UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLLIARD

Rapport de Projet

IA41 – Automne 2020

DUPAYRAT Antoine
CARON Allane
MOYSE Antoine
Informatique 1

Sujet :

Love Letter

Enseignants :

LAURI Fabrice
RUICHEK Yassine

utbm
université de technologie
Belfort-Montbéliard

Table des matières

1 - Présentation du projet

L'objectif du projet est de fournir un jeu Love Letter où le joueur joue contre une IA fonctionnelle. Nous l'avons choisi car celui-ci représentait un certain challenge pour nous, tout en étant réalisable sans expérience préalable en IA.

Love Letter est un jeu en tour par tour à deux joueurs. Ce jeu est à information parfaites mais non complètes. Le fait qu'il soit à informations non complète le distingue des projets plus simples d'intelligence artificielle, dans lesquels il est plus facile de déterminer un algorithme décidant des meilleures actions à effectuer.

Nous avons codé ce projet en Python, en utilisant la librairie Tkinter pour l'interface graphique. Pour l'IA, plusieurs solutions ont été abordés que nous voulions toutes implémentées sous formes de difficultés différentes. Nous n'avons finalement implémenté qu'un alpha beta.

1.1 - Architecture du projet

Nous avons utilisé le MVC pattern afin d'avoir un programme plus modulaire et plus propre en séparant la GUI des données relatives au déroulement du jeu.

Diagramme de classe : voir annexe 1.

Ainsi, les classes sous le répertoire model vont contenir toutes les informations relatives au jeu comme les données des joueurs, celles des cartes, l'état actuel du jeu… Les classes contenues dans le répertoire view serviront à afficher la GUI. Enfin, le controller sert à faire le lien entre le model et la view afin que ceux-ci ne communiquent jamais directement entre eux, ce qui les rend indépendants l'un de l'autre. En d'autres termes, le controller va s'occuper de mettre à jour la view et le model en accord avec les évènements déclenchés en interne par le model, ou par les inputs de l'utilisateur.

1.2 - La GUI (le répertoire view)

Comme celle-ci n'est pas l'objectif du projet, nous ne nous attarderons pas sur son implémentation et sa présentation sera brève.

Celle-ci est composée de trois scènes de jeu différentes : celle du menu, celle de jeu, et celle de fin de jeu. En effet, lorsque l'on lance l'application, nous avons la possibilité d'effectuer plusieurs actions avant de lancer la partie (regarder les règles par exemple). Une fois la partie lancée, la scène du jeu apparaît. Celle-ci est la scène principale dans laquelle chaque round se déroule (une partie étant composée de plusieurs rounds) :

Plusieurs informations y sont indiquées, comme la dernière carte à avoir été jouée, le nombre de points de chaque joueur ainsi que les 3 cartes dévoilées en début de round.

Il est également possible de consulter les cartes jouées durant le round par chaque joueur, ainsi qu'avoir un rappel de l'effet de chaque carte du jeu.

Une fois le round terminé, le menu de fin de jeu apparaît et indique le gagnant et son nombre de points pour finalement proposer de passer au round suivant (si la partie n'est pas terminée, i.e les joueurs ont moins de 6 points) ou de revenir au menu.

Ces scènes sont manipulées au moyen des classes MenuScene, GameScene et EndGameScene respectivement. La classe View permet de faire le lien entre ces trois scènes, elle constitue également la fenêtre du jeu.

La dernière classe du module view est utilisée chaque fois qu'il est nécessaire d'afficher une fenêtre d'information par-dessus la scène de jeu. Par exemple, lorsque l'on clique sur le bouton de rappel des cartes ou de consultation des cartes jouées, c'est cette classe qui va s'occuper d'afficher une nouvelle fenêtre avec les informations qui nous intéresse. Il en est de même pour tout évènement provoquant l'apparition d'une fenêtre en jeu (choix du camp lorsque le prince est joué par exemple…).

1.3 - Implémentation du jeu et de ses règles (le répertoire model)

Comme indiqué précédemment, le répertoire model contient toutes les informations relatives au déroulement du jeu, indépendantes de la GUI. Pour plus de clarté, nous avons décidé de séparer ces informations de telle manière :

-Un module (cards) pour gérer les cartes

-Un module (player) pour gérer les joueurs

-Un module (model) pour gérer les interactions entre éléments du model et le déroulement du jeu. Le module model est équivalent à un game manager.

Les cartes :

Il existe une classe par carte, toutes dérivant de la classe abstraite Card, classe qui oblige chacune de ses classes filles à avoir une propriété indiquant la valeur de la carte et une méthode correspondante à l'action de la carte. Cette classe comporte les éléments communs à toutes les cartes du jeu.

Certaines classes dérivent de la classe TwoActionCards (elle-même dérivant de la classe Card). Ces cartes correspondent aux cartes se déroulant en deux temps, par exemple le garde : on clique sur le garde, et ensuite un choix doit être fait. Il ne suffit pas de choisir la carte pour qu'elle soit jouée.

Cette classe abstraite permet simplement d'obliger ces classes filles à implémenter une méthode supplémentaire correspondante à la deuxième action. Elle permet aussi de différencier ces cartes des autres ce qui permet plus de clarté.

Les méthodes action et deuxième_action sont l'implémentation des effets des cartes sur le jeu.

Exemple du code de la carte Servante :

```
class Servante(Card):
    '''
    Classe définissant la carte servante
    '''
    def __str__(self):
        return "Servante"

    def __init__(self, model):
        Card.__init__(self, model)

    @classmethod
    def value(cls):
        return 4


    def action(self):
        Card.action(self)
        self._model.current_player.immune = True
```

Les joueurs :

Il existe une classe Player dont tout joueur dérive. Cette classe va définir les attributs, propriétés et méthodes communes à l'IA et à l'utilisateur. Autrement dit, la majeure partie du code se trouve dans cette classe, les différences entre le vrai joueur et l'IA n'étant que la manière de décider quelle carte jouer, ainsi que leurs noms.

Exemples de propriétés : -immune : true si le joueur a joué une servante, false sinon

-score : score du joueur

Exemples de méthodes : -add_card(card) : ajoute une carte à la main du joueur

-must_play_comtesse() : vérifie si le joueur est obligé de jouer la comtesse ou non

La classe RealPlayer correspond à la classe de l'utilisateur. Elle ne contient aucun code et ne sert finalement qu'à différencier le vrai joueur de l'ia via son instance.

La classe IA est une classe abstraite dont va dériver les différentes difficultés possibles de l'ia (une classe par difficulté), elle comporte une méthode abstraite correspondant à l'algorithme à utiliser notamment.

Les joueurs sont chacun stockés dans une liste circulairement chaînée, utilisée pour la gestion des tours. Ainsi, pour accéder au joueur suivant le joueur courant, il suffit d'écrire current_player.next.

Le module model :

Contient les attributs, propriétés et méthodes nécessaires au déroulement du jeu. Il instancie les cartes et les joueurs au début de chaque partie (à ne pas confondre avec les rounds) et répartie les cartes dans différentes autres listes permettant le fonctionnement du jeu au début de chaque round. Il se charge de mélanger et distribuer les cartes au début du round, de gérer la pioche, les tours, les joueurs et les différentes listes de cartes constituant le jeu.

Attributs du model :

_controller : permet l'accès aux méthodes de mises à jour du controller

_cards = liste des 21 cartes du jeu. Il n'existe pas d'autre instance de cartes que celles contenues dans cette liste. Ces cartes ne sont instanciées qu'une fois, lors du lancement de l'application.

Les variables que nous allons maintenant présenter ne contiennent pas des instances de cartes, mais des références vers les instances de cartes contenues dans la liste _cards.

_cards_played : liste de toutes les cartes ayant été jouées pendant le round.

_burnt_card : carte sélectionnée en début de round pour être « brulée »

_deck : liste de cartes représentant la pioche.

_cartes_defaussees : liste des cartes défaussées du round (lorsque le prince a été joué).

_players_list : liste circulairement chaînée des joueurs.

_victory : true lorsque la dernière carte jouée à permis au joueur de gagner, utilisée pour stopper le déroulement du jeu.

_issimul : nous verrons l'utilité de cette variable plus tard, elle est utilisée lors de l'algorithme alpha beta.

_islearning : utilisé lors de l'apprentissage par renforcement.

1.4 - Déroulement d'un tour

Lorsque le programme attend que l'un des joueurs joue, cela se passe au niveau du Controller dans la fonction start_turn. Cette fonction va permettre de mettre à jour la GUI en fonction des évènements du dernier tour (nombre de cartes affichés pour chaque joueur, dernière carte jouée..), et à faire la

distinction entre le tour de l'IA et le tour du joueur. Si le joueur courant est le vrai joueur, alors il faut attendre que le joueur clique sur une de ses cartes pour mettre à jour les informations contenues dans le model/la GUI, si c'est à l'IA, alors il faut directement mettre à jour la GUI et lancer l'algorithme permettant à l'IA de choisir sa carte.

De manière plus précise, voici le déroulement du tour dans le cas de l'utilisateur :

La fonction start turn va détecter que le joueur courant est l'utilisateur, le programme va alors attendre un input, le clic sur une des cartes va lancer la fonction card_played du controller qui va elle-même lancer la fonction play du model afin de mettre à jour les données du jeu, pour finalement passer au tour suivant.

Dans le cas de l'IA, le traitement est similaire mais il n'y aura pas d'attente d'input. La fonction start_turn va directement lancer la fonction card_playedAI du controller (équivalent à la fonction card_played, lancée lorsque l'utilisateur clique sur une de ses cartes) qui va mettre à jour la GUI en insérant une pause de 3 secondes à l'écran afin de laisser le temps à l'utilisateur de voir l'action de l'IA, puis lancer la fonction playAI du model permettant la mise à jour des données. Pour finalement passer au tour suivant.

On remarque ici le travail du MVC qui permet au controller de mettre à jour la GUI et les données du jeu en fonction des évènements du jeu.

Revenons maintenant sur la manière dont sont mises à jour les données. Lorsque l'utlisateur va choisir sa carte, la fonction play du model sera lancée avec en paramètre l'index de la carte à jouer (dans la liste des cartes du joueur), la fonction va alors effectuer un ensemble de traitement permettant de :

- Retirer la carte du jeu de l'utilisateur
- Effectuer l'action associée à cette même carte
- Mettre à jour les données des joueurs (retirer l'immunité de l'IA si celle-ci a joué la servante par exemple)
- Changer de joueur courant
- Faire piocher une carte au nouveau joueur courant

Pour l'IA le traitement est similaire, la fonction playAI du model est simplement appelée avant la fonction play afin que celle-ci appelle l'algorithme de l'IA lui permettant de choisir la carte à jouer.

Code de la fonction play : (ne pas prendre en compte le « if », utilisé dans le cas de l'apprentissage par renforcement).

```
#Choix de la carte jouée par l'IA
def playAI(self):
    #Si l'ia possède une comtesse et un prince ou un roi, alors pas besoin de lancer la simulation, la comtesse est jouée
    play_comtesse = self.current_player.must_play_comtesse()
    if play_comtesse != -1:
        self.play(play_comtesse)
    else:
        index = self.ia.algorithme()
        #self.play(randrange(0,2))
        self.play(index)


#Effectue l'action de la carte à l'index associée du joueur courrant
def play(self, index):

    last_card_played = self.current_player.cards[index]
    self.current_player.add_cards_played(last_card_played)

    self.current_player.remove_card(last_card_played)#Suppression de la carte dans la main du joueur courrant
    self._cards_played.append(last_card_played)#Ajout de cette carte à la liste des cartes jouées
    last_card_played.action()#Action de la carte
    if(not self._victorylearning):
        self.next_player.immune = False
        self.current_player.last_card_played = last_card_played

        self.next_turn()
    else:
        self._victorylearning = False
```

2 - L'IA

Notre but était d'implémenter différents algorithmes d'IA afin de comparer les résultats et performances obtenus avec chacun. Pour ce faire, nous avons étudiés plusieurs solutions possibles. Ces solutions étaient l'algorithme IDDFS, l'élagage alpha beta et l'apprentissage par renforcement. Dans leur approche, les deux premières solutions se ressemblent. En effet, un algorithme min max peut se rapporter à un algorithme DFS auquel on aurait ajouté une heuristique (donnée par la fonction éval).

Cependant nous devions répondre à plusieurs problèmes pour ces deux solutions :

2.1 - IDDFS et alpha beta

2.1.1 - La représentation d'un état

Dans les cas de l'IDDFS et de l'alpha beta, il est nécessaire de définir ce qu'est un état, et comment le coder. Nous sommes vite arrivés à la conclusion qu'un état correspondait à l'ensemble des données du jeu à instant t. Autrement dit, la pioche, les cartes de chaque joueur, les cartes précédemment jouées, les données de chaque joueur… Toutes ces données correspondent à un état. Un état du jeu correspond donc à l'ensemble des attributs de la classe Model à un instant t (cette classe ayant comme attributs toutes les listes de cartes, la liste circulaire des joueurs etc.. Pour rappel, cette classe fait office de game manager « interne » s'occupant du bon déroulement du jeu du point de vue des données (autrement dit, il ne s'occupe pas de la partie GUI)).

Cette représentation d'un état est partiellement fausse, car elle ne prend pas en compte la dimension non complète des informations du jeu. En effet, du point de vu des joueurs, des informations manquent. Par exemple, chaque joueur ne sait pas quelle est la carte qui a été brûlée en début de round, chaque joueur ne connaît pas non plus le jeu de l'autre, ni la carte qu'il va piocher.

Afin d'illustrer ce problème, prenons comme état courant un état où le joueur s'apprête à jouer. Il a donc deux cartes en mains. Instinctivement, nous pourrions penser que seulement deux états peuvent découler de son action (un par carte jouée). Si le jeu était à information complète, alors ça serait vrai. Cependant la recherche des états successeurs doit prendre en compte le point de vue du joueur, car celui-ci ne dispose pas de toutes les informations. Par exemple, il ne sait pas quelle carte a été brûlée en début de jeu, il ne sait pas non plus quelle carte possède son adversaire.

Ainsi, certaines variables doivent être ajoutées/modifiées afin de prendre en compte cette dimension du jeu :

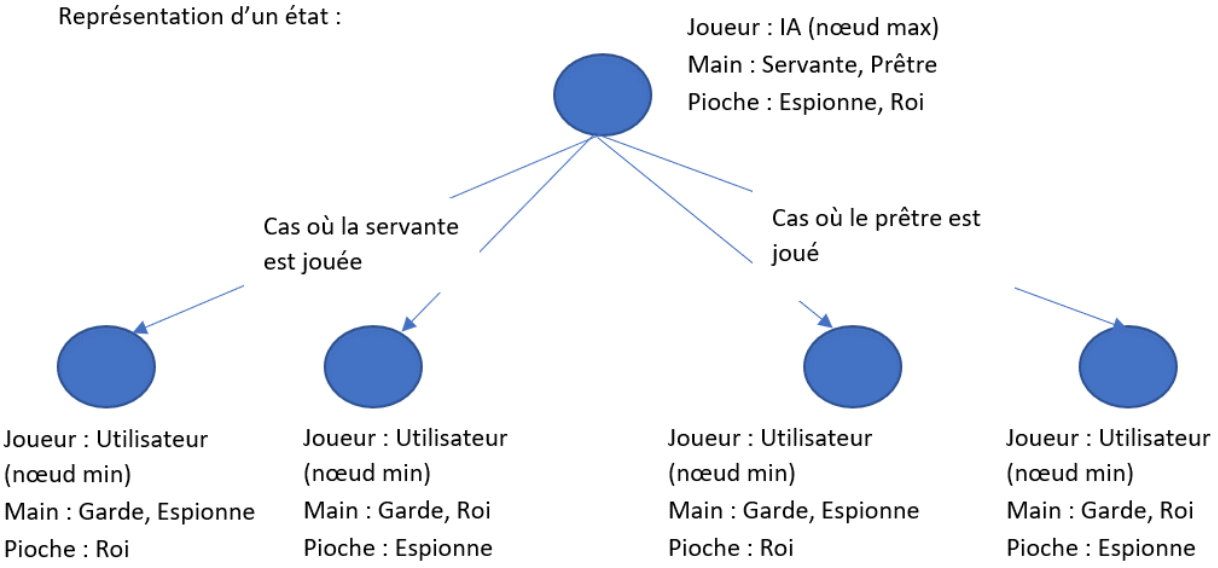
- La carte brûlée doit être ajoutée à la pioche, car du point de vue du joueur courant, celle-ci est théoriquement piochable puisqu'il ne la connaît pas. Ne pas l'inclure dans la pioche reviendrait à omettre une possibilité du point de vue du joueur.
- Ajout d'une variable correspondant à la liste des cartes piochables par l'adversaire, auxquelles sont associées une probabilité d'occurrence (nom de la variable et méthode permettant de la calculer : drawable_cards et get_drawable_cards)

Code de get_drawable_cards :

```
    #Fonction permettant de définir les cartes pouvant être piochée ainsi que leur probabilité d'être piochée
    def get_drawable_cards(self):
        drawable_cards = {}
        for card in self._deck:
            if(not any(isinstance(x, card.__class__) for x in drawable_cards)): #Vérifie si la carte possède déjà une instance dans la liste
                proba = sum(isinstance(x, card.__class__) for x in self._deck)/self._deck.__len__() #Calcul du pourcentage associé
                drawable_cards[card] = proba

            #Si la liste contient 10 cartes, elle contient tous les types de carte possible du jeu, on sort donc de la boucle
            if(drawable_cards.__len__() == 10):
                break
        return {k:v for k,v in sorted(drawable_cards.items(), key = lambda item:item[1], reverse = True)} #Tri du dictionnaire en fonction de ses valeurs (décroissant)
        #return drawable_cards
```

De plus, cela implique que d'un état peut découler autant d'état qu'il existe de cartes piochable pour le joueur adverse. En effet, du point de vue du joueur courant, les états qu'il engendre en jouant une carte sont multiples par la nature incomplète des informations qu'il possède du jeu.



Ici, lorsque le joueur joue la servante, il est possible que le joueur adverse ait pioché une espionne ou un roi. Le joueur courant ne peut pas savoir quelle carte a été piochée avec certitude. Voilà déjà deux états successeurs lorsque le joueur courant joue la servante. Il en existe également deux autres engendrés lorsque le joueur joue son prêtre. Ce qui nous donne 4 états successeurs (2 par cartes qu'il est possible de jouer).

Il est alors possible d'avoir un maximum de 20 états successeurs d'un état donné (dans le cadre où il est possible que l'adversaire pioche n'importe quelle carte du jeu).

Pour résumer : un état représente les informations auxquelles le joueur courant a accès, c'est-à-dire ses propres informations, les informations concernant le joueur adverse (sauf sa carte), les cartes qui ont été jouées, les cartes que le joueur adverse peut piocher, les cartes que le joueur adverse peut posséder

Cependant certaines cartes peuvent ajouter un certain nombre d'états « intermédiaires » :

Lorsque le prince est joué, il est possible de choisir de défausser le camp adverse ou son propre camp, cela devrait théoriquement doubler le nombre d'états successeurs lorsque l'on joue un prince. C'est également le cas lorsqu'un chancelier est joué (il est possible de piocher des combinaisons de carte différentes).

Nous avons fait le choix de ne pas prendre en compte ces états dans notre arbre de recherche. L'avantage de les prendre en compte est une meilleure vision à long terme des conséquences des actions de ces cartes (car on explore toutes les possibilités). Cependant la probabilité d'apparition de la majorité de ces états étant extrêmement faible (voir infime lorsque l'on commence à explorer les états successeurs des états successeurs), que nous avons préféré faire apparaître l'état le plus probable dans ces situations.

Cet allègement de l'arbre de jeu permet d'explorer des états à une profondeur plus élevée, qui auront finalement une importance plus forte que d'explorer toutes les possibilités de cartes dans le cadre où l'on joue le chancelier par exemple.

Pour simplifier, il est préférable pour l'IA de voir plusieurs coups à l'avance dans un arbre de jeu restreint, plutôt que d'explorer l'entièreté de l'arbre de jeu à une profondeur moins grande.

2.1.2 - Le problème de la simulation

Nous avons également rencontré un problème lors de l'implémentation de l'IA, qui est de l'ordre du code plutôt que de la théorie et de l'algorithmie.

Que ce soit pour l'IDDFS ou l'alpha beta, le problème de la simulation s'est posé. En effet, lorsqu'un algorithme est lancé, celui-ci va devoir simuler le jeu jusqu'à une certaine profondeur. Or cette simulation ne doit pas apparaître à l'œil de l'utilisateur, il faut également que les données ne subissent pas de modification à terme, le cours du jeu ne doit pas être modifié une fois l'algorithme terminé. Enfin, chaque état crée doit subsister tout au long de la simulation. Autrement dit, il doit être possible d'accéder depuis un état aux données de l'état parent. Le fait d'avoir générer un nouvel état en ayant simuler un tour de jeu ne doit pas perturber les données de l'état parent. Pour ce faire, nous avons utilisé une classe Save qui va utiliser la méthode deepcopy de la librairie copy afin de faire une copie du model à chaque nouvel état (ainsi, les références manipulées d'un état à un autre ne sont pas les mêmes). A chaque création d'état, une copie et faite automatiquement, cette copie sera manipulée lors de la génération des états successeurs de cet état puis finalement réinitialisée à chaque fois que les données ont été modifiées. (Grâce à la fonction backup). L'inconvénient de l'utilisation de deepcopy est que l'on effectue une copie d'un grand nombre d'éléments du jeu, ce qui n'est pas très optimisé, cependant nous n'avons pas réussi à mieux implémenter ce système de sauvegarde. Un command pattern aurait pu servir à annuler le dernier état généré jusqu'à ce que l'on revienne à l'état d'origine. Compte tenu du fait que l'IDDFS et l'alpha beta effectuent des parcours en profondeur, l'implémentation de ce pattern aurait pu amener à une meilleure optimisation des ressources.

Voici le code la classe Save :

```
    '''
    Classe permettant de sauvegarder l'état courant du jeu lors d'une simulation. Cette manière de faire est loin d'être la meilleure, implémenter un command
    pattern pour la simulation aurait probablement été plus bénéfique, bien que rendant l'architecture des fichiers moins lisible
    '''
    def __init__(self, model):
        #Création d'un environnement propre à l'état courant, afin de ne pas manipuler l'environnement hors de l'état courant
        self._model = copy.deepcopy(model)

        self._current_player = self._model.current_player
        self._next_player = self._model.next_player

        #Sauvegarde des éléments de l'environnement
        self._modelsave = self._model.save_attributes()
        self._current_player_save = self._model.current_player.save_attributes()
        self._next_player_save = self._model.next_player.save_attributes()

    def get_model(self):
        return self._model

    #Restauration de l'environnement
    def backup(self):

        self._current_player.set_attributes(self._current_player_save)
        self._next_player.set_attributes(self._next_player_save)
        self._model.set_attributes(self._modelsave)

        #La restauration ne se fait que sur les listes de carte, il faut revenir au joueur courant manuellement
        self._model.players_list.next_turn()
```

Un autre problème est également survenu du fait de la simulation. En effet, nous voulons effectuer un traitement similaire à celui lors du déroulement normal du jeu, mais nous voulons tout de même que ce traitement soit légèrement différent. C'est pourquoi le model contient un attribut is_simul qui permet d'empêcher les affichages de la GUI et d'effectuer des exceptions lors de la simulation.

Par exemple, la pioche contenant une carte en plus, la condition de victoire par pioche vide est différente en simulation. En effet, celle-ci apparaît lorsqu'il ne reste plus qu'une carte dans la pioche lors de la simulation. Nous voulons également contrôler quelle carte est piochée par l'adversaire etc..

2.1.3 - Le codage des états successeurs

La résolution des problèmes présentés nous permet finalement de générer les états successeurs d'un état quelconque, voici les méthodes utilisées :

```
    def next_states(self):

        drawable_cards = self._model.get_drawable_cards()
        states = []

        #Vérifie l'obligation de jouer la comtesse ou non
        play_comtesse = self._current_player.must_play_comtesse()
        if play_comtesse != -1:
            self.play_simu(states, drawable_cards, play_comtesse)
        else:
            #On boucle sur les cartes du joueur courant, pour chaque carte, on boucle sur toutes les cartes possibles que le prochain joueur peut piocher
            for i in range(0, self._model.current_player.cards.__len__()) :
                self.play_simu(states, drawable_cards, i)

        return states

    def play_simu(self, states, drawable_cards, i):
        for card in drawable_cards:
            #Simulation
            self._opponent.add_card(card)
            self._model.pick_card_simu(card)
            self._model.play(i)

            #Génération de l'état correspondant
            state = State(self._model, self, drawable_cards[card])
            states.append(state)

            #Restauration de l'environnement
            self._save.backup()
```

D'abord, on récupère les cartes piochables par le prochain joueur. Ensuite on vérifie si le joueur courant est obligé de jouer la comtesse ou non, si non, on lance la simulation. Cette simulation va générer un état par carte qu'il est possible de piocher pour chaque carte du joueur courant. Pour chaque carte possible de piocher, on va ajouter la carte au jeu adverse, retirer cette carte de la pioche, puis appeler la fonction play. La fonction play aura le même traitement que normalement (retirer la carte de la main du joueur courant, effectuer l'action correspondante, passer au joueur suivant puis

piocher une carte). Cependant, le booleen is_simul étant à true, le joueur ne piochera pas de carte dans la fonction play. L'état correspondant est ensuite créer, puis les données sont restaurées (l'attribut _save est créé dans la fonction init de la classe).

2.2 - Algorithme Iterative Deepening Depth First Search (IDDFS)

Cet algorithme permet en théorie de rechercher les états terminaux sans heuristique (en blind search) en DFS à une profondeur donnée. Le facteur de branchement de l'IA augmentant d'une puissance de 20 (maximum) à chaque profondeur, il n'est pas possible d'explorer l'arbre de recherche entièrement.

Si l'on cherche les états successeurs de l'état courant à une profondeur de 3, et qu'il est possible de piocher toutes les cartes pour chaque joueur, 3 tours de suites, il y aura un total de 20^3 états successeurs (ce n'est bien sûr théoriquement pas possible, car certaines cartes n'existent qu'en un exemplaire).

Cependant, plus le jeu avance, moins il y a de cartes disponibles. Lorsque toutes les instances d'une même carte ont été jouées, une branche de l'arbre de jeu correspondant au fait de piocher cette carte disparaitra.

Ainsi, plus l'on avancera dans le jeu, moins l'arbre de jeu sera grand. L'IDDFS est alors pertinent.

Le problème de cet algorithme vient de la recherche des états terminaux dans le jeu. En effet, les états terminaux ont une faible chance d'arriver. L'implémentation de cet algorithme risque de résulter en une succession de choix s'apparentant comme étant aléatoire de la part de l'IA.

C'est pourquoi nous ne l'avons pas implémenté.

2.3 - L'algorithme alpha beta

La majeure partie du travail de l'algorithme alpha beta était de trouver un moyen de simuler des tours à une profondeur donnée afin d'en tirer les états correspondants (expliqué précédemment). La deuxième partie la plus importante venait de l'implémentation de l'heuristique, ou de la fonction éval.

Nous avons également combiné les algorithmes alpha beta et IDDFS. En effet, l'algorithme étant de plus en plus décisif au fil des tours, et l'arbre de moins en moins grand au fil des tours, nous avons implémenté un compteur qui s'incrémente de 1 tous les deux tours. Celui-ci permet d'avoir une profondeur de recherche faible en début de round, et plus grande en fin de round où les coups sont plus décisifs. (La profondeur de recherche démarre à 3 et finit au maximum à 5).

2.3.1 - La fonction éval

La fonction évaluation renvoie un poids en fonction de l'état actuel du jeu. La fonction prend en paramètres les éléments que connaît et que peut déterminer l'IA, par exemple les éléments connus sont la main de l'IA et les trois cartes qui sont face découvertes. Les éléments que l'on peut déterminer est la pioche à l'aide des cartes déjà jouées par l'IA et le joueur, la main du joueur et la carte brulée. On détermine donc un deck avec toutes les cartes susceptibles d'être dans la main du joueur. C'est à l'aide de cette construction et de la main de l'IA que l'on peut déterminer un poids pour l'état du jeu. Par exemple un état avec une carte espionne et une autre carte sera plus avantageux qu'un état avec deux espionnes.

La fonction éval se présente par une succession de conditions if qui vérifient quelle carte l'IA a dans la main et en fonction de ces deux cartes et de la composition du deck, on calcule un poids pour chaque carte que l'on additionne en fin de fonction pour renvoyer le poids de l'état. Le poids des cartes est basé, pour la majorité des cartes, sur des probabilités de victoire. Par exemple, le garde, le baron et le prince ont leurs poids plus ou moins grand en fonction de la carte que l'on a dans la main ou le nombre de carte restante. Prenons l'exemple du baron, le poids calculé sera déterminé par rapport à la deuxième carte que l'on a dans la main mais aussi au nombre de carte restante. Si notre main est constituée d'un baron et d'un roi et que l'on sait que la comtesse et la princesse ne sont pas en jeu, le poids (uniquement du baron) devient alors très grand.

Il y a donc plusieurs cas particuliers en fonction des cartes. Comme dit précédemment certains poids de carte sont calculés en fonction de la probabilité de victoire mais d'autres cartes comme l'espionne ou le prêtre ont des poids fixes car ce sont des cartes qui, certes utiles, mais n'offre pas de victoire immédiate pour l'IA. Ces cartes ont donc un poids qu'il leur est propre tout en gardant une certaine probabilité d'être vraiment avantageuse ou non en fonction de l'état du jeu comme par exemple le roi.

D'autres cartes ont leur propre fonction évaluation à savoir le garde le prince et le chancelier car ce sont des cartes à double actions. Elles sont construites de la manière suivante : La fonction prend en paramètre un booléen qui permettra de décider si l'on se trouve dans le calcul du poids ou dans la décision de la carte à faire deviner dans le cas du garde, le camp à faire défausser dans le cas du chancelier ou des cartes à remettre dans la pioche pour le cas du chancelier. Ces fonctions sont à la fois appelées dans la fonction éval mais aussi quand le moment vient de jouer l'une de ces cartes et donc de renvoyer non plus un poids mais une action qu'exécutera l'IA. Cet algorithme de décision est donc très similaire au calcul du poids dans la fonction éval car plus leur poids est élevé (pour le garde et le prince) plus la décision de l'IA sera évidente. Si par exemple on connaît la carte de l'adversaire et que ce dernier ne l'a pas jouée lorsque c'était son tour, le poids du garde devient donc très grand grâce à la probabilité que le joueur ait une carte en particulier et au moment de la décision de l'IA, elle va choisir la carte qui aura renvoyée, dans la fonction éval, un poids très grand.

Un coefficient est appliqué à chaque poids calculé pour chaque carte. Ce coefficient est le nombre de cartes restant dans la pioche que l'on divise par le nombre de cartes total. Certains états sont beaucoup plus avantageux que d'autres comme par exemple la princesse avec le baron mais si la probabilité d'arriver à cet état est extrêmement faible, il est préférable de se diriger vers un état moins avantageux mais ayant plus de chances d'arriver. Cela permet d'optimiser l'IA dans son choix de carte pour arriver à un état avec la plus forte probabilité de gagner et qui soit atteignable. Sans ce coefficient, l'IA aurait tendance à se diriger vers des états très avantageux mais très peu probables, ce qui mènerait à des décisions moins bonnes.

2.3.2 – Améliorations du minmax

Nous avons trois variantes de l'algorithmes de l'alpha béta dont nous avons testé les performances :

- L'algorithme min max classique

-l'algorithme alpha béta classique

-l'algorithme alpha béta avec tri des états successeurs en fonction de la probabilité d'occurrence de ceux-ci : les états les plus probables d'arriver seront souvent ceux étant les plus avantagés, c'est pourquoi la liste des états successeurs est triée de manière décroissante de telle manière que l'algorithme alpha beta néglige le plus de branches possibles.

L'amélioration en algorithme alpha béta permet une importante amélioration du temps de calcul des états successeurs.

Lorsque l'on y ajoute le tri en fonction de la probabilité d'occurrence d'un état, nous avons également une légère amélioration du temps de calcul des états successeurs. Cela vient du fait que les états les plus avantageux seront souvent les plus probables d'arriver (bien que pas toujours, on peut avoir un état très probable d'arriver qui ne soit pas du tout avantageux)

2.4 - L'apprentissage par renforcement

Nous voulions implémenter un algorithme d'IA par renforcement en tant qu'IA à difficulté difficile, cependant nous avons rencontrés plusieurs problèmes.

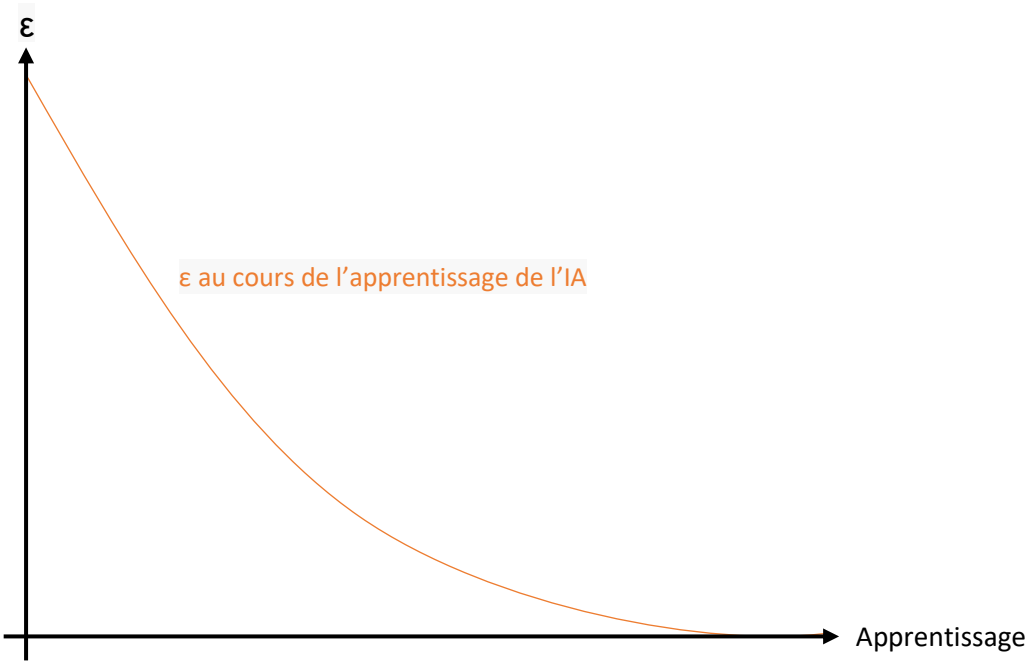
Nous avons implémenté la simulation de l'apprentissage de l'IA dans le programme. En effet, lorsque l'on choisit la difficulté difficile, une boucle lançant la fonction play jusqu'à ce qu'une des deux IA possède un score de 1000 démarre. Les booléen islearning de la classe Model est utilisé pour lancer la simulation lorsque la difficulté difficile est choisie. Le booléen victory learning de la classe Model est utilisé car la gestion de la victoire est différente lorsque l'IA apprend ou qu'elle joue dans une vraie situation. (Lorsque l'IA apprend, on ne veut pas qu'un écran de fin de round apparaisse, on cherche à ce que le prochain round apparaisse directement).

Le but de cette manière de procéder était de faire apprendre l'IA pendant un certain temps, de stocker les valeurs apprises dans un fichier grâce à la librairie pickle, pour finalement enlever le booléen islearning afin que l'IA joue normalement lorsque la difficulté difficile est choisie.

Le meilleur moyen de comprendre le fonctionnement d'un agent intelligent qui apprend par lui-même, simplement en jouant contre un autre agent, était d'en coder un sur un projet beaucoup plus simple. Un simple morpion suffisait donc pour appréhender la base de l'apprentissage par renforcement, même si ce dernier reste un jeu déterministe et possédant un nombre assez limité d'états possibles (de l'ordre de quelques centaines de milliers seulement, contre des milliards pour Love Letter).

2.4.1 – Exploration et exploitation : ε – greedy

Pour qu'un agent intelligent puisse apprendre, il faut bien sûr qu'il puisse tester, et donc explorer les différentes actions qui peuvent être effectuées. De plus, au bout d'un certain nombre de parties il faut bien que ce dernier puisse choisir la meilleure action à réaliser en fonction de ce qu'il a déjà fait dans des parties précédentes. C'est ici qu'intervient le système ε-greedy, il permet simplement de dire à l'IA, si elle doit explorer, c'est-à-dire choisir aléatoirement une action à réaliser, ou si elle doit exploiter les connaissances qu'elle a déjà acquise, c'est-à-dire choisir l'action qui l'arrange le plus, ou celle qui arrange le moins l'adversaire. Le ε représente une valeur comprise entre 0 et 1, qui indiquera quel type de choix doit faire l'IA. Nous pouvons par exemple initialiser ε à 0,995 et le faire diminuer progressivement jusqu'à 0 tout au long de l'apprentissage (des milliers, millions, ou même des milliards de parties). Ensuite, à chaque début de round, quand l'IA doit choisir une action à réaliser, on génère aléatoirement une valeur comprise entre 0 et 1, si cette valeur est plus petite que ε, l'IA explore, sinon elle exploite. Ainsi l'IA fera de moins en moins d'exploration et à la fin de son entrainement, elle choisira toujours les meilleures actions à réaliser.



Evolution d' ε durant l'apprentissage d'un agent intelligent

2.4.2 – La fonction d'évaluation (value function)

Maintenant que l'IA sait quand elle doit explorer ou bien exploiter, vient le problème de l'exploitation. Effectivement, pour l'exploration, rien de bien compliquer, l'IA a juste à choisir aléatoirement quelle action réaliser, néanmoins, durant l'exploitation, l'IA doit choisir le mieux possible, c'est donc là qu'intervient la fonction d'évaluation qui va s'occuper de donner une valeur à chaque état. Ici, la valeur d'un état détermine s'il est bon d'y être ou pas. Dans l'exemple du morpion, l'agent intelligent devra choisir entre un certain nombre d'états successeurs, il devra donc choisir celui qui possède la valeur la plus basse, pour que l'adversaire soit le plus désavantagé, et donc qu'il ait moins de chance de gagner.

La fonction d'évaluation prend place à la fin d'une partie, effectivement, il s'agira de partir de l'état final, qui s'avère être soit gagnant, perdant ou nul. Chaque état final devra posséder une valeur, pour le morpion, on a 2 pour la victoire, -1 pour l'égalité et -2 pour la défaite (valeurs choisies arbitrairement). Ainsi, il est mieux d'être dans l'état gagnant (ici la valeur est de 2) que dans l'état perdant (ici la valeur est -2).

Voyons maintenant comment fonctionne concrètement la fonction d'évaluation :

$$V(s) = V(s) + lr*(V(s') – V(s))$$

Avec :

- V la fonction d'évaluation
- s l'état courant
- lr le Learning rate
- s' l'état successeur

Une fois une partie terminée, on finit forcement sur un état possédant déjà une valeur, ainsi dans la formule ci-dessus on remplacera V(s') par la récompense (valeur de l'état final) donné par le dernier état. Ensuite on va simplement remonter les états jusqu'au premier état de la partie, et appliquer la formule pour affecter une nouvelle valeur à chaque état rencontré (chaque état est initialisé à 0, et on peut remonter la partie en sens inverse en la sauvegardant tout au long de cette dernière).

Finalement, chaque état aura été parcouru plusieurs fois et aura une valeur plus précise et donc l'agent intelligent pourra sans problèmes choisir la meilleure action à réaliser dans un état donné.

2.4.3 – Résultats du Morpion / problèmes pour LoveLetter

Pour le Morpion, un taux de 100% de victoire ou nul est facilement atteignable pour un agent intelligent qui apprend sur quelques dizaines de millions de parties, du fait que le Morpion est un jeu avec un nombre d'état assez limité. De plus, le Morpion est un jeu où il suffit simplement de jouer chacun son tour, contrairement à LoveLetter où il peut y avoir des actions double (cas du Garde), ce qui rajoute une difficulté supplémentaire.

Un autre problème est survenu lors de l'implémentation de l'agent intelligent dans le Morpion, effectivement ce dernier n'essaye pas de gagner, il fait en sorte que l'adversaire ne gagne pas, ce qui mène principalement à des égalités. Ce phénomène était assez prévisible dans le sens où pendant l'exploitation, l'IA va choisir l'action qui mène à l'état le plus désavantageux pour l'adversaire.

Le principal problème pour l'implémentation d'un tel agent sur LoveLetter, est le nombre incalculable d'état possible, ce qui fait qu'il aurait fallu des jours, voir des semaines pour entrainer une IA qui puisse éventuellement faire quelques bonnes actions. Effectivement, un simple entrainement de cinquante millions de partie pour le Morpion à pris pas loin de 2h. Une solution aurait été d'implanter un système d'approximation, étant donné que certaine situation, certains états se ressemble.

Conclusion

Dans ce projet, nous avons pu prendre en main différents algorithmes dans le cadre d'un projet concret. Il nous a également permis de nous familiariser avec github et d'améliorer notre communication et travail de groupe. Enfin, il nous a permis d'apprendre à coder en python. Cependant nous avons eu certaines difficultés, et plusieurs améliorations sont possibles.

1-Les difficultés :

- L'implémentation de la GUI : Celle-ci a pris du temps. Nous avons pris un long moment de réflexion avant de commencer le projet afin que le code soit le plus modulaire et propre possible. En effet, ne connaissant pas python (et par extension la librairie Tkinter) et n'ayant pas beaucoup d'expérience dans l'implémentation du pattern MVC et l'élaboration d'une interface graphique, il était difficile de se projeter quant à la manière de résoudre nos problèmes et au temps qu'allait prendre nos tâches respectives.
- L'implémentation de l'IA : Nous avons implémenté l'IA en dernier (après le codage de la GUI et des règles). Malgré nos efforts pour garder un code efficace et modulaire, le codage de l'IA s'est avéré fastidieux et s'apparentant à du « bricolage » par moment. Ceci est à la fois dû à la nature de l'IA (qui ne se joue finalement que sur des probabilités rentrées à la main), et à

l'ajout d'une fonctionnalité que l'on n'a pas préparée à l'avance. Nous n'avions en effet pas prévu ce qu'impliquait l'implémentation de l'algorithme de l'IA (la simulation des états successeurs, le fait qu'il fallait que cette simulation soit cachée aux yeux de l'utilisateur, les problèmes que posent la simulation par rapport à la manipulation des références de certaines variables…).

Cela a mené à des « if » qui parsèment le code afin de régler nos problèmes sur le tas, ce qui le rend moins clair et moins modulaire.

- Le travail de groupe. Il est difficile de planifier le travail équitablement lorsque l'on ne possède pas toutes les connaissances nécessaires au codage en amont (pattern MVC, librairie Tkinter, python...). Il est également difficile de fournir un travail équitable quand tout le monde n'a pas les mêmes charges de travail au même moment dans le semestre (du fait que tout le monde n'a pas les mêmes matières). Enfin tout le monde n'a pas forcément les mêmes ambitions lors d'un tel projet, ni envie d'apprendre les mêmes choses.

2-Les améliorations :

- Une des améliorations découlant directement de nos difficultés vient du code qui est devenu de moins en moins clair au fur et à mesure que l'on y a ajouté des fonctionnalités
- L'utilisation de la fonction deepcopy lors de la sauvegarde des états utilise un surplus de ressources, l'implémentation d'un command pattern serait potentiellement préférable.
- L'ajout d'une difficulté avec un apprentissage par renforcement aurait été bénéfique et aurait pu nous servir de comparaison avec l'alpha beta. De plus, notre début d'implémentation n'est pas optimal car il nécessite que l'on intervienne dans le code entre la phase d'apprentissage et la phase de jeu avec un vrai joueur
- La fonction éval de l'algorithme alpha beta pourra toujours être amélioré. Elle dépend entièrement des probabilités rentrées par un humain donc elle sera toujours faillible et dépendra de la compétence de celui qui implémente ces probabilités (plus le codeur est fort plus l'IA sera forte). De plus, il existe tellement de cas particuliers que l'algorithme ne se résume finalement qu'à un bricolage de probabilités afin d'avoir le meilleur rendu possible.
- Les algorithmes de décision pour les cartes à doubles actions. Tout comme la fonction éval, ils pourront toujours être améliorés. Il y a énormément de paramètre a prendre en compte au-delà du nombre de cartes restantes et de la main actuelle de l'IA.
- Le calcul des états successeurs pourraient se faire grâce à des threads, cela pourrait grandement améliorer la vitesse d'exécution du programme et donner une IA plus performante et difficile à battre.

Annexe 1 : UML de la partie model du MVC

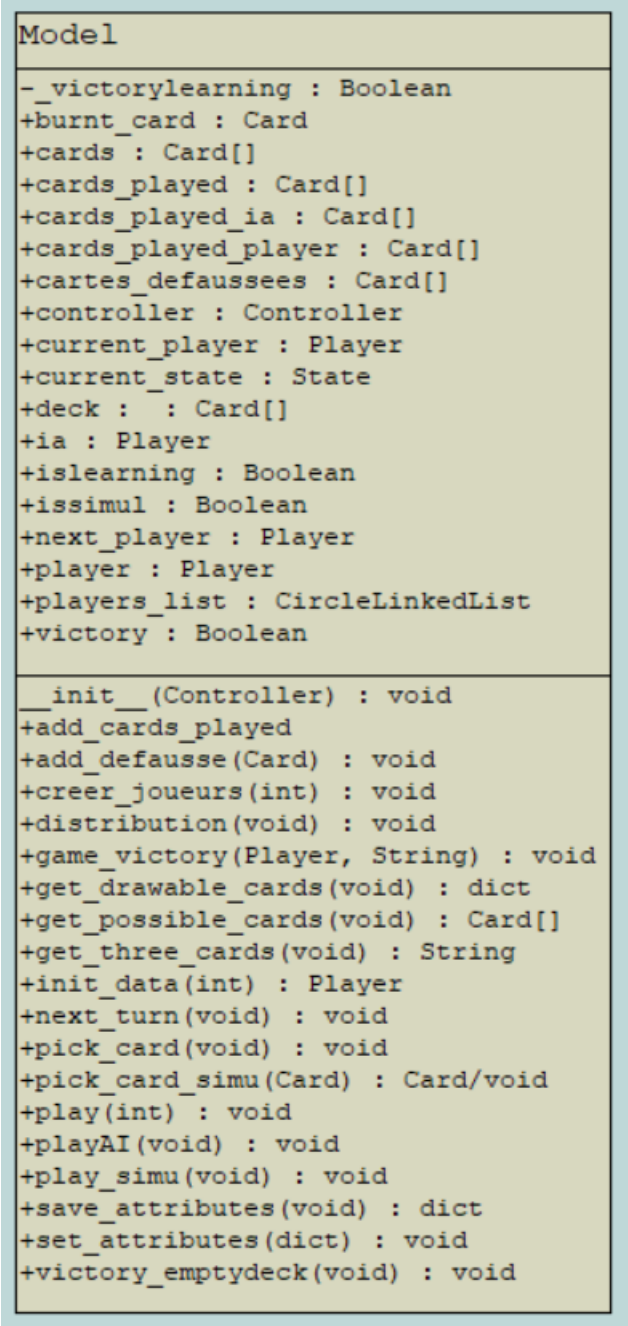


Figure 1. Diagramme de la classe model

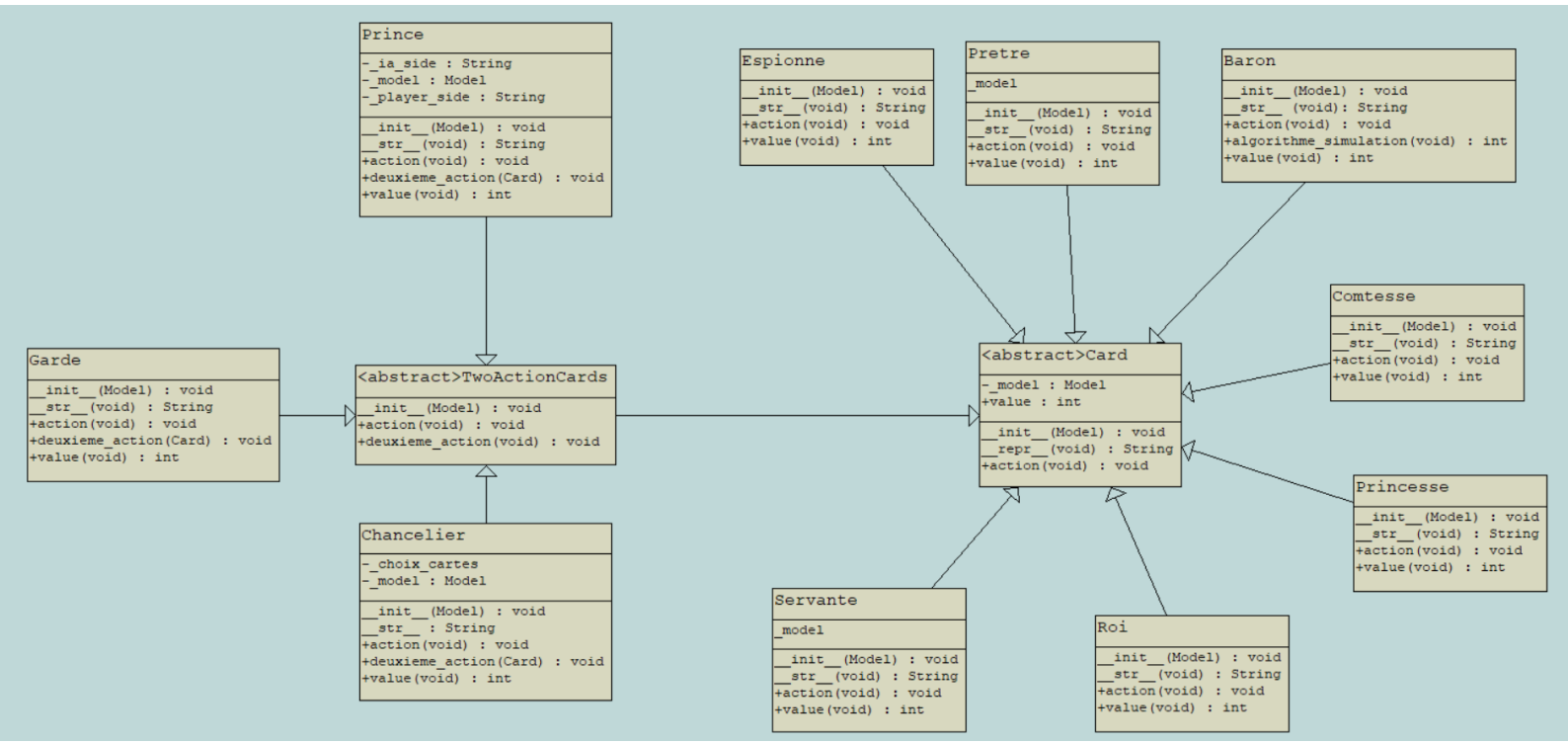


Figure 2. Diagramme du module cards

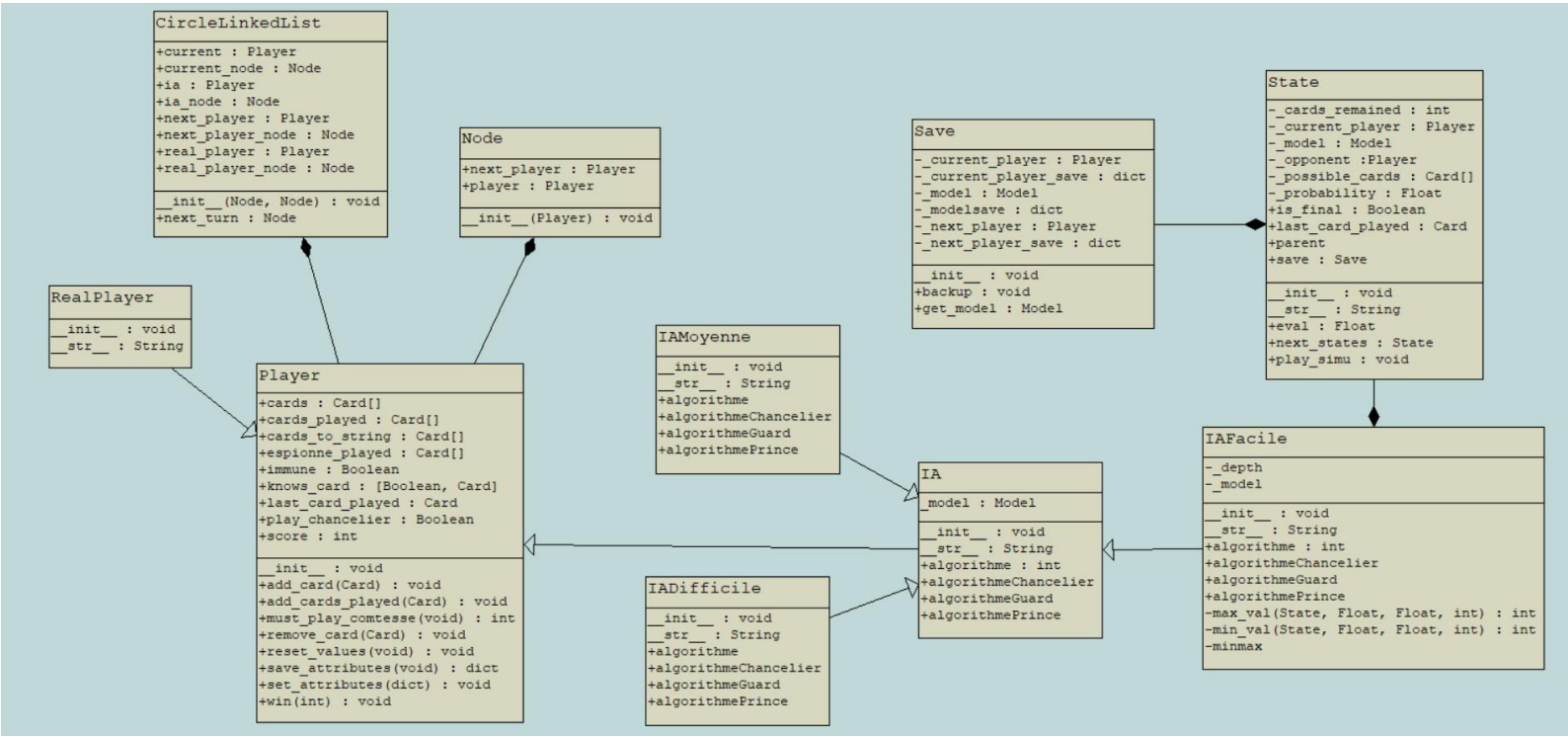


Figure 3. Diagramme du module player

Annexe 2 : UML de la partie view du MVC

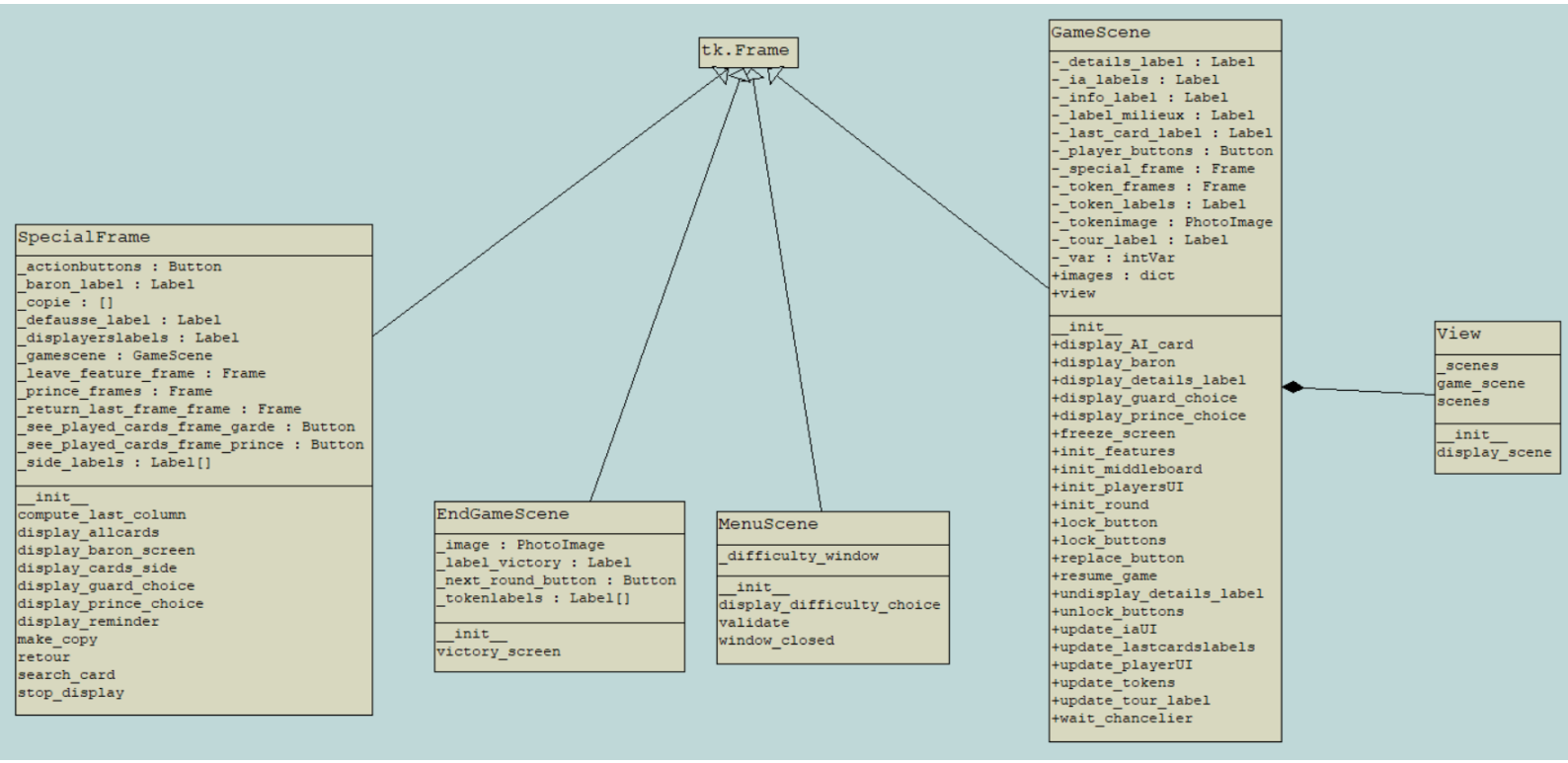


Figure 4. Diagramme du module view

Annexe 3 : UML de la partie controller du MVC

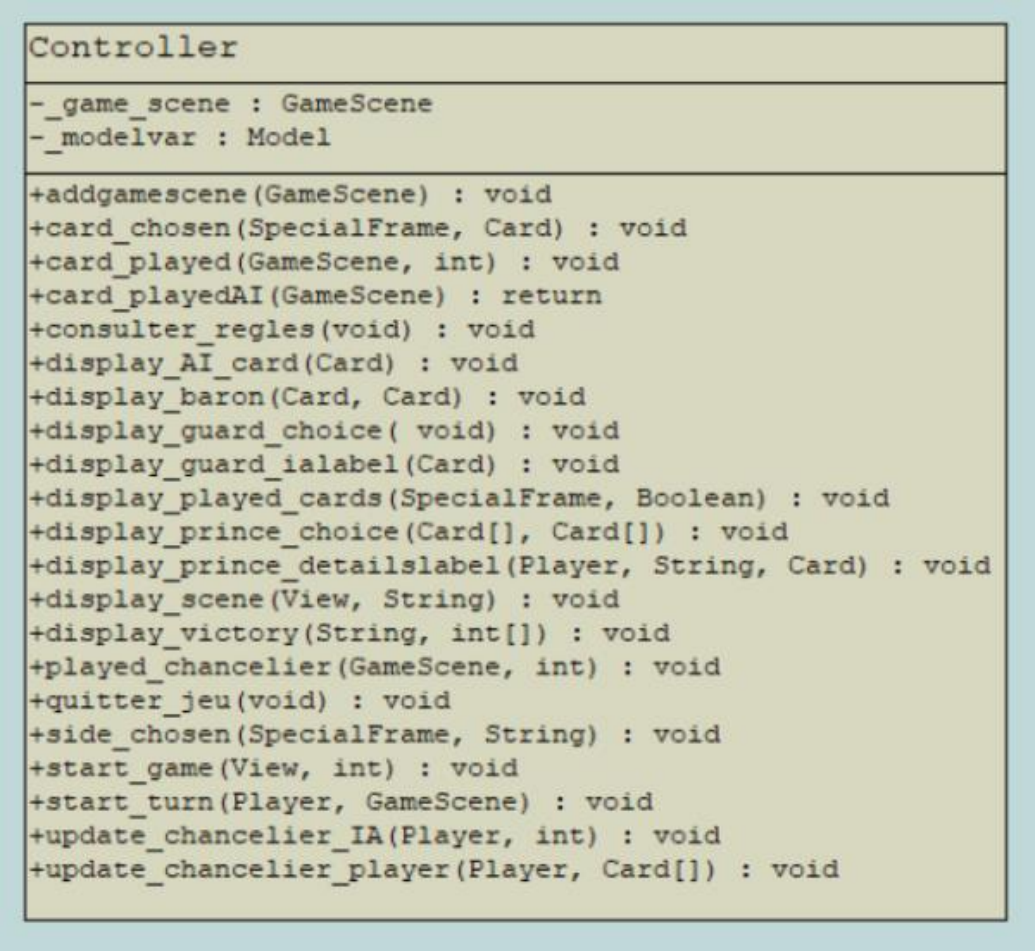


Figure 5. Diagramme de la classe Controller

Auteurs :
DUPAYRAT Antoine
CARON Allane
MOYSE Antoine

IA41 – Automne 2020

Rapport de Projet

Sujet :

Love Letter

utbm
université de technologie
Belfort-Montbéliard