UV 2.1 Projet Python

Groupe 4

RoboRally

Code disponible sur github à l'adresse suivante:
https://github.com/Chojnacki/RoboRally

Alexandre Chojnacki
Alexandre Corazza

Table des matières

I. Introduction 3
II. Présentation du programme 4
    1. Diagramme des classes 4
    2. Description des classes et des méthodes principales 5
III. Intelligence Artificielle 6
IV. Tests unitaires 6
    1. Test des classes Case et Carte 6
    2. Test des matrices de déplacement 6
    3. Test des priorités des cartes 6
    4. Test d'autres fonctions élémentaires indispensables 6
V. Interface PyQt 7
VI. Conclusion : problèmes rencontrés, perspectives et améliorations 8

Code disponible sur github à l'adresse suivante:
https://github.com/Chojnacki/RoboRally

Alexandre Chojnacki
Alexandre Corazza

I. Introduction

Notre programme simule un jeu de stratégie de course de robots. L'objectif étant de positionner son robot sur une case d'arrivée le plus rapidement possible. Chaque joueur choisi 5 cartes parmi 9 qui lui permettent de programmer les déplacements de son robot sur un plateau. Les différentes cartes sont des cartes « Avancer » de 1, 2 ou 3 cases, « reculer » ou « tourner » de +/- 90° ou 180°.

Des éléments du plateau peuvent modifier l'état du robot (points de vie, coordonnées, orientation) : des trous, des cases de réparation, des tapis roulants et des pivots sont implémentés.

Case réparation
Rend des points de vie au robot

Trou
Détruit le robot

Pivot droite/gauche
Modifie l'orientation du robot

Tapis roulant
Déplace et fait pivoter le robot lors d'un virage

Des obstacles bloquent également les coordonnées du robot, comme des murs ou des robots adverses. La collision entre les robots doit également être prise en compte.

Au début de la partie, le robot dispose de 9 points de vie et lorsqu'il atteint 5 points de vie ou moins, il peut seulement choisir autant de cartes qu'il n'a de points de vie.
Lors du déroulement de la partie, le joueur ayant la carte qui fait avancer le plus a la priorité et déplace son robot en premier.
La partie touche à sa fin lorsqu'un robot se place sur la case d'arrivée.

Case d'arrivée

II. Présentation du programme

1. Diagramme des classes

Toutes les entités présentes dans le jeu nous ont permis d'organiser les différentes classes, les liens entre elles et les méthodes principales.

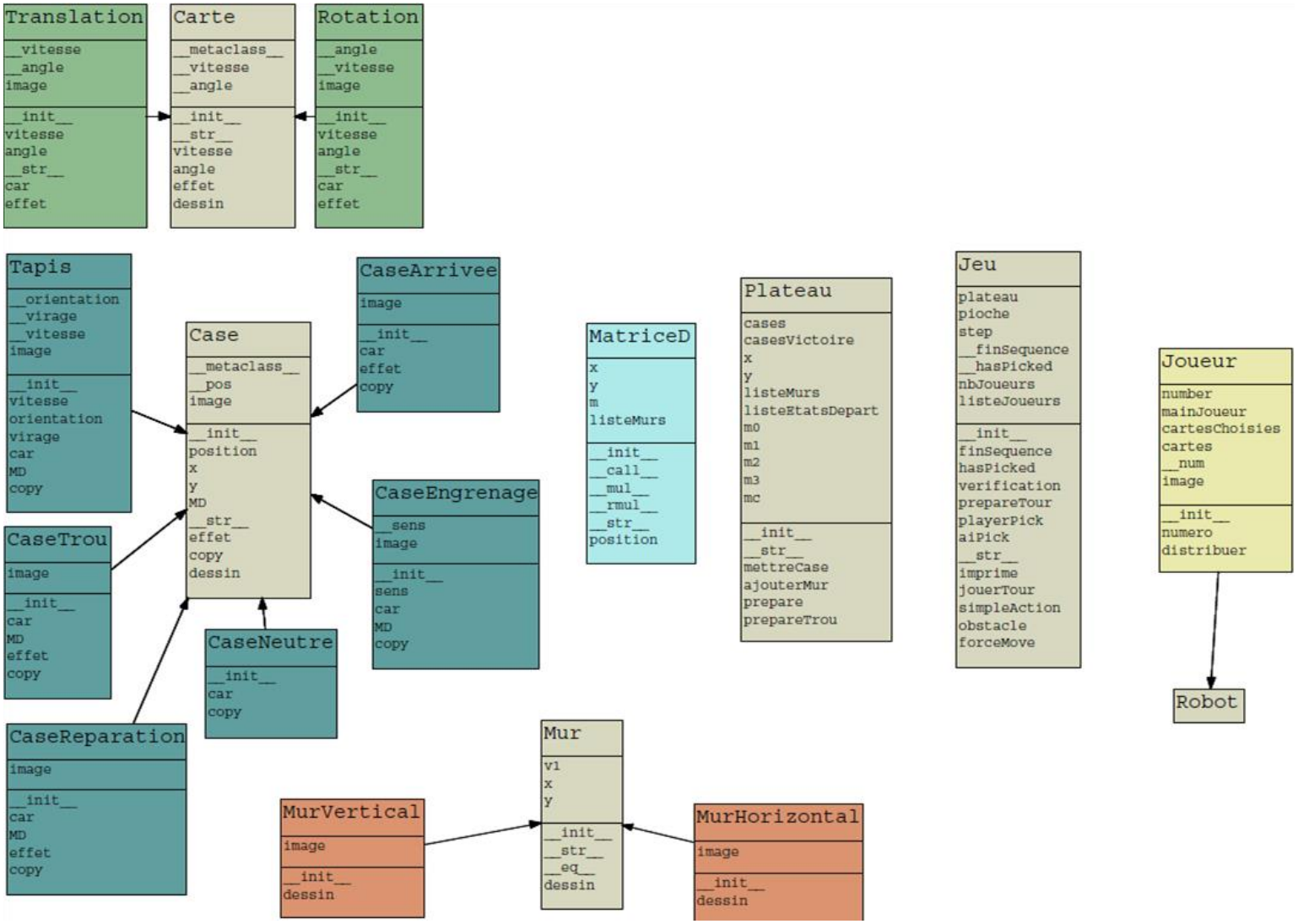

Figure II-1.1 - Classes du programme

2. Description des classes et des méthodes principales

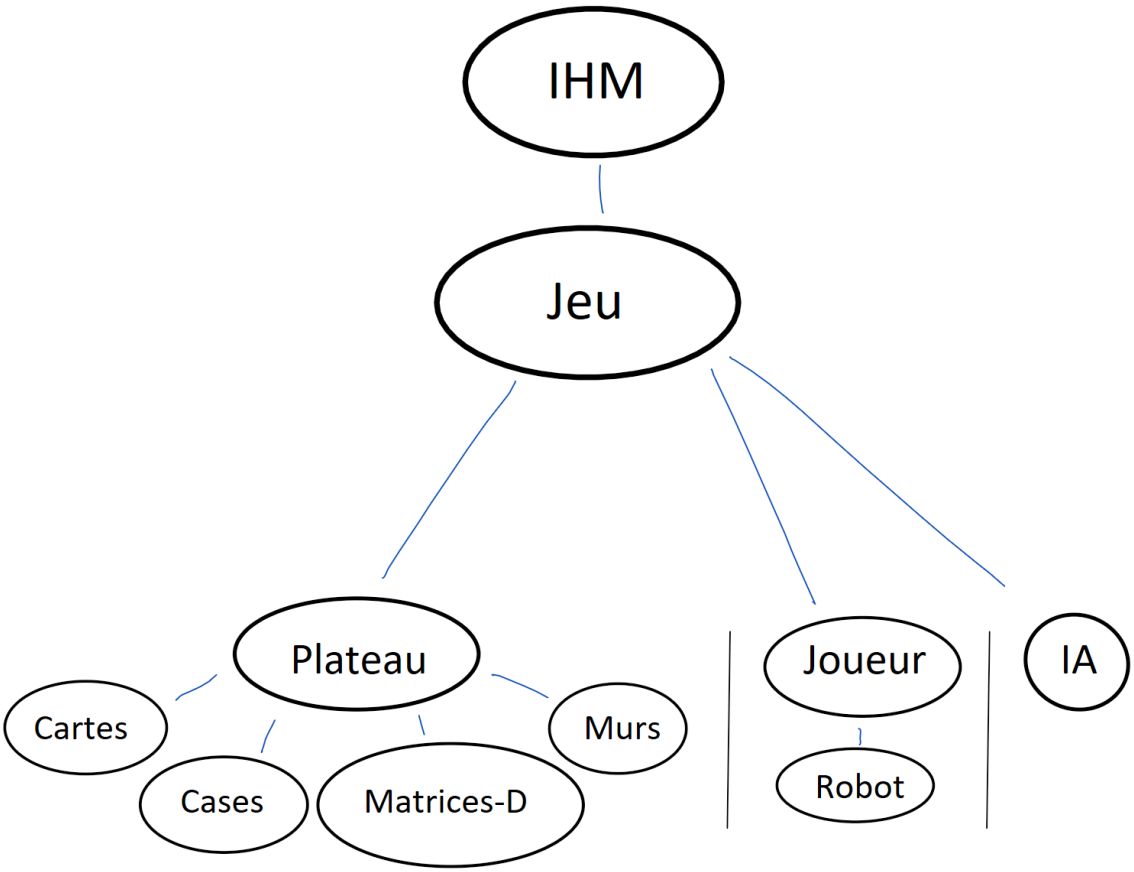

Figure II-2.1 - Graphe des classes

La classe IHM :
Cette classe permet de faire le lien entre le code python qui fait tourner le jeu et le fichier .ui qui correspond à l'interface graphique.
Elle contient une machine à états finis (Finite State Machine (FSM) en anglais). La FSM permet de lancer les étapes du jeu en fonction des interactions avec l'utilisateur. Elle est détaillée dans la partie V. Interface PyQt.

La classe Jeu :
Elle contient toutes les variables nécessaires au déroulement du jeu ainsi que les méthodes pour lancer les phases de jeu.

La classe Plateau :
Cette classe contient toutes les variables liées au plateau de jeu choisi. Nous y trouvons les cartes de la pioche, les cases et murs présents sur le plateau et les matrices de déplacement du plateau.

La classe MatriceD :
Nous avons créé cette classe pour pouvoir gérer les déplacements de robot sur terrain. Leur structure mathématique permet de rapidement calculer les positions atteignables par l'IA.
Elles sont appelées à chaque déplacement pour bloquer le robot lorsqu'il veut passer un mur ou pour lui appliquer l'effet des cases.

La fonction forceMove :
Elle implémente le déplacement qui pousse les robots adverses. Elle est programmée de façon récursive et ne permet le mouvement que si les robots sur le chemin ont été poussés préalablement. Elle fonctionne en appelant les fonctions transitionPositions et la fonction obstacle : elle détermine quel robot doit être déplacé pour permettre le déplacement du robot actuel et elle tente de déplacer les robots qui bloquent le passage.

La fonction transitionPositions :
Cette fonction sert à évaluer la direction d'un robot ainsi que les cases sur lesquelles il va passer en effectuant un déplacement. Nous pouvons ainsi en déduire quels robots doivent être poussés par le robot en déplacement, la direction dans laquelle ils seront poussés et la case finale où ils devraient se trouver à l'issue

III. Intelligence Artificielle

Le jeu sous sa présente forme fait jouer le joueur (robot représenté par un Gwen Ha Du) contre une intelligence artificielle (robot représenté par un Triskel).
L'intelligence artificielle (IA) imite le comportement du joueur : elle se voit distribuée 9 cartes et effectue son choix de cartes. L'IA est basée sur l'algorithme suivant : elle cherche à se rapprocher le plus possible de la case d'arrivée sans mourir. Elle explore toute les possibilités que lui offrent les cartes qu'elle a en main et choisit la meilleure solution. Si aucune solution n'améliore sa situation, elle passe son tour.

IV. Tests unitaires

Nous avons effectué des tests sur les fonctions élémentaires simples à vérifier, très nombreuses et essentielles au bon déroulement du programme. Ainsi les vérifications restantes ne sont à effectuer que sur quelques fonctions principales, plus compliquées à valider à l'aide de tests unitaires.

1. Test des classes Case et Carte

Au commencement du projet, nous avions programmé ces de manière à ce qu'elles appliquent un effet sur le robot et effectué des tests qui validaient nos codes. Nous avons ensuite modifié la structure de programme, ce qui rend ces fichiers test obsolètes. Ces tests peuvent être lancés à partir du dossier « Anciens fichiers tests ».
La structure du code des cartes a été légèrement modifié, chaque carte renvoi maintenant un état estimé au lieu de modifier directement l'état du robot.
La vérification de bon fonctionnement se fait maintenant sur les matrices de déplacement pour les cases.

2. Test des matrices de déplacement

La validation de cette entité est impérative pour le programme. Nous vérifions à travers ces tests la multiplication d'un état du robot avec la matrice du plateau. Le résultat doit correspondre à l'effet de la case où le robot se situe.

3. Test des priorités des cartes

Nous avons choisi un simple tri à bulle pour trier l'ordre des cartes afin d'assurer les priorités des joueurs. La taille des listes de cartes et de joueurs n'étant pas très grande, cette méthode de tri suffit. Nous avons testé tous les cas possibles pour valider ce tri.

4. Test d'autres fonctions élémentaires indispensables

Nous avons testé d'autres méthodes comme la « TransitionPosition » qui est utile pour la collision des robots, ou encore des produits scalaires ou de calcul de distance entre deux cases utiles à l'intelligence artificielle

V. Interface PyQt

L'interface Homme-Machine de notre projet étant la vitrine de notre travail, nous l'avons particulièrement soignée.

L'affichage du plateau s'effectue case par case. Nous avons utilisé l'avantage de l'héritage des classes ici. Tous les types de cases héritent de la méthode « dessin » de leur classe mère. Il suffit alors de préciser le nom de l'image pour chaque sous case. Nous avons adapté ce nom afin d'utiliser la méthode « format » et ainsi simplifier le code.

En ce qui concerne l'affichage du robot, la rotation de l'image nous a posé problème. En effet, la méthode « rotate » de PyQt effectue une rotation autour d'un axe qui n'est pas le centre de l'image du robot, il est donc nécessaire de repositionner le robot à la bonne position sur l'interface. Nous avons finalement adapté la fonction « dessin » de robot qui gère ce problème.

Pour l'exécution des effets des cartes, il a été difficile d'afficher les déplacements case par case. Nous avons alors décidé d'intégrer une Finite State Machine (FSM) à notre programme. Cette solution nous permet uniquement d'afficher les déplacements « carte par carte » et non pas case par case. Par exemple, lors de l'exécution d'une carte d'avance de 3 cases, le robot se téléporte 3 cartes plus loin.
Notre FSM présente 3 états : Pick qui est l'état d'attente du choix de carte du joueur, Play qui représente l'état d'exécution du choix précédent. Nous nous trouvons dans l'état endGame lorsque l'évènement « case d'arrivée atteinte » est détecté.

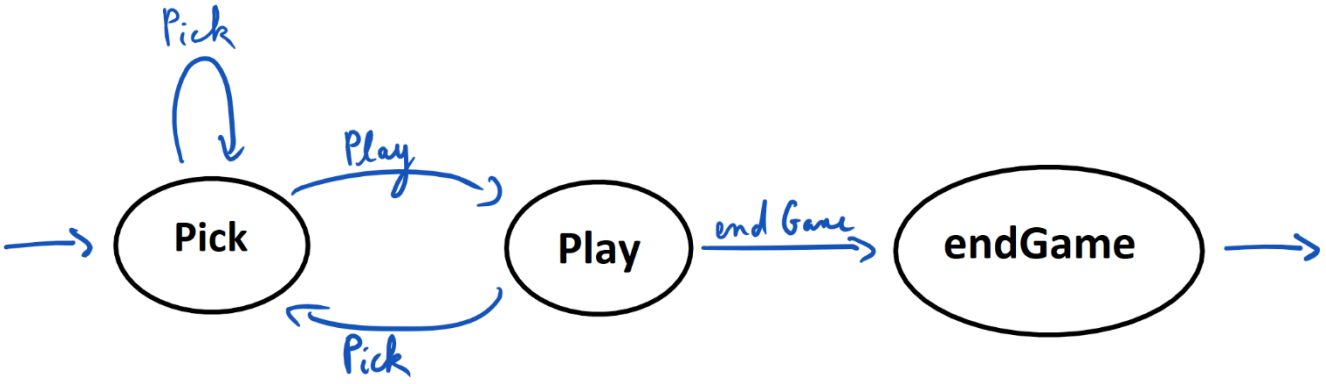

Figure V.1 - Finite State Machine

L'interface Homme-Machine nous a permis de contrôler le bon fonctionnement de notre programme en plus des tests unitaires. Nous avons également pensé à construire des plateaux symétriques automatiquement (fonction copy de la classe Case) et à placer les robots aux coins de la plateforme pour comparer les comportements des intelligences artificielles.

VI. Conclusion

Ce projet nous a permis de nous sensibiliser sur l'importance des tests unitaires et du travail en amont sur la création des classes : on ne peut pas coder au fur et à mesure, il est important de commencer par une analyse du sujet en passant par un diagramme des classes. Toutefois, malgré ce travail, nous avons été piégés par notre programme de base, que nous avons dû modifié en milieu de projet afin de simplifier la programmation de l'intelligence artificielle. Cette dernière a maintenant accès à toutes les combinaisons possibles et a la capacité de choisir la mieux adaptée. Une stratégie couvrant tous les points du cahier des charges était donc indispensable dès le départ.

Il reste cependant des points à améliorer dans notre programme. L'afficher des déplacements du robot case par case améliorerait la qualité de notre interface Homme-Machine. De plus, les lasers n'ont pas été implémentés puisque nous avons choisi de nous concentrer sur les méthodes clés.