



Love Letter

Rapport • Projet

IA41 - Intelligence artificielle
Concepts fondamentaux

Responsable UV : Fabrice LAURI

Alexandre DESBOS

Thomas SIRVENT

Martin BLANCKAERT

Introduction	2
Réalisation	3
Analyse du problème à résoudre	
Représentations des connaissances adoptées	
Définition d'un état	
Les états terminaux	4
Passage d'un état à un état suivant	
Evaluation d'un état	5
Algorithme utilisé	7
Résultats obtenus	8
Situations particulières.	8
Difficultés	12
Les références	
Le jeu virtuel	
Gestion de l'adversaire	13
Suite de la partie.	14
Annexe	17
[1] generateChildren	
[2] nextStates	19
[3] evaluate	20
[4] chancellor_power	21
[5] negamax	22
[6] powerPrinceAI	23

*Toutes les fonctions affichées en bordeaux sont cliquables et disponibles dans l'annexe du rapport.

Introduction

Ce projet ayant pour objectif de découvrir les concepts fondamentaux de l'intelligence artificielle et de travailler sur les algorithmes vus en cours, nous avons fait le choix ambitieux de travailler sur **Love Letter**, un des nouveaux sujets expérimentaux de l'UV proposés ce semestre. Avec seulement un membre du groupe ayant déjà travaillé sur les algorithmes d'intelligence artificielle, ce projet était pour nous un véritable défi, permettant de faire le plein d'expériences.

Love letter est un jeu de déduction pouvant être joué jusqu'à 6 joueurs. L'objectif est de faire perdre ses adversaires en réussissant à garder la carte ayant la plus grosse valeur jusqu'à la fin de la manche ou en les faisant défausser jusqu'à devenir le dernier joueur en lice.

Le jeu est composé de 10 types de cartes et de 21 cartes en tout :

- 0 - Espionne (x2)
- 1 - Garde (x6)
- 2 - Prêtre (x2)
- 3 - Baron (x2)
- 4 - Servante (x2)
- 5 - Prince (x2)
- 6 - Chancelier (x2)
- 7 - Roi (x1)
- 8 - Comtesse (x1)
- 9 - Princesse (x1)

Chaque carte possède des capacités lui permettant d'exécuter une action lorsqu'elle est jouée. Certaines d'entre elles permettent d'éliminer directement un adversaire, d'autres permettent d'obtenir des informations sur le jeu (le Prêtre permet de connaître la carte de l'adversaire par exemple).

Pour simplifier le problème et respecter l'échéance du projet, nous avons décidé de développer le jeu pour seulement 2 joueurs.

Puis, nous avons défini au début du projet nos priorités le concernant :

- ☒ Faire fonctionner le jeu dans la console (pour deux humains seulement).
- ☒ Implémenter une (ou plusieurs) IA.
- ☒ Créer une interface graphique pour lier le tout.

Réalisation

Analyse du problème à résoudre

L'intelligence artificielle que nous devions développer pour jouer à Love Letter devait être en mesure de se comporter comme un joueur.

Elle devait être capable de :

- Connaître les cartes qui sont sur la table (les cartes isolées au début de la partie et toutes les cartes jouées jusqu'au moment où elle doit prendre une décision).
- Piocher une carte dans le deck .
- Décider quelle carte jouer parmi les cartes qu'elle a dans la main en s'adaptant à tous les cas particuliers liés aux capacités des cartes et **en favorisant la situation qui l'avantage le plus** dans la suite du jeu en fonction de toutes les situations possibles.
- Subir l'effet des cartes.

Elle doit donc être en mesure de générer l'arbre de jeu en considérant toutes les possibilités. Autrement dit, pour décider quelle carte sera la plus intéressante à garder dans sa main, elle doit générer toutes les possibilités de cartes que pourrait jouer l'adversaire et voir dans quelles situations elle aura les meilleurs résultats. Pour cela, elle peut déduire, grâce aux cartes qui ont déjà été jouées, grâce à celles qui sont déjà sur la table et celles qu'elle a dans la main, quelle carte peut jouer son adversaire et donc connaître quelle situation risque de la faire perdre.

Représentations des connaissances adoptées

Nous avons décidé que l'IA serait considérée comme une instance de la classe Player() et donc qu'elle aurait les mêmes méthodes et attributs que le joueur humain.

Définition d'un état

Nous avons décidé de définir une classe Node qui a les attributs suivants et va être utilisée pour représenter l'ensemble des éléments de l'arbre de jeu pendant la partie :

- `state` : L'état correspondant au noeud (nous reviendrons sur cette classe ensuite)
- `parent` : Correspond au noeud parent du noeud actuel
- `children` : La liste d'enfants du noeud actuel
- `value` : Représente la valeur du nœud, qui va être obtenue grâce à la fonction **evaluate(node)** que nous définirons ensuite.

Chaque noeud dispose d'un attribut "state", qui est une instance de la classe State qui a comme attribut :

- `deck` : Le deck de la manche.
- `isolatedCards` : Les cartes isolées au début de la manche
- `listOfCards` : La liste des cartes (les 10 types de cartes du jeu)
- `player` : Le joueur
- `opponent` : L'adversaire du joueur

Comme son nom l'indique, State définit l'état du jeu. Cet état représente le jeu à l'instant T et va être utilisé par l'intelligence artificielle pour décider quelle carte de sa main elle souhaite jouer.

Les états terminaux

Les états terminaux correspondent à deux situations possibles :

- Le deck est vide
- Un des joueurs a gagné la partie, l'autre joueur n'étant plus en lice.

Pour vérifier le premier point, une simple vérification `if not deck` suffit.

Pour vérifier ce deuxième point, on se réfère aux attributs `hasWon` et `isAlive` de la classe Player.

```
def isTerminal(node):  
  
    cond1 = not node.state.deck # deck vide  
    cond2 = (not node.state.player.isAlive) or node.state.player.opponent.hasWon  
    cond3 = (not node.state.player.opponent.isAlive) or node.state.player.hasWon  
  
    return cond1 or cond2 or cond3
```

Cette fonction va retourner un booléen utilisé pour vérifier si un nœud est terminal ou non.

Passage d'un état à un état suivant

Il est nécessaire que l'intelligence artificielle soit en mesure de générer tous les états possibles de l'arbre de jeu. Autrement dit, elle doit être capable, à partir d'un état donné, de générer les états suivants. La fonction chargée de réaliser cette partie est `nextState()`, qui elle-même fait appel à `generateChildren()`. L'objectif est ici, de créer une partie virtuelle à partir du "virtualNode" envoyé en argument à la fonction. Ce virtualNode est créé dans la fonction `negamax()`, à l'aide du package `copy` et de la

fonction `deepcopy()` qui permet de copier un objet et ses attributs en étant certain qu'il n'existe plus de relations entre les deux objets. L'intérêt ici est d'être certain que la modification du `virtualNode` ne va pas impacter le nœud original.

Une fois envoyé dans `nextState()`, on peut manipuler le deck, les cartes connues et les mains des joueurs sans se préoccuper du nœud original tout en le conservant pour savoir quelle carte était dans la main du joueur à l'origine et pouvoir choisir la bonne carte à jouer pour la suite.

Une fois dans la manche virtuelle, l'IA va générer tous les coups et les pioches possibles à partir de l'instant où elle a copié l'état du jeu. Pour cela, il faut d'abord définir le joueur qui est en train de jouer, que nous nommerons `activePlayer` et qui sera envoyé à la fonction `generateChildren()`. Cette fonction, pour chaque carte de la main du joueur, va créer un nouveau nœud virtuel dans lequel le joueur va jouer la carte et utiliser sa capacité. En réalisant cette étape pour chaque carte de sa main et chaque carte possible du jeu, `generateChildren()` va générer tous les nœuds de l'arbre et les ajouter à la liste `nextNodes` envoyée en paramètre. Finalement, la fonction `nextState()` retournera dans la fonction negamax cette liste correspondant à l'ensemble des enfants du noeud originel.

Evaluation d'un état

C'est en faisant cette fonction `evaluate()` que nous nous sommes rendu compte à quel point l'algorithme MinMax est limité pour des jeux avec une notion de chance, où l'on ne connaît ni les coups possibles de l'adversaire ni les siens plus d'un tour à l'avance. Plus on réfléchissait à cette fonction, plus on trouvait de cas particuliers où elle n'allait pas fonctionner à son plein potentiel, sans pour autant pouvoir la modifier pour prendre en compte les cas en question.

Cette fonction `evaluate()` retournant le score d'un état, nous avons dû la réécrire lorsque nous avons modifié le fonctionnement de `nextState()`. En effet, lorsque nous avions initialement imaginé la meilleure façon d'attribuer un score à un état, la représentation la plus fidèle que nous avions trouvée était de le calculer à partir du début du tour après la pioche, c'est-à-dire avec deux cartes en main. Cependant, cette modélisation représente un problème : lorsque le point de vue change, l'autre joueur se retrouve face à un adversaire à deux cartes - ce qui est impossible.

Nous avons donc réinventé `nextState()` afin qu'au début ainsi qu'à la fin de la fonction, le joueur ait une seule carte dans sa main. Cette nouvelle manière de procéder nous a

cependant rajouté une contrainte supplémentaire : on doit piocher et jouer au milieu de `nextState()`. Afin de simuler tous les cas possibles, nous avons imaginé un système de pioche “virtuelle”, qui fait piocher au joueur chaque carte encore en jeu et crée un enfant intermédiaire pour chacun de ces cas. On retire ensuite cette carte du deck (quel que soit son index) afin de maintenir la cohérence du jeu : la carte piochée est désormais dans la main du joueur. Chacun des enfants intermédiaires se dédouble encore finalement en deux enfants couvrant les deux cas possibles restants : la carte jouée est la première ou la deuxième. Cette façon de procéder est la plus exhaustive possible, générant chaque possibilité à partir d'un état.

Malheureusement, il est nécessaire de prendre en compte le cas particulier du Prince qui fait défausser le joueur qui vient de le jouer ou fait défausser l'adversaire. Quand c'est le cas, un test vérifie la taille de la main des joueurs dans `nextState()` et fait piocher le joueur qui a la main vide.

Cette refonte de `nextState()` nous a donc fait modifier la fonction `evaluate()` afin qu'elle se plie également au nouveau format visant à toujours n'avoir qu'une seule carte dans la main à la fin de `nextState()`. Cependant, nous souhaitions conserver le calcul du score d'un binôme de cartes grâce à `evaluate()`, car lorsqu'on joue dans Love Letter, on a deux cartes en main. Notre calcul de score se base donc sur tous les binômes faisables à partir de la carte que le joueur a en main, et retourne une moyenne de tous ces scores. Afin de rendre ce score le plus adapté possible, nous pondérons également ces moyennes en fonction de la probabilité de piocher la carte associée (ex : avec un Baron en main, le score du binôme Baron/Princesse est très élevé, et celui du binôme Baron/Garde très faible car la valeur de la carte Garde est bien plus faible que celle de la carte Princesse. Cependant, comme il y a beaucoup plus d'exemplaires de la carte Garde que de la carte Princesse, la moyenne globale de l'état sera relativement faible). Le calcul de la valeur de chaque binôme prenant également en compte de multiples facteurs comme par exemple l'avancement de la manche et le nombre de cartes connues, notre fonction `evaluate()` se veut la plus **réaliste** possible.

Pour ce faire, nous avons trié les cartes en différentes catégories :

- les cartes “constantes” : Prêtre, Servante, Prince et Chancelier. Leur valeur reste la même tout au long de la manche, plus ou moins grande en fonction de leur utilité à nos yeux.
- les cartes à haute valeur : Roi, Comtesse et Princesse. Dans le calcul de leur valeur est prise en compte une variable calculant l'inverse du nombre de cartes restantes en jeu. Plus concrètement, elles ont une valeur infime en début de manche, mais

qui devient très grande lorsque le deck se vide, car c'est la carte dont la valeur est la plus élevée qui gagne si le paquet est vide.

- les cartes "situationnelles" : Garde et Baron. Leur valeur est celle de la probabilité de deviner la bonne carte et celle de la probabilité de gagner le duel respectivement. Ces valeurs sont donc elles aussi modifiées en fonction de la connaissance que l'IA a du jeu.
- L'Espionne. Bien que nous trouvions son utilité minimale au début du projet, nous nous sommes vite rendu compte de l'importance de prendre un point supplémentaire à chaque tour si possible, notamment dans un jeu où il y a des coups qui ont un impact moindre. Nous avons donc attribué à l'Espionne une valeur fixe, élevée si l'IA n'a pas encore récupéré de point supplémentaire, nulle si elle a déjà le point.

Algorithme utilisé

Nous avons décidé d'utiliser l'algorithme **negamax**, une variante simplifiée de l'algorithme MinMax qui se base sur le principe du jeu à somme nulle (comme MinMax).

Le principe mathématique du negamax est le suivant :

$$\max(a, b) = -\min(-a, -b)$$

Autrement dit, le joueur A en train d'effectuer la recherche dans l'arbre de jeu cherchera toujours à maximiser ses coups et à minimiser ceux de son adversaire. L'idée est de ne plus avoir le principe de joueur "max" et de joueur "min".

La simplification par rapport à l'algorithme MinMax est donc le fait qu'au niveau du code, il n'est pas nécessaire de considérer que le joueur A doit maximiser la valeur du nœud dans lequel il souhaite arriver et que le joueur B veut minimiser celle-ci. Dans les deux cas, le joueur (A ou B) veut choisir le nœud dans lequel il souhaite arriver avec la plus grande valeur possible.

Voici l'implémentation de negamax en pseudo-code :

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := -∞
    for each child of node do
        value := max(value, -negamax(child, depth - 1, -color))
    return value
```

Bien qu'il n'y ait pas beaucoup de tours dans une manche de Love Letter (un maximum de 8 tours), l'algorithme negamax requiert beaucoup trop de calculs pour prévoir toutes les issues possibles de la manche dans les cas où les joueurs ont épuisé le deck et où la victoire se joue à la carte haute.

Nous avons donc été contraints d'ajouter une profondeur limitée pour éviter une trop grande quantité de calculs à réaliser.

La profondeur de l'arbre est statique et de 4 étages. De plus, à score égaux, l'IA tentera de se diriger vers le score qui est le plus éloigné du tour actuel. Nous avons décidé que les bons scores qui apparaissent trop tôt sont souvent ceux qui incluent une grande prise de risque. Notre IA peut donc être considérée comme "prudente".

Pour diminuer le nombre d'états générés par l'intelligence artificielle, nous avons décidé d'implémenter un élagage **alpha-beta**, qui permet de conserver seulement les branches de l'arbre prometteuses par rapport aux noeuds visités précédemment. On ne conserve que les noeuds qui ont une valeur supérieure à ceux évalués précédemment. voir [annexe \[5\]](#) pour l'algorithme negamax appliqué à notre projet.

Résultats obtenus

Le jeu en mode joueur contre joueur est totalement fonctionnel mais sans interface graphique. Nous avons préféré privilégier une IA fonctionnelle sans interface graphique plutôt qu'un jeu fonctionnel avec UI mais sans IA correcte.

Pour ce qui est de l'intelligence artificielle, elle est capable de prendre part à la partie, elle peut tirer une carte, jouer une des cartes de sa main en gérant les cas particuliers comme le Prince ou le Chancelier et choisir ses coups en fonction des poids des binômes de cartes et des situations analysées.

Pour réaliser un essai, nous vous invitons à télécharger le projet en suivant le lien hypertexte suivant : [Gitlab - LoveLetter AI Project](#)

Le fonctionnement de l'IA est affichée de cette manière lorsqu'elle joue :

```
--- Time for Terminator(AI) to play! ---  
  
Terminator is thinking  
Just a sec ...  
  
The AI played a Chancellor !
```

Situations particulières.

Un des problèmes auxquels nous avons été confrontés est le fait que Love Letter est un jeu où il existe beaucoup de cas particuliers très précis et plutôt rares auxquels on peut difficilement penser si on est n'est pas un habitué du jeu ou que l'on ne rencontre pas la situation en jouant. Tous ces cas particuliers nous ont obligés à remanier la structure du code plusieurs fois et à chercher à faire un code plus souple et apte à être modifié et adapté facilement au fur et à mesure que nous rencontrions les cas.

Les principaux cas ci-dessous ont pu être gérés assez facilement mais requièrent de nombreux contrôles dans le code :

Le premier cas correspond à la capacité de la carte 8 - la Comtesse - qui indique que dans le cas où un Prince ou un Roi est pioché alors que la Comtesse est en main, le joueur est obligé de défausser sa Comtesse.

Ce cas a été géré avec des tests à chaque fois qu'un tour est joué pour chaque joueur, dans la méthode `playTurn()` :

```
cardValues = []  
  
for j in range(len(self.hand)):  
    cardValues.append(self.hand[j].value)  
  
cond5 = 5 in cardValues # detects a Prince  
cond7 = 7 in cardValues # detects a King  
cond8 = 8 in cardValues # detects a Countess  
  
if cond8 and (cond5 or cond7):  
  
    # Search to find the countess  
    for i in range(len(self.hand)):  
  
        if cardValues[i] == 8:
```

```
index = i

print("Countess was discarded !\n" if real else "", end="")
```

Le cas de la Princesse :

Cette carte fait perdre la manche au joueur qui la défausse.

Il faut donc vérifier si la Princesse est défaussée à chaque fois qu'une carte est jouée. C'est le cas dans la méthode `playCard()` :

```
if cardPlayed.value == 9:
    self.isAlive = False

    print(f"\n{self.name} played a Princess !" if real else "", end="")
```

Nous reviendrons sur la variable `real` dans la partie sur les difficultés.

Un des cas les plus difficiles à gérer est celui du Chancelier.

Pour rappel, la capacité du Chancelier est la suivante :

Le joueur qui joue le Chancelier pioche 2 cartes du deck et les ajoute à sa main. Il choisit ensuite la carte qu'il souhaite conserver parmi les 3 qu'il a dans la main et replace les deux autres cartes dans le deck dans l'ordre de son choix.

Il est nécessaire de gérer le cas où il ne reste qu'une carte dans le deck, ou qu'il ne n'en reste plus.

Nous avons donc imaginé des tests qui contrôlent la longueur du deck et qui font tirer au joueur un bon nombre de cartes en fonction de la longueur dans la fonction `chancellor_power()`. (voir [annexe \[4\]](#) pour la fonction complète)

Un dernier exemple de cas qui oblige à gérer des exceptions pendant la partie est celui où le deck est vide alors qu'un Prince est joué. Lorsque le Prince est joué par un joueur, celui-ci fait défausser la main d'un joueur cible (pouvant être lui-même). Le problème survient lorsque la dernière carte piochée dans le deck est un Prince et que le joueur qui l'a piochée décide de le jouer. Puisque le deck est vide, le joueur visé n'a pas de nouvelle carte à piocher et ne peut pas comparer sa main. Le joueur sans main doit donc piocher la carte cachée qui est sur le plateau.

Dernier cas, celui des capacités des cartes :

Nous avons décidé d'écrire une fonction power commune à toutes les cartes qui exécute seulement le bout de code correspondant à la carte qui est jouée. Pour le Prince, le Chancelier et le Roi, nous avons écrit leurs fonctions séparément car ces cartes nécessitent un nombre conséquent d'instructions et de contrôles. Le problème est que dans ces fonctions, une interaction avec le joueur est nécessaire.

Par exemple, dans le cas du Prince qui est joué (figure ci-contre), le joueur doit choisir le joueur qu'il souhaite viser.

Quand l'IA doit faire un choix, que ce soit dans un cas virtuel ou réel, elle doit être en mesure de se décider par elle-même. Pour cela nous avons créé des versions "AI" des pouvoirs de certaines cartes qui vont être appelées dans la fonction power de la carte en question.

```
Time for Alex (Human) to play!

Alex's hand is :
0. Guard [1]
1. Prince [5]

What card do you want to play ? (0/1)
1

Who do you want to target ? [You/Opponent]
```

Voici un exemple :

Dans la fonction power de la carte Prince, dans le cas où le joueur est l'IA ou dans le cas où la manche est virtuelle, la fonction `powerPrinceAI()` est exécutée et permet à l'IA de faire le choix "you" ou "opponent".

```
if activePlayer.gender == "Human" and real:
    choice = input("\nWho do you want to target ? [You/Opponent]\n")
    #some code
else:
    choice = powerPrinceAI(activePlayer)
```

Difficultés

Les références

Une première difficulté que nous avons rencontrée est le fait que, en python, les variables peuvent avoir des références vers un même objet. Lorsque nous avons défini les listes `playedCards`, nous pensions que chacun des joueurs avait sa propre liste de cartes jouées.

```
self.player1.playedCards = self.player2.playedCards = []
```

Mais à cause des références, nous avons compris que l'attribut `playedCards` des deux joueurs était le même. Cette difficulté nous a posé problème par rapport à la connaissance du nombre de cartes jouées pendant les parties au début, mais nous a finalement permis de bien comprendre le système de références et de l'utiliser à notre avantage dans le reste du projet.

Nous avons décidé qu'il serait intéressant que les `playedCards` soient communes aux deux joueurs, cela permet de représenter les cartes qui sont sur la table sans avoir besoin d'utiliser de variables globales.

Le jeu virtuel

Une seconde difficulté à laquelle nous n'avions pas du tout pensé lorsque nous avons imaginé la structure du projet est apparue lorsque nous avons commencé à réfléchir à notre fonction `nextstate()`. Dans cette fonction, l'IA doit être en mesure de jouer tout le jeu, avec toutes les possibilités afin de générer tous les noeuds de l'arbre de jeu avant de pouvoir faire le bon choix de carte à jouer. Le problème était que certaines fonctions (comme `playcard`) devaient pouvoir être jouées **virtuellement**.

Nous avons finalement adapté nos fonctions avec une variable booléenne nommée `real`, qui permet de faire des tests et d'aller dans les bouts de codes correspondant à la situation dans laquelle l'IA doit jouer (virtuelle ou réelle). Plus concrètement, cela permet à la simulation d'éviter les print, réservés au jeu dans la console.

Voici un exemple : La fonction `Prince_power` ne doit pas être exécutée de la même manière si c'est un tour virtuel ou réel. Lors d'un tour réel, les choix doivent être proposés au joueur pour qu'il puisse jouer. Sinon, on appelle la fonction `powerPrinceAI()` qui va permettre à l'IA de faire le choix qui lui est demandé.

```
def Prince_power(activePlayer, deck_arg, real=True):
    if activePlayer.gender == "Human" and real:
        choice = input("\nWho do you want to target ? [You/Opponent]\n")

        # some code
    else:
        choice = powerPrinceAI(activePlayer)
```

Gestion de l'adversaire

Un problème que nous avons rencontré assez vite était celui de la gestion du point de vue lors du tour d'un joueur, notamment pour le cas des cartes ayant une capacité visant l'adversaire. Après avoir contourné le problème, nous avons fini par créer une variable `opponent` dans la classe `Player`.

Lorsque les deux instances d'objet de `Player` sont créées (à l'initialisation de la partie), on définit en même temps les deux joueurs comme étant adverses. Lors du tour d'un joueur, ce dernier endosse le rôle de `activePlayer`, ainsi, en cas d'utilisation d'une carte agressive, elle aura pour cible `activePlayer.opponent`.

Suite de la partie.

Ce projet nous a permis de mettre en œuvre nos compétences en programmation orientée objet, d'apprendre et de comprendre certains concepts de python qui nous étaient inconnus, de travailler à l'aide de Git sur un travail collaboratif et d'améliorer notre connaissance de l'IDE Pycharm de Jetbrain.

Finalement, nous aurons appliqué les notions vues en cours et en particulier, une dérivée de l'algorithme MinMax à un projet concret et ambitieux pour le niveau de connaissances que nous avions.

Nous avons pu aussi comprendre à quel point il est important de choisir un algorithme dimensionné exactement pour la tâche pour laquelle il a été conçu. Comme expliqué dans la partie sur les difficultés rencontrées, nous nous sommes rendu compte que choisir MinMax/negamax comme l'algorithme pour le développement de notre IA n'était pas un choix très judicieux et que certains algorithmes correspondent bien mieux au problème à résoudre dans ce projet.

Après quelques recherches, nous avons lu quelques études indiquant que l'algorithme le plus adapté à un problème à information non complet asynchrone serait l'algorithme ISMCTS¹ (Information Set Monte Carlo Tree Search).

Pour la suite du projet, il aurait donc été intéressant d'implémenter cette solution afin de comparer en quoi les deux IA diffèrent au niveau de la qualité de leur choix, du temps de recherche de solutions lorsqu'elles doivent jouer et du temps d'implémentation des deux algorithmes.

En comparant negamax et ISMCTS, nous aurions pu voir lequel est le plus adapté au cas de Love Letter.

Pour ce qui est de l'interface utilisateur, nous avions imaginé quelques prototypes rapides réalisés avec Figma afin de se donner une idée de ce que nous souhaitions implémenter.

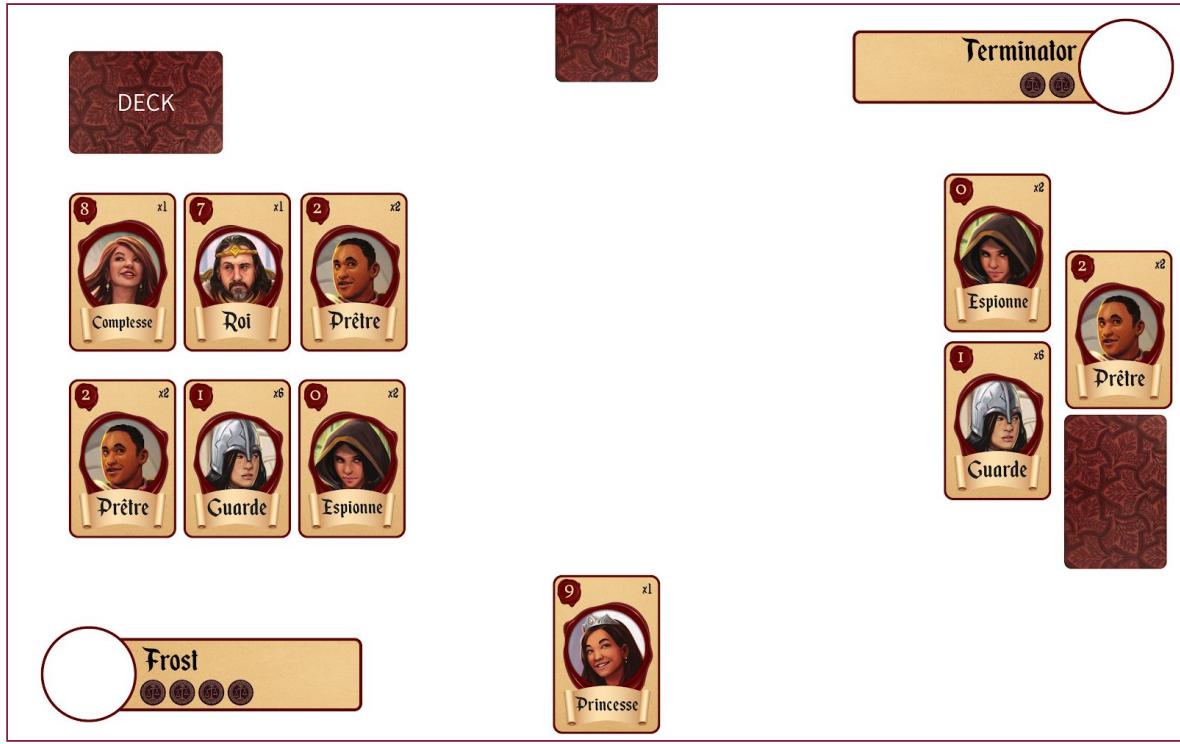
¹ Jack Reinhardt. Competing in a Complex Hidden Role Game with Information Set Monte Carlo Tree Search [online]. [ISMCTS](#) (consulté le 22/12/20)



Nous avons créé les cartes du jeu numériquement en se basant sur le jeu original.

Nous avons aussi imaginé le plateau et la page d'accueil du jeu :





Finalement, nous avons réalisé un prototype interactif pour illustrer la manière dont nous voulons qu'une manche soit jouée. Ici, le prototype représente un tour où un Baron est joué, pour l'essayer, il suffit de suivre le lien hypertexte suivant (Il est peut être nécessaire de redimensionner le prototype avec le menu “options” en haut à droite) :

[Prototype Baron](#)

Pour implémenter cette interface utilisateur, nous avons pensé à travailler avec PySimpleGUI, un framework python qui simplifie Qt, Remi et Wxpython de façon à implémenter une interface graphique plus rapidement, mais par manque de temps, nous avons préféré nous concentrer sur l'intelligence artificielle et sur la préparation du rapport écrit et oral dans le cadre de l'UV.

Annexe

[1] generateChildren

```
def generateChildren(virtualNode, next_nodes, color, knownCards, firstTurn, *simulatedCard):

    for i in range(2): # Pour chaque carte de la main

        newVirtualNode = copy.deepcopy(virtualNode)
        newVirtualNode.parent = virtualNode

        # on définit le parent du nouveau noeud

        if firstTurn:
            activePlayer = newVirtualNode.state.player
            index = -1
            # print("_____ FIRST TURN _____")

        else:
            if color == 1:
                activePlayer = newVirtualNode.state.player
            else:
                activePlayer = newVirtualNode.state.player.opponent

            # print(f"player name : {activePlayer.name}")
            index = findCard(simulatedCard[0].value, newVirtualNode.state.deck)

        if index != -1:
            activePlayer.hand.append(newVirtualNode.state.deck[index])
            del newVirtualNode.state.deck[index]

        knownCards = activePlayer.isolatedCards + activePlayer.hand + activePlayer.playedCards

        if index == -1 and not firstTurn:
            del newVirtualNode

        else:
            activePlayer.playTurn(newVirtualNode.state.deck, i, real=False)
            if not activePlayer.hand:

                # cas ou Le Prince a été joué
                for drawnCard in virtualNode.state.listOfCards:
                    n = findOccurrences(drawnCard, knownCards)

                if drawnCard.totalNumber - n > 0:
                    PrincedNode = copy.deepcopy(newVirtualNode)
                    PrincedNode.parent = virtualNode
```

```
    if color == 1:
        activePlayer = PrincedNode.state.player
    else:
        activePlayer = PrincedNode.state.player.opponent

    index = findCard(drawnCard.value, PrincedNode.state.deck)

    if index == -1 and not firstTurn:
        del PrincedNode
    else:
        activePlayer.hand.append(PrincedNode.state.deck[index])
        del PrincedNode.state.deck[index]

    next_nodes.append(PrincedNode)

elif not activePlayer.opponent.hand:

    drawnCard = newVirtualNode.state.deck.pop(0)
    activePlayer.opponent.hand.append(drawnCard)

    next_nodes.append(newVirtualNode)

else:
    next_nodes.append(newVirtualNode)
```

[2] nextStates

```
def negamax(node, depth, alpha, beta, color):
    # Création d'une copie permettant de ne pas toucher au noeud original

    virtualNode = copy.deepcopy(node)

    # Vérification du nœud (est il terminal? Est ce que la profondeur limite est atteinte?)
    if isTerminal(node) or depth == 0:
        node.value = color * evaluate(node)
        return color * evaluate(node)

    node.children = nextStates(virtualNode, color)

    bestValue = -1000000

    print(f"Number of babies : {len(node.children)} ; floor {node.floor} :")

    for child in node.children:
        # Définition de l'étage des enfants dans l'arbre
        child.floor = virtualNode.floor + 1

        # Récursivité de Negamax
        negaValue = - negamax(child, depth - 1, -beta, -alpha, -color)
        child.value = negaValue

        bestValue = max(bestValue, negaValue)

        alpha = max(alpha, negaValue)

        # Élagage Alpha/Beta

        if beta <= alpha:
            return alpha

    return bestValue
```

[3] evaluate

```
def evaluate(node):

    knownCards = node.state.player.isolatedCards + node.state.player.hand +
node.state.player.playedCards
    m = 0
    impact = 0.75

    if isTerminal(node):

        if node.state.player.isAlive:
            if node.state.player.extraPoint:
                impact = 1

        elif node.state.player.opponent.isAlive:
            impact = 0

        elif not node.state.deck:

            for Card in node.state.player.listOfCards:

                if node.state.player.hand[0].value > Card.value:

                    n = findOccurrences(Card, knownCards)
                    m += Card.totalNumber - n

            impact = m / (1 - len(knownCards))

        if node.state.player.extraPoint:
            impact += 0.5
    else:
        impact = weights(node.state.player, knownCards, False)

    # arrondi pour faciliter la lecture

    impact = round(impact, 5)
return impact
```

[4] chancellor_power

```
def chancellor_power(activePlayer, deck_arg, real=True):
    if len(deck_arg) > 1:
        k = 2
    else:
        k = len(deck_arg)
    for i in range(k):
        activePlayer.draw(deck_arg)
    print("\nThere are no more cards in the deck !" if not k else "")

    if activePlayer.gender == "AI" or not real:
        listOfIndex = powerChancellorAI(activePlayer)
        print("list of index is : ", listOfIndex)
    aiCount = 0
    p = 0

    while k != 0:
        if activePlayer.gender == "Human" and real:
            print(f"\nYou need to put {k} card(s) in the deck !"
                  f"\n\nYour hand is :\n")
            for i in range(len(activePlayer.hand)):
                print(f"{i}. {activePlayer.hand[i].title} [{activePlayer.hand[i].value}]")

            playerInput = input(f"Which card do you want to place at the bottom of the deck ? Pick
between 0 and {k}\n")
            numbers = "0123456789"
            numberList = list(numbers)
            availableIndexes = [numberList[i] for i in range(k+1)]

            while len(playerInput) != 1 or playerInput not in availableIndexes:
                playerInput = input(f"\nWrong input ! Remember, between 0 and {k}\n")

            index = int(playerInput)

        else:
            index = listOfIndex[aiCount]
            if listOfIndex[0] == 2:
                p = 1
            elif aiCount == 1 and index != 0 and p != 1:
                index -= 1
            print("index is : ", index)
            aiCount += 1

            placedCard = activePlayer.hand.pop(index)
            deck_arg.append(placedCard)
            k -= 1
```

[5] negamax

```
def negamax(node, depth, alpha, beta, color):
    # Cr ation d'une copie permettant de ne pas toucher au noeud originel

    virtualNode = copy.deepcopy(node)

    # V rification du n eud (est il terminal? Est ce que la profondeur limite est atteinte?)
    if isTerminal(node) or depth == 0:
        node.value = color * evaluate(node)
        return color * evaluate(node)

    node.children = nextStates(virtualNode, color)

    bestValue = -10000000

    print(f"Number of babies : {len(node.children)} ; floor {node.floor} :")

    for child in node.children:
        # D efinition de l' tage des enfants dans l'arbre
        child.floor = virtualNode.floor + 1

        # R cursivit  de Negamax
        negaValue = - negamax(child, depth - 1, -beta, -alpha, -color)
        child.value = negaValue

        bestValue = max(bestValue, negaValue)

        alpha = max(alpha, negaValue)

        #  lagage Alpha/Beta

        if beta <= alpha:
            return alpha

    return bestValue
```

[6] powerPrinceAI

```
def powerPrinceAI(activePlayer):
    """
    power of the Prince (AI version)
    :param activePlayer: player that is playing the Prince
    :return: The person that will be targeted with the Prince.
    """

    if activePlayer.hand[0].value in [4, 6, 9]:
        return "opponent"
    elif activePlayer.hand[0].value == 0:

        if activePlayer.extraPoint:
            return "you"
        else:
            return "opponent"

    elif activePlayer.hand[0].value == 1:

        knownCards = activePlayer.isolatedCards + activePlayer.hand + activePlayer.playedCards

        probs = []
        b = 21 - len(knownCards)

        for Card in activePlayer.listOfCards:
            n = findOccurrences(Card, knownCards)
            a = Card.totalNumber - n
            probs.append((a / b))

        impact = max(probs)

        if impact > 0.2:
            return "opponent"
        else:
            return "you"

    else:
        return "you"
```