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.

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 ultérieurement 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 ou PvIA (*à finaliser*)
- 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 (*à effectuer*)
- Créer un système de sauvegarde d'une partie en cours (*à effectuer si possible*)

c. Figures imposées

Les 4 conditions prérequises sont respectées :

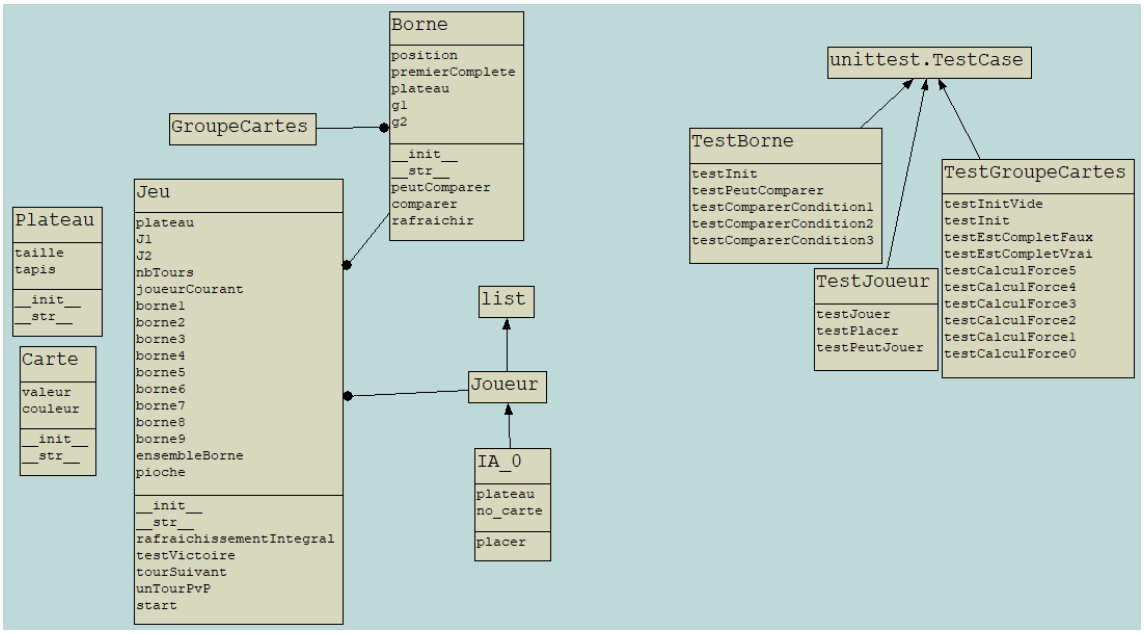
1. Factorisation du code : au moins trois modules et noms de classes distincts
2. Documentation et commentaires du code
3. Tests unitaires : (au moins 4 méthodes avec au moins 2 cas testés par méthode)
4. Création d'un type d'objet (classe) : il devra contenir au moins deux variables d'instance

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 écrit à ce jour permet le jeu à deux joueurs avec une interface minimaliste. Toutes les actions de jeu peuvent être convenablement réalisées et les conditions de victoire sur une borne ou sur le jeu sont fonctionnelles, mais les commandes se font encore via la console *Python*. 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 et Joueur 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.

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. Classe *IA_0*

Premier niveau d'IA programmé, cette classe hérite de Joueur et modifie à l'aide du polymorphisme la fonction **placer()** pour la prédéterminer en un choix aléatoire. Cette classe ira de pair avec la méthode **unTour()** modifiée pour ne pas demander graphiquement à l'IA son choix de carte et de position (à implémenter)

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.

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. Work In Progress

Plusieurs gros objectifs sont encore à concevoir :
-La mise en place des différents modes de jeu
-Différents niveaux d'IA
-Une possibilité de sauvegarder une partie en cours
-Une interface graphique interactive et en cohérence avec notre code déjà écrit