UNIVERSITÉ
CAEN
NORMANDIE

- Rapport de projet -
Conception d'un solveur de ricochet robots

André Lorada
Auvray Théo
Licence 2 informatique - Groupe 1A

3 mai 2020

Table des matières

1 Introduction

Le Ricochet Robots est à l'origine un jeu de plateau. D'après Wikipédia[1], ce jeu de société est composé d'un plateau, de tuiles représentant chaque case du plateau, et de pions appelés "robots" et "jetons". Différents jetons sont imprimés dans certaines des cases (au nombre de 17). Le jeu commence en piochant un pion "jeton" parmi les 17. La case à aller correspond au symbole du jeton qui a été tiré aléatoirement. La partie est décomposée en tours de jeu, un tour consistant à déplacer les robots sur le plateau afin d'emmener le robot de la même couleur du jeton tiré sur la case correspondante. Les robots ne peuvent que se déplacer qu'en ligne droite jusqu'à rencontrer un obstacle (un robot ou un mur).

Le Ricochet Robot peut aussi bien être joué seul qu'avec un grand nombre de participants. Le jeton revient à la personne qui aura trouvé la séquence de mouvement qui permettra à un robot donné (parmi quatre), d'atteindre la case, en moins de coups possible dans un délai de temps imparti. La partie se termine lorsque tout les jetons auront été tirés. Le gagnant est la personne qui aura récolté le plus de jetons.

Le but de ce projet à été de développer un programme permettant de trouver une solution optimale pour toute situation du jeu. La conception de ce projet à été réalisée en plusieurs temps :

1. Le développement du moteur du jeu suivant les règles du Ricochet Robots
2. La réalisation d'une interface graphique.
3. L'implantation d'un algorithme de résolution naïf, appelé A*.
4. Proposer des méthodes d'optimisation de l'algorithme, par soucis de complexité pour être résolu dans de bonnes conditions.

2 Organisation du projet

2.1 Gestion du projet

Afin de faciliter la communication et le bon déroulement de la conception de notre jeu, divers moyens ont été mis en oeuvre.

2.1.1 Hébergement du code

Afin de faciliter la gestion du projet, nous avons utilisé à la fois la Forge d'Unicaen et Github qui permettent de créer et d'administrer des dépôts sous Git très facilement par l'intermédiaire d'une interface web. D'autres fonctionnalités sont disponibles sur ces plateformes comme une gestion des permissions, une visualisation des différents commits, la visualisation de l'activité du projet, etc. L'utilisation supplémentaire de Github permettait de centraliser les projets du cursus de la licence sur une seule plateforme ainsi que l'utilisation de "Webhooks" (envoi de notifications sur Discord lors de la mise en ligne d'une fonctionnalité).

2.1.2 Gestionnaire de version

Nous avons utilisé un gestionnaire de version afin de permettre la centralisation du code et rendre le travail en équipe bien plus efficace. Nous avons opté pour Git, qui est un gestionnaire de version que nous avons utilisé dans un précédent projet. L'utilisation de Git rend l'utilisation des branches plus facile, permet de faire des commits sans pour autant être connecté sur le serveur. Cela permet de faire plus de commits, qui sont enregistrés localement et de les envoyer sur le serveur en une seule fois, au moment où nous sommes sûrs que la fonctionnalité ajoutée fonctionne. Git permet également de transférer facilement son code vers un autre hébergeur en ajoutant simplement une "route" (remote), tout en conservant la totalité des commits réalisés.

2.1.3 Trello

Concernant la répartition et le listage du travail à effectuer, nous avons fait le choix d'utiliser Trello, une plate-forme qui nous permet l'utilisation de tableaux afin de planifier le projet.

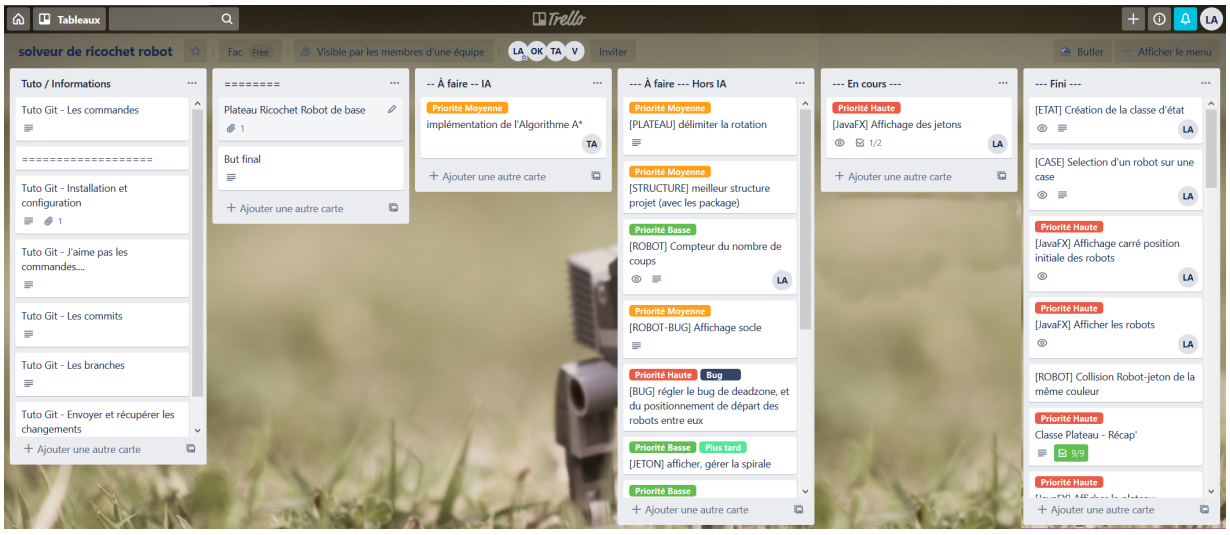


FIGURE 1 – Notre tableau Trello

Ainsi, comme nous pouvons le constater sur l'image ci-dessus, les différentes tâches passent par différents états appelés "*À faire*", "*En cours*", "*À modifier et à vérifier*", et "*Fini*".

La colonne "*À modifier et à vérifier*" est utilisée lorsqu'une tâche est réalisée, mais doit être modifiée (car elle n'est pas optimale) ou doit être soumise à évaluation et/ou relecture. Cela permet d'éviter d'éventuels bugs par la suite mais aussi de garder une cohérence au travers du code. Lorsque cette tâche est modifiée et/ou vérifiée, elle est déplacée dans la colonne "*Fini*".

2.1.4 Discord

Afin de faciliter la communication au sein du groupe, nous avons utilisé le service de messagerie Discord car tous les membres du groupe l'utilisaient déjà de manière personnelle. Ce service permet de parler par le biais de "serveurs" gratuits dans lesquels nous pouvons ajouter des salons textuels ou des salons vocaux à volonté.

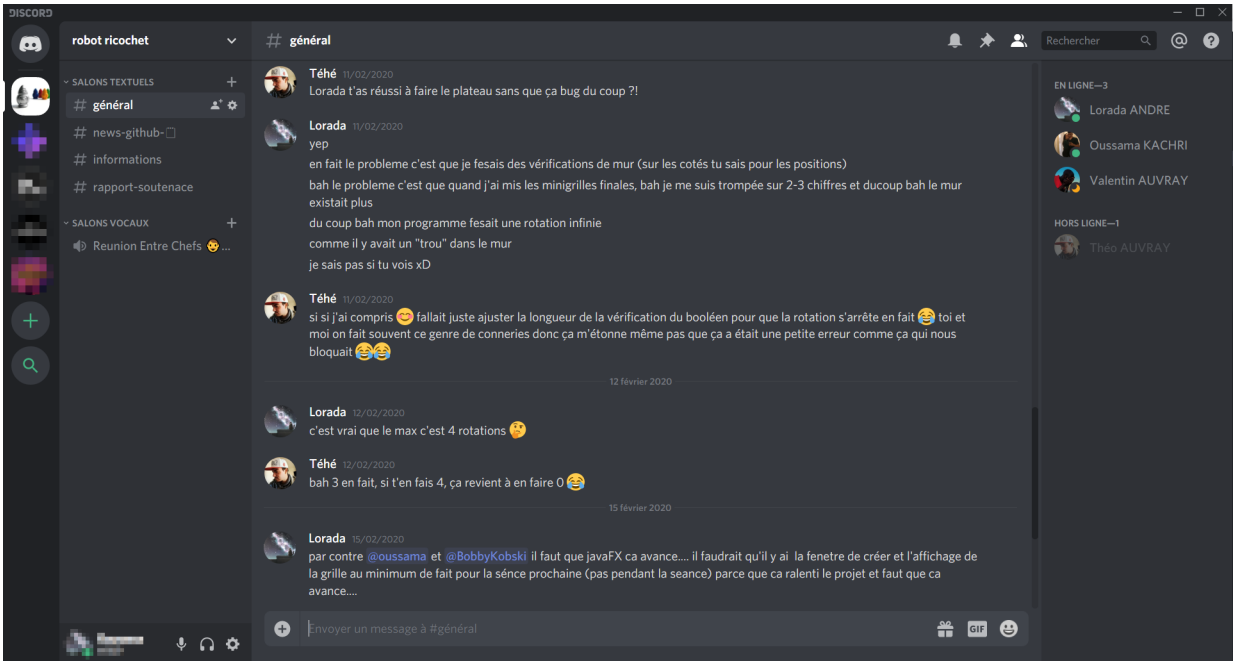


FIGURE 2 – Notre serveur Discord

Ainsi, nous avions un salon de discussion, nommé "*informations*". Comme son nom l'indique, il permet de transmettre des messages importants sur ce qui a été fait, sur des changements importants concernant le projet, etc. Un deuxième se nommant "*news-Github*" servait à recevoir une notification dès qu'une personne ajoutait du code sur Github, afin de connaître l'avancement de chaque personne sans pour autant devoir aller sur les hébergeurs, ou bien pour qu'une personne sache à quel moment elle devait reprendre le relais sur une fonctionnalité. Le troisième salon se nommant"*général*" était utilisé pour les discutions beaucoup plus générales. Sur ce salon, nous pouvions discuter du projet, de certains choix à faire, ou bien demander de l'aide ou aider des membres en difficulté.

2.2 Répartition des tâches

Tâches	Lorada	Théo
Plateau		
Création de la classe Plateau	X	
Création des quarts de plateau	X	
Assemblage des quarts de plateau		X
Génération aléatoire du plateau	X	
Rotation des quarts de plateau	X	
Affichage graphique du plateau	X	
Robot		
Création de la classe		X
Positionnement aléatoire des robots	X	
Collisions des robots avec les éléments	X	
Déplacement des robots	X	
Sélection d'un robot	X	
Affichage des robots et des socles	X	
Jeton		
Création de la classe Jeton	X	
Génération aléatoire du jeton tiré		X
Affichage graphique du jeton tiré	X	
Case		
Création de la classe Case	X	
Positionnement des cases	X	
Rotation des cases	X	
Affichage graphique des cases	X	
Case Jeton		
Création de la classe Case Jeton	X	
Positionnement des jetons		X
Rotation des cases jeton	X	
Affichage graphique des cases jetons	X	
Pattern Observer		
Implémentation du pattern observer	X	
State		
Création de la classe State	X	
Création de l'état initial du jeu	X	
Déplacement du robot avec le clavier	X	
Séléction du robot qui doit jouer	X	
Définition de l'état gagnant	X	
Algorithme A*		
Algorithme BFS	X	
Sortie Anticipée	X	
Mise en place de l'heuristique	X	
Documents		
Rédaction du rapport	X	
Rédaction du support de soutenance	X	

3 Architecture du projet

3.1 Arborescence du projet

Le projet est structuré de la manière suivante :

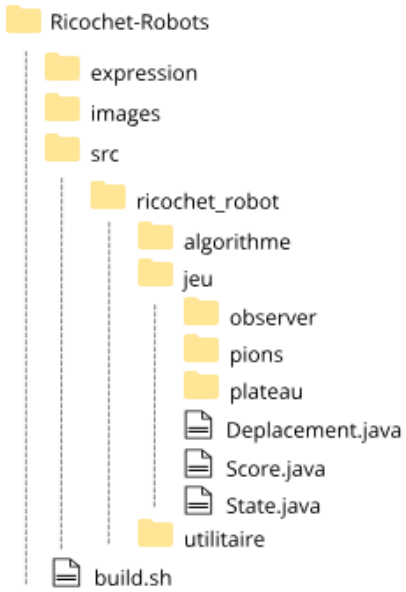


FIGURE 3 – Arborescence du projet

Á la racine de ce projet, nous y retrouvons :

expression : contient ce rapport et le diaporama pour la soutenance.

images : contient l'ensemble des images utilisées pour la partie interface graphique du jeu.

src : contient le code source du projet.

build.sh : le script de compilation du projet.

3.2 Architecture du programme

3.2.1 Diagramme des packages

Le code source est organisé dans différents packages :

algorithme : contient les classes relatives à l'implémentation de l'algorithme A*.

jeu : contient tout les éléments relatifs au jeu.

- **observer :** regroupe les différentes classes utilisées pour l'implémentation du pattern observer.
- **pions :** contient les classes représentants les éléments mobiles du Ricochet Robots, correspondant aux robots et aux jetons du jeu.
- **plateau :** contient les classes relatives à la création du plateau

utilitaire : contient une classe utilitaire.

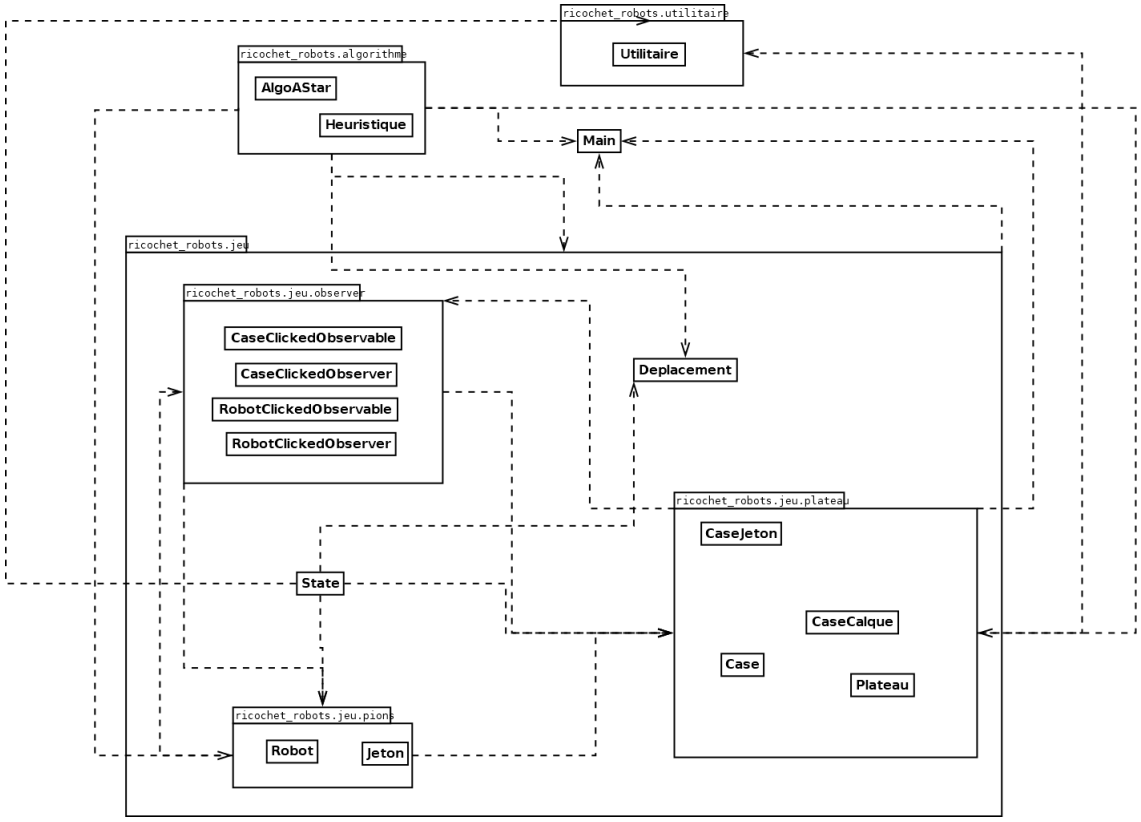


FIGURE 4 – Diagramme des packages

3.2.2 Diagramme des classes

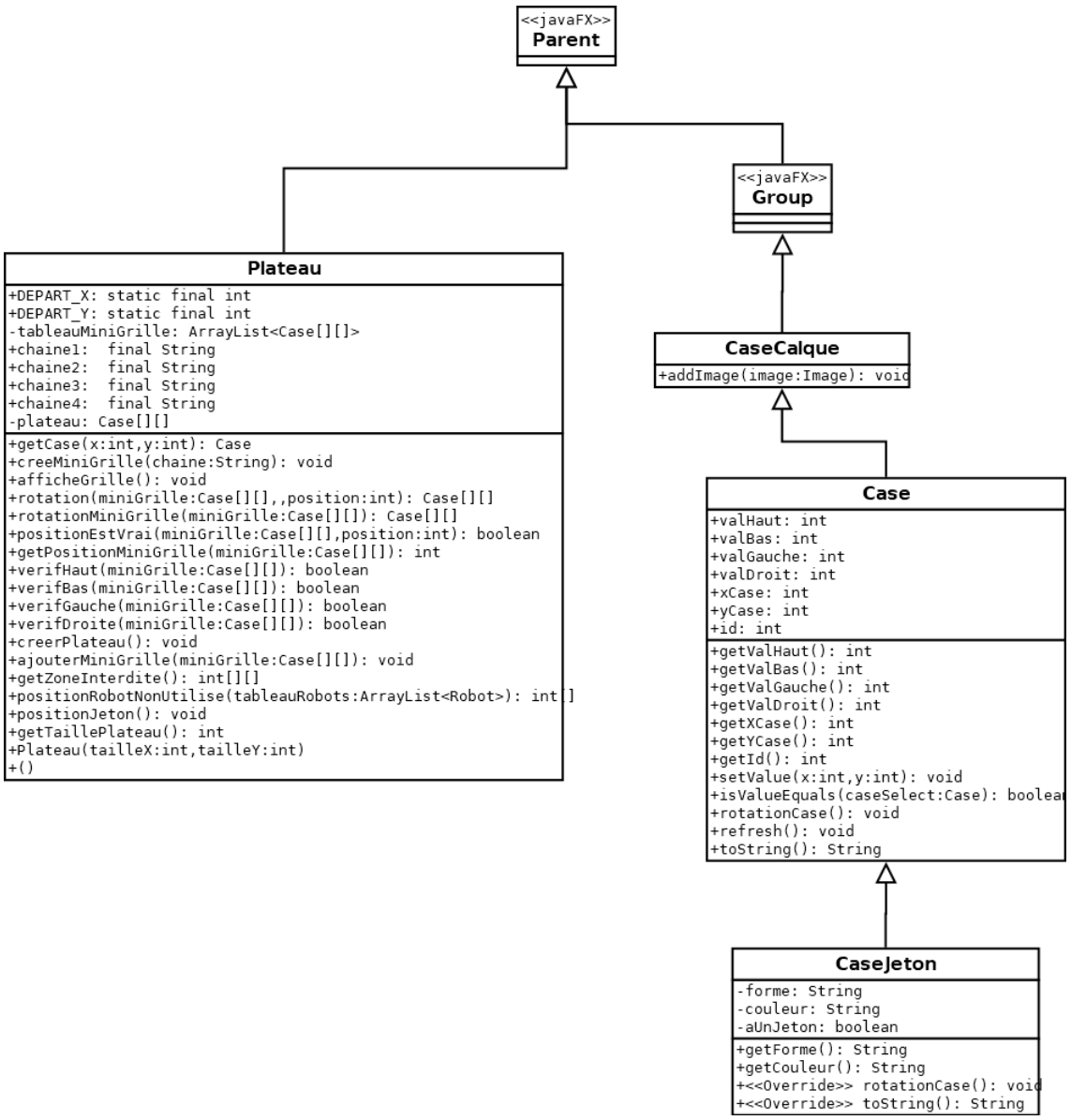


FIGURE 5 – Diagramme du package plateau

4 Développement du jeu

4.1 Création du plateau

4.1.1 Création des mini-plateaux

Pour concevoir le Ricochet Robot, la première question que nous nous sommes posé à été : "comment devons-nous créer le plateau ?". Le plateau du Ricochet Robots est un ensemble de "mini-plateaux" qui, une fois assemblés, forment ce plateau. Dans la version du jeu de société, il existe quatre morceaux de plateaux, chaque morceau ayant deux faces. Dans notre version, nous avons voulu garder que quatre faces, une pour chaque morceau, qui étaient suffisantes pour la réalisation du Ricochet Robot.

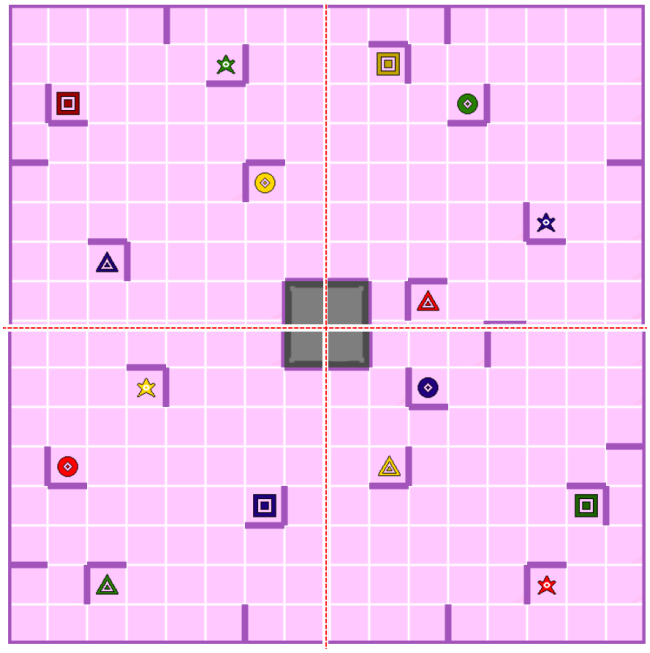


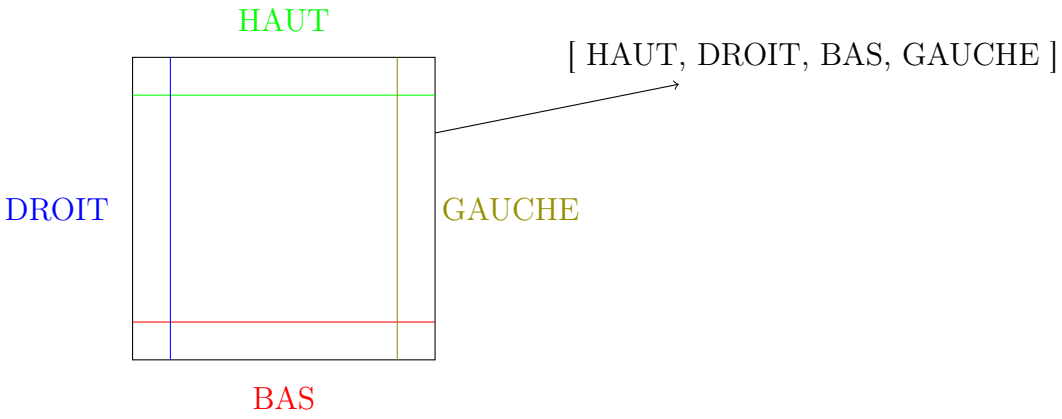
FIGURE 6 – Composition du plateau

Pour créer le plateau, nous devions commencer par la création de ces mini-plateaux. Ces quatre mini-plateaux sont chacun représenté sous forme de tableaux à deux dimensions, contenant des cases et des murs.

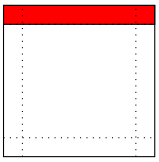
C'est à ce moment là que nous avions réfléchi sur la représentation de ces cases et de ces murs. Après observation du plateau, nous avions remarqué que ce plateau était composé que de onze cases différentes.

FIGURE 7 – Ensemble des cases existantes

Ces cases ont toutes un élément en commun : la possibilité d'avoir un mur en haut, en bas, à gauche, ou à droite.



Nous avons donc décidé de représenter chaque case de cette manière : la présence d'un mur en haut entraînait un 1 à la valeur de HAUT, la présence d'un mur à droite entraînait un 1 à la valeur de DROIT, et ainsi de suite pour les autres côtés. Ce qui donne par exemple :



[True, False, False, False] ⟶ [1,0,0,0]

FIGURE 8 – Cas de figure pour un mur en haut de la case

Pour ne pas alourdir chaque mini-plateau en créant pour chacun d'entre eux un tableau à trois dimension (un tableau en deux dimensions dont chaque élément contient elle-même un tableau), nous avons donc créé une classe Case, contenant les quatre variables précédemment évoquées, nommées "valHaut, valBas, valGauche et valDroite" qui remplacent le format de tableau [HAUT, DROIT, BAS, GAUCHE]. Nous obtenons alors avec un tableau à deux dimension dont chaque élément contient une instance de Case.

Á ce moment là, c'était au tour de la création des quatre mini-tableaux. La position de chaque case de chaque mini-tableau est importante et inchangeable, car les modèles des mini-plateaux du jeu de société sont réalisés de manière à avoir une solution à toute situation de jeu. Nous avons donc gardé les modèles existants.

Nous avons donc cherché une manière de sauvegarder ces quatre modèles de mini-plateau dans le code, et ainsi qu'à partir de cette sauvegarde, les mini-plateaux se créent.

Nous avons opté pour un modèle suivant :

```
1 public final String chaine1 = "9,8,8,12,9,...,0,4,9"; //haut, gauche
2 public final String chaine2 = "8,12,9,8,8,...,8,0,0,4"; //haut droit
3 public final String chaine3 = "6,1,0,0,0,...,2,2,6"; //bas droit
4 public final String chaine4 = "1,0,0,0,0,...,6,3,2,2"; //bas gauche
```

Cette manière de faire permettait qu'à partir d'une suite de nombres, le programme créé un mini-plateau. Le code correspondant est le suivant :

Algorithme 1 : CRÉATION D'UN MINI-PLATEAU

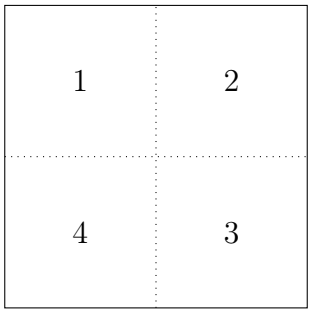
Entrées : Une chaine de caractère représentant le mini-plateau
Sortie : Le mini-plateau

```
1 tableauDeChiffre ← chaineDeCaractere.split(",")
  tailleTableau ← √tableauDeChiffre.length() index ← 0
  miniGrille[][] ← nouvelle instance de Case pour y ← 0 à tailleTableau faire
2 |  pour x ← 0 à tailleTableau faire
3 |  |  miniGrille[x][y] ← new Case(Utilitaire.intToBinary(tab1D[index]), x, y)
  |  |  index++
4 |  fin
5 fin
6 retourner miniGrille
```

Chaque nombre correspond à un identifiant d'une case. Le numéro n'est pas pris au hasard : en reprenant la figure 8 ci-dessus, nous avons vu que une case ayant un mur en haut correspondait à [1,0,0,0]. L'idée était de récupérer ces chiffres, formant "1000", qui en nombre binaire donne un "8". Une case avec un mur en haut correspond donc au chiffre "8". Le système est le même pour chaque case.

4.1.2 Positionnement des mini-plateaux

Comme le plateau est composé de quatre parties, chaque mini-plateau ont quatre possibilités de positionnement afin de former le plateau final (plus exactement : 4 - le nombre de mini-plateau déjà positionné).



Pour les positionner de manière aléatoire, nous avons fait en sorte que le programme tire une position pour chaque mini-plateau. Cependant ce n'est pas suffisant, puisqu'il fallait que les murs de chaque mini-plateau soient bien positionnés, sinon nous nous retrouverions avec des "mauvais plateaux" de cette forme :

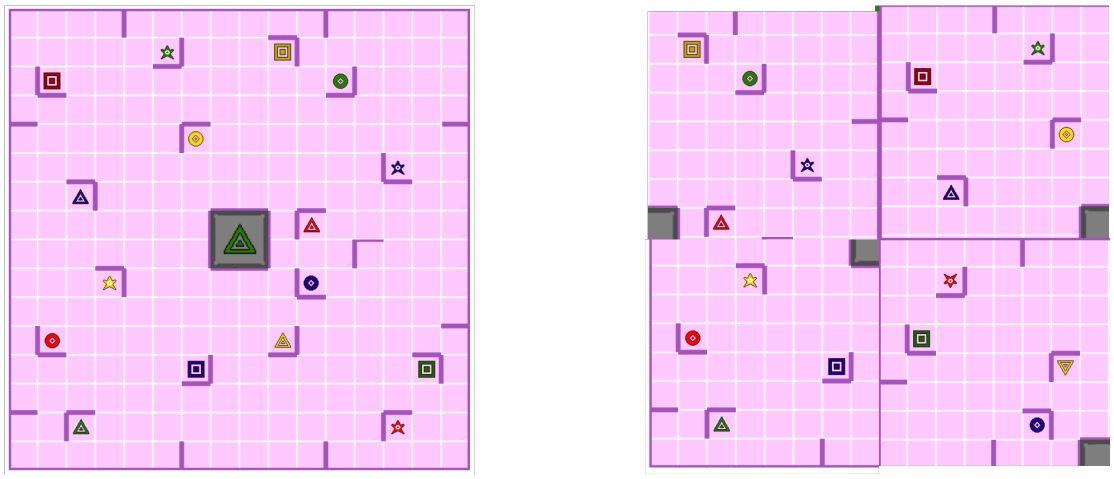


FIGURE 9 – Bon (à gauche) et mauvais plateau (à droite)

Pour cela, il fallait vérifier l'emplacement des murs qui composent les bordures, et affecter une rotation à ces mini-plateaux tant que le mini-plateau n'est pas valide par rapport à sa position affectée. Les algorithmes qui représentent ce dispositif sont les suivants :

Algorithme 2 : Rotation d'un mini-plateau

Entrées : le mini-plateau, la position affectée au mini-plateau
Sortie : Le mini-plateau

```
1  miniGrilleRota ← new Case(miniGrille.length(), miniGrille.length())
2  pour y ← 0 à miniGrille.length() faire
3  |  pour x ← 0 à miniGrille.length() faire
4  |  |  miniGrilleRota[x][y] ← miniGrilleRota[x][y]
5  |  fin
6  fin
7  tant que positionEstVraie(miniGrilleRota, position) != true faire
8  |  miniGrilleRota[x][y] ← rotationMiniGrille(miniGrilleRota)
9  fin
10 retourner miniGrilleRota
11 Avec positionEstVrai(), une méthode qui permet de vérifier les murs des bordures
```

Algorithme 3 : Rotation du mini-plateu de 90 degrés

Entrées : un mini-plateau
Sortie : Le mini-plateau dans une rotation de 90 degrés

```
1  miniGrilleRota ← new Case(miniGrille.length(), miniGrille.length())
2  pour y ← 0 à miniGrille.length() faire
3  |  pour x ← 0 à miniGrille.length() faire
4  |  |  miniGrilleRota[x][y] ← miniGrilleRota[y][miniGrille.length − x − 1]
   |  |   miniGrilleRota[x][y].rotationCase()
5  |  fin
6  fin
7  tant que positionEstVraie(miniGrilleRota, position) != true faire
8  |  miniGrilleRota[x][y] ← rotationMiniGrille(miniGrilleRota)
9  fin
10 retourner miniGrilleRota
```

La méthode rotationCase consiste à décaler d'un cran les valeurs de la classe Case selon le modèle suivant :

Le code est le suivant :

```java
public void rotationCase(){
    int temp = this.valGauche;
    this.valGauche = this.valBas;
    this.valBas = this.valDroit;
    this.valDroit = this.valHaut;
    this.valHaut = temp;

    this.id = Utilitaire.CaseToInt(this);
}
```

La méthode caseToInt permet de recalculer l'identifiant de la case, qui aura changé après la rotation d'une case.

4.1.3 Création du plateau

Une fois que chaque mini-plateau se trouvait dans la bonne rotation correspondant à sa position, c'est à ce moment là que nous devions créer le plateau, en assemblant ces mini-plateaux entre eux. Pour cela, nous avons crée un tableau à deux dimensions de type Case, représentant le plateau et chaque coin de ce plateau est parcouru pour y affecter les valeurs contenues dans chaque mini-plateau. L'algorithme utilisé est le suivant :

Algorithme 4 : Création du plateau

```
   Entrées : Le mini-plateau à placer
 1 plateau[][] ← nouvelle instance de Case
     demiTabX ← divise le tableau en largeur par 2
     demiTabY ← divise le tableau en hauteur par 2
 2 si la position du mini-plateau doit être à 1 alors
 3 |   pour y ← 0 à demiTabY faire
 4 |   |   pour x ← 0 à demiTabX faire
 5 |   |   |   plateau[x][y] ← miniPlateau[x][y]
 6 |   |   fin
 7 |   fin
 8 fin
 9 sinon si la position du mini-plateau doit être à 2 alors
10 |   pour y ← 0 à demiTabY faire
11 |   |   pour x ← demiTabX à tableau.length faire
12 |   |   |   plateau[x][y] ← miniPlateau[x − demiTabX][y]
13 |   |   fin
14 |   fin
15 sinon si la position du mini-plateau doit être à 3 alors
16 |   pour y ← demiTabY à tableau.length faire
17 |   |   pour x ← demiTabX à tableau.length faire
18 |   |   |   plateau[x][y] ← miniPlateau[x − demiTabX][y − demiTabY]
19 |   |   fin
20 |   fin
21 sinon si la position du mini-plateau doit être à 4 alors
22 |   pour y ← demiTabY à tableau.length faire
23 |   |   pour x ← 0 à demiTabX faire
24 |   |   |   plateau[x][y] ← miniPlateau[x][y − demiTabX]
25 |   |   fin
26 |   fin
```

4.2 Positionnement des robots

Lors du lancement du jeu, les robots ne pouvaient pas être positionnés sur n'importe quelle case. Certaines cases étaient dites "interdites". Il s'agissait des cases à jetons (en rouge sur l'image ci-dessous), ainsi que le centre du plateau (en jaune sur l'image ci-dessous), comme montré ci-dessous :

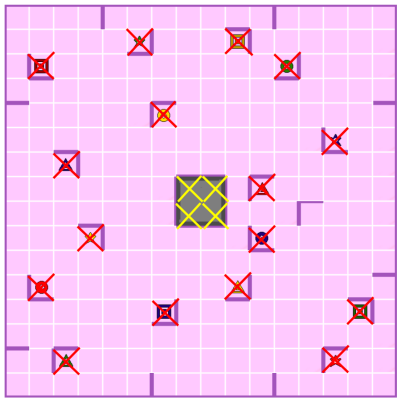


FIGURE 10 – Positions initiales interdites au lancement du jeu pour les robots

Bien évidement, deux robots ne pouvaient pas se situer sur la même case. Il fallait donc également vérifier qu'à chaque nouveau robot créé, les cases utilisées par les autres robots, ne soient plus disponibles pour ce nouveau robot.

Étant donné que les positions interdites étaient qu'une minorité, il était plus judicieux de d'abord tirer une position de manière aléatoire, c'est-à-dire un nombre représentant la position en abscisse du robot, et un autre pour la position en ordonnée. Á partir de ces deux nombres, il fallait ensuite vérifier si la coordonnée correspondante n'était pas similaire à une case interdite ou une position précédemment donnée à un robot. Si les coordonnées étaient correctes, alors on appliquer au robot ces coordonnées, et dans le cas où la coordonné tirée n'était pas valide, le processus recommençait tant que celles-ci n'étaient pas satisfaisantes.

L'algorithme correspondant à l'affectation des positions pour les robots est la suivante :

Algorithme 5 : POSITIONNEMENT DES ROBOTS À L'ÉTAT INITIAL

Sortie : Les coordonnées correctes pour le robot

```
1  surJeton ← false surRobot ← false surCaseInterdite ← false
2  do
3  |  aleaX ← tirage d'un nombre aleatoire entre 0 et 15
   |   aleaY ← tirage d'un nombre aleatoire entre 0 et 15 si surJeton alors
4  |  |  continue
5  |  fin
6  |  surCaseInterdite ← estSurCaseInterdite() si surCaseInterdite alors
7  |  |  continue
8  |  fin
9  |  si robotsExistants! ← ∅ alors
10 |  |  surRobot ← estSurAutresRobots(aleaX, aleaY)
11 |  |  si surRobot alors
12 |  |  |  continue
13 |  |  fin
14 |  fin
15 |  position[] ← aleaX, aleaY
16 while surJeton OR surCaseInterdite OR surRobot
17 retourner position[]
```

4.3 Déplacements et collisions des robots

Lorsque l'on veut déplacer un robot vers une direction, le déplacement devait s'effectuer qu'en ligne droite jusqu'à ce qu'il rencontre un obstacle. L'obstacle peut être un mur ou bien un autre robot. Pour que le robot s'arrête à un obstacle, il fallait mettre en place un système de collision.

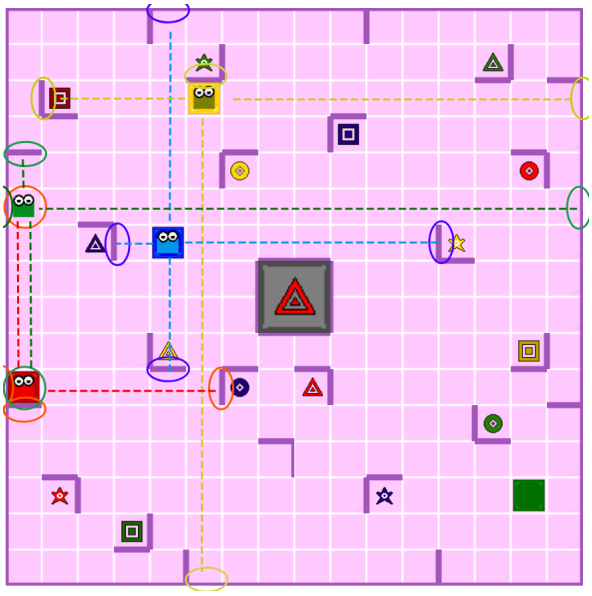


FIGURE 11 – Ensemble de collisions pour chaque déplacement des robots à un état donné

Pour pouvoir mettre en place ce système de collision lors d'un déplacement, il a donc fallu faire en sorte de vérifier la case actuelle du robot à déplacer, si elle contenait un mur sur son chemin et si la case suivante contenait un mur sur son chemin ou un autre robot.

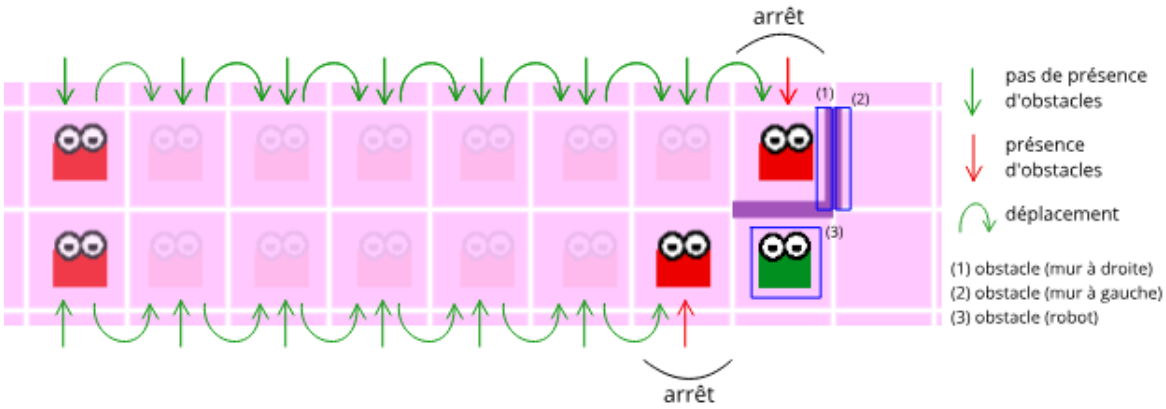


FIGURE 12 – Vérifications lors d'un déplacement d'un robot (vers la droite sur cette image)

l'algorithme correspondant à ces vérification est le suivant :

Algorithme 6 : DÉPLACEMENT D'UN ROBOT JUSQU'À COLLISION

Entrées : La direction de déplacement

```
1  si direction == haut alors
2  |  tant que pas de mur en haut de la case actuelle et
   |   pas de mur en basde la case suivante et !estUneCollisionRobot(direction, robot)
   |   faire
3  |  |  deplacement d'une case vers le haut
4  |  fin
5  fin
6  sinon si direction == bas alors
7  |  tant que pas de mur en bas de la case actuelle et
   |   pas de mur en haut de la case suivante et
   |   !estUneCollisionRobot(direction, robot) faire
8  |  |  deplacement d'une case vers le bas
9  |  fin
10 sinon si direction == gauche alors
11 |  tant que pas de mur a gauche de la case actuelle et
   |   pas de mur a droite de la case suivante et
   |   !estUneCollisionRobot(direction, robot) faire
12 |  |  deplacement d'une case vers la gauche
13 |  fin
14 sinon si direction == droite alors
15 |  tant que pas de mur a droite de la case actuelle et
   |   pas de mur a gauche de la case suivante et
   |   !estUneCollisionRobot(direction, robot) faire
16 |  |  deplacement d'une case vers la droite
17 |  fin
```

4.4 Sélection des robots

Pendant une partie, il est souvent nécessaire de déplacer les autres robots afin qu'ils fassent office d'obstacles, ce qui peut permettre au robot devant jouer (celui qui doit aller sur la case d'objectif) de récupérer le jeton en réalisant moins de mouvements qu'en temps normal.

Pour cela, nous voulions faire en sorte qu'au moment où le joueur clique sur un autre robot, c'est ce nouveau robot qui à la possibilité de se déplacer, et si le joueur clique ailleurs sur le plateau (en l'occurrence sur une case du plateau), ou sur le robot qui doit attraper le jeton, c'est de nouveau le robot devant jouer qui a la possibilité de se déplacer.

Pour mettre en place ce système, nous avons donc mis en place le design pattern *Observer*. Ce design pattern permet dans notre projet de gérer des événements de la souris sur un robot et une case. Dans notre projet, nous avons la classe State qui est à la fois l'observeur de l'événement sur un robot et l'observeur de l'événement sur une case. D'un autre côté, du côté des observables, nous avons la classe Case (qui signale un clic sur les cases) et la classe Robot (qui signale un clic sur les robots). Le pattern observer est mis en place de la manière suivante :

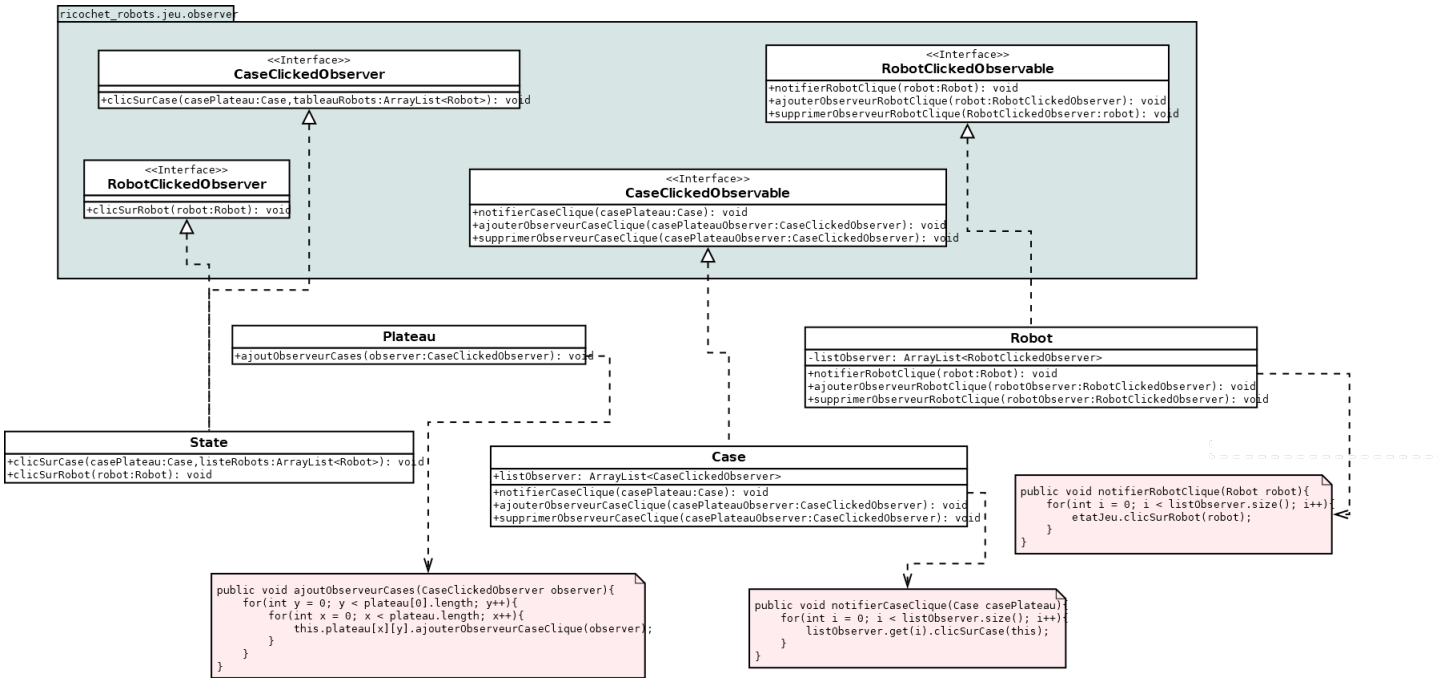


FIGURE 13 – Diagramme de l'implémentation du pattern Observer

Lors de la création de chaque robot, nous ajoutons à chacun d'entre eux, la classe State comme observeur. Dans la classe Plateau, nous réalisons la même chose sur chacune des cases (en ajoutant la classe State comme observer). La classe State implémente les interfaces CaseClickedObserveret RobotClickedObserver. Nous retrouvons dans cette classe les méthodes à redéfinir de ces interfaces, c'est-à-dire la méthode *clicSurCase(Case case)* et la méthode *clicSurRobot(Robot robot)*. Cet deux méthodes contiennent l'action réalisée lors du clic sur l'un de ces objets. Dans les classes Robot et Classe, nous avons une méthode qui est redéfini des interfaces RobotClickedObservable (pour la classe Robot) et CaseClickedObservable (pour la classe Case). Ces méthodes sont appelées lorsque le robot (ou la case) est cliqué(e). Cela permet de prévenir les observeurs du clic réalisé sur l'objet (en l'occurrence ici il s'agit uniquement de la classe State). Plus précisément, la méthode notifierRobotClique appelle directement la méthode *clicSurRobot()* pusqu'elle à accès à la classe State, et la méthode *notifierCaseClique()* récupère dans son ArrayList d'observeurs, les différents observeurs et appelle la méthode *clicSurCase()* de chaque observateur (la classe State).

Le contenu de la méthode *clicSurRobot()* est le suivant :

```
1 @Override
2 public void clicSurRobot(Robot robot){
3   robotSelect = robot;
4 }
```

Cela permet de changer la valeur de la variable robotActuel, puisque le robot qui se déplace est celui qui est sauvegardé dans la variable robotActuel. Ceci permet donc de changer de robot à déplacer.

Le contenu de la méthode *clicSurCase()* est le suivant :

```
1 @Override
2 public void clicSurCase(Case casePlateau){
3   robotAJouer();
4 }
```

Cet appel à la méthode *robotAJouer()* permet de rechercher le robot qui doit atteindre l'objectif parmi tout les robots existants. Une fois le robot trouvé, ce robot est affecté à la variable robotActuel, ce qui permet de récupérer le contrôle du robot devant jouer sans pour autant directement le sélectionner.

5 Implémentation de l'algorithme

5.1 Présentation de l'algorithme A*

L'algorithme A*[2] est un algorithme qui, à partir d'un état initial et d'un état final, permet de trouver le chemin le plus court sur une carte, un plateau, etc. La recherche de chemin s'effectue dans un graphe. Dans ce domaine, l'algorithme A* est un des algorithmes les plus efficaces. Il s'agit d'un algorithme qui est relativement rapide et s'il est bien optimisé, permet de trouver très rapidement la bonne solution.

Pour mettre en place l'algorithme A*, nous avons procédé en différentes étapes. Les étapes seront présentées tout au long de cette partie.

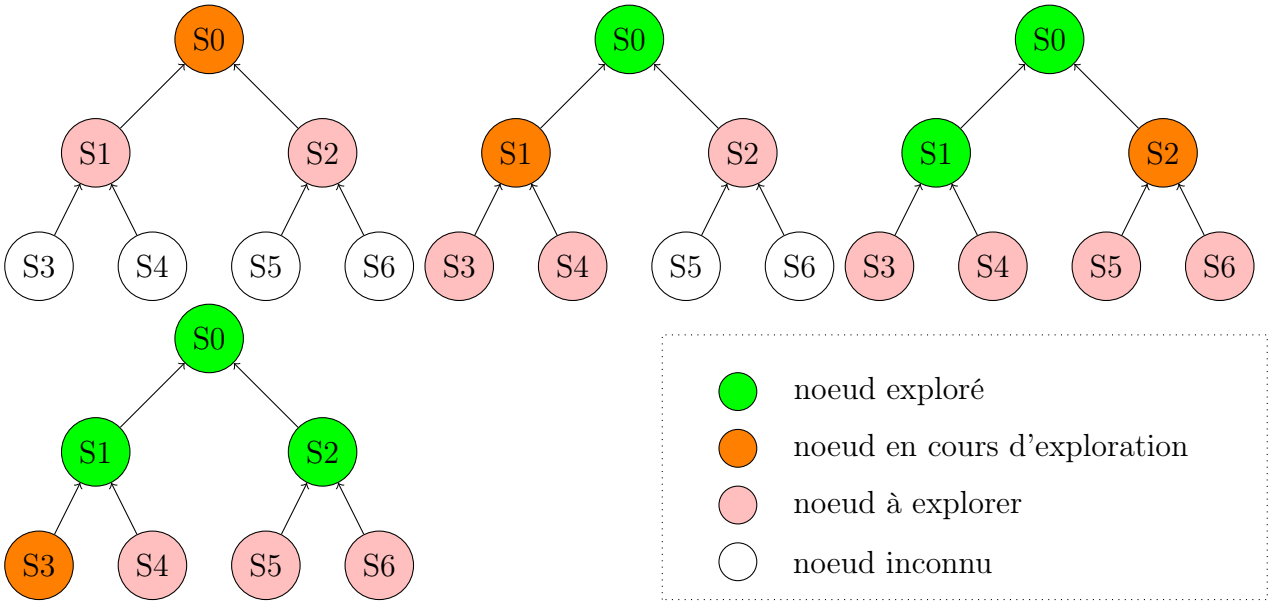
5.2 Première approche : l'algorithme de parcours en largeur (BFS)

5.2.1 Principe

Pour implémenter l'algorithme A*, nous sommes parti sur l'idée de commencer par implémenter un algorithme de parcours de noeud et de le modifier afin de le transformer en algorithme A*, qui est également un algorithme de parcours de noeud. Cela nous a permis de bien comprendre le principe de parcours de noeud de ce type d'algorithme. Un des algorithme les plus classiques dans ce domaine est l'algorithme de parcours en largeur, nous avons donc opté pour ce choix, qui était également proposé par un site présentant l'algorithme A*[3] (certaines des illustrations suivantes seront issues de ce site).

D'après Wikipédia[4], l'algorithme de parcours en largeur ou BFS, pour Breadth First Search en anglais, est un algorithme permettant le parcours d'un graphe ou d'un arbre étage par étage. L'idée est de commencer par explorer un nœud parent, puis l'ensemble de ces successeurs, puis les successeurs non explorés de ces successeurs, et ainsi de suite jusqu'à ne plus trouver de noeud.

Pour cela, l'algorithme fait en sorte de déplacer un noeud dans toute les directions possibles, génère un nouveau noeud à chaque déplacement, qu'il sauvegarde dans une liste de noeuds à explorer. Une fois le noeud exploré, il est retiré de cette liste. L'algorithme se termine lorsque la liste de noeud à explorer est vide.



5.2.2 Mise en place dans le Ricochet Robots

Dans le cadre du Ricochet Robots, l'idée était de déplacer chaque robot dans chaque direction, de générer un nouvel état après chaque déplacement et de le conserver dans une liste, si ce noeud est nouveau. Puisque nous avons quatre robots et quatre déplacements possibles, une profondeur contient 16x(profondeur de l'arbre), ce qui sera relativement long à résoudre si la solution se situe à une grande profondeur. Nous avons ci-dessous une représentation de l'application du BFS sur une plus petite grille de ricochet robots et avec seulement un robot représenté, pouvant se déplacer.

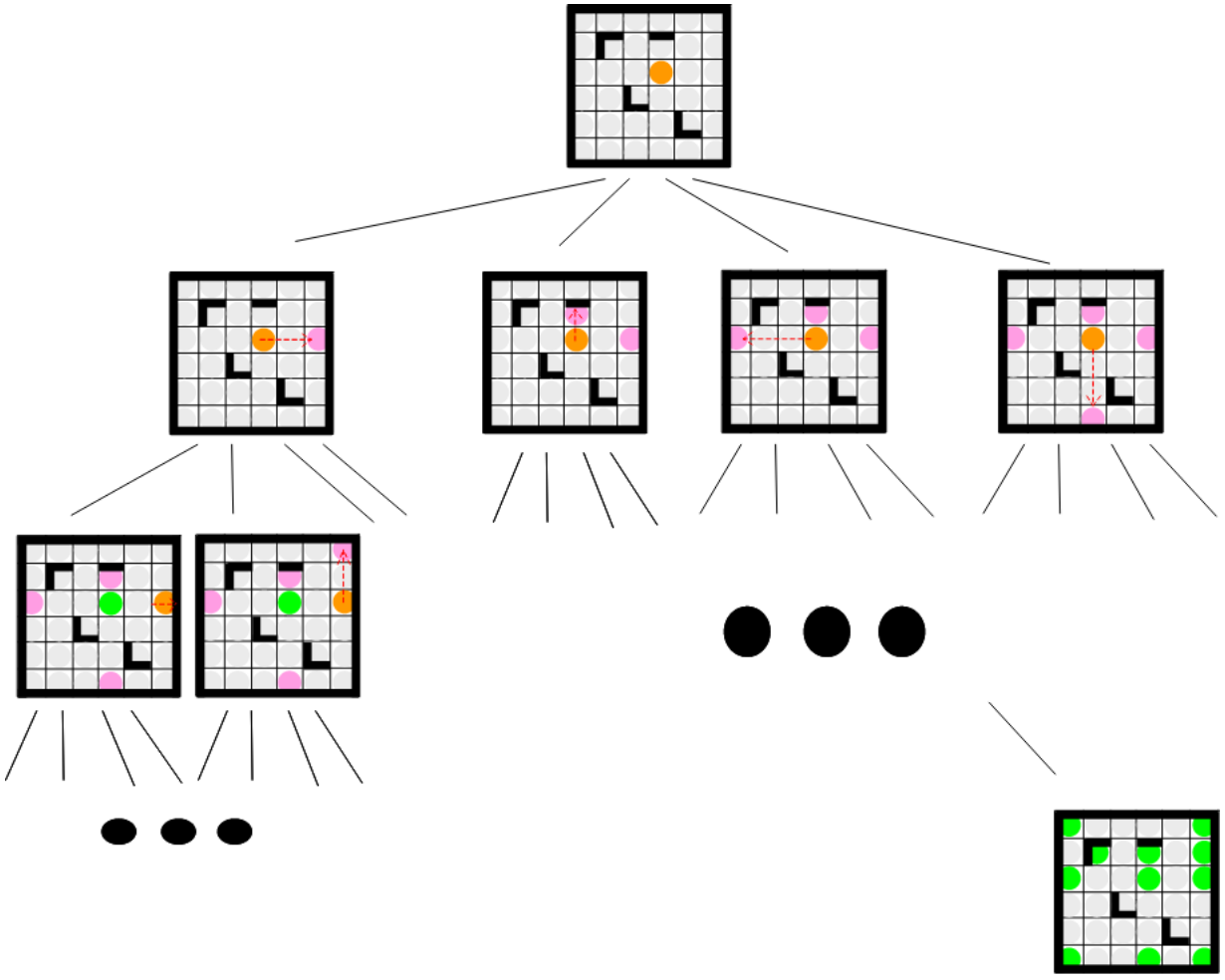


FIGURE 14 – Visualisation de l'algorithme BFS sur le Ricochet Robots pour un robot

5.3 Implémentation de l'algorithme A*

Au début, nous avons implémenté l'algorithme BFS tel quel, car cet algorithme était intéressant et présentait des similarités avec l'algorithme A*. Dans les deux algorithmes, le noeud suivant qui est pris correspond au premier élément d'une liste de noeud à explorer. Cependant, dans l'algorithme BFS, cette liste contient suite à suite les voisins du noeud courant. Ce procédé peut devenir très long et nous nous retrouvions à parcourir des noeuds qui n'étaient pas intéressants à la résolution du problème. Il a donc fallu le modifier pour se rapprocher du modèle de l'algorithme A*.

L'idée est alors d'affecter un coût à chaque noeud, afin de prioriser les noeuds ayant un coût moindre, ce qui nous permettait de nous rapprocher des noeuds solutions. Dans l'algorithme A*, le coût d'un état est alors calculé à la fois par rapport à la profondeur du noeud mais aussi d'une heuristique. Pour que les noeuds soient pris selon le coût du noeud il ne fallait plus ranger

la liste des noeuds à visiter selon les éléments voisins. Nous avons donc transformé cette liste en une liste à priorité (PriorityQueue). L'utilité est que chaque nouveau noeud parcouru est choisi comme étant celui qui minimise à la fois la distance parcourue et l'heuristique, qui est le principe même de l'algorithme A*.

Dans le cas du Ricochet Robots, un noeud est représenté par un état du jeu et la flèche par un déplacement d'un robot. Lorsque un robot réalise un déplacement, un nouvel état est créé. L'algorithme vérifie que ce nouvel état ne fait pas parti des noeuds à explorer, ou alors dans le cas ou il est déjà existant, il vérifie si ce nouveau noeud est plus avantageux que celui enregistré ou non. S'il s'agit d'un nouvel état ou d'un noeud avantageux, nous l'ajoutons à la liste des noeuds à explorer, sinon il est ignoré et nous passons à la direction suivante. Lorsque nous ajoutons un élément à la liste des noeuds à explorer, nous ajoutons également dans une HashMap à la fois ce nouveau noeud mais également l'état qui précédait celui-ci, ce qui permet par la suite de retrouver le chemin parcouru qui aura été fait pour atteindre cet état.

Á la fin de ce procédé, nous récupérons l'état gagnant de cette partie. Ensuite, nous recherchons dans la HashMap l'état gagnant (la valeur dans l'HashMap) ainsi que l'état qui a permis d'atteindre cet état (la clé qui est associé à la valeur), puis nous recherchons cet état précèdent et d'où il venait, et ainsi de suite jusqu'à obtenir l'état initial. L'ensemble des états successeurs sont ainsi conservés dans une Arraylist de State. Nous inversons ensuite cette ArrayList puisque le cheminement des états est inversé, nous avons commencé de la fin pour aller au départ.

Une fois ce procédé terminé, nous parcourons cet ArrayList et nous récupérons dans chaque état le cheminement des robots qui ont été bougés durant l'état et le cheminement de directions effectuées. Ces deux cheminement sont également conservés dans des ArrayList. Une fois terminé, nous affichons dans le terminal, la couleur du robot à déplacer ainsi que la direction à effectuer à chaque étape afin d'indiquer à l'utilisateur le chemin à parcourir pour atteindre la case d'objectif.

Le code de l'algorithme A* est le suivant :

Algorithme 7 : ALGORITHME A*

```
   Entrées : L'etat initial
 1 etatInitial ← l'etat initial ;
 2 PriorityQueue < State > frontier ← new PriorityQueue <> () ;
 3 HashMap < State, State > cameFrom ← new HashMap <> () ;
 4 HashMap < State, Integer > costSoFar ← new HashMap <> () ;
 5 ArrayList < State > path ← new ArrayList <> () ;
 6 ArrayList < Robot > listRobot ← new ArrayList <> () ;
 7 ArrayList < Deplacement > listDeplacement ← new ArrayList <> () ;
 8 etatInitial.setValCost(0) ;
 9 frontier.add(etatInitial) ;
10 cameFrom.put(etatInitial, null) ;
11 costSoFar.put(etatInitial, cout de l'etat initial) ;
12 tant que frontier n'est pas vide faire
13 |   noeudEnCours ← frontier.poll() ;
14 |   si objectif atteint alors
15 |   |   etatFinal ← noeudEnCours ;
16 |   |   break ;
17 |   fin
18 |   pour chaque direction faire
19 |   |   pour chaque robot faire
20 |   |   |   etatSuivant ← noeudEnCours.etatSuivant(direction, robot) ;
21 |   |   |   newCost ← cout de l'etat suivant + 1 + heuristique de l'etat suivant ;
22 |   |   |   si l'etat suivant est nouveau OR
   |   |   |    le cout de l'etat suivant est plus petit que celui delaliste alors
23 |   |   |   |   costSoFar.put(etatSuivant, newCost) ;
   |   |   |   |    etatSuivant ← cout de l'etat suivant + 1 ; frontier.add(etatSuivant) ;
   |   |   |   |    cameFrom.put(etatSuivant, noeudEnCours) ;
24 |   |   |   fin
25 |   |   fin
26 |   fin
27 fin
28 current ← etatFinal ;
29 tant que current! = etatInitial and current! = null faire
30 |   path.add(current) ;
31 |   current = cameFrom.get(current) ;
32 fin
33 si l'etat actuel existe alors
34 |   path.add(etatInitial) ;
35 |   pour chaque etat du chemin faire
36 |   |   listDeplacement.add(dernier deplacement de l'etat) ;
37 |   |   listRobot.add(dernier deplacement de robot de l'etat) ;
38 |   fin
39 |   reverse(listRobot) ; reverse(listDeplacement) ;
40 |   pour chaquedeplacementderobot faire
41 |   |   affiche le robot et la direction à effectuer
42 |   fin
43 fin
44 sinon
45 |   affiche "pas de solution"
46 fin
```

5.4 Mise en place de l'heuristique

Pour que l'algorithme A* soit opérationnel, cet algorithme a besoin d'une heuristique. L'heuristique permet d'aller plus rapidement vers la solution, et ainsi permet d'éviter d'explorer des noeuds qui n'ont pas d'intérêt à être visité. Parmi les heuristiques existants, le plus connu est celui du "vol d'oiseau". Cet heuristique permet de calculer la distance du pion actuel par rapport à l'objectif et ainsi, plus la distance est petite, plus le coût sera moindre et plus l'état sera priorisé. Par exemple dans l'image ci-dessous, dans un déplacement d'une case par une case, le noeuds explorés se dirigent directement vers la cible puisque que ce sont des états où le jeton est le plus proche de la solution, la distance est plus petite en allant directement vers sa cible.

La mise en place de cet heuristique permet une réelle optimisation. Nous pouvons ainsi le voir sur cette même image l'importance de l'heuristique.

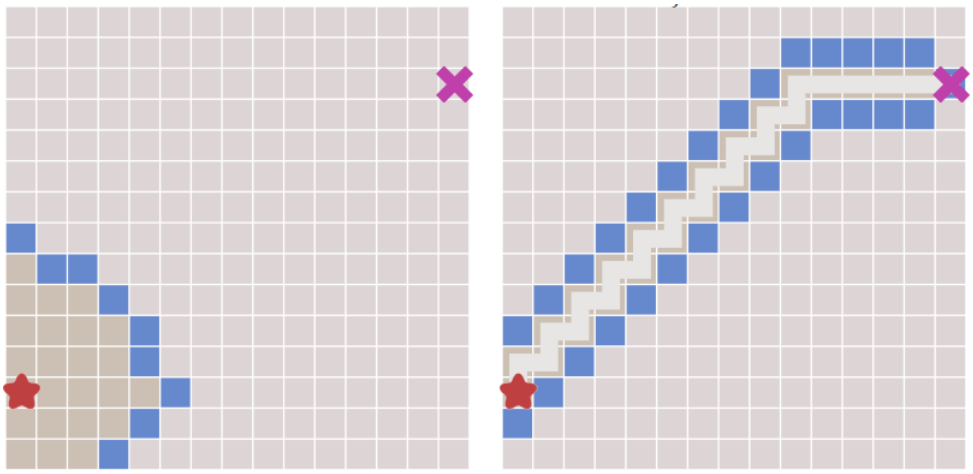


FIGURE 15 – image[3] sans et avec heuristique après 23 noeuds explorés

Cependant, ce type d'heuristique n'est pas adapté pour le ricochet robot, puisque le plateau n'est pas construit selon le modèle d'une géométrie standard comparé a une map par exemple. Le fait de se rapprocher de la case d'objectif ne permet pas forcément de récupérer plus rapidement le jeton. Souvent, il est nécessaire de s'éloigner du jeton pour mieux se rapprocher de celui-ci.

Nous avons donc mis en place une carte heuristique. La carte heuristique du ricochet robots consiste à donner à chaque case une valeur heuristique égale au nombre minimum de rebond qu'il faut faire pour atteindre l'objectif. Pour construire cette carte, il faut partir de la fin :

1. Toutes les cases qui sont à la verticale ou l'horizontale de l'objectif sans être bloquées par un mur ont une heuristique de 1.
2. Toutes les cases qui sont à la verticale ou l'horizontale d'une case possédant une heuristique de 1 et qui ne possèdent pas encore d'heuristique (ont une heuristique de -1) sont à une heuristique de 2.
3. Toutes les cases qui sont à la verticale ou l'horizontale d'une case possédant une heuristique de 2 et qui ne possèdent pas encore d'heuristique (ont une heuristique de -1) sont à une heuristique de 3.
4. Toutes les cases qui sont à la verticale ou l'horizontale d'une case possédant une heuristique de 3 et qui ne possèdent pas encore d'heuristique (ont une heuristique de -1) sont à une heuristique de 4.
5. Et ainsi de suite jusqu'à ce que toutes les cases du plateau qui sont accessibles aux robots aient une heuristique différente de -1.

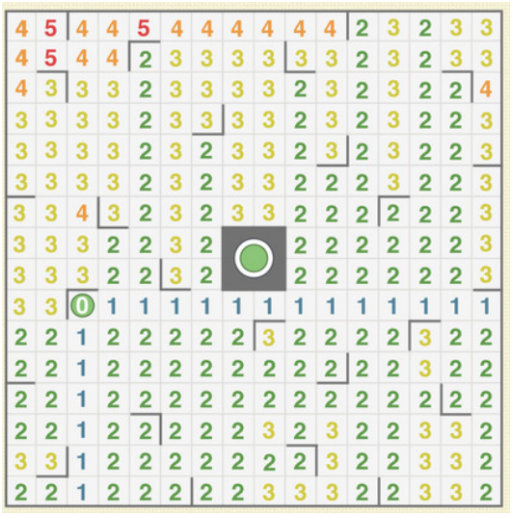


FIGURE 16 – Exemple de carte heuristique pour le Ricochet Robots[5]

Ainsi l'heuristique d'un état correspond à la valeur d'heuristique de la case sur laquelle se trouve le robot de la couleur de l'objectif.

L'heuristique doit être recalculé après chaque déplacement puisque le moindre mouvement d'un autre robot peut avoir un réel impact sur la partie, qui fausse la valeur de l'heuristique de la case puisque cet autre robot peut à la fois bloquer le chemin vers la solution ou bien favoriser le ricochet du robot.

6 Utilisation de l'application

6.1 Lancement du programme

Le lancement du jeu se fait par l'exécution du script de compilation, c'est-à-dire en réalisant la commande suivante à la racine du projet :

```
./build.sh run
```

Le lancement peut également se faire en cliquant directement sur le ficher "launcherBuild.sh", qui exécute la commande indiquée ci-dessus. Si nous ne fournissons pas d'arguments à cette commande, alors la taille de la fenêtre sera par défaut de 980x980 pixels. Si nous ajoutons comme argument par exemple 300 comme ci-dessous, alors la fenêtre fera du 300x300 pixels au lancement du jeu.

```
./build.sh run --args "300"
```

Le contenu de la fenêtre (le plateau, les robots et les autres éléments visuels relatifs au jeu) sont alors adaptés selon la taille de la fenêtre.

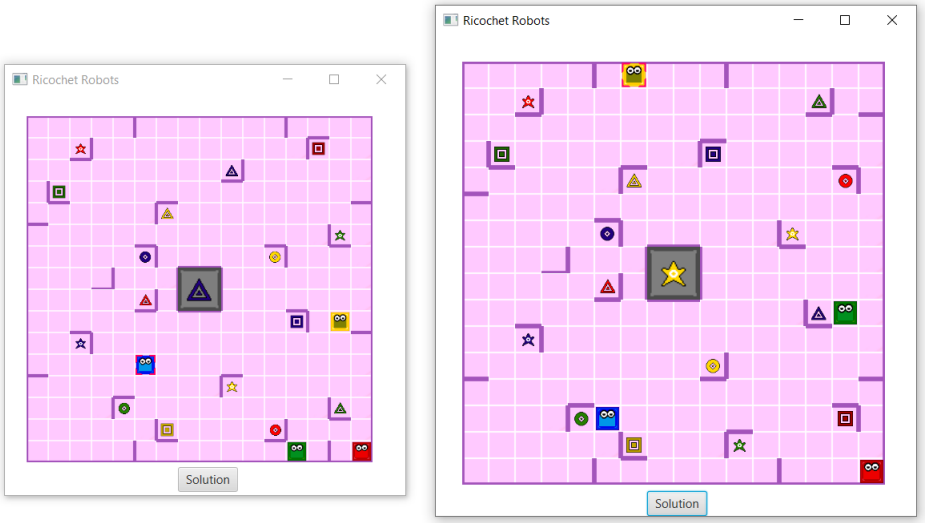


FIGURE 17 – Deux lancements avec 500px (à gauche) et 600px (à droite)

6.2 Jouabilité

Au lancement, nous avons un fenêtre affichant un plateau, quatre robots de différentes couleurs, un jeton au centre correspondant à la case sur laquelle le robot de la même couleur doit aller et un bouton "solution" en dessous de ce plateau.

FIGURE 18 – Exemple de fenêtre au démarage

Le robot pouvant se déplacer est celui qui a la marque de sélection. Sur l'image ci-dessus, il s'agit du robot bleu.

L'utilisateur peut déplacer le robot sélectionné dans les quatre directions (en haut, en bas, à gauche et à droite) avec les touches directionnelles du clavier.

L'utilisateur peut sélectionner un autre robot à déplacer en cliquant avec la souris sur cet autre robot.

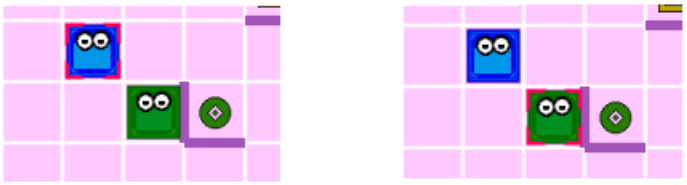


FIGURE 19 – Avant et après clic sur le robot vert. Le robot vert peut désormais être déplacé

Lorsque le robot atteint la case d'objectif, le socle de ce même robot se positionne sur la case d'objectif, les autres robots retournent sur leur socle et un nouveau jeton est tiré aléatoirement au centre du plateau, désignant ainsi la nouvelle case à aller.

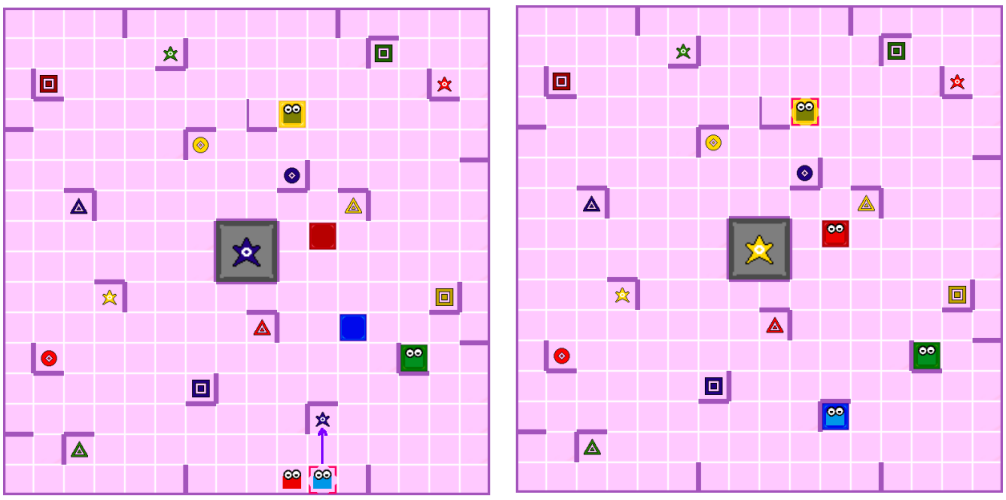


FIGURE 20 – Avant et après mouvement pour atteindre l'objectif

6.3 Lancement de l'algorithme A*

L'algorithme A* est utilisé dans cette application comme "outil de solution". Lorsque l'on veut savoir les étapes à effectuer pour atteindre l'objectif, l'utilisateur doit cliquer sur le bouton "solution" situé en dessous du plateau de jeu. Il faut ensuite que l'utilisateur regarde le terminal afin de connaître la solution nécessitant le moins de mouvements.

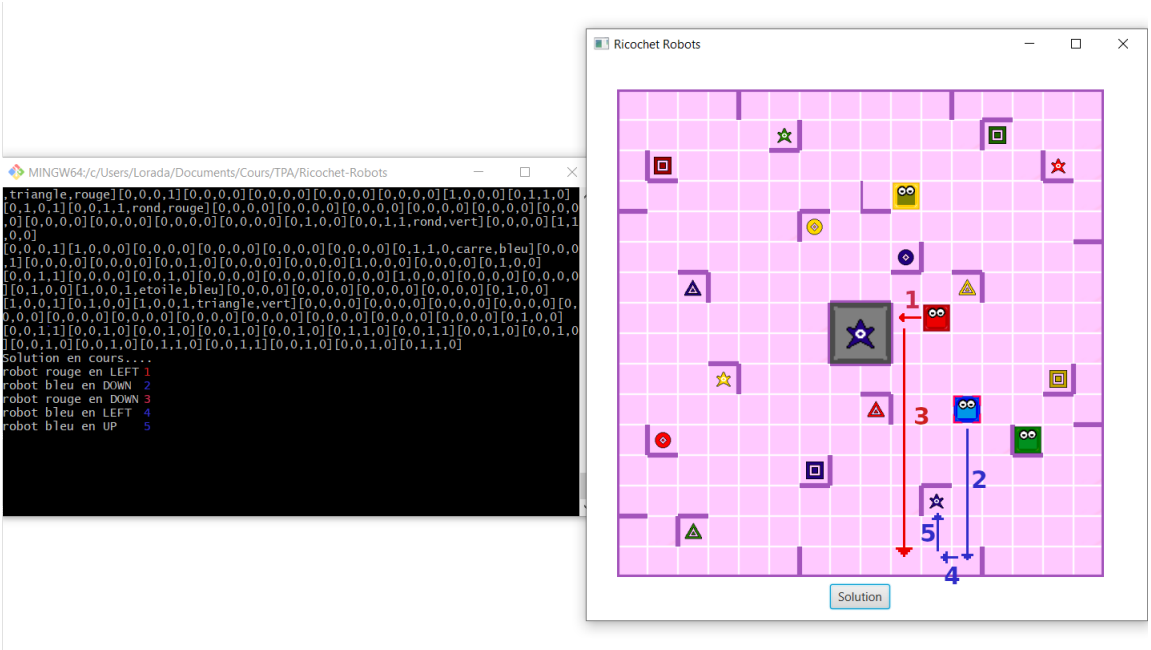


FIGURE 21 – Étapes successives indiquées par l'algorithme A*

7 Problèmes rencontrés

La partie suivante sera rédigée au nom de Lorada, car elle a été la seule personne à réaliser l'IA.

Au cours de ce projet, j'ai rencontré différents problèmes que je détaillerai ci-dessous :
Au niveau de la partie technique, un problème est survenu à partir de l'implémentation de l'algorithme A*, qui m'a pris un certain temps à résoudre.

Dans l'algorithme , au moment de comparer deux noeuds afin de savoir si le noeud en cours faisait était déjà présent parmi les noeuds explorés, j'ai utilisé la méthode *contains*. Cependant, lors du lancement de l'algorithme, après un certain temps, l'algorithme plantait, en indiquant un StackOverflow comme erreur. J'ai donc regardé la taille du tableau contenant les noeuds restants à explorer au fur et à mesure de l'algorithme et j'ai remarqué que la taille augmentait trop rapidement, comme si tout les noeuds en cours d'exploration s'ajoutaient. J'ai donc regardé le comportement de la méthode *contains*. j'ai remarqué que cette méthode ne marchait pas puisque quelle renvoyait toujours false lors de la vérification d'un nouveau noeud dans la HashMap des noeuds à explorer et ce, même si deux noeuds étaient identiques. Après différentes recherches, j'ai remarqué que j'avais redéfini les méthodes *equals* de l'état, du jeton tiré et de chaque robot, mais que je n'avais pas redéfini la méthode *hashcode* pour que la méthode *contains* fonctionne. J'ai donc regardé le comportement de la méthode *contains* et j'ai remarqué qu'il acceptait certains noeuds mais pas d'autres. J'ai donc sauvegarder dans un fichier une partie des noeuds de la liste des noeuds à explorer, en affichant le hashcode et la position de chaque robot pour chaque état. J'ai remarqué qu'il y avait beaucoup de noeud en plusieurs exemplaires alors que chaque noeud devaient être unique. La méthode *contains* n'était donc pas vraiment fonctionnelle. Au début je me suis demandée si le hashcode n'était pas bon, puisque je n'avais jamais redéfini cette méthode jusqu'à ce moment. Après avoir affiché ce que valait equals et hashcode pour chaque comparaison, les hashcodes étaient tous similaires pour deux états identiques et la méthode équals renvoyait bien true, à la différence de la méthode *contains* qui continuait de me retourner false. Après un long moment de recherche et de tests en vain, j'ai fais en sorte d'agrandir le hashcode de toute les classes ou celui-ci était redéfini, le problème s'est réglé de lui-même, bien que je n'ai pas d'explication à cela.

8 Conclusion

8.1 Conclusion générale

Ce projet était très intéressant. Nous ne connaissions pas le Ricochet Robots auparavant et c'était intéressant de se pencher, sur les règles de ce jeu et de le recréer afin de pouvoir y jouer autrement que sous forme physique. Cependant, malgré cette bonne expérience en terme de conception, l'implication faible ou inexistante des personnes du groupe à été un réel frein pour terminer le projet dans les temps et ce malgré la mise en place de différents dispositifs permettant une bonne cohésion de groupe. Le report de la date de rendu a permis de pouvoir corriger les manquements de l'algorithme A*, ce qui n'aurait pu se faire dans les conditions de temps initiales.

8.2 Partie personnelles

Lorada : Pour ma part, je connaissais déjà le principe de l'algorithme A*, à quoi il servait, mais je ne m'étais jamais réellement penchée dessus et je ne l'avais jamais implémenté auparavant. Bien que j'ai eu quelques difficultés à bien saisir dans le détail certains aspects de cet algorithme, j'ai trouvé très intéressant de le mettre en place dans ce projet.

8.3 Pistes d'améliorations

Bien que nous avons remplis les différents objectifs données pour ce projet, voici une liste non-exhaustive d'améliorations possibles :

- La mise en place du jeton spirale.
- La mise en place d'un système de score.
- Une mise en duel entre l'IA et le joueur : si le nombre de mouvements effectués par le joueur est supérieur au nombre minimum de mouvements prévus par l'algorithme A*, alors le joueur gagne un certain nombre de points. Dans l'autre cas, le joueur perd un certain quota de points selon le nombre de déplacements supplémentaires.
- L'ajout ou la suppression de robots sur le plateau, bien que ce soit différents du principe de base du Ricochet Robots.
- L'ajout d'obstacles (et/ou de malus) ou de bonus sur le plateau.
- Mise en place d'un système de difficulté, rendant plus ou moins punitif les déplacements supplémentaires réalisés par l'utilisateur.
- Mise en place du délai imparti à la réflection au nombre de mouvements à faire et un autre délai pour la réalisation du mouvement.

Références

[1] Wikipedia, ricochet robots. https://fr.wikipedia.org/wiki/Ricochet_Robots/.

[2] Wikipedia, algorithme a*. https://fr.wikipedia.org/wiki/Algorithme_A*.

[3] Red Blob Games. Introduction to the a* algorithm. https://www.redblobgames.com/pathfinding/a-star/introduction.html.

[4] Algorithme de parcours en largeur. https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur.

[5] A* algorithm. https://speakerdeck.com/randycoulman/solving-ricochet-robots?slide=108.