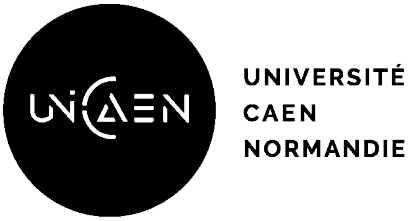

Rapport projet conception logicielle avancée

Ricochet Robots

Nguyen Phuong Vy VU

L2 Informatique - Groupe 2A - Année 2020-2021

22 avril 2021

Table des matières

1 Introduction **2**
1.1 Ricochet Robots . . . 2
1.2 But du projet . . . 2

2 Organisation du projet **3**
2.1 Hébergement du code . . . 3
2.2 Tableau des tâches . . . 3

3 Architecture du projet **4**
3.1 Directoires du projet . . . 4
3.2 Diagramme des classes . . . 5

4 Développement du projet **6**
4.1 Création du plateau . . . 6
4.2 Déplacement et collision des robots . . . 6
4.3 Design Pattern Modèle-Vue-Contrôleur (MVC) . . . 7
4.3.1 Modèle . . . 7
4.3.2 Vue . . . 7
4.3.3 Contrôleur . . . 9

5 Algorithme A* **10**
5.1 Présentation d'algorithme A* . . . 10
5.2 Algorithme de parcours en largeur . . . 10
5.2.1 Fonction de coût $f(c)$ . . . 10
5.2.2 Fonction de déplacement $g(c)$ . . . 11
5.2.3 Fonction heuristique $h(c)$ . . . 11
5.3 Implémentation d'algorithme A* . . . 12

6 Instruction du jeu **13**
6.1 Lancer le jeu . . . 13
6.2 Jouer le jeu . . . 13

7 Captures d'écran **14**

8 Conclusion **15**

1 Introduction

1.1 Ricochet Robots

En 1999 Hans im Glück / Tilsit ont édité un jeu de société qui a été créé par Alex Randolph et illustré par Franz Vohwinkel. Il est Ricochet Robots. D'après Wikipédia[1] le jeu est composé d'un plateau, de tuiles représentant chacune une des cases du plateau, et de pions appelés « robots ».

FIGURE 1 – Ricochet Robots

Le jeu se décompose en tours, un tour de déplacement des robots sur le plateau pour en amener un sur l'une des cases du plateau. Les robots se déplacent en ligne et avancent jusqu'à rencontrer un obstacle (un robot ou un mur).

À chaque tour, l'un des joueurs a remis une tuile cible. Le but est alors de faire en sorte que le robot de la couleur de la tuile sur la case cible dont le symbole est même que le symbole tuile. Si la tuile multicolore est tirée, le but est d'amener n'importe quel robot sur la case multicolore du plateau.

La tuile est remportée par le joueur qui a la solution comptant le moins de mouvement dans délai de temps impartis. Le jeu se termine lorsque tous les jetons auront été tirés. Le gagnant est la personne qui aura récolté le plus jetons.

Le Ricochet Robots peut aussi bien être joué seul qu'avec un grand nombre de participants.

1.2 But du projet

Dans ce projet, on a développé un programme qui permet de trouver la meilleure solution pour toute situation du jeu. On a lancé un planning de la conception de ce projet en plusieurs temps :

- Lancer de la conception du projet (diagramme, librairie, méthode)
- Développer l'interface du jeu
- Développer du moteur du jeu, selon les règles du jeu
- Implémenter l'algorithme A* pour chercher le chemin le plus court
- Optimiser les méthodes de l'algorithme et vérifier le code pour éviter les bugs

2 Organisation du projet

2.1 Hébergement du code

On a choisit le forge d'Unicaen pour faciliter la gestion du projet. Il permet de créer et d'administrer des dépôts sous le forge que l'on peut en faire avec terminal. De plus, le forge offre les autres fonctionnalités qui sont une gestion des permissions, une visualisation des différents commits, le suivi de l'activité du projet, etc.

2.2 Tableau des tâches

Classes	**Tâches**
Création des modèles	
Case	Contenir un point des coordonnées Créer des murs et des variables nécessaires
Robot	Distinguer entre la classe Target
Target	Distinguer entre la classe Robot
Name	Énumération des symboles différents des targets
Création du plateau	
Board	Construire les éléments Initialiser les éléments Déplacer les robots Rechercher les targets Réinitialiser le plateau Exécuter A*
Direction	Énumération des quatres directions
Création de MVC	
Draw	Dessiner l'interface Déplacer le robot avec clavier Sélectionner le robot avec souris Lancer A*
Algorithme A*	
AStar	Créer une liste des coups validés Initialiser les valeurs heuristiques sur chaque cases Implémenter A* Récupérer la liste des cases du chemin
Main	Lancer le jeu

3 Architecture du projet

3.1 Directoires du projet

La structure du projet est non seulement de diviser le projet en des parties spécifiques, mais encore doit adopter la structure de Gradle. Il est structuré en quatre parties.

```
assets
expression▸rapport
gradle▸wrapper
src▸main▸java
  board
    Board.java
    Direction.java
  draw
    Draw.java
  model
    Case.java
    Name.java
    Robot.java
    Target.java
  solve
    AStar.java
  Main.java
```

On remarque les packages utilisés :
— board : initialiser le plateau avec les robots, les tuiles et les fonctions du jeu
— draw : dessiner le plateau, ajouter l'écouteur de clavier et l'écouteur de souris
— model : créer modèle du jeu
— solve : implémenter l'algorithme A*

3.2 Diagramme des classes

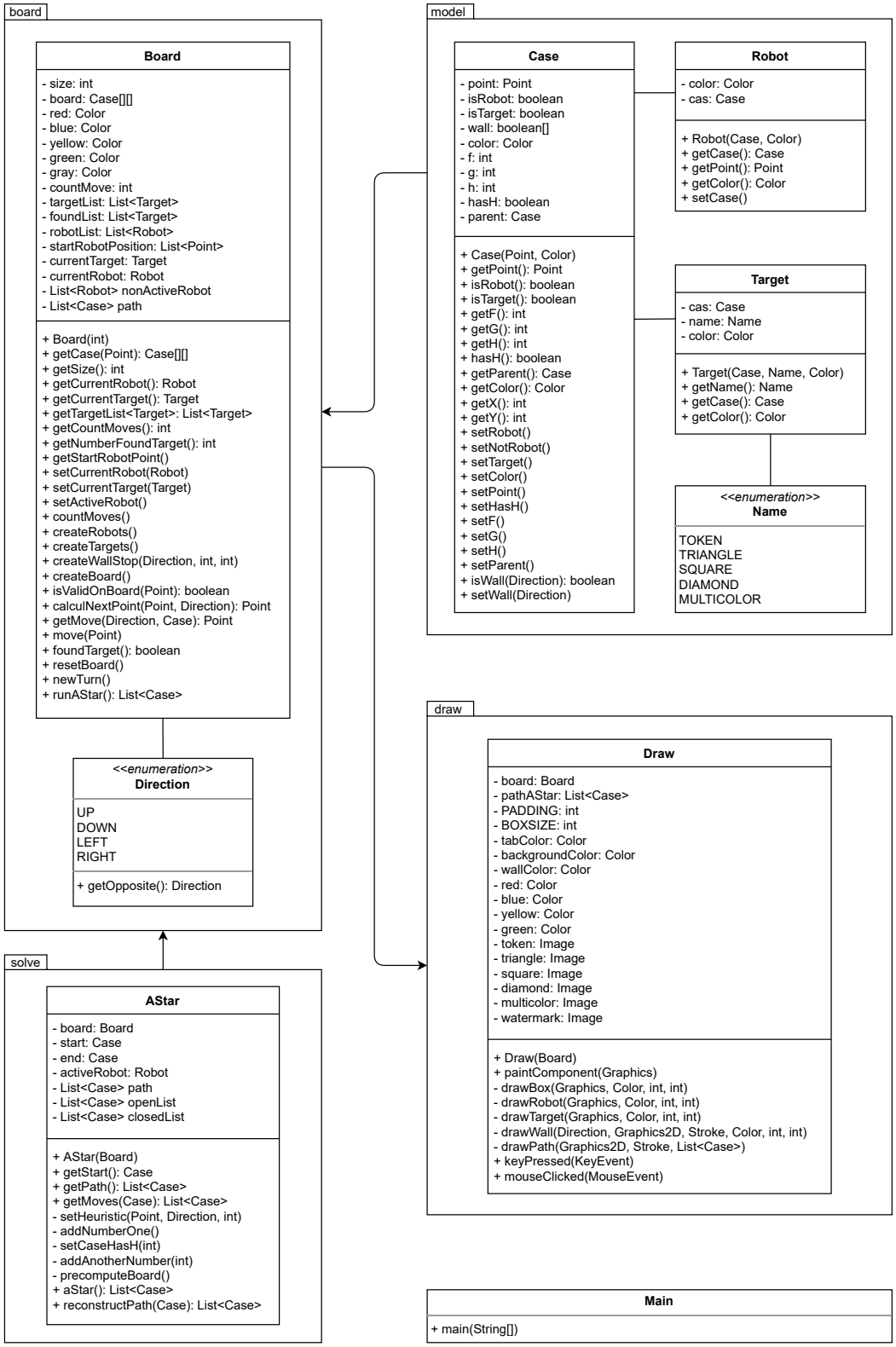

FIGURE 2 – Diagramme des classes

4 Développement du projet

4.1 Création du plateau

Au début, on sait que la taille du plateau est de 16x16, 256 carrées, 5 robots et 16 tuiles. Donc, on crée la classe des cases nommées **Case** qui sera une petite carrée du plateau. On trouve que le plateau a neuf cases différentes et les cases contiennent les tuiles ou les robots ou sont nulles. (Image à 4.3.2 - Figure 5)

En effet, on utilise le booléen dans les cases pour contrôler les robots, les tuiles et les positions du mur. On a un commun des cases du mur, la possibilité d'avoir un mur en haut, en bas, à gauche et à droite avec le format de direction `{up, down, left, right}`. Par exemple, le mur en haut à la liste `{true, false, false, false}`, la case de robot est `true`, et la case de tuile est `true`.

FIGURE 3 – Format de direction

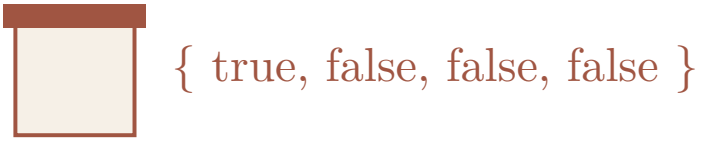

FIGURE 4 – Exemple d'une case

On initialise les robots dans la classe **Robot** avec les couleurs et les rôles qui se mettent en lien avec la classe **Case** par la méthode `setRobot():Case`. Avec les tuiles, on écrit une classe **Target** représentant les tuiles comme le robot, ajoute le nom de type et met en lien avec les cases en méthode `setTarget():Case`.

Finalement, on construit le plateau avec les cases en défaut. Après avoir créé les cases sur le plateau, c'est à ce moment-là que l'on place les éléments qui sont les murs, les robots et les tuiles. On utilise une variable intermédiaire pour déterminer les robots et les tuiles dans la case après avoir créé et mettre dans un `ArrayList` de chaque type `Robot` ou `Target` pour les manager.

4.2 Déplacement et collision des robots

Selon aux règles du jeu, on déplace un robot vers une direction, ce robot se déplace jusqu'à ce qu'il rencontre un obstacle (le mur ou les autres robots). Grâce aux rôles des cases, on met la capacité de distinguer les obstacles dans ce robot

pour le stopper. On construit le mur d'arrêt qui a deux directions opposées afin de contrôle la direction du robot.

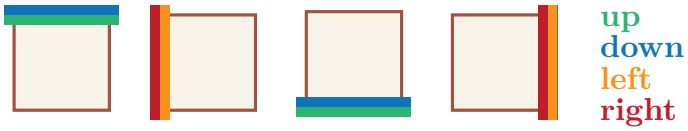

FIGURE 5 – Cases avec murs pour stopper le robot

On contrôle la situation d'une case qui contient seulement un robot. C'est pourquoi le robot ne peut pas être dans une même case.

4.3 Design Pattern Modèle-Vue-Contrôleur (MVC)

Dans cette partie, on construit l'interface du jeu par la librairie Java Swing qui a été inclus au JDK depuis la version 1.2, cette libraire est moins compliquée d'installer que la librairie JavaFX, avec la conception Modèle-Vue-Contrôleur (MVC). Et on souhaite de ne pas lancer la ligne de commande quand on joue le jeu.

Java Swing[2] fait partie de la bibliothèque Java Foundation Classes (JFC). C'est un API dont le but est similaire à celui de l'API AWT, mais son fonctionnement et son utilisation sont complètement différentes.

Modèle-Vue-Contrôleur ou MVC[3] qui est composé de trois types de modules ayant trois responsabilités différentes : les modèles, les vues et les contrôleurs (un motif d'architecture logicielle destiné aux interfaces graphiques).

4.3.1 Modèle

Un modèle est les données à afficher. Par conséquent, on développe quatre classes de modèle qui sont **Case**, **Name**, **Robot** et **Target**.

4.3.2 Vue

Une vue présente l'interface graphique. Pour simplifier, on divise les murs à quatre cases de switch avec les directions et dessine en fonction `drawLine` et `setStroke` de la classe **Graphics2D**.

FIGURE 6 – Neuf cases avec mur

En outre, les robots sont des cercles et les targets sont des carrées avec les symboles différents. On va dessiner avec les fonctions principales `fillOval`, `fillRect` et `drawImage` de la classe **Graphics**

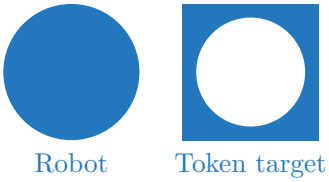

FIGURE 7 – Modèle d'un robot et d'une tuile de couleur bleu

C'est un jeu pour tout le monde, par suite on choisit le minimalisme avec une palette ci-dessous, car ce sont des couleurs douces et non sombres. Elles donnent une atmosphère relax quand on joue.

FIGURE 8 – Palette de couleurs

En résumé, on indique les remarques de l'interface graphique sur l'image ci-dessous et "Mochi Shiba Pun Pun" qui apporte l'amusement et la chance devient l'icônes du jeu.

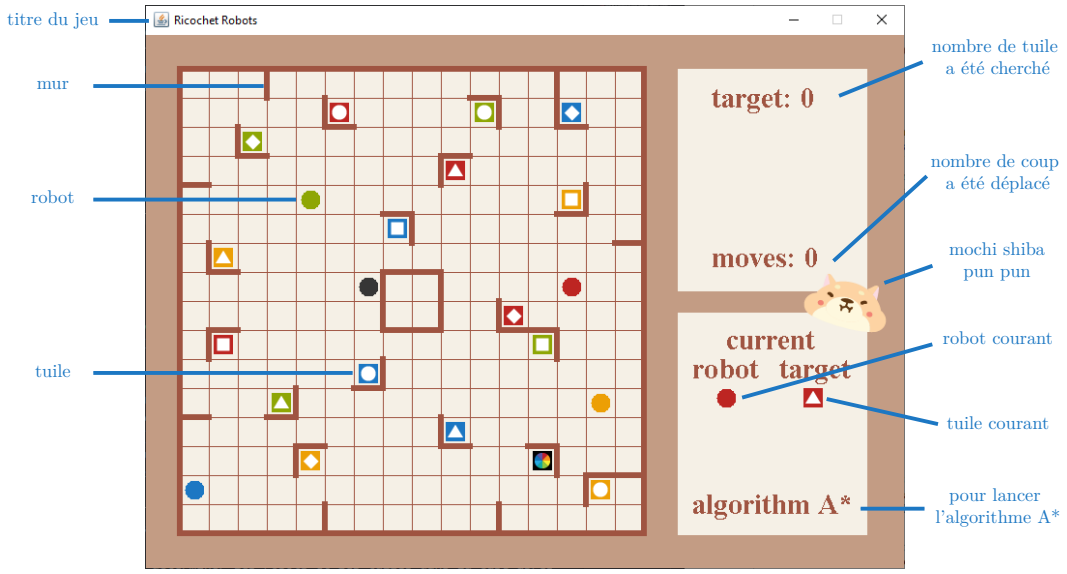

FIGURE 9 – Remarques de l'interface graphique

FIGURE 10 – "Mochi Shiba Pun Pun"

4.3.3 Contrôleur

Un contrôleur contient la logique concernant les actions effectuées par l'utilisateur. Pour faire les contrôleurs, on doit implémenter des interfaces **ActionListener** et **KeyListener**.

Premièrement, pour choisir les autres robots sur le plateau, on doit calculer le point de case par rapport les coordonnées de souris. Par exemple la coordonnée qui correspond au point de robot (4, 4) est de (170, 170). On propose la formule pour calculer le point (x, y) de robot de la coordonnée (xPos, yPos) :

$$x = (xPos - PADDING)/BOXSIZE$$
$$y = (yPos - PADDING)/BOXSIZE$$

Deuxièmement, on met les conditions des coordonnées pouvant fonctionner dans les zones autorisées de la souris et utilise la méthode `mouseClicked(MouseEvent)` pour appeler la fonction de changement le robot et la fonction de recherche le chemin.

Enfin, pour terminer cette partie, on met les méthodes de déplacer le robot, de compter les coupes et faire nouveau état du plateau quand les tuiles ont été cherchées dans les navigateurs de clavier avec la méthode `keyPressed(KeyEvent)`.

5 Algorithme A*

Intelligence artificielle peut résoudre des problèmes d'une énorme complexité combinatoire. Dans Ricochet Robots, un robot doit chercher la tuile cible de même couleur que l'on l'a tiré. De plus, pour aider les joueurs qui peuvent trouver facilement la solution du jeu, on construit une fonction avec l'algorithme A*. Cet algorithme doit être très simple, ne nécessite pas de prétraitement et consomme très peu de mémoire.

5.1 Présentation d'algorithme A*

Algorithme A*[4] est un algorithme de recherche de chemin dans un graphe pondéré : à partir d'un nœud de départ spécifique d'un graphique, il vise à trouver un chemin vers le nœud de but donné ayant le plus petit coût (distance-la moins parcourue, temps le plus court, etc). A* est basé sur l'utilisation de méthodes heuristiques pour atteindre l'optimalisé et l'exhaustivité et est une variante de l'algorithme best-first.

Il faut conserver un arbre de chemins provenant du nœud de départ et en étendant ces chemins un bord à la fois jusqu'à ce que son critère de terminaison soit satisfait. À chaque itération de sa boucle principale, A* doit déterminer le chemin à développer. Il le fait en fonction du coût du chemin et d'une estimation du coût requis pour étendre le chemin vers la cible. Plus précisément, l'algorithme choisit le plus petit chemin.

Lorsqu'un algorithme de recherche a la propriété d'optimalisé, cela signifie qu'il est garanti de trouver la meilleure solution possible, dans notre cas le chemin le plus court vers l'état d'arriver. Et quand un algorithme de recherche a la propriété d'exhaustivité, cela signifie que s'il existe une solution à un problème donné, l'algorithme est assuré de la trouver.

5.2 Algorithme de parcours en largeur

On utilise les cases `c` de classe **Case** qui contient les valeurs des fonctions $f(c)$, $g(c)$ et $h(c)$ comme un noeud de l'algorithme.

5.2.1 Fonction de coût $f(c)$

On veut déterminer dans quelle case se déplacer à chaque étape. Pour ce faire, on va concevoir une fonction mathématique $f(c)$ qui mesurera à quel point une case candidate est bonne pour être inclus dans notre chemin le plus court.

C'est la fonction de coût et on veut la minimiser pour produire un résultat optimal. La fonction de coût est la somme d'une fonction de déplacement et d'une fonction heuristique.

$$f(c) = g(c) + h(c)$$

5.2.2 Fonction de déplacement $g(c)$

On appelle cette fonction de déplacement $g(c)$ est de 0 en défaut. $g(c)$ est la distance parcourue entre la case de départ A et la case courante. Ainsi, les cases voisins de la case de départ auront une distance qui est la longueur entre le case de départ et eux.

Pour calculer $g(c)$, on obtient les coordonnées $g(c)$ de la case avant la case actuelle, la distance entre la case précédente et la case actuelle. Chaque case a sa propre valeur $g(c)$ de la case de départ à cette case.

5.2.3 Fonction heuristique $h(c)$

La fonction heuristique est utilisée pour estimer à quel point la case que l'on examine est proche de la cible. On choisit l'heuristique grâce au diaporama de Michael Fogleman [5] et à la présentation de Randy Coulman [6].

D'abord, on a une liste d'états à prendre en compte au lieu de choisir simplement celui qui a la plus courte distance. On ajoute un facteur qui est une estimation de combien on reste à parcourir et tant que cette estimation n'est jamais une surestimation. On peut en quelque sorte calculer si les robots pourraient s'arrêter n'importe où où ils le voulaient, combien de mouvements ont pris pour atteindre la tuile si évidemment.

Sur le plateau, l'heuristique des cases $h(c)$ en défaut est de 0, on commence à partir du robot au choix et doit déjà aller c'est 0 tout en ligne droite est un 1 tout en ligne droite à partir de là il y a un 2 puis 3, 4 et 5 et dans ce plateau $h(c)$ est de jusqu'à 5.

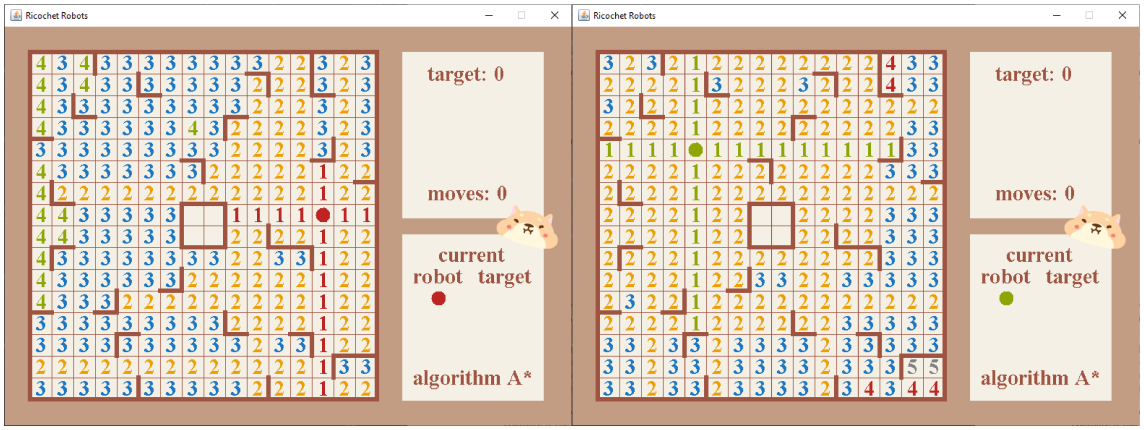

FIGURE 11 – Affichage des valeurs heuristiques

Donc, à partir de ces cases marquées cinq, même si les robots pouvaient s'arrêter où ils le souhaitent, il faudrait cinq mouvements pour entrer dans la case de but à partir de là, à cause de la structure du plateau et de l'emplacement des murs.

5.3 Implémentation d'algorithme A*

Premièrement, on crée la méthode `aStar(): List<Case>` pour construit l'algorithme. On simplifie la zone de recherche pour gestion plus facile. On divise la zone de recherche en cases. Chaque case est un élément de l'algorithme.

Deuxièmement, on doit avoir 2 list :
- Un pour l'enregistrement de tous les cases que l'on doit examiner pour trouver le chemin le plus court (`openList`).
- Un pour l'enregistrement de tous les cases que l'on n'a pas besoin de revoir (`closedList`).

Troisièmement, on ajoute la position de la case de départ à `closedList`. Aussi, ajouter tous les cases que le robot peut arriver de la position actuelle à `openList`. N'oubliez pas que ce n'est pas un obstacle.

Maintenant, on calcule la valeur du chemin et met une valeur de $f(c)$ à chaque case. Ici, on examine seulement que la case ne peut aller qu'en droit entre la case.

Pour trouver le chemin le plus court, on répète les étapes suivantes :
- Prendre la case en `openList` avec une valeur $f(c)$ minimale. Appeler cela la case B.
- Supprimer B de `openList` et ajouter B à `closedList`.
- Si M est la liste des cases que B peut déplacer (des coups validés), alors soit I est l'examen de chaque case dans M.
- Si I est déjà dans `closedList`, ignorer.
- Si I n'est pas dans `openList`, ajouter-le pour calculer les valeurs de distance.
- Si I est dans `openList` : vérifier-le, si la valeur $f(c)$ est moins à la valeur $f(c)$ du point actuel, de même mettre à jour le nouveau point actuel.

Pour conclure, on écrit une méthode `reconstructPath(Case):List<Case>` pour obtenir la chemin final par rapport aux cases parents lorsque la case courant a le même point à la case de la tuile courant.

6 Instruction du jeu

6.1 Lancer le jeu

Il y a beaucoup du logiciel de construction automatisée d'application qui automatisent la création d'applications exécutables à partir du code source sur la marché comme ANT, Maven et Gradle. Sur ce projet, on décide d'utiliser le Gradle, car c'est simple pour écrire le script et pour lancer.

Entrez la commande `./gradlew run` pour lancer le jeu. Ça fonctionne avec le Linux Shell et Windows Powershell.

6.2 Jouer le jeu

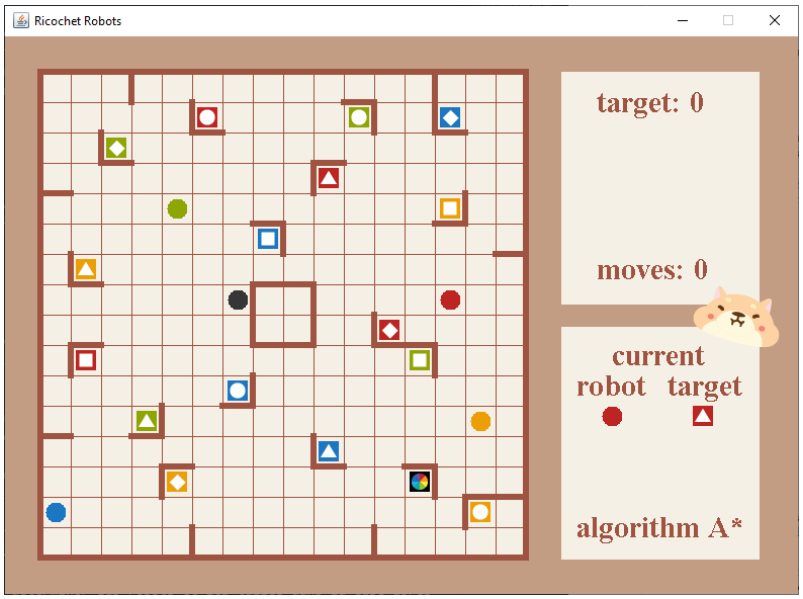

FIGURE 12 – Initialisation du plateau

On utilise quatre navigateurs sur le clavier pour déplacer les robots. On note si le clavier ne fonctionne pas, relancer le jeu. Ensuite, pour changer le robot courant, on clique sur les autres robots. Pour l'algorithme A*, cliquez sur le texte "algorithme A*" sur écran, les chemins vont afficher et le robot peut déplacer sur ces chemins.

7 Captures d'écran

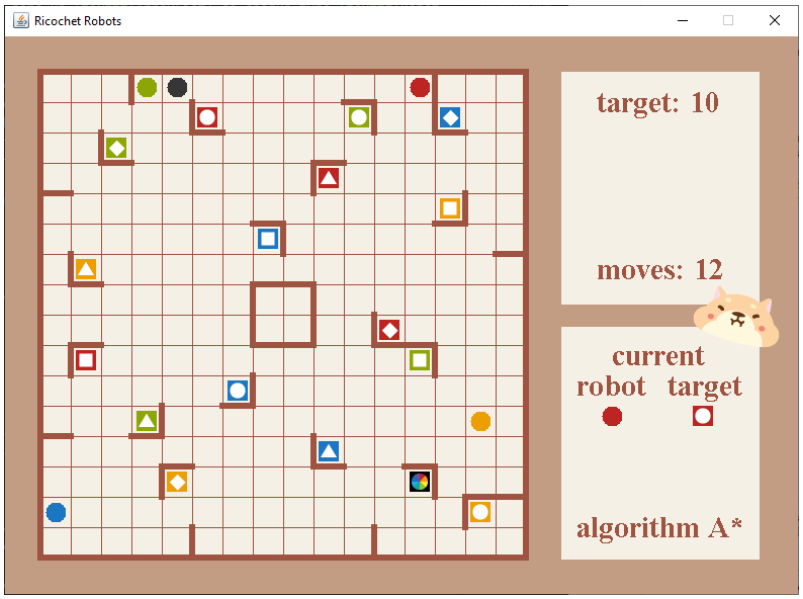

FIGURE 13 – Jouer en normal

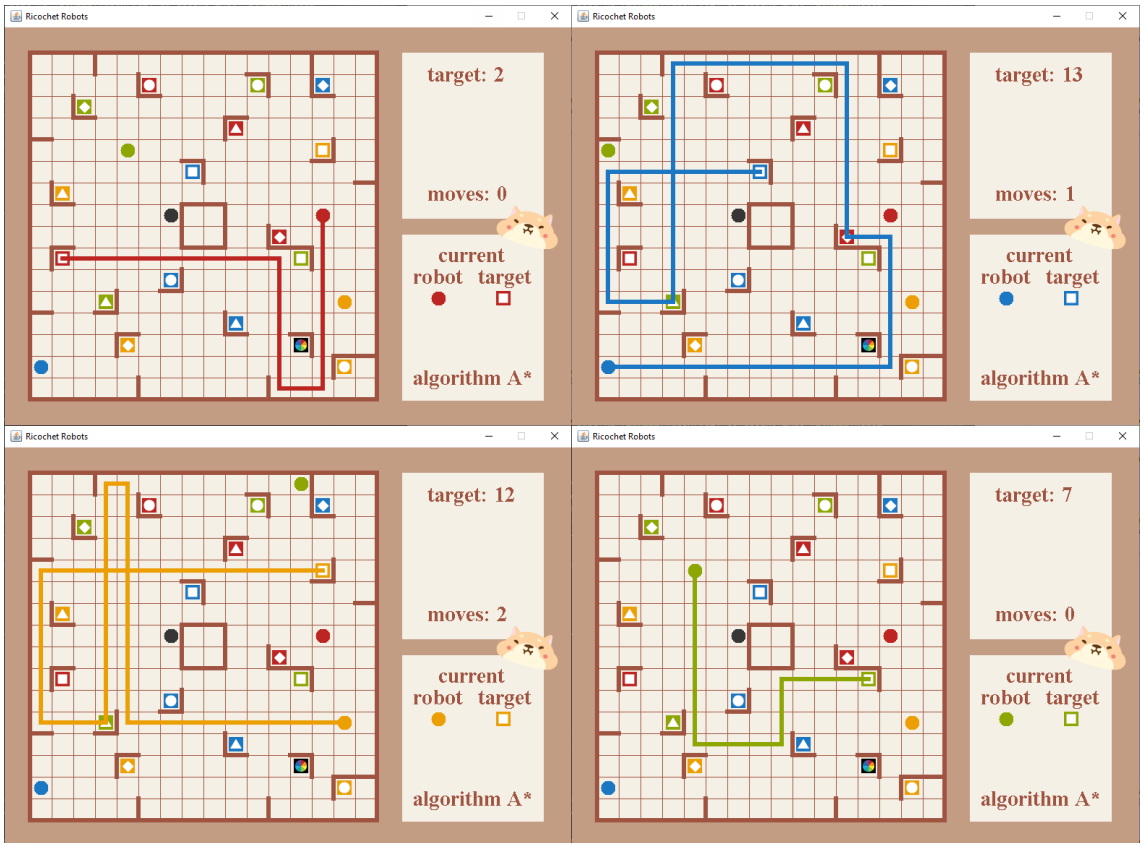

FIGURE 14 – Jouer avec algorithme A*

8 Conclusion

Après ce projet, j'ai mieux compris la libraire Java Swing, le Java Build Tools Gradle, comment l'algorithme A* fonctionne et le jeu de société Ricochet Robots, je n'ai pas connu ce jeu en précédent. En autre, j'ai avancé la capacité de débugger, d'optimiser le code avec la conception MVC. Malgré que je n'aie pas fini la partie de déplacer multi-robots avec l'algorithme A*, je continuerai de terminer.

Enfin, je veux remercier Mme DHAOU de me conseiller pendant les cours de travaux pratiques ainsi que M. BONNET pour ses cours magistraux. Merci d'avoir pris le temps de lire ce rapport.

Références

[1] Wikipédia. Ricochet robots.
https://fr.wikipedia.org/wiki/Ricochet_Robots.

[2] Jean-Michel DOUDOUX. Le développement d'interfaces graphiques avec swing.
https://www.jmdoudoux.fr/java/dej/chap-swing.htm.

[3] Wikipédia. Modèle-vue-contrôleur.
https://fr.wikipedia.org/wiki/Modèle-vue-contrôleur.

[4] Wikipédia. Algorithme a*.
https://fr.wikipedia.org/wiki/Algorithme_A*.

[5] Michael Fogleman. Ricochet robots - solver algorithms.
https://speakerdeck.com/fogleman/ricochet-robots-solver-algorithms, 2012.

[6] Randy Coulman. Mountainwest rubyconf 2015 - solving ricochet robots.
https://www.youtube.com/watch?v=fvuK0Us4xC4.