permet de distribuer les cartes sur les joueurs.

Et finalement, on a une classe **Card** qui représente les cartes du jeu et qui contient deux attributs principaux :

— `Color :` représente la couleur de la carte.
— `Rank :` représente le rang de la carte.

Le fonctionnement de l'application ainsi expliqué, la communication client-serveur qui permet d'illustrer réellement les relations du diagramme UML sera expliquée dans la partie suivante.

2.4 Communication client-serveur

Une bonne communication entre le client et le serveur est importante car elle garantit une expérience de jeu fluide. Du côté serveur, il y a la mise en place d'un fichier principal Python qui orchestre le serveur Flask. Ce fichier permet de créer et de lancer le serveur avec les lignes de code clés suivantes :

```
# Création de l'application Flask
app = Flask(__name__)
CORS(app)
if __name__ == '__main__':
    # Permet de lancer le serveur Flask sur l'hôte 127.0.0.1 (localhost) et sur le port 5000
    app.run(host='127.0.0.1', port=5000)
```

L'utilisation de `CORS(app)` dans Flask est important pour résoudre les problèmes de Cross-Origin Resource Sharing (CORS), qui surviennent lorsque des requêtes sont effectuées entre des domaines différents dans une application web. En activant `CORS(app)`, on permet au serveur Flask de répondre aux requêtes provenant de domaines autres que celui sur lequel il est hébergé, ce qui est essentiel pour autoriser des interactions sécurisées entre notre serveur Flask et notre client Flutter. Ensuite, ce fichier contient les endpoints nécessaires pour notre application, définis à l'aide de l'annotation `@app.route`. Parmi ces endpoints, nous avons :

- `/play_move` : Cet endpoint permet à une intelligence artificielle de jouer un coup, et le résultat est renvoyé au client pour l'affichage à l'utilisateur.
- `/initialisation` : Lorsque l'utilisateur a choisi les types d'IA contre lesquelles il souhaite jouer, cet endpoint prévient le serveur de l'initialisation de ces intelligences artificielles.
- `/scores` : À la fin de la partie, cet endpoint est utilisé pour transmettre les scores au serveur, qui seront utilisés pour l'entraînement de l'IA.

Du côté du client Flutter, nous utilisons la bibliothèque `http.dart` pour émettre des requêtes HTTP vers le serveur Flask. Par exemple, lorsque l'utilisateur valide les types d'IA qu'il souhaite affronter, une requête HTTP est envoyée avec un corps de requête JSON pour spécifier ces choix. En voici un exemple :

```
final Map<String, dynamic> requestBody = {
  "ia_players": [
    {"id": 1, "type": ia1Type},
    {"id": 2, "type": ia2Type},
    {"id": 3, "type": ia3Type},
  ]
};

final String requestBodyJson = json.encode(requestBody);

try {
  final response = await http.post(
    Uri.parse('http://localhost:5000/initialisation'),
    headers: <String, String>{
      'Content-Type': 'application/json',
    },
```

```
    body: requestBodyJson,
  );

  // Gestion de la réponse en fonction du statut de la requête
  if (response.statusCode == 200) {
    // La requête a réussi, vous pouvez maintenant naviguer vers la page de jeu
    Navigator.pushNamed(context, "/game");
  } else {
    // La requête a échoué, affichez un message d'erreur
    print("Erreur lors de la requête d'initialisation : ${response.statusCode}");
  }
} catch (e) {
  // Gestion des erreurs en cas de problème de réseau, etc.
  print("Erreur lors de la requête d'initialisation : $e");
}
```

De plus, des requêtes HTTP sont également émises lorsque c'est le tour d'une IA de jouer afin de pouvoir afficher la carte jouée par l'IA à l'utilisateur et lorsque la partie se termine afin de transmettre au serveur les scores de la partie qui sont nécessaires à l'entrainement de l'IA.

Cette architecture de communication client-serveur assure une coordination fluide entre l'interface utilisateur et le serveur Flask.

3 Les intelligences artificielles

3.1 Outils utilisés

Nous avons opté pour l'utilisation de la bibliothèque open-source de machine learning Pytorch pour développer notre réseau de neurones au vu de sa facilité d'utilisation et de son interface intuitive, offrant ainsi une expérience de développement plus conviviale.

Concernant l'algorithme MCTS, aucune bibliothèque existante n'a retenu notre attention car elles n'étaient soit pas suffisamment intuitive permettant un gain de temps significatif soit pas compatible avec notre jeu des Coeurs (MCTS pour des jeux à information parfaite, ce qui ne correspond pas à notre cas).

Nous avons ensuite décidé d'utiliser poetry pour gérer les dépendances de notre projet Python. En effet, la déclaration des dépendances est réalisée dans un fichier nommé "pyproject.toml" et poetry s'occupe d'installer les bonnes versions de ces dépendances, rendant ainsi le projet Python moins susceptible à l'obsolescence de certaines bibliothèques avec le temps. Voici un exemple d'un fichier pyproject.toml :

```toml
[tool.poetry]
name = "pythonproject"
version = "0.1.0"
description = ""
authors = ["Noha <noha.doaif@insa-rennes.fr>"]
readme = "README.md"

[tool.poetry.dependencies]
python = "^3.10"
torch = "^2.1.2"
matplotlib = "^3.8.2"
numpy = "^1.26.3"


[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

FIGURE 4 – Exemple d'un fichier pyproject.toml

- python = "^3.10" : signifie que les versions de python pouvant être utilisées sont les versions superieures ou égales à la version 3.10 .

3.2 Implémentation du jeu des coeurs

Afin de construire et entraîner les deux intelligences artificielles, il était impératif de disposer de l'intégralité de la logique et de l'architecture du jeu. C'est la raison pour laquelle nous avons entrepris de recoder le jeu des cœurs sans interface, en adoptant une approche minimaliste pour faciliter l'entraînement du modèle. Bien que des codes préexistants de ce jeu soient disponibles en ligne, la variation des règles nous a conduit à préférer consacrer du temps à cette tâche, garantissant ainsi une maîtrise totale du code à utiliser.

Cette structure a été formalisée à travers le diagramme de classes suivant :

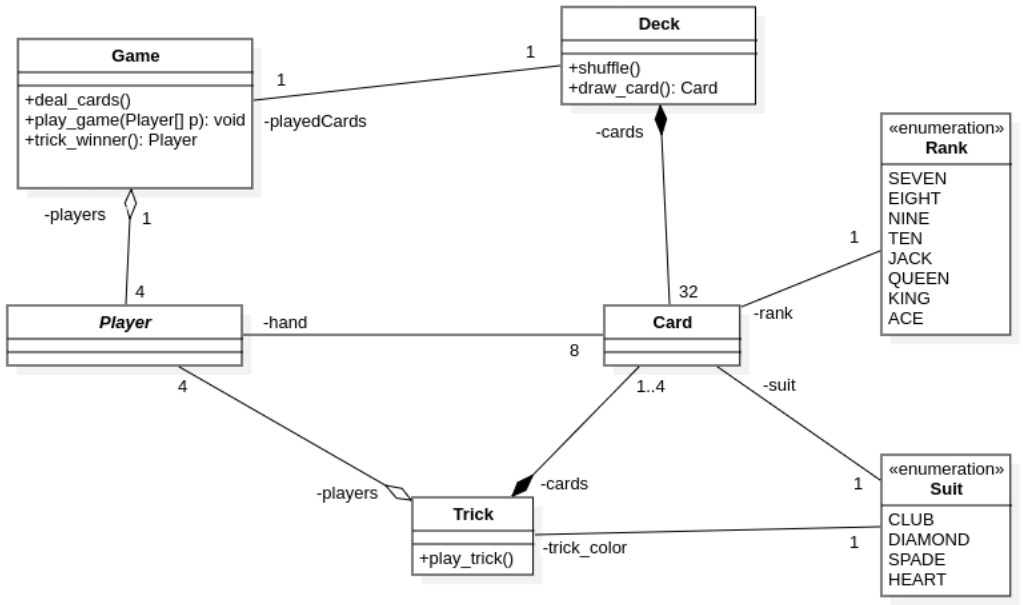

FIGURE 5 – Diagramme de classes du jeu

Notre conception implique la représentation de chaque élément du jeu, à savoir la carte, le paquet, le pli, la partie et le joueur.

La classe `Deck` est responsable de la génération des 32 cartes initiales et du mélange de l'ordre de ces cartes au début de chaque partie avec la méthode `shuffle()`. Les deux énumérations `Suit` et `Rank` définissent respectivement les quatre couleurs de cartes et les valeurs hiérarchiques des cartes dont a besoin le jeu des Cœurs. Cela permet également une représentation visuelle des cartes, par exemple, `HEART = '♡'`, `JACK = 'J'`, ce qui rend une main de joueur comme suit :
$[J♡, K♡, A♢, 7♢, Q♠, 8♢, 8♡, A♠]$.

Pour démarrer une partie, il est essentiel d'avoir quatre joueurs. Actuellement, la classe `Player` est vide, car son implémentation dépend des besoins spécifiques des deux intelligences artificielles. Nous allons donc détailler les deux approches d'implémentation de cette classe dans les sections dédiées de chacune des deux intelligences artificielles.

Enfin, une partie est lancée avec quatre joueurs. La classe `Game` contient un paquet avec lequel elle distribue les cartes à chaque joueur en prenant les cartes avec la méthode `draw_card()` de la classe `Deck`. La partie est lancée avec `play_game()`, une méthode qui elle-même lance 8 plis. La classe `Trick` contient une méthode `play_trick()` qui fait jouer chaque joueur à tour de rôle, si c'est le début d'un pli, la couleur est enregistrée dans un attribut et le reste des joueurs doivent respecter cette couleur. À la fin de chaque pli, un vainqueur est déterminé avec la méthode `trick_winner()` pour ouvrir le pli suivant, et un score est calculé.

Ce code nous permet ainsi de générer des parties avec des joueurs aléatoires, une base essentielle pour le développement de l'IA. Par la suite, nous allons intégrer le réseau de neurones et l'algorithme MCTS à ce code et ajouter un joueur capable de jouer de manière autonome.

Le code ne donne pas la possibilité à un humain de jouer car ceci n'est pas nécessaire dans ce contexte, cette fonctionnalité est bien sûr assurée du côté de l'application Flutter.

3.3 Réseau de neurones

Dans cette section, nous présenterons l'IA à base de réseau de neurones. Tout d'abord, nous examinerons en détail le diagramme de classes de notre code, offrant ainsi une vue d'ensemble de la structure organisationnelle de notre système. Ensuite, nous explorerons les différentes couches qui constituent notre modèle, notamment les couches d'entrée, de sortie et cachées, ainsi que les fonctions d'activation utilisées pour façonner le flux d'informations à travers le réseau. Enfin, nous nous attarderons sur le processus d'entraînement du réseau en abordant deux approches, à savoir une approche reposant sur la génération d'une base de données puis, une deuxième approche suivant le principe du Qlearning. Cette section vise à fournir une base solide pour la compréhension du fonctionnement interne du réseau de neurones, ainsi que du processus d'entraînement qui lui permet d'acquérir les compétences nécessaires pour jouer efficacement.

3.3.1 L'architecture du réseau de neurones

3.3.1.a Structure du code du JoueurIA

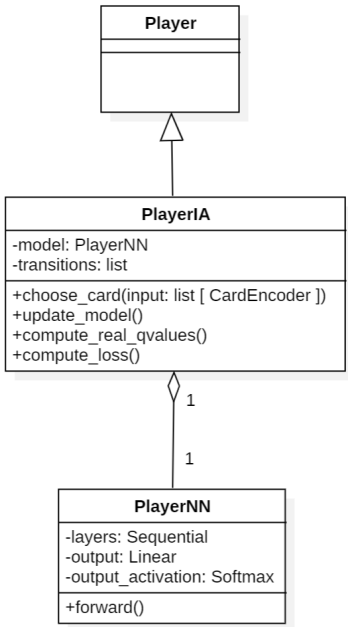

FIGURE 6 – Diagramme de classes du PlayerIA

- Les joueurs contrôlés par l'IA sont des instances de la classe **"PlayerIA"**, qui hérite des fonctionnalités de la classe de base **"Player"**.

- La classe **"PlayerIA"** utilise un objet **"PlayerNN"** pour prendre des décisions basées sur un réseau de neurones.

3.3.1.b Classe Player

La classe `Player` utilisée dans cette partie représente un joueur par défaut qui joue des cartes au hasard. Ce comportement est nécessaire pour l'entraînement de l'IA plus tard. Cette classe comprend un nom, un score, une liste de cartes en main et une liste de cartes remportées pendant la partie. Cette dernière liste est nécessaire pour le calcul du score que réalise la méthode `calculate_score()` à la fin de chaque pli, conformément aux règles du jeu présentées dans le premier rapport. La classe `Player` contient également une méthode `play_card()` lui permettant de jouer une carte de manière aléatoire. S'il ouvre le pli, il jouera sa première carte de la liste ; sinon, il jouera la première carte correspondant à la bonne couleur.

3.3.1.c Les couches du réseau de neurones

Comme décrit dans notre rapport de specification, notre **vecteur d'entrée** pour le réseau de neurones sera constitué de :

- **Numéro du Pli :** int

- **Main du Joueur :** list [Card]

- **Cartes Déjà Jouées dans le Pli :** list [Card]

- **Cartes Restantes dans les Mains des Autres Joueurs :** list [Card]

- **Couleur Demandée dans le Pli Courant :** string

- **Ordre du joueur actuel :équivalent au nombre de cartes sur la table :**list [boolean]

- **Flag : Le joueur actuel a remporté tous les cœurs** : boolean

Ces informations devront être encodées et envoyées en entrée au réseau de neurones. Nous avons pensé à deux encodages differents. Ces deux derniers seront détaillés dans la section suivante. La taille du vecteur d'entrée dépendera de la méthode d'encodage utilisée.

Nous nous sommes inspirés d'un article publié sur le blog Towards Data Science pour la structure de notre réseau de neurones. L'article intitulé **"Teaching a Neural Network to Play Cards"**, explique les étapes de développement d'un réseau de neurones capable de jouer au jeu **"Queen of Spades"**. Etant donné que le **"Queen of Spades"** présente la même complexité que le **jeu des "cœurs"** et que d'après l'auteur la structure des couches cachées n'avait pas beaucoup d'importance, nous avons décidé d'adopter la même structure citée dans cet article tout en l'adaptant à nos entrées et sorties.

Les couches cachées se composeront donc :

- **1ère couche cachée** : 384 unités avec une activation ELU

- **2ème couche cachée** : 384 unités avec une activation ELU

- **3ème couche cachée** : 256 unités avec une activation ELU

- **4ème couche cachée** : 128 unités avec une activation ELU

- **5ème couche cachée** : 32 unités avec une activation Tanh

En ce qui concerne la **couche de sortie** nous y appliquerons la fonction d'activation Softmax.

Cette sortie sera de taille 32, en référence aux 32 cartes du paquet du jeu de cartes. Notre vecteur de sortie sera donc constitué dans ce cas de 32 probabilités. Il existera pour chaque joueur des cartes non jouables, notamment les cartes des autres joueurs ou alors les cartes déjà jouées. Dans ce cas nous appliquerons un masque sur les probabilités des cartes non jouables en les multipliant par 0 en ne gardant que les probabilités des cartes jouables qu'on multipliera par 1. Nous choisirons ensuite la valeur maximale parmi ces probabilités. Cette valeur maximale correspondra à la carte choisie par le playerIA en fonction des entrées.

3.3.1.d Encodage

Pour la premiere approche d'encodage nous avons décidé d'encoder les cartes avec des vecteurs de taille 12 en référence à 8 hauteurs et 4 couleurs. Pour chaque carte la couleur et l'hauteur correspondante à cette carte seront mises à 1 tandis que les autres éléments seront mis à 0. La taille du vecteur d'entrée après encodage sera 430 :

-**numero_pli** : (int) taille = 1

-**main_joueur** : (list [Card]) taille = 8 (cartes) x 12 (taille du vecteur) = 96

-**cartes_deja_jouees** : (list [Card]) taille = 3 (cartes) x 12 (taille du vecteur) = 36

-**cartes_restantes_autres_joueurs** : (list [Card]) taille = 24 (cartes) x 12 = 288

-**couleur_demandee** : (string) taille = 4 (quatre couleurs)

-**ordre_du_joueur actuel** : (list [boolean]) taille = 4 (quatre joueurs)

-**flag : le joueur actuel a remporté tous les cœurs** : (boolean) taille = 1

Taille du vecteur d'entrée = 1 + 96 + 36 + 288 + 4 + 4 + 1 = 430

Pour illuster cet encodage voici quelques exemples :

- En supposant que les couleurs sont représentées dans l'ordre suivant :[♠, ♣, ♢, ♡]

La répresentation de la couleur ♢ après encodage serait : **[0, 0, 1, 0]**

- En supposant que les couleurs et hauteurs sont representées dans l'ordre suivant :

couleurs, hauteurs = [♠, ♣, ♢, ♡, 'A', '7', '8', '9', '10', 'J', 'Q', 'K']

La représentation de la carte (K,♠) après encodage serait : **[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]**

Dans le cas par exemple où le joueur joue une de ses cartes et que cette carte est retirée de la main du joueur, ajoutée aux cartes déjà jouées puis retirée des cartes restantes des autres joueurs. Nous sommes susceptibles d'avoir un vecteur de taille variable. Pour rendre la taille de ce vecteur fixe nous allons remplacer les cartes retirées de la main du joueur par des vecteur de 0 **[0,0,0,0,0,0,0,0,0,0,0,0]** étant donné que ce vecteur de représente aucune carte. Nous procèderons de la même façon pour les cartes déjà jouées tel que nous initialiserons ce vecteur avec trois vecteurs **[[0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0],[0,0,0,0,0,0,0,0,0,0,0,0]]** étant donné que la liste ne contient aucune carte au début d'un pli. Le vecteur représentant la couleur sera également initialisé avec un vecteur [0,0,0,0] dans le cas où aucune couleur n'a encore été choisie.

En début du premier pli par exemple le joueur reçoit 8 cartes :

[K♡, A♠, J♢, 8♣, 8♡, 7♣, 7♢, J♣]

Ce vecteur sera encodé de la manière suivante :

[[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0], [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0], **[0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]**, [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]]

En supposant que le joueur joue la carte 7♣ mise en gras dans le vecteur ci-dessus, le vecteur de la main du joueur sera encodé de la manière suivante :

[[0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0], [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0], **[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]**]

Les informations de type **boolean** seront encodés par des 0 et des 1 tel que **true** sera représentée par **1** et **false** par **0**. Le **numéro_pli** quant à lui étant un entier entre 1 et 8 ne sera pas encodé. Nous avons décidé de l'envoyer tel quel au réseau de neurones.

Dans la deuxième méthode d'encodage que nous envisageons, elle pourrait être moins précise mais elle permettrait de considérablement réduire l'espace mémoire occupé par un vecteur d'entrée. L'approche ici consiste à encoder une carte ou une liste de cartes sous la forme d'un vecteur de 32 objets, chacun associé à une carte. Chaque objet prend la valeur de 1 ou -1, où 1 signifie que la carte est présente.
Expliquons cela plus clairement avec un exemple :
Supposons que nous ayons un ensemble de cartes avec les valeurs suivantes :
$[7\heartsuit, 8\heartsuit, 9\heartsuit, 10\heartsuit, J\heartsuit, Q\heartsuit, K\heartsuit, A\heartsuit]$
Le vecteur d'encodage donne cela :
[1, 1, 1, 1, 1, 1, 1, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1] On observe que les huit premières valeurs du vecteur représentent toutes des cœurs, et ainsi de suite.
Pour les autres informations qui ne sont pas des cartes, nous utilisons le one hot encodage, qui est une technique de représentation où chaque catégorie est représentée par un vecteur binaire unique. Par exemple le numero de pli encodé sur 8 bits, par exemple le pli numéro 3 ressemble à :
[-1,-1,1,-1,-1,-1,-1,-1]
L'ordre du joueur et la couleur sont codés sur 4 bits de la même manière. Le flag ne change pas.
Taille du vecteur d'entrée = 8 + 32 + 32 + 32 + 4 + 4 + 1 = 113
Avec cet encodage nous avons seulement 113 bits pour le vecteur d'entrée.

3.3.2 Entraînement du réseau de neurones

3.3.2.a Traduction des données pour l'entraînement

Pour pouvoir coder et decoder des cartes ou des ensembles de cartes nous avons créé une classe CardEncoder. Cette classe a pour rôle de transformer une carte ou une liste de cates en un vecteur de 32 coordonnées, comme expliqué précédemment.

Ensuite, il est important de recueillir toutes les informations nécessaires pour construire les vecteurs d'entrée et les vecteurs de sortie associés, en vue de les utiliser ultérieurement dans l'apprentissage par renforcement. Pour ce faire, nous devons construire le vecteur d'entrée à partir de toutes les informations disponibles. A chaque fois qu'un joueur doit jouer, nous commençons par construire le vecteur d'entrée en encodant les cartes qu'il a en main, les cartes sur la table, et les cartes qui n'ont pas encore étées jouées, toutes étant encodées par la classe CardEncoder. Ensuite, à ce vecteur, nous ajoutons le reste des informations : la couleur du pli, l'ordre du joueur courant

que nous obtenons en fonction du nombre de cartes sur la table, le numéro de pli en cours et les indicateurs.

3.3.2.b Entraînement avec une base de données

L'étape la plus importante dans la réalisation d'un réseau de neurones est son entraînement. Pour le jeu de cœurs il n'existe pas de base de données permettant d'entraîner le modèle directement, il faut donc trouver une solution. Pour augmenter nos chances de trouver une bonne solution permettant d'avoir une IA puissante nous avons décidé de partir sur deux approches différentes : une avec base de données générée et l'autre avec q-learning.

La première approche adoptée est assez classique, où l'IA s'entraîne à l'aide d'une vaste base de données d'entrées/sorties, cherchant à se rapprocher le plus possible de la sortie attendue. Cependant, le défi dans notre cas réside dans l'absence d'une telle base de données pour le jeu de cœurs, nécessitant ainsi sa création. Pour ce faire, nous allons générer plusieurs parties avec des joueurs jouant de manière aléatoire. À chaque itération, nous construirons le vecteur d'entrée et nous le stockerons temporairement dans une variable. Cette variable sera donc une hashmap associant des vecteurs d'entrée à des vecteurs de sortie. Pendant le déroulement d'une partie nous allons enregistrer les cartes jouées dans les vecteurs de sortie seron remplis de zéros, à l'exception de l'élément représentant la carte qui a été jouée qui lui sera temporairement mis à 1. Une fois la partie terminée, nous pourrons changer les vecteurs de sortie et les multiplier par le score du joueur. Par exemple, si un joueur a terminé une partie avec un score de -10 et qu'à un moment donné il avait joué la quatrième carte, le vecteur de sortie correspondant à cette entrée sera [0, 0, 0, -10, 0, ...].

Nous nous doutons que l'utilisation de données d'entraînement provenant uniquement de jeux aléatoires ne conduit pas à un modèle qui joue très bien. Par exemple, il est très improbable qu'un joueur remporte tous les plis lorsque tout le monde joue simplement des cartes au hasard. Le modèle ne parvient donc pas à apprendre à gagner de cette manière.

Il est préférable d'adopter une approche itérative, où la première série d'entraînement utilise simplement des données aléatoires, mais la série suivante de données d'entraînement est produite en faisant jouer le modèle résultant contre lui-même ou contre des joueurs aléatoires.

Algorithme :

1. Générer des données d'entraînement à partir de jeux aléatoires.
2. Entraîner un nouveau modèle avec ces données.
3. Évaluer le nouveau modèle par rapport au modèle actuel le plus performant. Si le nouveau modèle est meilleur, le sauvegarder pour remplacer le meilleur modèle actuel.
4. Arrêter après un nombre défini d'itérations ou s'il n'y a pas eu d'amélioration pendant un certain temps.
5. Générer de nouvelles données d'entraînement à partir de jeux où le modèle actuel joue contre lui-même.
6. Continuer à entraîner le modèle actuel avec les nouvelles données générées pendant quelques époques.
7. Revenir à l'étape 3.

3.3.2.c Entraînement par Q-learning

La deuxième approche pour l'entraînement de notre réseau est d'utiliser les principes de l'algorithme du Qlearning.

D'abord nous identifions les états, actions et récompenses dans notre jeu. Les états sont représentés par notre vecteur d'entrée, les actions sont représentées par les cartes jouables et les récompenses quant à elles sont définies en fin de partie en fonction du score final du playerIA (notre agent).

Notre réseau de neurones est utilisé pour estimer les valeurs Q pour chaque action à partir de l'état actuel. Les valeurs Q estimées sont simplement les sorties du réseau de neurones pour chaque action possible à partir de l'état donné.

L'agent (notre playerIA) interagit avec l'environnement en choisissant des actions à partir d'un état donné, guidé par les valeurs Q prédites par le modèle. Nous collectons également dans notre classe PlayerIA les informations nécessaires, telles que les états observés et les actions prises afin de s'en servir pour la mise à jour du modèle en fin de partie. Cette collecte se fait dans la liste **transitions** (cf.diagramme de classe PlayerIA) à la fin de la méthode **choose_card** où à chaque action prise par le playerIA nous stockons l'état du jeu avant l'action , l'action prise et l'état du jeu après l'action.

À la fin de chaque partie, les poids du modèle sont mis à jour. Cette mise à jour est réalisée dans la méthode **update_model** qui calcule pour chaque action dans la liste "transitions" la valeur Q réelle. La logique du calcul est implementée dans la méthode **compute_real_qvalues**.

En effet, pour chaque action prise par notre agent au cours de la partie, nous utiliserons la formule de mise à jour de la valeur Q de l'algorithme Q-learning pour calculer la valeur Q réelle de cette action, en utilisant la récompense finale et en considérant que l'état suivant est l'état du jeu après que l'action a été prise par l'agent.

Comme cité ci-dessus les valeurs Q predites par le modèle sont les sorties du réseau de neurones, tandis que les valeurs Q réelles sont calculées par cette formule :

$Q(s, a) = r + \gamma \max_a Q(s', a')$

Où :

- r est la récompense obtenue après à la fin de la partie

- γ est le facteur de remise.

- s' est l'état suivant après avoir pris l'action a dans l'état s.

- a' est l'action maximisant la valeur Q dans l'état suivant s'.

La mise à jour du modèle se fera donc en ajustant les poids du réseau de neurones pour minimiser une fonction de perte qui mesure l'écart entre les valeurs Q prédites par le modèle et les valeurs Q réelles calculées. Cette perte sera calculée dans la fonction **compute_loss** . Le calcul de la loss se fera de la manière suivante :

loss = $(Q_{réelle}$ - $Q_{estimée})^2$

Nous obtiendrons une perte pour chaque valeur Q présente dans la liste **transitions**, nous calculerons une moyenne de toutes les pertes obtenues puis appliquerons une retropropagation pour ajuster les poids du réseau de neurones en minimisant la moyenne de pertes calculée.

Ce processus d'interaction avec l'environnement, d'estimation des valeurs Q par le modèle, et de mise à jour du modèle est répété itérativement au fil de l'apprentissage à chaque partie jouée, permettant au modèle d'apprendre à estimer les valeurs Q de manière plus précise au fil du temps.

Bien que cela ne soit pas directement l'algorithme Q-learning et que l'algorithme n'est pas explicitement utilisé pour estimer les valeurs Q à la sortie du réseau de neurones, le processus d'optimisation des poids du réseau de neurones suit les principes de l'algorithme Q-learning en cherchant à minimiser l'erreur entre les valeurs Q prédites et les valeurs Q réelles, permettant en fin d'entraînement, une estimation plus precise des valeurs Q par le réseau de neurones.

3.3.3 Synthèse

Pour développer un réseau de neurones pour jouer au jeu de cœurs, nous avons recodé le jeu de cœurs en Python, en adoptant une approche minimaliste pour faciliter l'entraînement du modèle. L'architecture du jeu a été définie à l'aide d'une classe pour chaque composant du jeu, connectées de manière efficace. Deux méthodes d'encodage des données ont été explorées. La première utilise un vecteur de 12 éléments pour encoder chaque carte tel que la hauteur et la couleur de la carte sont à 1 et le reste à 0, tandis que la deuxième utilise un vecteur de 32 objets, où chaque objet est à 1 ou -1 selon la présence de la carte. Pour entraîner le réseau de neurones, deux stratégies distinctes ont été envisagées. La première approche comprend la création d'une base de données par la génération de parties avec des joueurs aléatoires. Les vecteurs d'entrée sont associés aux vecteurs de sortie en fonction du score des joueurs. Ensuite, l'IA est entraînée sur cette base de données, et une nouvelle base de données est générée avec l'IA entraînée. La deuxième approche quant à elle, repose sur les principes du Qlearning pour minimiser l'erreur entre les valeurs Q prédites et les valeurs Q réelles, permettant la mise à jour des poids du réseau de neurones à chaque fin de partie, assurant ainsi un apprentissage par renforcement sans avoir recours à des données d'entrainement.

3.4 Monte Carlo Tree Search

Dans cette section, nous allons explorer les détails de l'intelligence artificielle basée sur l'algorithme MCTS qui sera utilisée pour affronter les utilisateurs de notre application. Dans un premier temps nous allons décrire le système complet de cette IA en visualisant les différents objets le constituant. Puis, nous regarderons plus particulièrement les classes liées au fonctionnement de l'algorithme Monte Carlo Tree Search. Finalement nous verrons les différentes possibilités concernant l'entraînement de l'IA ainsi que l'élaboration de différents niveaux de performance/difficulté.

3.4.1 Système de l'IA

3.4.1.a Implémentation du jeu des cœurs

De manière similaire au réseau de neurones, notre choix a été de disposer de la logique du jeu des cœurs dans le système de notre IA à base de MCTS. Nous avons donc codé notre propre version du jeu des cœurs, avec ses règles spécifiques au sein de l'algorithme MCTS.

La partie du système correspondant à cette tâche a été détaillée précédemment.

3.4.1.b Implémentation du Monte Carlo Tree Search

À partir de l'implémentation du jeu des cœurs que nous avons présentée plus tôt, nous pouvons maintenant décrire les joueurs qui vont constituer une partie pour l'IA MCTS.

Ainsi, la classe **Player** détaille leur comportement avec plusieurs attributs : un nom, un score et une liste de cartes, sa main. Elle possède plusieurs méthodes reliées à cette main : *addHand(Card c)* qui ajoute une carte depuis le paquet à la main du joueur, *clearHand()* et *hasSuit(Suit s)* qui

vérifie si la main possède une carte d'une couleur donnée. Ensuite, les méthodes *addPoints(int p)* et *getPoints()* permettent le suivi des scores et leur mise à jour.

Enfin, la méthode *doAction(State s)* est certainement la plus importante. Elle est responsable de la prise de décision de jouer une carte. Elle prend en paramètre l'état de la partie. Nous détaillerons cet état plus tard mais, en attendant, nous pouvons considérer un état comme une combinaison d'informations permettant de décrire une partie à un instant donné.

Cette méthode est abstraite et par conséquence la classe **Player** également. Cette méthode est celle qui détermine toute la stratégie d'un joueur, la façon dont il prend une décision et représente donc la différence entre un joueur jouant aléatoirement et un joueur jouant à partir d'un algorithme MCTS par exemple. La classe **Player** défini donc le comportement général d'un joueur du jeu des cœurs pour cette IA, quelle que soit sa façon de jouer.

On peut maintenant créer une classe **MCTSPlayer** qui représente le joueur jouant à partir de l'algorithme MCTS. Comme nous l'avons expliqué dans notre rapport de spécification, cet algorithme se décompose en 4 étapes : la sélection, l'expansion, la simulation et la backpropagation. Le choix a été fait d'ajouter un attribut à ce joueur, *simulationHand*, qui nous permet de ne pas manipuler/modifier directement la main du joueur mais une copie de celle-ci lors de l'algorithme et plus particulièrement lors de l'étape de simulation. Il est également composé d'un noeud qui est la racine de l'arbre MCTS où chacun des noeuds représente une action possible caractérisée par un état unique.

En plus de toutes les méthodes d'un joueur de base, il faut donc ajouter celles permettant le déroulement de l'algorithme : *selection(Node root)*, *expansion(Node parent, int childNum)*, *simulation(Node start)* et *backProp(Node start, int reward)*. Ces 4 méthodes définissent chacune le fonctionnement d'une étape de l'algorithme. *selection* rend ainsi un noeud à étendre en parcourant l'arbre à partir de la racine. Cette méthode sélectionne successivement le meilleur enfant de chaque noeud jusqu'à l'obtention d'une feuille à l'aide d'une autre méthode : *bestChildUCT(Node parent, double weight)*. Celle-ci rend le meilleur enfant à visiter lors de la sélection et est basée sur une fonction : **UCT** (Upper Confidence applied to Trees). Cette formule répond à la difficulté qui est de trouver un équilibre entre exploitation et exploration :

$$UCT = \frac{recompense}{visites} + c \times \sqrt{\frac{\ln(visitesparent)}{visites}}$$

Elle est ainsi basée sur le nombre de visites du noeud (*visites*) et de son noeud parent (*visitesparent*), la récompense du noeud (*recompense*) et d'un paramètre c qui représente le poids (paramètre *weight*) de l'exploration.

Après avoir sélectionné le noeud à étendre, la méthode *expansion* ajoute un noeud à l'arbre MCTS à savoir un enfant au noeud sélectionné.

À cet instant, l'arbre a été étendu et nous allons maintenant réaliser une simulation. Cette étape consiste à, en partant d'un noeud donné, terminer la partie en jouant des coups aléatoires pour chacun des joueurs. Cette méthode *simulation* rend le score final du joueur après la fin de cette simulation aléatoire.

Finalement, une fois la simulation effectuée on rétro-propage le score final du joueur à chacun des noeuds en partant de celui ajouté à l'arbre précédemment jusq'à la racine. Après cette étape, une itération complète de l'algorithme MCTS a été effectuée. La méthode *runMCTS(State origin)*,

qui prend en paramètre l'état actuel de la partie centralise l'utilisation de ces méthodes pour effectuer un certain nombre d'itérations de l'algortihme. Elle représente le squelette de l'algorithme MCTS dans son ensemble et rend finalement le numéro du meilleur enfant à l'aide de la méthode *bestRewardChild(Node root)*. Cette dernière parcourt les différents noeuds et sélectionne celui avec la meilleure récompense.

La méthode qui caractérise chaque joueur, *doAction*, joue donc la carte en basant sa décision sur la méthode *runMCTS*.

Comme nous pouvons le remarquer, le fonctionnement de ce **MCTSPlayer** est fondamentalement basé sur la manipulation d'un arbre des actions possibles. Cet arbre est composé de noeuds qui représente chacun un état possible de la partie. Le diagramme de classes suivant nous permet de visualiser la structure du Monte Carlo Tree Search :

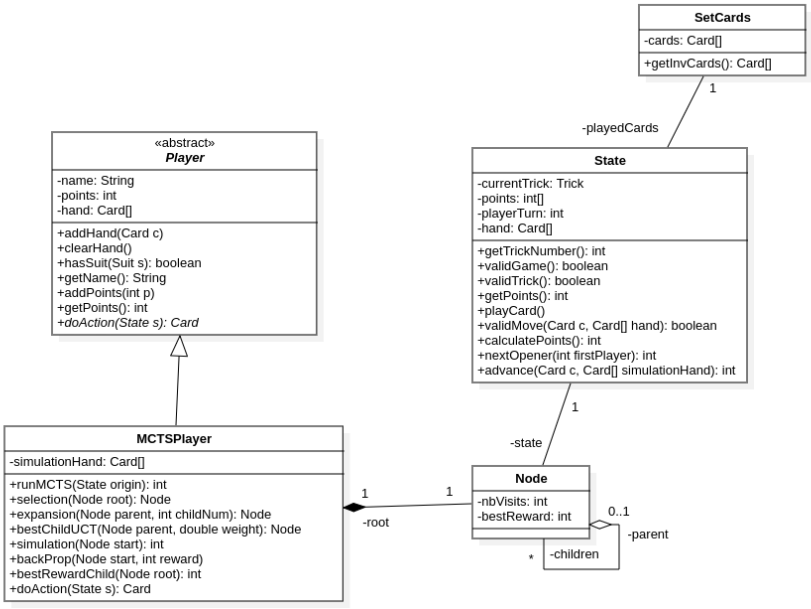

FIGURE 7 – Diagramme de classes du jeu des cœurs

La classe **Node** est composée d'un état, de deux attributs cruciaux pour l'utilisation d'UCT : le nombre de visites et la récompense, et enfin d'un noeud parent et de ses noeuds enfants.

La classe **State** est elle essentiellement basée sur les entrées de notre IA, tous ses attributs caractérisant la situation de la partie à un instant donné. Ainsi l'attribut *points*, un tableau des scores de tous les joueurs renferme l'information sur le nombre de cœurs remportés depuis le début de la partie et *a fortiori* par chacun des joueurs. L'attribut *playedCards*, un **SetCards** (objet composée de cartes), permet de connaître à la fois toutes les cartes déjà jouées et celles restantes mais également l'information sur le numéro du pli en cours. Justement, *currentTrick* représente les cartes qui sont actuellement sur la "table" dont on peut donc tirer la couleur du pli (celle de la première carte du pli) et *playerTurn* l'ordre du joueur dans le pli. Enfin l'état est composé de la main du joueur.

Les méthodes *getTrickNumber()*, *validGame()*, *validTrick()* et *getPoints()* permettent tout simplement de récupérer les informations sur le numéro du pli en cours, le score du joueur et de vérifier

si le/la pli/partie est terminé(e) ou non. La méthode *playCard(Card c)* permet de jouer une carte donnée, sans oublier de mettre à jour l'attribut *playedCards* et par conséquent l'information sur les cartes qui restent à jouer également. La méthode *validMove(Card c, Card[] hand)* vérifie si une carte donnée peut être jouée par rapport au pli en cours. Globalement on vérifie que la carte est soit de la couleur du pli, soit dans le cas contraire que le joueur n'avait pas une autre carte de la couleur du pli dans sa main. La méthode *calculatePoints()* calcule la valeur du pli en cours en faisant la somme des valeurs de chaque carte le composant (-5 pour les cœurs et 0 pour toutes les autres couleurs). Quant à*nextOpener(int firstPlayer)*, elle détermine l'ouvreur du pli suivant en cherchant le joueur qui a posée la carte de la couleur du pli courant avec la hauteur la plus élevée (7 étant la hauteur la plus basse et l'as la plus haute). La dernière méthode, *advance(Card c, Card[] simulationHand)*, est très importante pour le bon fonctionnement de l'algorithme MCTS. Plus tôt, nous avons indiqué que lorsque l'on étend l'arbre, on doit mettre à jour l'état du noeud auquel on ajoute un enfant pour définir l'état de ce nouveau noeud une fois que l'action est faite (qu'une carte est jouée). Cet état mis à jour correspond donc à l'état d'origine à partir duquel on joue la carte choisie puis on effectue un tour de table pour revenir à un moment de décision du joueur. C'est ce que fait cette méthode *advance*, elle fait avancer la partie le plus possible jusqu'à une nouvelle situation dans laquelle le joueur doit prendre une décision. Elle retourne ainsi le nombre de points que le joueur a obtenu dans ce processus (si le joueur a remporté le pli terminé pendant cette avancée il a potentiellement récupéré des cœurs par exemple).

3.4.2 Entraînement du MCTS

3.4.2.a Stratégies possibles

L'architecture de l'intelligence artificielle étant définie, il reste à entraîner notre MCTS en générant des parties. Une première possibilité serait de faire jouer plusieurs **MCTSPlayer** les uns contre les autres. La principale limite à cette possibilité serait le temps de décision multiplié par la présence de plusieurs intelligence artificielle, chose qui reste à mesurer et ne sera potentiellement pas un problème. Dans le cas contraire, une solution pour réduire le temps de décision serait de poser une limite à notre algorithme MCTS lors de la phase de sélection : fixer une profondeur de l'arbre limite jusqu'à laquelle on étend avant de réaliser des simulations. Cette solution pourrait néanmoins créer une nouvelle problématique : trouver un équilibre entre temps de décision et précision/qualité de décision. Car limiter la profondeur jusqu'à laquelle on étend notre arbre MCTS limite par la même occasion la qualité de la décision prise par l'algorithme. Cette problématique sera au coeur de réflexions au sein de notre équipe lors du développement de notre IA MCTS.

Sinon, une autre possibilité serait de définir un nouveau type de joueur : **RandomPlayer**. Nous entraînons alors un **MCTSPlayer** contre d'autres **RandomPlayer** qui prennent leurs décisions de façon aléatoire.

3.4.2.b Niveaux de difficulté/maîtrise de l'intelligence artificielle

Lors de la phase d'entraînement, après avoir déterminé la méthode optimale parmi celles présentées précédemment, il existe différents paramètres avec lesquels nous pouvons "jouer" pour aboutir à différents niveaux d'intelligence artificielle.

Tout d'abord, l'algorithme MCTS est fondamentalement dépendant d'un paramètre : le nombre d'itérations de l'algorithme à réaliser avant de prendre la décision. Donc plus le nombre de d'itérations augmente, meilleure sera la décision car le nombre de simulations augmente. Mais, à l'instar de la

profondeur limite, il faut trouver un équilibre entre temps de décision et qualité car nous ne pouvons pas simplement fixer le nombre d'itérations le plus élevé possible.

Un autre paramètre plus spécifique à notre algorithme MCTS est le paramètre *weight* de la méthode *bestChildUCT*. Ce paramètre correspond au poids d'exploration utilisé dans la formule UCT. Théoriquement ce paramètre vaut racine de 2 mais sa valeur peut être modifiée et détermine l'importance que l'on souhaite accorder à l'exploration, i.e, aux coups avec peu de simulations. Plus la valeur de ce paramètre est élevée plus on donne d'importance à ces coups et inversement.

Ainsi, en jouant avec ces deux paramètres nous pouvons essayer plusieurs combinaisons : un MCTS avec peu d'itérations et favorisant l'exploration ou un autre avec un nombre élevé d'itérations favorisant l'exploitation... dans le but d'aboutir à plusieurs IA basée sur un algorithme MCTS commun mais avec une qualité de prise de décision différente pour offrir une expérience riche à l'utilisateur de notre application.

3.4.3 Synthèse

Lors du développement d'un algorithme MCTS pour jouer à notre version du jeu des cœurs, nous avons d'abord décomposé le système en deux parties, l'une étant consacrée à l'implémentation de la logique du jeu en elle-même et l'autre à l'algorithme spécifiquement. Un état représentant la façon dont on caractérise une partie à un instant donné a été défini avec un nombre d'informations essentielles jugé minimal pour faciliter l'implémentation dans un premier temps. Pour l'entraînement de notre algorithme, deux méthodes ont été détaillées et seront explorées chacune en profondeur pour déterminer celle la plus efficace. La première consiste à entraîner des joueurs prenant leur décision à partir de l'algorithme MCTS les uns contre les autres tandis que la seconde vise à entraîner un seul joueur dit "MCTSPlayer" contre d'autres joueurs prenant leurs décisions de façon aléatoire. Ensuite, deux paramètres influant sur la performance de l'algorithme ont été identifiés et feront l'objet de différents tests pour créer finalement différents niveaux de difficultés/maîtrise de l'intelligence artificielle.

4 Conclusion

La phase de conception du projet a joué un rôle important dans l'établissement des bases du développement de l'application mobile dédiée au jeu de cartes "Cœurs". Les objectifs spécifiés lors de la pré-étude ont été clairement définis, mettant en avant trois points essentiels : le développement de l'application sur Flutter, la création d'une intelligence artificielle basée sur un réseau de neurones ou sur l'algorithme MCTS, et l'implémentation de la communication client/serveur entre les IA et l'application.

Dans la première partie du rapport, nous avons exposé la conception de l'application en résumant son fonctionnement global, présentant les différentes vues de l'interface graphique, et détaillant l'architecture à l'aide d'un diagramme de classes. La section a également abordé en détail la communication client/serveur avec Flask, en spécifiant la structure des requêtes.

La deuxième partie du rapport a été consacrée à la conception du réseau de neurones. Nous avons commencé par expliquer les outils nécessaires à son implémentation, détaillant le code du jeu "Cœurs" recodé en Python. Nous avons ensuite présenté l'architecture du réseau de neurones, spécifiant sa structure, ses entrées/sorties, ses couches cachées, ainsi que les deux méthodes d'encodage de l'état du jeu. Les deux approches d'entraînement du réseau, le Q learning et la génération d'une base de données, ont été définies pour être testées ultérieurement.

Enfin, nous avons détaillé la deuxième intelligence artificielle que nous allons développer, basée sur l'algorithme Monte Carlo Tree Search (MCTS). De la même manière que pour le réseau de neurones, nous avons commencé par expliquer les outils utilisés pour cette IA. Ensuite, nous avons détaillé la mise en œuvre du jeu pour pouvoir développer l'algorithme MCTS. Nous avons présenté en détail le déroulement de cet algorithme, avec les formules qu'il utilise et les étapes qu'il enchaîne, pour finir par expliquer la manière dont cette IA va être entraînée.

Avec ce rapport, nous avons conclu la phase de conception de notre projet. Les architectures des composantes sont clairement définies, et les étapes à suivre pour les développer sont également claires. Nous sommes donc prêts à entamer la dernière phase, qui est celle de l'implémentation.