Exposé de projet : Implémentation du jeu de cartes *Schotten-Totten* en langage *Python*

ENSTA Bretagne
2 rue F. Verny
29806 Brest Cedex 9, France

DUCASSE Quentin,
quentin.ducasse@ensta-bretagne.org

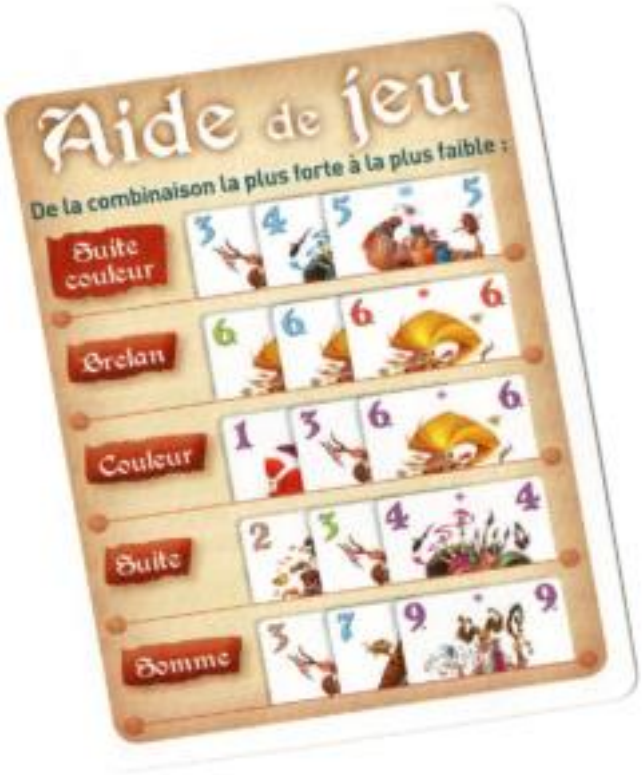
BOUVERON Matthieu,
matthieu.bouveron@ensta-bretagne.org

Sommaire

Introduction

« *Lancez-vous dans une lutte où tous les coups sont permis pour gagner le contrôle de la frontière qui vous sépare de votre adversaire. Envoyez les membres de votre tribu défendre les bornes et déployez vos forces en réalisant les meilleures combinaisons de cartes. Pour gagner, soyez le premier à contrôler cinq Bornes dispersées le long de la frontière ou trois Bornes adjacentes.* »

Telle est la présentation officielle de ce jeu de carte, dans lequel deux joueurs s'affrontent par le biais de combinaisons de cartes pour le contrôle du terrain de jeu. Le but du projet décrit dans le présent rapport est de permettre une confrontation joueur contre joueur par le biais d'un ordinateur dans un premier temps, puis une confrontation joueur contre intelligence artificielle dans un deuxième temps. Il est à noter qu'une version en ligne de ce jeu permettait une confrontation entre joueurs, mais a été retirée pour des questions de droits d'auteur.



I. Analyse du problème

a. Analyse générale

La procédure d'écriture du code se sépare en différentes phases. Il faut tout d'abord chercher à cerner le fonctionnement du jeu en assimilant ses règles et le comportement des joueurs au cours des différents instants de jeu. On en vient alors à imaginer un partitionnement du jeu autour de différents objets corrélés entre eux, qui engendreront diverses classes s'appelant les unes les autres par des jeux d'héritages et de complétion. On s'intéresse ensuite à la décomposition en fonctions des différentes actions qui peuvent être réalisées par les joueurs ainsi qu'à la synchronisation de ces actions avec le déroulement du jeu. Pour s'assurer du bon fonctionnement du code dans une majorité de cas de figures, il devient donc nécessaire d'implanter des tests unitaires qui vérifieront que les retours de chaque fonction sont bien ceux attendus.

Dans le cadre de ce projet, nous nous intéresserons exclusivement à la construction du jeu selon des règles légèrement simplifiées qui ne le modifient pas exagérément. Ainsi, on ne considérera pas les règles optionnelles concernant les cartes à effets spéciaux, ni la règle précisant que l'on peut démontrer que l'adversaire ne peut pas remporter une borne pour la revendiquer même s'il a posé moins de trois cartes.

Une fois toutes ces fonctionnalités réalisées, le jeu *player v player* devrait fonctionner convenablement et permettre à deux joueurs de s'opposer avec une interface graphique basique. On s'intéressera alors au développement d'une intelligence artificielle, ou plutôt de plusieurs intelligences artificielles de niveaux différents, permettant à un joueur de s'entraîner seul. Une première *IA* servira de référence en posant aléatoirement ses cartes. Les *IA* développées ensuite seront progressivement plus performantes en appliquant des stratégies de calcul plus élaborées.

Par ailleurs, on s'intéressera au développement d'une interface homme-machine graphiquement plus agréable que la console *Python*, autorisant une véritable ambiance de jeu. Elle intégrera autant que possible les éléments du jeu d'origine et un mode de jeu instinctif.

b. Objectifs

Les objectifs fixés sont donc :

- Compréhension du fonctionnement du jeu (**OK**)
- Mise en place d'un environnement de jeu entre deux joueurs avec interface minimaliste dans la console Python (**OK**)
- Mise en place d'un choix entre différents modes de jeu permettant de choisir entre PvP, PvIA ou IAvIA (**OK**)
- Développement d'un premier niveau d'IA (**OK**)
- Réflexion sur les stratégies plus poussées inhérentes au changement de règle imposé (*à effectuer*)
- Développement de plusieurs IA plus compétentes (2 supplémentaires + 1 incomplète)
- Créer un système de sauvegarde d'une partie en cours (**OK**)
- Créer un environnement graphique plus confortable que la console *Python* de base, en utilisant *QTDesigner*. (**Incomplet**)

c. Figures imposées

Les 4 conditions prérequises sont respectées :

1. Factorisation du code : au moins trois modules et noms de classes distincts
2. Documentation et commentaires du code
3. Tests unitaires : (au moins 4 méthodes avec au moins 2 cas testés par méthode)
4. Création d'un type d'objet (classe) : il devra contenir au moins deux variables d'instance

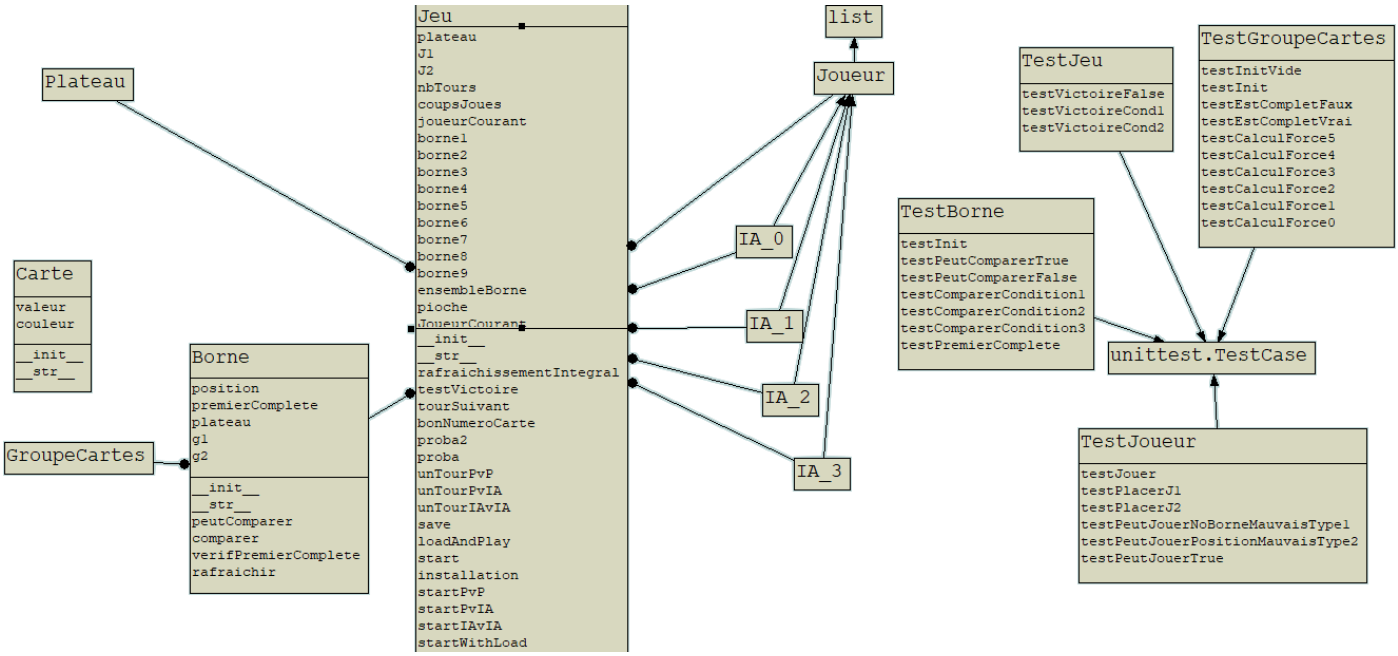
De plus, nous avons choisi de respecter les figures imposées suivantes :

- Héritage au moins entre deux types créés — *Joueur et IA*
- Héritage depuis un type intégré (hors en IHM) — *Joueur et List*
- Lecture/ écriture de fichiers — *Sauvegarde d'une partie*

II. Description des classes mises en jeu

Le code que nous avons écrit permet à deux joueurs de s'opposer par clavier interposé sur ce jeu de carte, en bénéficiant d'une interface graphique plus confortable que les consignes précédemment tapées directement en console. De plus, il est possible de jouer seul face à différents algorithmes capables d'opposer des niveaux de jeu variés. Enfin, il est possible de comparer des *'Intelligences Artificielles'* entre elles en les faisant jouer les unes contre les autres.

Les différentes classes mises en jeu et les relations qui les lient sont synthétisées dans la figure suivante, obtenue en utilisant l'outil *PyNSourceGui*. Malheureusement, les méthodes et variables d'instance des classes *GroupeCartes, Joueur* et *IA* n'apparaissent pas pour des raisons obscures.



a. Classe *Jeu*

La classe *Jeu* centralise l'objet et les commandes qui assurent le bon déroulement général du jeu, telles que la situation globale des bornes, le changement de tour de jeu, la pioche ou la vérification des conditions de victoire. Ses principales variables d'instance sont le plateau, les joueurs l'ensemble des bornes, la pioche et une trace du joueur courant.

b. Classe *Borne*

La classe *Borne* rassemble des données locales sur les actions effectuées autour d'une borne donnée. L'objet *borne* comprend ainsi les cartes posées sur les emplacements liés à cette borne par chacun des deux joueurs ainsi que d'autres informations (en particulier, la première complétion d'un groupe de cartes). Ses méthodes agissent sur les groupes de cartes situés de part et d'autre de la borne. Ses variables d'instance principales sont les deux groupes de cartes de part et d'autre de la borne ainsi que sa position.

c. Classe *GroupeCartes*

GroupeCartes regroupe les cartes en vérifiant si le groupe est complet ou non. Le cas échéant, les méthodes de cette classe permettent l'affichage du groupe et sa force. Les principales variables d'instance d'un groupe de cartes sont les trois cartes qui le composent ainsi que sa force, caractérisée en utilisant la classification proposée par les règles du jeu.

d. Classes *Carte* et *Plateau*

La classe *Carte* donne les caractéristiques des cartes, à savoir leur valeur et leur couleur, ainsi qu'une méthode d'affichage. La classe Plateau permet notamment la création de l'objet plateau, qui est utilisé tout au long de la partie pour la disposition des cartes. La méthode d'affichage qui s'y trouve sert de base à l'interface graphique actuelle. Ces deux classes sont les plus courtes du programme et en sont la base, elles ne possèdent qu'une méthode d'initialisation et une d'affichage.

e. Classe *Joueur*

La classe *Joueur* est, avec *Jeu*, l'une des deux grandes classes décrites dans le code actuel. Elle est complémentaire à la première dans le sens où elle recense les caractéristiques des joueurs (le numéro, le contenu de leur main) en faisant le lien avec la partie en cours par des méthodes qui l'affectent directement (*placer, piocher*) ou assurent certaines vérifications (*peutJouer()*). Cette carte hérite de List, et en effet, un joueur est intégralement représenté par sa main et son numéro.

f. Classes relatives aux 'Intelligences Artificielles'

Toutes les classes d'*Intelligence Artificielle* héritent de la classe *Joueur* et jouent bien au moment où un joueur réel le ferait face à son adversaire. Ainsi, les affichages se font de la même manière, que ce soit le tour d'un joueur ou de l'ordinateur. Toutefois, les demandes de choix sont désactivées et le traitement se fait directement dans des méthodes redéfinies par polymorphisme à partir de celles de *Joueur*. Cela est rendu possible en particulier par l'utilisation des méthodes *UnTourPvIA()* et *UnTourIAvIA()*.

Les différentes IA sont numérotées par ordre croissant de niveau de difficulté, chacune se basant sur une stratégie différente de la précédente et étant supposée remporter quasi-systématiquement la victoire face à elle du fait de la réalisation de meilleures combinaisons.

i. Classe IA_0

Premier niveau d'IA programmé, cette classe sert de référence à l'évaluation des performances de l'IA1. À partir de l'ensemble de cartes qui lui est attribué comme main, l'IA en choisit aléatoirement une et la place sur un emplacement aléatoire dans la zone de jeu qui lui est attribuée. Le placement se fait cependant bien dans l'ordre de remplissage des bornes, du plus proche de la borne au plus éloigné, par l'appel à la variable *carteCourante* dans la méthode *placer()*.

ii. Classe IA_1

Suivant une stratégie simple, l'IA1 cherche à réaliser un total de points plus important que la somme des points des cartes posées par l'adversaire sur une borne, sans prendre en compte la

possibilité de réaliser des combinaisons. Telle qu'implantée actuellement, cette stratégie autorise environ 80% de victoire face à l'IA0, le résultat étant légèrement meilleur si IA1 joue en tant que joueur 1.

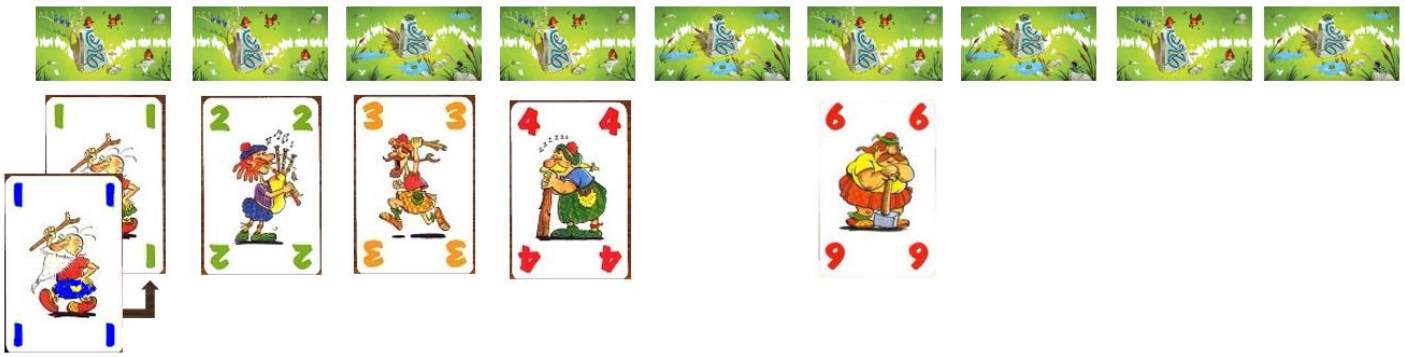
Le choix de la carte et de la borne se fait dans la fonction *placer***()**. La première étape du traitement est de trier les cartes par ordre croissant de valeur dans la 'main' de l'IA en modifiant directement la liste *self*. Pour rappel, les IA héritent de la classe *Joueur*, qui hérite elle-même de *List*.

Le numéro de borne est choisi en considérant l'ensemble des bornes où l'adversaire compte plus de points que l'IA, puis en ne retenant que celle où cet écart est le plus faible. Connaissant cet écart, il ne reste plus qu'à parcourir la main de l'IA jusqu'à en trouver une qui permette de renverser l'écart en sa faveur. Si cela est impossible, alors une carte et une borne sont choisies aléatoirement comme pour l'IA précédente.

iii. Classe IA_2

Celle-ci repose sur une stratégie performante d'après le classement des puissances des différentes combinaisons du jeu : l'établissement de brelans (3 cartes de même valeur). C'est également une stratégie qui émerge naturellement dans les premières parties sur jeu de carte réel humain contre humain (malgré l'apparition de quelques suites couleur). Cela autorise 100% de victoire face aux deux IA précédentes.

Cette stratégie est particulièrement simple à mettre en place dans une IA en lui faisant placer les cartes de valeur X sur la borne indexée par X. En effet, il y a 9 valeurs possibles pour les cartes et autant de bornes.



Pour le choix de la carte, et donc de la borne, on parcourt la main de l'IA en cherchant une carte que l'on peut poser sur la borne correspondante. Si toutes les bornes correspondantes sont déjà remplies, alors on choisit au hasard la carte et la borne.

iv. Classe IA_3

Cette IA a pour but de favoriser l'apparition des meilleures combinaisons sur chaque borne, en s'adaptant aux évolutions du jeu. Ainsi, une fonction de recherche des débuts de combinaisons est mise en place, qui reconnait à la fois sur le plateau et dans la main du joueur les cartes qui peuvent se combiner, même s'il manque une carte pour compléter la combinaison.

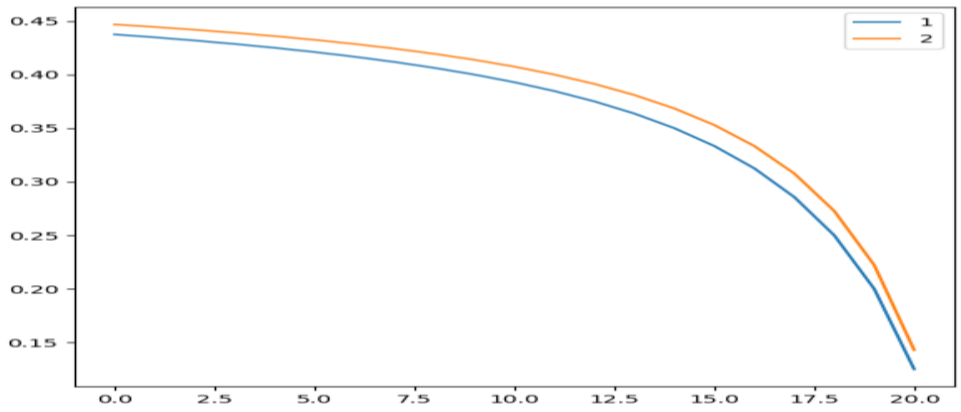
Le choix de considérer une combinaison comme réalisable même s'il manque une carte repose sur l'utilisation d'une fonction de calcul de la probabilité d'obtention d'une carte qui n'est pas encore apparue, qui se trouve donc soit dans la main de l'adversaire soit dans la pioche.

La méthode *proba()* :

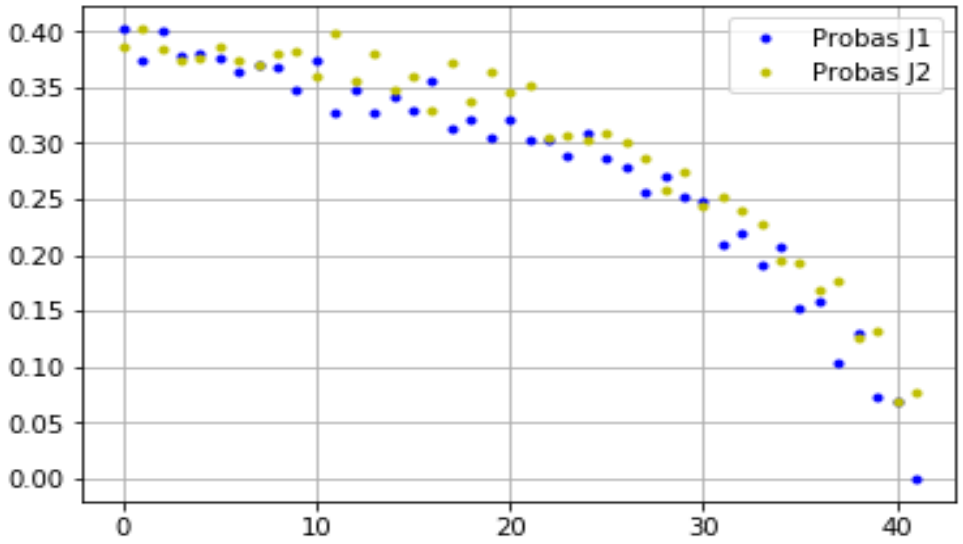
Elle calcule les probabilités (égale à 1 si la carte est déjà dans la main) d'obtenir chacune des cartes du jeu entre le tour en cours et la fin du jeu. L'intérêt est de récupérer par traitement simple les probabilités d'obtenir une couleur ou une valeur donnée. Les probabilités sont regroupées sous forme d'un tableau de la forme suivante :

```
  | 1 2 3 4 5 6 7 8 9
__|___________________
A | X X X X X X X X X
B | X X X X X X X X X
C | X X X X X X X X X
D | X X X X X X X X X
E | X X X X X X X X X
F | X X X X X X X X X
```

Ces probabilités sont calculées par application d'une formule de probabilités composées et indique les probabilités suivantes pour l'obtention d'une carte donnée, exprimée en fonction du nombre de tours de pioche passées :



Il en ressort que le joueur 2 a légèrement plus de chances que le joueur 1 d'obtenir une carte n'étant pas encore apparue. Bien que ce résultat soit quelque peu surprenant au premier abord, il est globalement confirmé par des mesures réalisées sur un grand nombre d'expériences (divisez les abscisses par 2 pour la correspondance entre les graphiques):



Après avoir déterminé quelle combinaison favoriser, l'IA pose une carte de manière à en compléter une ou bien joue de manière à ce qu'une combinaison ait une bonne probabilité d'apparaître par la suite, en préférant l'établissement de suites couleur.

g. Classes de test

Les trois classes possédant des méthodes testées actuellement sont *GroupeCartes*, *Borne* et *Joueur*. Arriveront plus tard des tests sur les classes *Jeu* et les différentes IA (à implémenter)

III. Description des fonctions et méthodes les plus importantes

Voici une description des méthodes les plus importantes de notre projet, réparties par classe :

a. Classe *GroupeCartes*

-estComplet()
Vérifie que chacune des trois cartes composant le groupe est différente d'un emplacement vide, défini comme base du plateau et en tant que Carte(0,'X'). Cette méthode compare les str() des cartes et vérifie qu'ils diffèrent de ' '. Renvoie True ou False.
-calculForce()
Première condition : Si une des cartes a pour valeur 0 (c'est-à-dire la carte Carte(0,'X') correspondant à un emplacement vide.
Deuxième condition : *Couleurs identiques* et *valeurs consécutives* correspondant à une force de **5**. Pour vérifier que les valeurs sont consécutives, on utilise la condition min(l)==(max(l)-2)) avec l la liste des valeurs des trois cartes.
Troisième condition : *Couleurs différentes* et *valeurs identiques* correspondant à une force de 4
Quatrième condition : *Couleurs identiques* et *valeurs sans lien apparent* pour une force de 3
Cinquième condition : *Couleurs sans lien apparent* et *valeurs consécutives* pour une force de 2
Enfin, si aucun des cas ci-dessus n'est apparu on accorde au groupe la force de 1

b. Classe *Borne*

-peutComparer()
Calcule tout d'abord les forces de chacun des groupes de la borne avec *calculForce***()**. Ensuite, la méthode vérifie que chacun des groupes est complet via la méthode *estComplet***()** appliquée à g1 et g2.
-comparer()
Si la méthode *peutComparer***()** est vérifiée, on compare la force des deux groupes autour de la borne. En cas d'égalité, on compare le total des points de chacun des groupes de cartes et si on a de nouveau affaire à une égalité, c'est le premier joueur à complété sa borne qui gagne la borne.

-verifPremierComplete(jeu)
Cette fonction est uniquement utilisée au moment de placer une carte via *Joueur.placer()* et permet d'attribuer la variable d'instance *Borne.premierComplete* au joueur correspondant.

-rafraichir()
Remet à jour les cartes dans la borne et donc dans chacun des groupes de cartes g1 et g2 par rapport aux cartes enregistrées dans le plateau.

c. Classe *Joueur*

-placer(no_carte,position)
Remplace la Carte(0,'X') (emplacement vide) du plateau par la carte sélectionnée dans la main du joueur. Ensuite, la méthode procède à un rafraîchissement intégral des bornes du plateau, à la *verifPremierComplete(jeu)* sur la borne en question puis à *comparer()* su cette même borne.

-peutJouer(position)
Vérifie que l'emplacement visée par la fonction *jouer()* (et donc *placer()*) est bien disponible ; c'est-à-dire que la position entrée est bien du bon côté du plateau, n'est pas sur une borne, que le type de cette position correspond bien à un tuple et que l'emplacement est libre (égal à Carte(0,'X')).

-piocher()
Ajoute la première carte de la pioche à la main du joueur dont c'est le tour.

-jouer(no_carte, position)
Combinaison des fonctions *placer(no_carte, position)* et *piocher()*.

d. Classe *Jeu*

-testVictoire()
Vérifie les deux conditions de victoire, c'est-à-dire 5 bornes appartiennent à un joueur ou 3 consécutives. Pour ce faire, on prend etatBornes qui correspond à la troisième liste du plateau, les bornes initialement affichées comme 'XX', plus tard remplacées par 'J1' ou 'J2'.

-unTourPvP()
Affichage du plateau et de la main du joueur en cours. Ensuite, deux *input()* se suivent pour obtenir la carte sélectionnée et la position (vérifiée à l'aide de *peutJouer(position)*. On rafraîchit ensuite les bornes du jeu et on utilise la fonction *tourSuivant()* qui incrémente le nombre de tours et change le joueur courant.

-start()
Lance le jeu en demandant la sélection d'un mode : PvP, PvIA (à implémenter) ou IAvIA (à implémenter) et ensuite le(s) niveau(x) de(s) IA. On distribue *Joueur.taille* carte à chaque joueur et on lance une boucle sur *unTour()* ne s'arrêtant qu'en cas d'absence de cartes dans la main des joueurs ou de la victoire d'un des joueurs.

-save()
Sauvegarde dans un fichier les différents objets représentant notre jeu à l'aide du module *pickle*. Initialement, nous avions sauvegardé les représentations graphiques des objets via *str()* puis permis la relecture de ces différents objets par une fonction spécifique mais le 'pickler' est nettement plus simple et permet des résultats similaires.

-**loadAndPlay(fichier) :**
Permet la reprise de la partie en cours à partir d'un des fichiers de *saves*.

IV. Description des tests et résultats

La majorité des méthodes listées ci-dessus sont testées afin d'optimiser la détection d'erreurs en cas de changement ou de vérifier le bon comportement de notre code.
Pour l'instant, ce sont les classes Joueur, GroupeCartes et Borne dont les méthodes sont testées avec notamment :

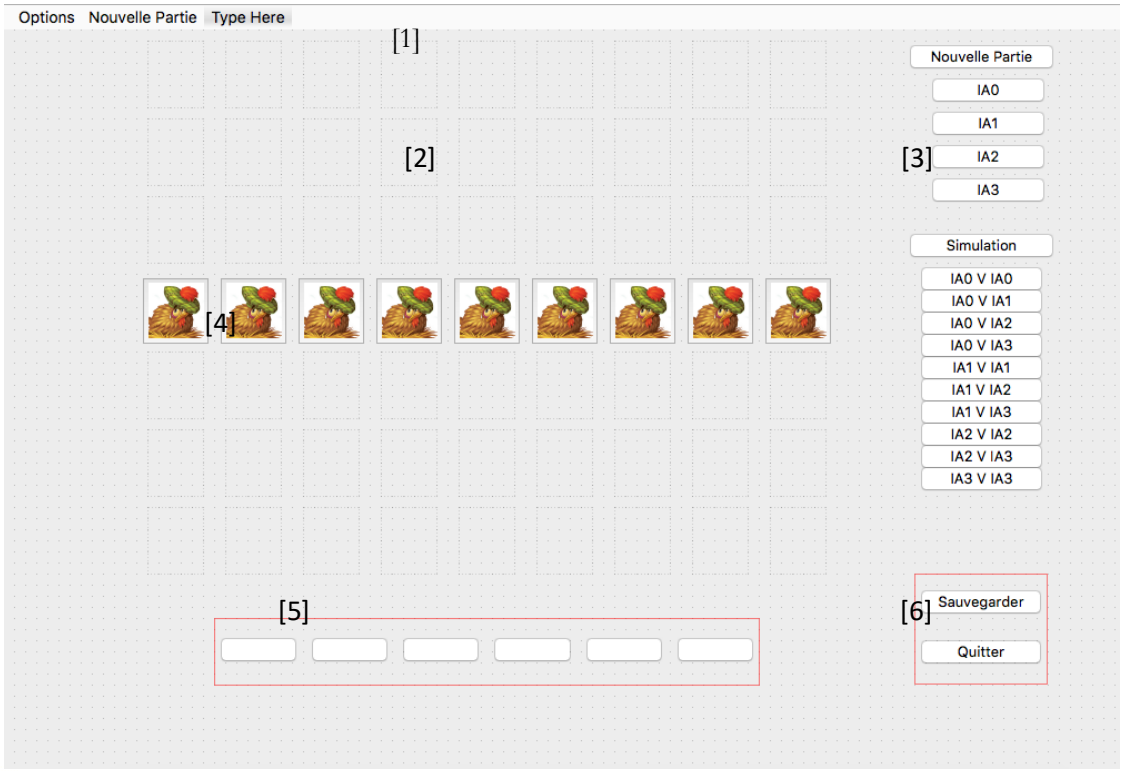
TestGroupeCartes :
-Deux tests **d'initialisation**, un avec des 'cartes vides' et un avec des cartes normales
-Deux tests de *estComplet()* : un True et un False
-Six tests de *calculForce***() :** un pour chacune des configurations (de 0 à 5)

TestBorne :
-Un test **d'initialisation**
-Deux tests de *peutComparer()* : un True et un False
-Trois tests de *comparer()* : un dans chaque cas d'égalité (pas d'égalité, égalité, égalité du total de points)
-Un test de *premierComplete()*

TestJoueur :
-Plusieurs tests pour *placer()* (à finaliser)
-Deux tests pour *peutJouer()* (à implémenter)
-Un test pour *piocher()* (à implémenter)

Restent à implémenter des tests sur le comportement de *Jeu()* en général ainsi que sur les futures IA.

V. IHM

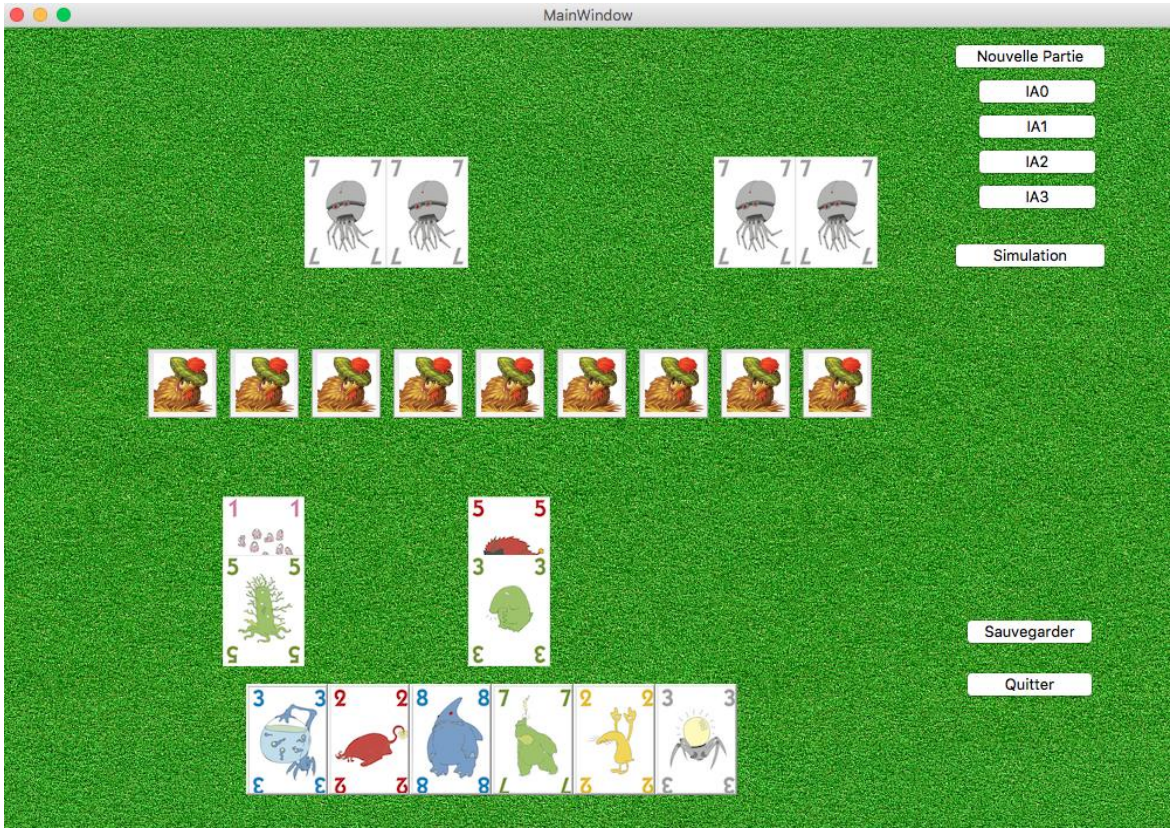


La mise en place de l'IHM n'a pas permis de rendre un modèle confortable. Cependant, voici ci-dessus le schéma idéal souhaité.

Avec ainsi :

[1] La barre de Menu permettant Options(Quitter ou Charger une partie) et le lancement d'une nouvelle partie selon les critères désirés à l'aide des menus déroulants
[2] Les 27 conteneurs de chaque côté servant de réceptacles pour les cartes
[3] Le menu à l'aide de boutons (servant d'alternative au menu supérieur qui ne fonctionne pas)
[4] Les bornes, seuls boutons du plateau qui servent à jouer une carte de notre main
[5] La main du joueur, composée uniquement de boutons poussoirs
[6] Les fonctions de sortie et de sauvegarde qui peuvent être utilisée avec ces deux boutons

On associe le tout à l'aide d'une variable globale qui retient la carte en cours de placement et met à jour le plateau en conséquence. Cependant, le tour des IA n'est pas reconnu et ne permet donc pas de procéder à une partie complète. L'ensemble des images utilisées est disponible dans le dossier IHM du fichier fourni.



Les deux modes différents permettent d'effectuer une partie contre une IA au tour par tour ou une simulation dont le plateau résultant nous est affiché. On constatera sur la capture d'écran quelques soucis de l'IA. Un meilleur contrôle des différentes actions que le joueur peut faire pour ainsi permettre d'en interdire certaines aurait été intéressant.

VI. Perspectives

Au stade actuel, ce projet présente encore quelques lacunes. En effet, la règle que nous avons omise, à savoir la possibilité de déclarer une borne avant que l'adversaire n'ait posé ses trois cartes s'il ne peut plus la remporter, peut présenter un intérêt crucial dans la stratégie à adopter au cours du jeu. De plus, l'implantation des cartes spéciales et de leurs règles associées permet de complexifier le jeu et donc de le rendre plus distrayant que la version de base que nous avons implémentée. Enfin, le nombre de niveaux associés à l'ordinateur en tant qu'adversaire est limité et il pourrait être intéressant de développer plus cet aspect du projet.

Toutefois, le projet que nous proposons présente l'intérêt d'être relativement léger d'utilisation pour le joueur, y compris lors du jeu en console puisqu'il n'a réellement qu'à choisir un numéro de carte de sa main et la borne sur laquelle la placer pour que le programme puisse extraire toutes les informations nécessaires. De plus, il est possible de quitter une partie en cours et d'y revenir plus tard grâce au système de sauvegarde qui accompagne le jeu.

Dans une optique de poursuite du projet, il serait bon de régler les défauts présentés précédemment et d'apporter encore plus de souplesse au programme, en faisant en sorte que toutes les IA ainsi que fonctionnalités de l'IHM soient opérationnelles.

Ce projet nous a cependant permis d'effectuer un travail d'ampleur dans l'environnement de Python et ce sur bien des aspects : Design du jeu dans la console Python, Mise en place des différents niveaux d'IA, Mise en place d'une IHM à l'aide de Qt et QtDesigner, Réflexion sur les différentes fonctionnalités d'un jeu –même simple comme celui-.