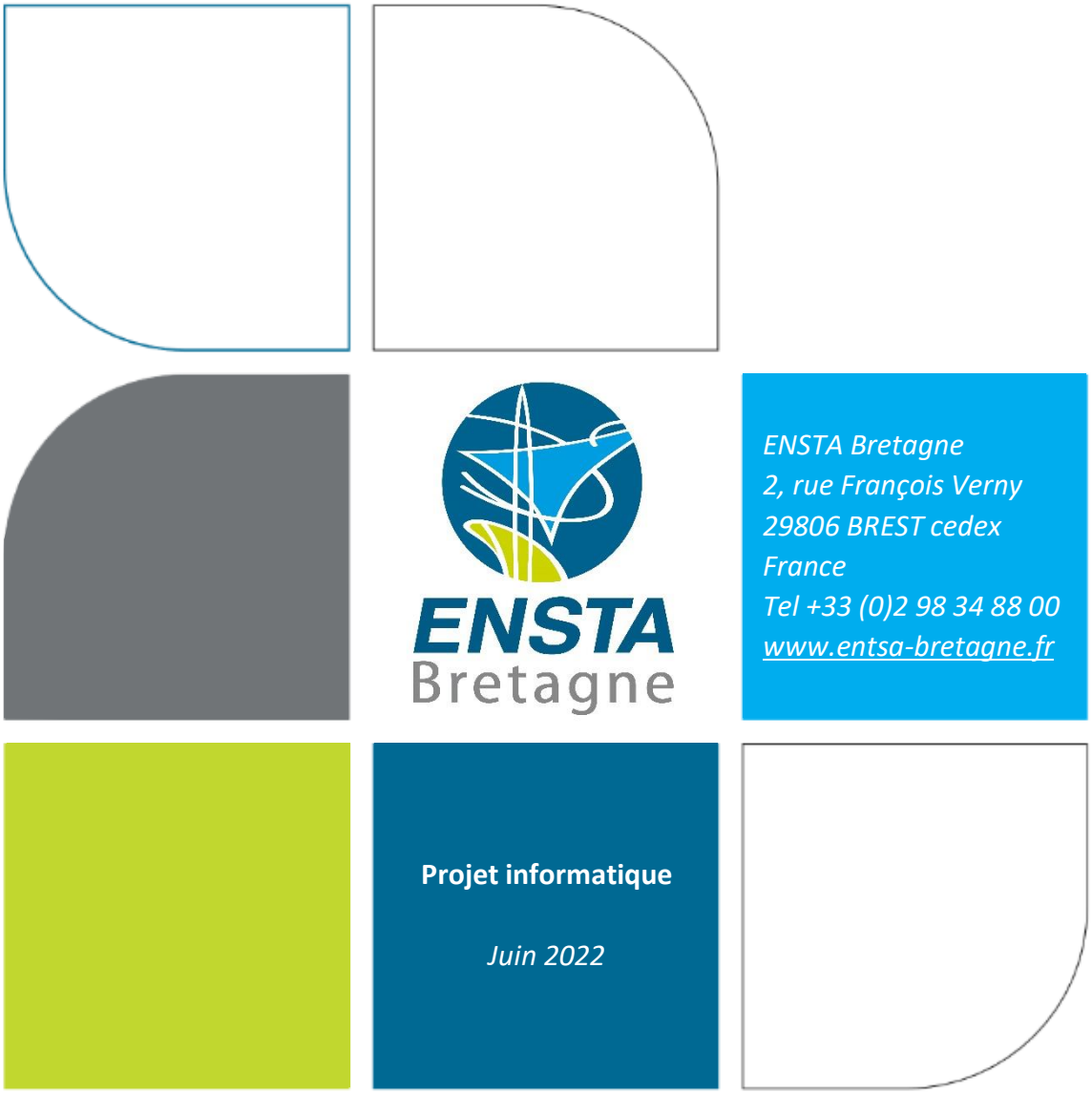


Sujet 4 – Les Aventuriers du rail

Rapport final

NOEL Océan, LEVRIER-MUSSAT Gautier
ocean.noel@ensta-bretagne.org
gautier.levrier-mussat@ensta-bretagne.org

Sommaire

1. Introduction

Le but de notre projet était de réaliser une version numérique du jeu « Les Aventuriers du rail » en python en implémentant une IHM, une IA et la possibilité à un joueur de jouer.

Les Aventuriers du rail est un jeu de société dans lequel il faut avoir un maximum de points pour gagner. Pour cela, il faut relier des villes entre elles avec des wagons et en fonction des objectifs à réaliser, on gagne plus ou moins de points. Les règles du jeu sont disponibles sur https://www.jeuxavolonte.asso.fr/regles/les_aventuriers_du_rail.pdf.

Notre jeu est une version moins complète que le vrai jeu, il y a une seule carte possible, avec 11 villes et 20 routes, que nous avons créés en suivant le modèle de la figure1.

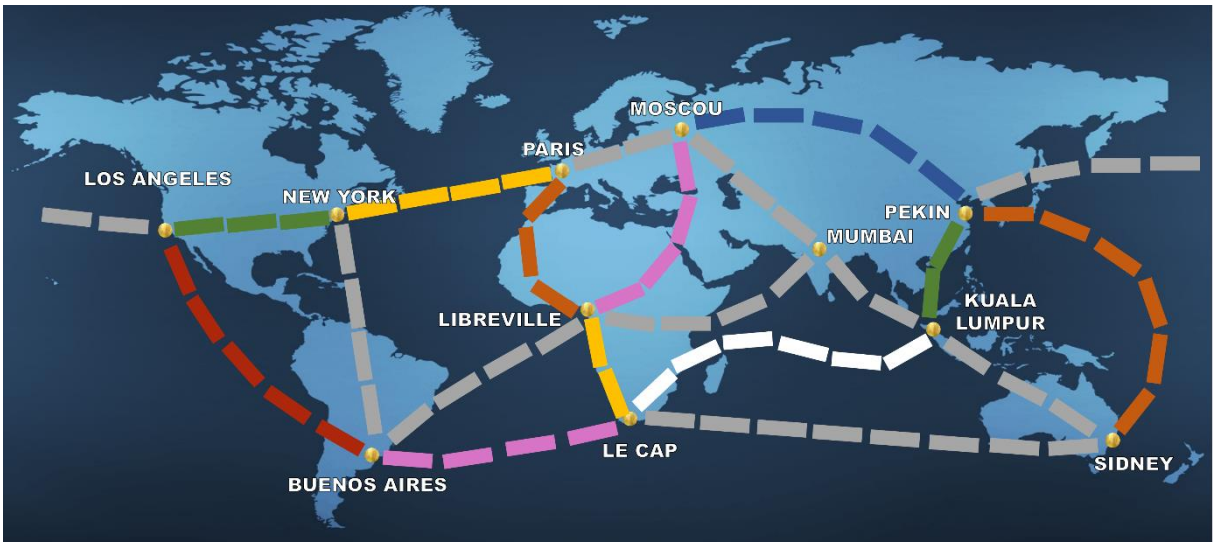


Figure 1 : Carte du plateau de notre jeu

Ensuite, nous avons implémenté 14 cartes destination ou objectif et 108 cartes wagon de couleurs différentes en respectant les quantités du jeu de base, un inventaire de nos cartes est disponible en annexe.

De plus notre jeu se joue uniquement contre une IA, nous n'avons pas implémenté la structure pour supporter une version joueur-joueur, cependant nous avons implémenté toutes les actions possibles du vrai jeu dans notre projet.

2. Description du programme

2.1. Structure du programme

Structure principale

Tout d'abord, notre projet est séparé en 5 modules :

- objects.py (il contient toutes les classes créées)
- functions.py (il contient les différentes fonctions créées)
- game.py (il contient le programme d'une partie avec l'initialisation des objets)
- menu.py (il contient le programme du menu)
- tests.py (il contient les tests unitaires)

Pour réaliser notre adaptation numérique, nous avons donc implémenté la structure représentée figure 2 :

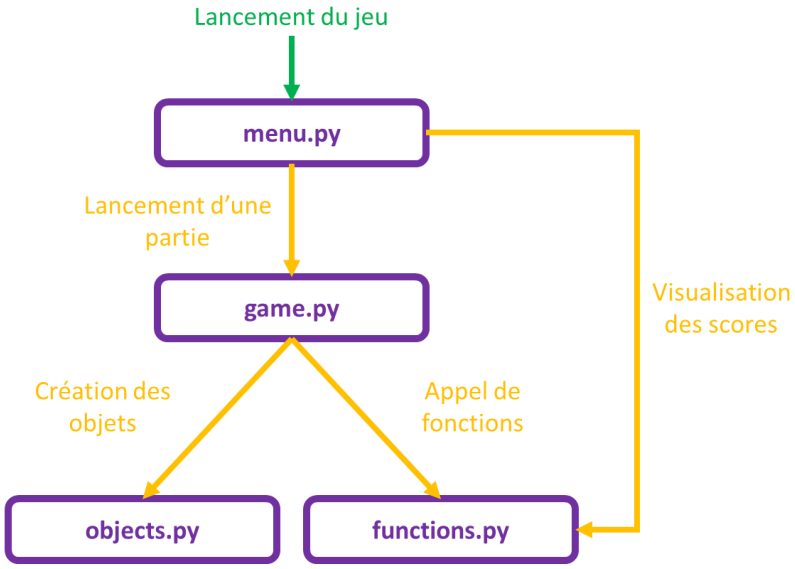


Figure 2 : Diagramme d'appel de nos modules

Ainsi, le joueur arrive sur un menu dans lequel il peut : lancer une partie avec deux difficultés aux choix, visualiser les scores des parties précédentes, voir les règles du jeu, accéder aux options ou encore quitter le jeu.

S'il décide de lancer une partie, il peut choisir la difficulté « Débutant » ou la difficulté « Intermédiaire ». Dans le premier cas, le joueur affronte une IA qui fait des choix aléatoires, dans le deuxième cas, l'IA choisis ses coups en fonction de ses cartes objectif et optimise ses trajets.

Pour lancer une partie, le module « menu.py » importe le module « game.py » et lance une fonction game() avec le niveau choisi en paramètre qui structure les tours de jeu. Cette fonction appel d'autres fonctions player_turn() et IA_turn() qui définissent le programme à exécuter au tour du joueur et de l'IA respectivement, puis elle vérifie si des conditions de fin de partie sont atteintes à chaque tour, la figure 3 schématise le principe.

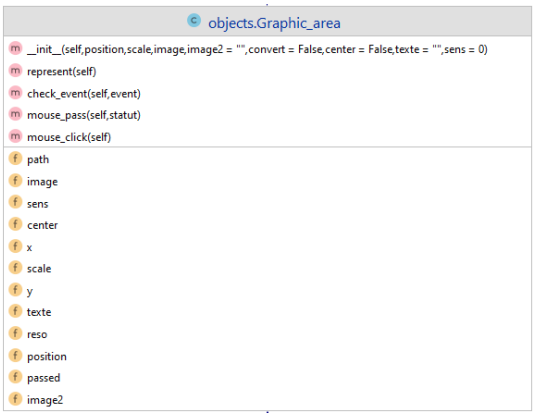
Figure 3 : Schéma du déroulement d'une partie

Nous avons décidé de terminer la partie lorsqu'un des joueur (IA ou vrai joueur) ne peut plus prendre aucune route sur le plateau, donc quand son nombre de wagons est inférieur à la longueur de la route la plus petite disponible.

Enfin, pour réaliser notre projet nous avons eu besoin de créer 9 classes, nous présentons brièvement ces classes dans ce rapport, mais nous vous invitons à consulter notre documentation html pour plus

de précisions. Cette documentation est accessible via la méthode décrite en partie 5 de ce rapport ou dans le fichier « README.md » du projet transmis avec ce rapport.

Graphic_area()



Classe mère de tous les objets qui sont affichés dans l'IHM, elle définit des méthodes d'initialisations et d'utilisations communes à tous ces objets.

Par exemple elle définit la méthode represent() qui permet d'afficher un objet.

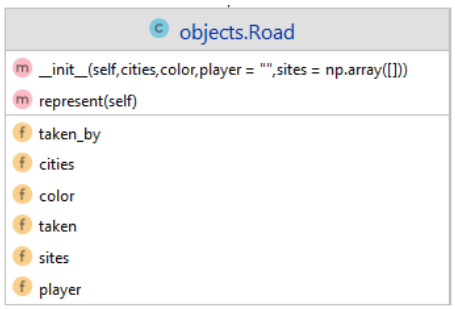
Card(Graphic_area)

Classe qui décrit l'objet carte, qui peut être soit une carte destination, soit une carte wagon. Elle hérite de la classe Graphic_area puisque certaines cartes sont affichées à l'écran pour que le joueur puisse les piocher.

Draw_pile(Graphic_area)

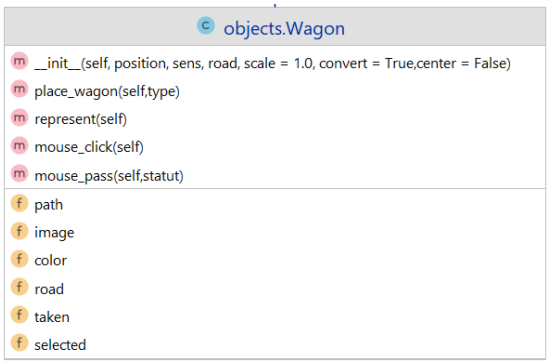
Classe qui décrit l'objet « paquet de carte », elle permet de définir les différentes pioches et mains des joueurs. Elle hérite aussi de Graphic_area car le joueur peut interagir avec les pioches sur l'IHM.

Road()



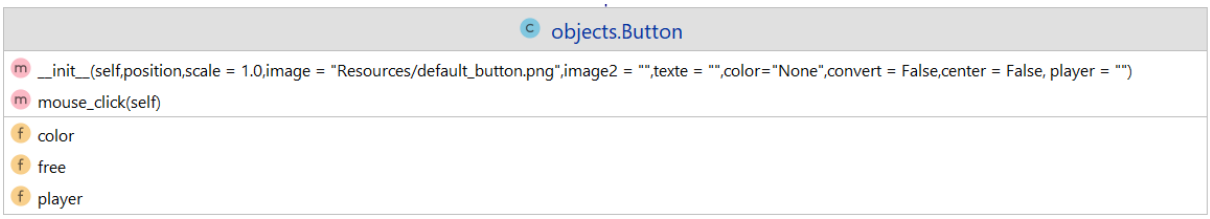
Classe qui permet de créer des routes, chaque route est constituée d'un couple de deux villes, qu'elle relie, d'une couleur et d'une liste d'emplacements correspondant aux cases où le joueur peut poser ses wagons pour prendre la route.

Wagon(Graphic_area)



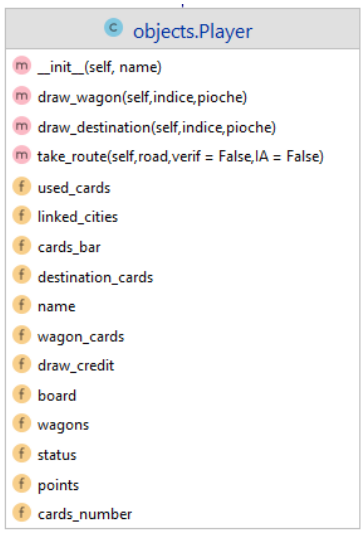
Cette classe permet de créer les emplacements qui appartiennent à une route, chaque emplacement peut être affiché grâce à l'héritage de Graphic_area et on peut donc représenter nos routes de cette manière. De plus chacun des emplacements sont interactifs pour que le joueur puisse cliquer dessus s'il veut prendre la route.

Button(Graphic_area)



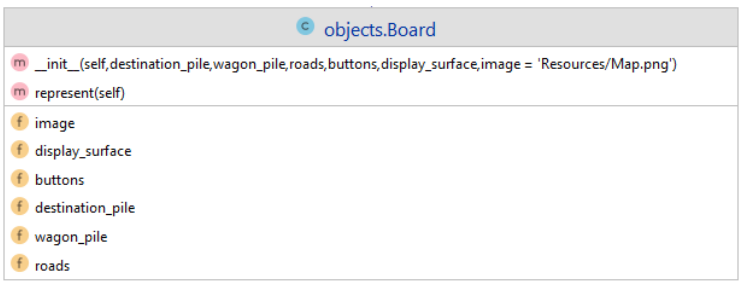
Cette classe permet de créer des boutons interactifs avec du texte et une image dans notre IHM pendant une partie, elle hérite naturellement de la classe Graphic_area.

Player()



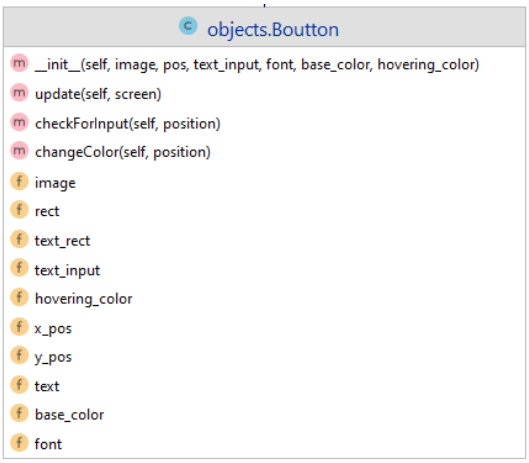
Classe qui définit un joueur, elle permet de créer un joueur, de lui donner des cartes et de lui faire réaliser différentes actions (piocher, prendre une route).

Board()



Classe qui définit le plateau de jeu, elle permet de créer un plateau de jeu avec ses pioches, ses routes et ses boutons.

Boutton() :



Classe spécifique pour les boutons du menu. Elle permet de créer des boutons interactifs pour le menu.

On obtient donc le diagramme de classe globale suivant :

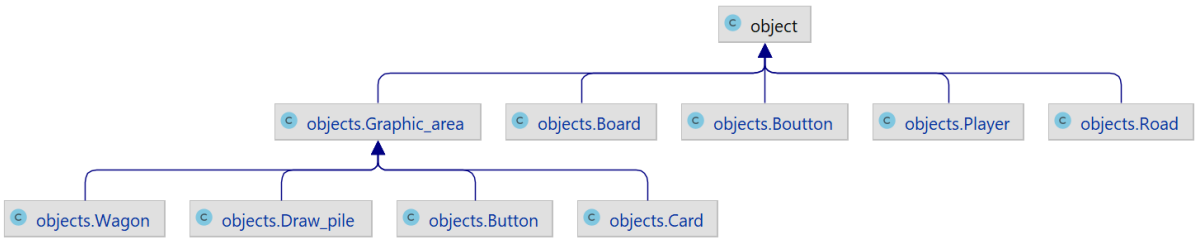


Figure 4 : Diagramme de classe du projet

2.2. Implémentation des actions du joueur

Lors d'une partie et à chaque tour, le joueur peut soit piocher deux cartes wagons, soit piocher des nouvelles cartes objectif, soit prendre possession d'une route. Nous avons implémenté ces trois actions dans notre jeu. Nous avons attribué au joueur un nombre de crédits à chaque tour pour contrôler ses actions. Son tour s'arrête lorsqu'il n'a plus de crédits.

Piocher des cartes wagons

Pour cette action, le joueur peut piocher des cartes wagons face visibles ou des cartes wagons dans la pioche face non visible. S'il pioche une carte joker visible, il ne peut plus piocher, et s'il pioche d'abord une carte wagon (visible ou non), alors il ne peut pas prendre une carte joker visible. Au contraire, s'il pioche une carte joker non visible dans la pioche. Cette carte ne compte pas pour deux.

Pour réaliser cette action, le joueur doit soit cliquer sur la pioche, soit cliquer sur une des 5 cartes wagons visibles dans l'IHM. Nous avons défini des méthodes pour vérifier si le joueur à le droit de piocher la carte sélectionnée, et le cas échéant, alors cette carte est piochée. La figure suivante illustre notre algorithme.

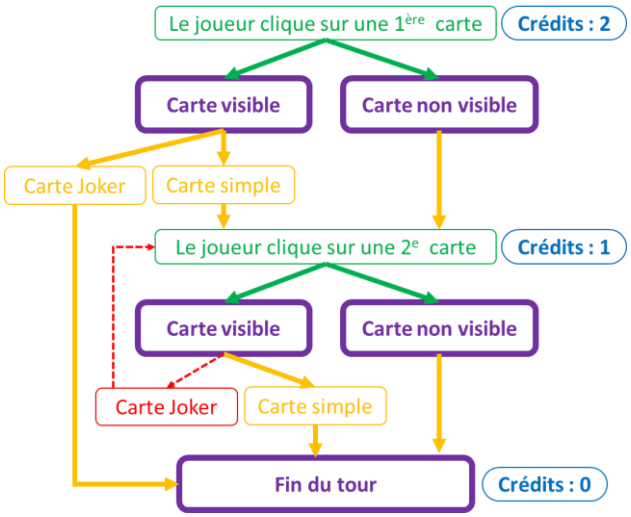


Figure 5 : Schéma de l'algorithme de pioche de carte wagon

Piocher d'autres cartes objectif

Pour cette action, le joueur a le choix entre trois cartes objectif. Il peut piocher une, deux ou trois carte mais doit en prendre au moins une. Pour cela, le joueur doit cliquer sur la pioche de carte objectif. Il est alors automatiquement débité de 2 crédits car il ne peut plus jouer après cette action. Une fenêtre apparait avec trois cartes : s'il clique sur une d'entre elles, elle est piochée. La même fenêtre réapparait avec les deux cartes restantes, mais cette fois le joueur peut aussi fermer la fenêtre s'il ne veut pas les piocher. Le principe est le même s'il continue de piocher jusqu'à ce qu'il n'y ai plus de cartes. Dans ce cas, la fenêtre se ferme toute seule et le tour du joueur est terminé.

Prendre possession d'une route

Lorsque le joueur souhaite s'emparer d'une route il doit cliquer sur l'un des emplacements de cette route sur la Carte depuis l'IHM directement. C'est alors toute la route qui est sélectionnée. Nous avons défini une méthode qui vérifie que le joueur possède les cartes et le bon nombre de wagon pour s'en emparer. Le cas échéant, la route est prise et sa variable est modifiée (« Taken » en True pour indiquer au programme que cette route ne peut plus être prise). S'affiche alors une mise à jour graphique avec les wagons du joueur posés dessus. Si le joueur ne peut pas prendre la route, un message lui indiquant les cartes manquantes est affiché.

Cette méthode prend en compte les cartes jokers mais elle utilise en priorité les cartes simples, le joueur n'est pas libre de choisir combien de cartes jokers il souhaite utiliser. Par exemple, si le joueur veut une route verte de 5 emplacements, et qu'il a dans sa main 4 cartes vertes et 3 cartes jokers, le jeu posera automatiquement 4 cartes vertes et 1 carte joker.

2.2. Implémentation des actions de l'IA

L'IA a le droit de réaliser les trois mêmes actions que le joueur à son tour.

IA Aléatoire

L'IA au niveau « Débutant » est une IA aléatoire. A chaque tour, elle peut piocher des cartes wagons ou prendre une route, elle commence avec 2 cartes objectif et n'en prend jamais d'autres.

Elle essaie d'abord de prendre une route en regardant deux routes au hasard sur le plateau, et, en prenant une des deux si elle le peut. Sinon, elle pioche des cartes wagons. Dans ce cas, elle pioche deux

cartes au hasard parmi celles visibles et celles face cachée du dessus de la pioche. Les règles liées aux cartes jokers sont respectées. Enfin le programme affiche au joueur dans une fenêtre uniquement les cartes visibles qu'elle a piochées. Son tour est terminé.

IA Intelligente

L'IA de niveau « Intermédiaire » peut réaliser les trois actions du jeu : la plus simple est la pioche de nouvelles cartes objectif puisqu'elle en pioche simplement une autre dès qu'elle a terminé les deux cartes qu'elle a prise en début de partie ou qu'elles sont infaisables.

Puis, pour les autres actions nous avons programmé l'IA tel qu'elle essaie d'abord de s'emparer d'une route et, si elle échoue, elle pioche des cartes wagons. Ces méthodes ont été codées en plusieurs étapes :

Etape 1 : Initialisation d'un graphe pour modéliser les routes

Nous avons d'abord créé un graphe pour modéliser les routes du plateau et leur attribuer une difficulté. Pour cela, nous avons utilisé le module *dijkstra*. Il permet de créer un graphe vierge auquel on peut ajouter des sommets. Ces derniers sont liables et on peut attribuer une certaine difficulté aux liaisons. Dans notre cas nous avons assimilé les sommets à des villes, les liaisons aux routes et les difficultés aux longueurs des routes. Cependant, nous avons aussi initialisé nos liaisons en enlevant un point de difficulté aux routes jokers, en mettant une difficulté nulle aux routes déjà prise par l'IA et en mettant une difficulté infini aux route prises par le joueur, ainsi, lorsque l'IA utilisera la graph, qui est recréé à chaque tour, pour trouver le chemin le plus intéressant, elle ne considérera pas les chemin qui contienne des routes prises par le joueur et elle privilégiera les routes jokers ou les routes qu'elle a déjà prises pour son parcours.

Etape 2 : Détermination des routes à prendre

L'IA doit choisir quelle carte objectif elle décide de réaliser en premier. Pour cela, à chaque tour, nous utilisons le module *dijkstra* pour calculer les distances entre les villes des cartes objectif en s'appuyant sur le graph créé avant et sur un algorithme de *dijkstra* [1]. Ensuite nous choisissons l'objectif le plus facile à réaliser qui a la distance la plus courte. Puis, le module *dijkstra* nous renvoie directement une liste des villes par lesquelles passer pour atteindre cet objectif rapidement. Nous utilisons une de nos fonctions pour extraire, à partir de cette liste, les routes de notre plateau qui nous intéressent. Nous retirons à celles-ci les routes déjà prise par l'IA. L'IA n'a plus qu'à essayer de prendre chacune de ces routes, et si c'est un échec pour toutes, alors elle pioche des cartes wagons.

Etape 3 : Détermination des couleurs à piocher

Pour que l'IA pioche des cartes wagons de manière cohérente, nous récupérons la couleur des routes intéressantes déterminées précédemment. Ensuite, nous établissons une liste des couleurs des cartes que nous voulons. Pour chacune de ces couleurs, l'IA vérifie si elle figure parmi les cartes visibles et pioche la carte si oui. Si aucunes des couleurs voulues n'est disponible, elle regarde s'il y a un joker face visible et le pioche. S'il n'y a ni un joker, ni les couleurs voulues, alors l'IA pioche une carte face caché dans la pioche. Cette méthode est contrainte par les règles liées aux cartes joker comme pour le joueur. Lorsqu'elle pioche sa deuxième carte, c'est la fin de son tour, et elle affiche les cartes visibles qu'elle a pioché.

3. Bilan de nos réalisations

3.1. Table des figures réalisées

Figure	Noms	Validation
1	Factorisation du code	Notre projet est séparé en 5 modules et contient 9 classes comme présenté dans les parties précédentes.
2	Documentation et commentaires du code	Toutes nos fonctions et classes sont commentées et il est possible de visualiser cette documentation au format html en suivant la démarche décrite dans le README du projet joint ou dans la partie 5 de ce rapport.
3	Tests unitaires	Nous avons réalisé des tests pour 4 méthodes avec au moins 2 cas testés par méthode. Nos tests sont présenté en partie 3.2 de ce rapport.
4	Création d'un type d'objet	Nous avons créer des classes avec plus de 2 variables d'instances et nous avons instancié plusieurs objets.
5	Héritage au moins entre deux types créés	Comme présenté dans la partie 2.1 de ce rapport, plusieurs de nos classes héritent de la classe Graphic_area que nous avons créée.
6	Structure de données dynamique	Nous utilisons des Array du module Numpy plutôt que d'utiliser des listes python parfois pour faciliter certains calculs ou manipulations.
7	Lecture/écriture de fichiers	Nous utilisons deux fichiers textes dans notre programme, d'abord pour sauvegarder les scores des différentes parties et pour les afficher dans le menu, puis pour créer nos cartes destinations. Les codes pour créer nos cartes destination sont représentés figure 6 et 7,

```
def destination_cards_f():
    """
        Renvoie la liste des paramètres de toutes les cartes destinations à créer depuis un fichier texte.

        :return: Renvoie la liste des paramètre issus du fichier texte.
        :rtype: list

        Auteur : LEVRIER-MUSSAT Gautier
    """
    f = open("Resources/logs/Destination.txt", 'r')
    ligne = f.readlines()
    f.close()
    ligne = [c.strip().split(';') for c in ligne]
    return ligne
```

Figure 6 : Fonction qui extrait une liste de paramètres depuis un fichier texte pour l'initialisation de nos cartes destinations

```
destination_cards = []

cards = destination_cards_f()

for card in cards:
    destination_cards.append(Card("destination", destination=(card[0],card[1]),points=int(card[2])))
```

Figure 7 : Code qui créer nos cartes destinations en fonction des paramètres renvoyé par la fonction de la figure 6

3.2. Tests unitaires

Pour les tests unitaires nous avons testé les méthodes liées aux classes Card() et Draw_pile(). Ce sont des méthodes importantes pour le bon fonctionnement de notre programme puisqu'elles sont utilisées partout ensuite. Ainsi nous présentons ici les tests des différentes méthodes liées à ces classes.

Test de la classe Card()

```
class TestCard(unittest.TestCase):
    """
        Test de l'objet Card, qui représente une carte. Vérification de leur bonne initialisation.
    """
    def setUp(self):
        """
            Création de toutes les cartes possibles pour les tests

             Auteur : NOEL Océan
        """
        #création de toutes les cartes destinations
        self.destination_cards = []
        cards = destination_cards_f() #renvoie les cartes destination possibles depuis le fichier texte fait pour
        for card in cards:
            self.destination_cards.append(Card("destination", destination=(card[0], card[1]), points=int(card[2])))

        #création de toutes les cartes wagon possibles
        self.wagon_cards = []
        color_list = ["rose", "blanc", "bleu", "jaune", "orange", "noir", "rouge", "vert", "tout"]
        for color in color_list:
            for i in range(1):  # 1 carte de chaques couleurs
                self.wagon_cards.append(Card("wagon", color=color))

    def test_init(self):
        """
            Vérification que la variable path des cartes créés est bien assignée automatiquement
            en fonction de la couleur ou de la destination de cette dernière.

             Auteur : NOEL Océan
        """
        #TEST du bon paramétrage des cartes wagons
        for card in self.wagon_cards:
            self.assertEqual(card.path, "Resources/Card_"+card.color+".png")

        #TEST du bon paramétrage des cartes destinations
        for card in self.destination_cards:
            self.assertEqual(card.path, "Resources/Card_"+card.destination[0]+"_"+card.destination[1]+".png")
```

Ce test permet d'être sûr que nos cartes créées reçoivent bien automatiquement le chemin vers leur image.png en fonction de leur couleur ou de leur destination.

Test de la classe Draw_pile()

Pour cette classe nous avions trois méthodes à tester. Nous avons d'abord testé la méthode d'initialisation pour vérifier que nos pioches contenaient bien les cartes voulues, puis, nous avons testé les méthodes de mélange de pioche et de prise de cartes dans une pioche en utilisant les code ci-dessous.

```python
def test_mix(self):
    """
        Test la méthode pour mélanger les cartes

         Auteur : NOEL Océan
    """
    # ////INITIALISATION////
    self.pile1.mix() #on mélange les carte
    self.pile2.mix()
    # ////TESTS////
    #on vérifie que les paquets sont différents
    self.assertNotEqual([card.color for card in self.pile1.cards], [card.color for card in list_cards_sav])
    self.assertNotEqual([card.destination for card in self.pile2.cards], [card.destination for card in list_cards2_sav])
```

Ce test mélange deux pioches, puis compare ces pioches avec une sauvegarde d'elle-même avant mélange pour vérifier que les paquets sont bien différents.

```python
def test_draw(self):
    """
        On test la méthode pour piocher, elle renvoie une liste des cartes qu'on a pioché et les enlève de la pioche cible.

         Auteur : NOEL Océan
    """
    for i in range(10): #on fait 10 tests différents avec des valeurs aléatoires
        amount1 = random.randint(0,len(self.pile1.cards)-1) #nombre de carte à piocher dans la pioche1
        amount2 = random.randint(0,len(self.pile2.cards)-1) #nombre de carte à piocher dans la pioche2
        pos1 = random.randint(0,len(self.pile1.cards)-1-amount1) #a partir de quelle position piocher dans la pioche 1
        pos2 = random.randint(0,len(self.pile2.cards)-1-amount2)#a partir de quelle position piocher dans la pioche 1

        # ////INITIALISATION////
        to_draw1 = self.pile1.cards[pos1:pos1+amount1] #résultat auquel on s'attend après avoir piocher amount1 cartes depuis la carte de position pos1 de la pioche 1
        to_draw2 = self.pile2.cards[pos2:pos2+amount2] #résultat auquel on s'attend après avoir piocher amount2 cartes depuis la carte de position pos2 de la pioche 2
        draw1 = self.pile1.draw(amount1,pos1) #on pioche amount1 cartes depuis la carte de position pos1
        draw2 = self.pile2.draw(amount2,pos2) #on pioche amount2 cartes depuis la carte de position pos2
        # ////TESTS////
        #vérification des tailles
        self.assertEqual(len(draw1), len(to_draw1))
        self.assertEqual(len(draw2), len(to_draw2))

        #verification qu'on a les cartes voulus en piochant
        self.assertEqual([card.color for card in draw1], [card.color for card in to_draw1])
        self.assertEqual([card.destination for card in draw2], [card.destination for card in to_draw2])

        #remise des cartes dans les pioches pour prochain test
        self.pile1.cards = np.append(self.pile1.cards,draw1)
        self.pile2.cards = np.append(self.pile2.cards,draw2)
```

Ce test pioche des cartes dans les pioches avec une quantité aléatoire et depuis une position aléatoire. Ensuite il vérifie si les cartes sont bien piochées comme voulues. Le test est répété 10 fois.

4. Limitations et perspectives

Perspectives d'amélioration pour l'IA :

- Prise en compte de ses cartes wagons actuelles lorsqu'elle choisie une route à prendre. Effectivement, le Graph créé à chaque tour pourrait considérer la main de l'IA pour déterminer la difficulté de chaque route, ainsi, si l'IA a beaucoup d'une couleur, elle essaiera de jouer les routes de cette couleur en priorité. Nous n'avons pas pu intégrer cette fonctionnalité car elle pose des problèmes de redondance lorsque plusieurs routes de la même couleur sont sur le même parcours choisi par l'IA, et nous n'avions accès à la fonction qui détermine les parcours les plus court pour corriger ce problème.
- Réalisation des deux cartes objectif simultanément :
  L'IA pourrait essayer de réaliser ses deux cartes objectif en même temps en fonction des cartes qui tombent au hasard. Au lieu de ça, elle se focalise actuellement sur celle qui à le parcours

le moins long. De plus elle ne prend pas en compte les routes dont elle aura besoin pour ses autres objectifs lorsqu'elle place ses routes.

Puis pour l'expérience utilisateur :

- Ajout de la possibilité de jouer contre d'autres joueurs.
- Ajout de la possibilité de jouer à plus que 2.
- Ajout de nouvelles cartes avec d'autres routes et villes.
- Ajout de paramètres de personnalisation du joueur (nom, avatar)
- Ajout du détail des points gagnés à la fin, avec un résumé des cartes destinations qui ont été faites pour les différents joueurs.

5. Instructions de lancement

Prérequis

Python 3.7

Modules supplémentaires nécessaires pour lancer le jeu :

numpy (dernière version)
random (dernière version)
playsound (version 1.2.2)
pygame (dernière version)
copy (dernière version)
dijkstra (dernière version)
sys (dernière version)
datetime (dernière version)
itertools (dernière version)
screeninfo (dernière version)
time (dernière version)
importlib (dernière version)

Modules nécessaires pour les tests unitaires :

unittest (dernière version)
random (dernière version)
pygame (dernière version)

Pour ajouter des nouveaux modules sur Pycharm il faut aller dans Fichier -> Settings -> Project -> Python Interpreter, puis cliquer sur l'icône « + » , ensuite rechercher le module, le sélectionner et cliquer sur « Install Package ».

Lancement

Pour lancer le jeu, il suffit de lancer le module "menu.py" depuis pycharm.
Le module "test.py" permet de réaliser les tests, pour cela il suffit de le lancer.

Après le lancement du jeu, en jouant contre une IA de niveau intermédiaire, il est possible de visualiser ses objectifs et ses choix dans la console.

Documentation

Pour obtenir la documentation, il faut lancer le fichier "index.html" qui se trouve dans le chemin donné ci-dessous :

chemin d'accès ...\Doc sphinx\build\html\index.html

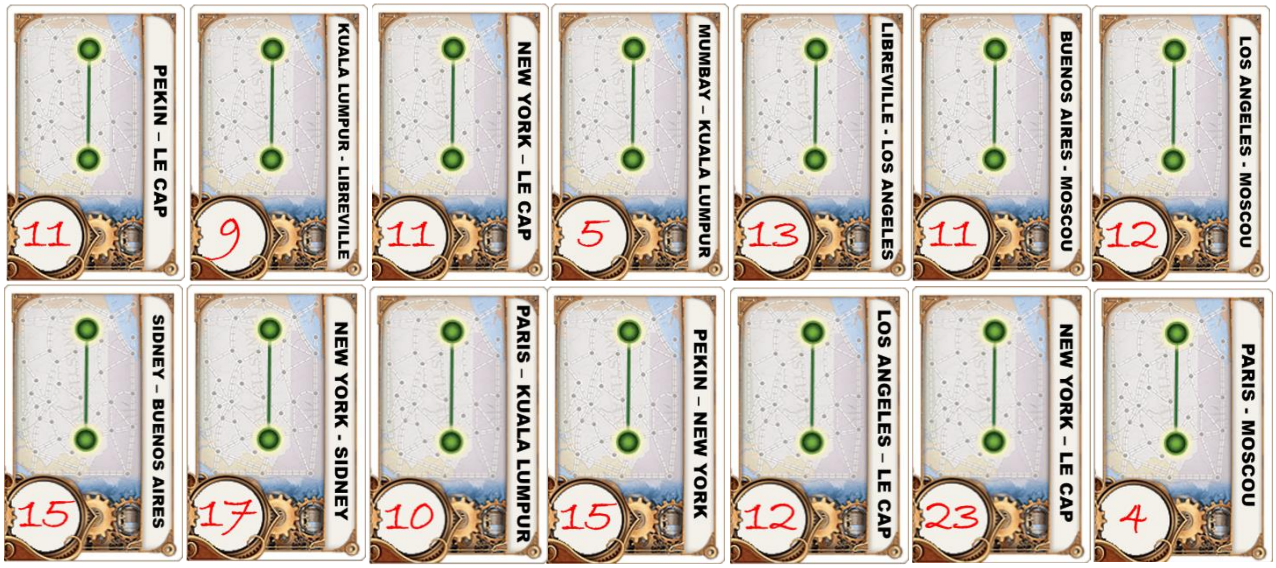
Elle se visualise sur un navigateur.

6. ANNEXE

[1] « Algorithme de Dijkstra », *Wikipédia*. 30 mai 2022. Consulté le: 1 juin 2022. [En ligne]. Disponible sur: https://fr.wikipedia.org/w/index.php?title=Algorithme_de_Dijkstra&oldid=194104287

INVENTAIRE DE NOS CARTES :

Cartes destinations



Cartes wagons (12 exemplaires de chaque)