UNIVERSITÉ
CAEN
NORMANDIE

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

André Lorada
Auvray Théo
Licence 2 informatique - Groupe 1A

7 avril 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 chacune des cases 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 :

- Le développement du moteur du jeu suivant les règles du Ricochet Robots
- Réalisation d'une interface graphique.
- Implantation d'un algorithme de résolution naïf, appelé A*.
- 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 plate-formes 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 de la licence sur une seule plate-forme 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.

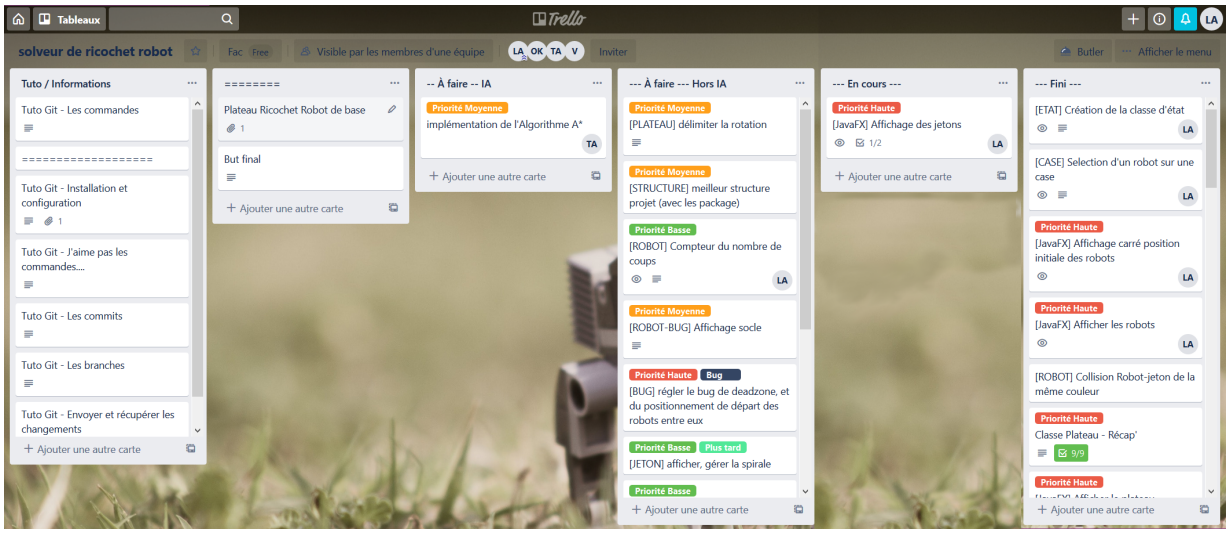


FIGURE 1 – Notre tableau Trello

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.

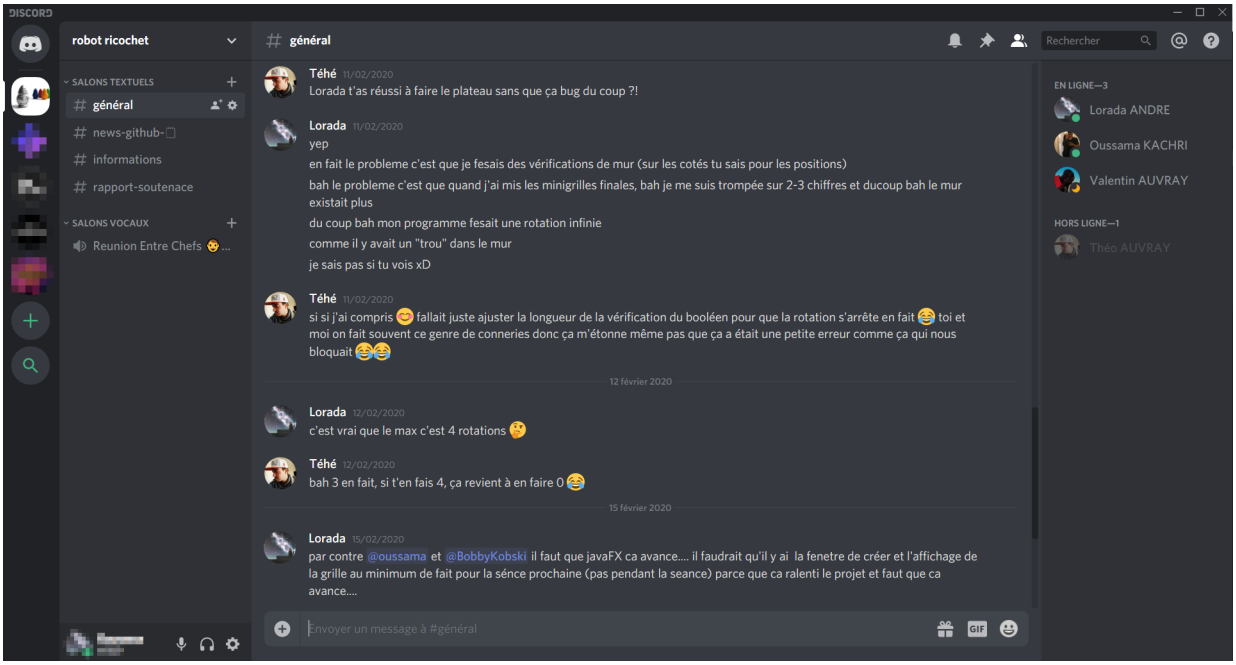


FIGURE 2 – Notre serveur Discord

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	
Score		
Création de la classe Score	X	
Création du système de score	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 , partiellement fontionnel	
Sortie Anticipée	X	
Mise en place de l'heuristique	manque de temps	
Rapport		
Rédaction du rapport	X	

3 Architecture du projet

3.1 Arborescence du projet

Le projet est structuré de la manière suivante :

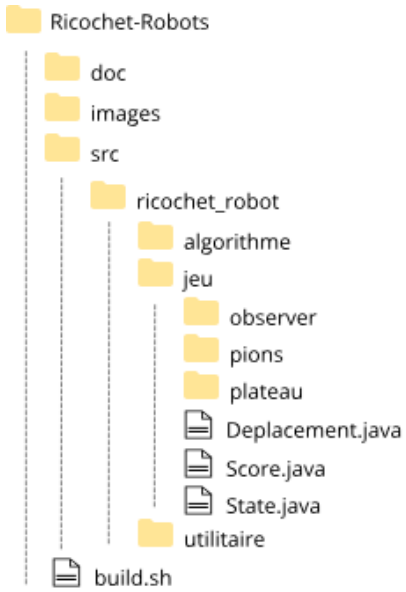


FIGURE 3 – Arborescence du projet

Á la racine de ce projet, nous y retrouvons :

doc : contient ce rapport et prochainement 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 au robot et au jeton
- **plateau :** contient les classes relatives à la création du plateau

utilitaire : contient une classe utilitaire.

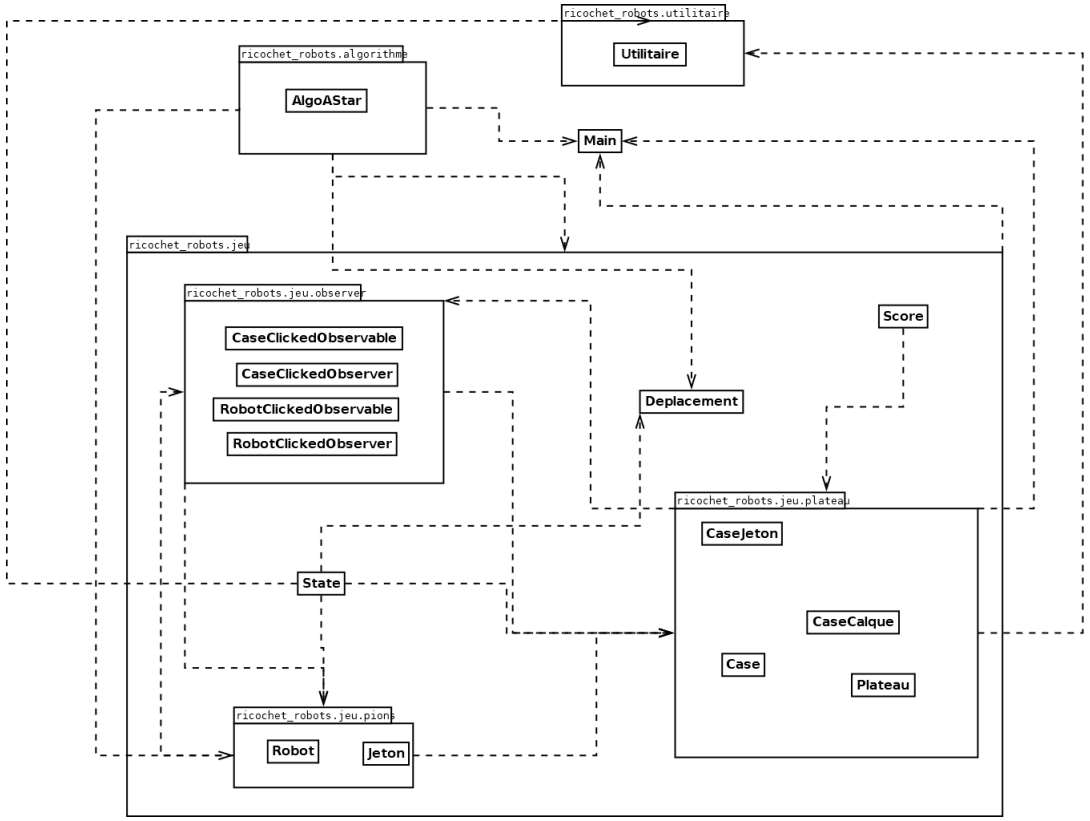


FIGURE 4 – Diagramme des packages

3.2.2 Diagramme des classes

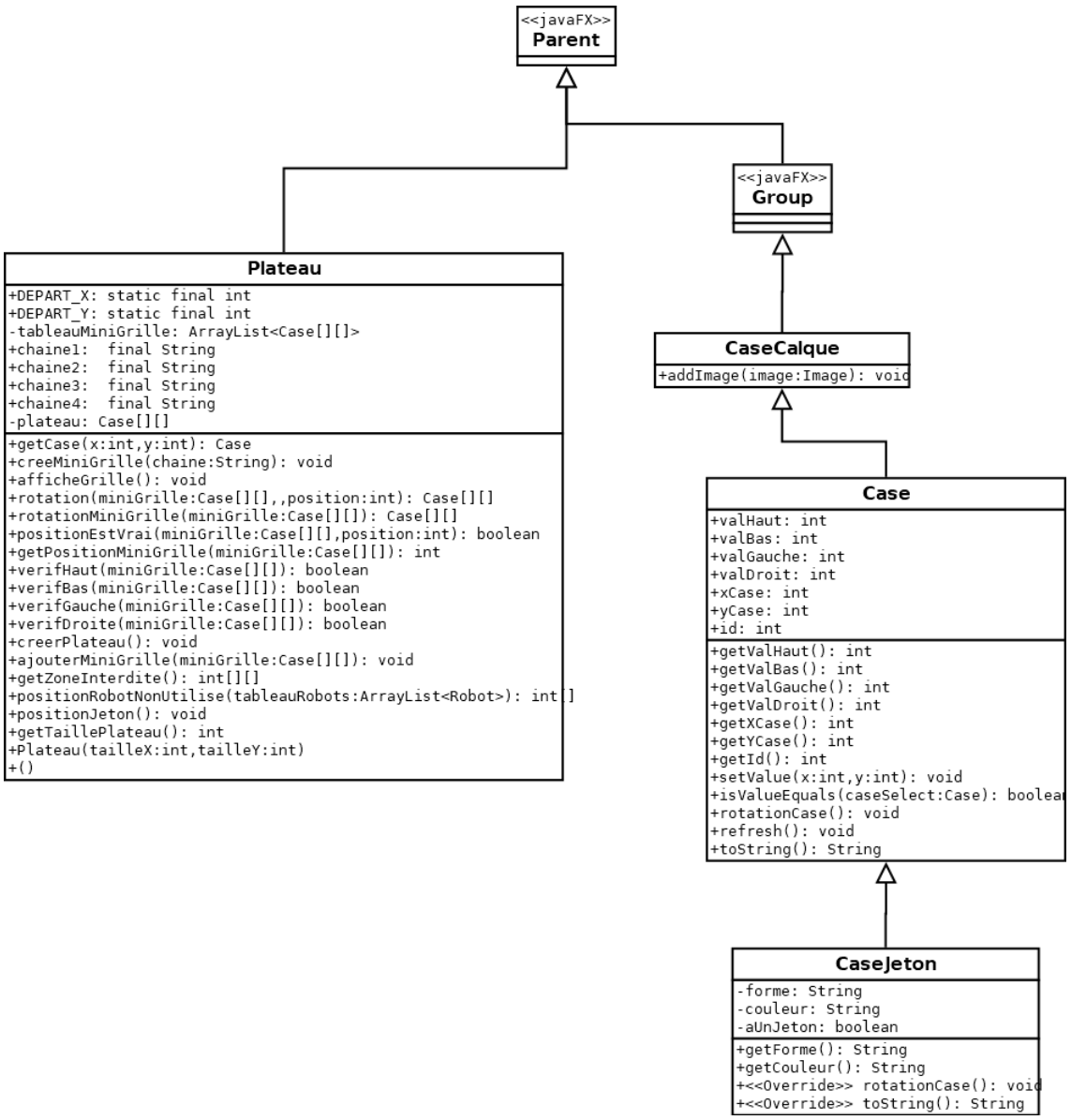


FIGURE 5 – Diagramme du package plateau

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 posés 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.

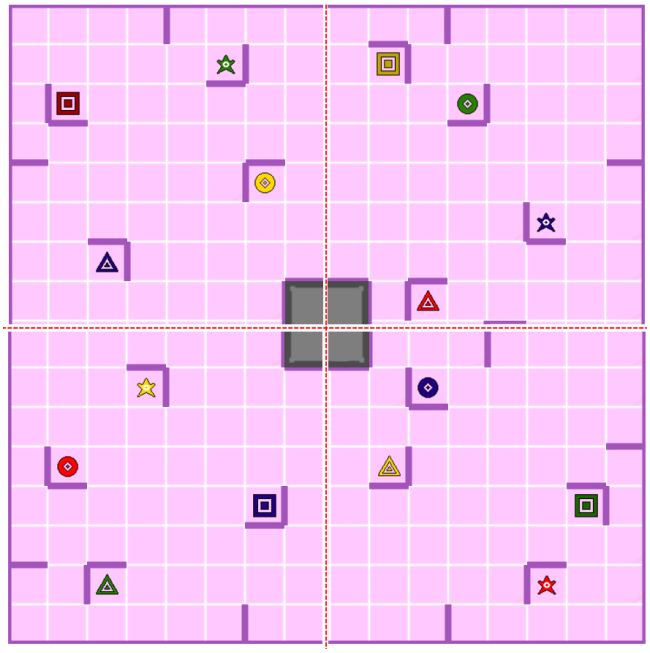


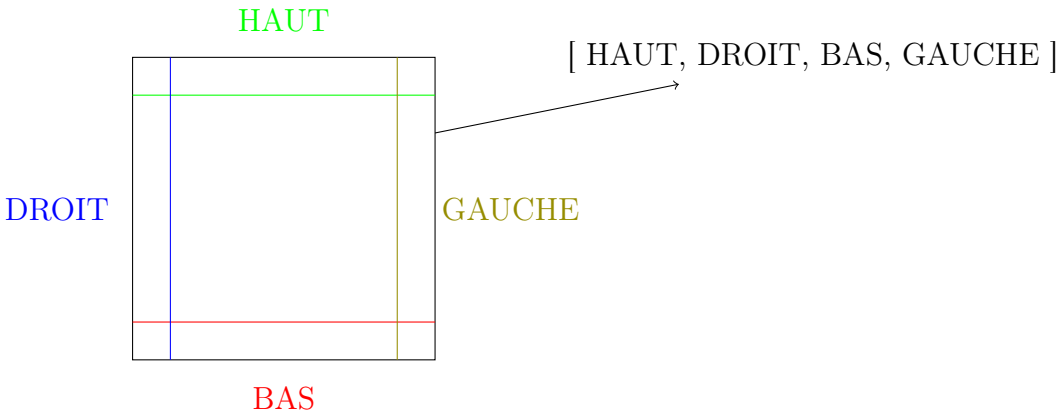
FIGURE 6 – Composition du plateau

Pour créer le plateau, nous devions commencer par la création de ces mini-plateaux. Ces quatre mini-plateaux ont été chacun, représentés par des tableaux en deux dimensions, contenant des cases et des murs.

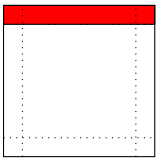
C'est à ce moment là que nous devions réfléchir sur la représentation de ces cases et de ces murs. Après observation du plateau, nous avons remarqué que ce plateau était composé que de onze cases différentes.

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 en haut 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 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 quatre variables nommées "valHaut, valBas, valGauche et valDroite" qui remplace le format de tableau [HAUT, DROIT, BAS, GAUCHE]. Nous avons alors avec un tableau à deux dimension dont chaque élément contient une instance de Case.

Á ce moment là, il fallait donc créer les 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'a partir ce cette sauvegarde, les mini-plateaux se créent.

Nous avons opté pour un modèle suivant :

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

Cette manière de faire permettait qu'a 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

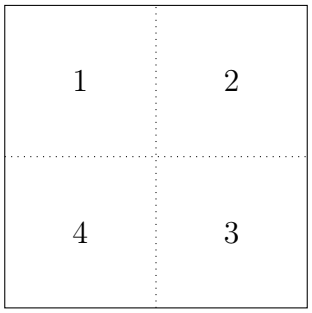
Entrées : Une chaine de caractère représentant le mini-plateau
Sortie : Le mini-plateau

```
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 type de 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 est 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.



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 faut que les murs de chaque mini-plateau soient bien positionnés, sinon nous nous retrouverions avec des plateaux de cette forme :

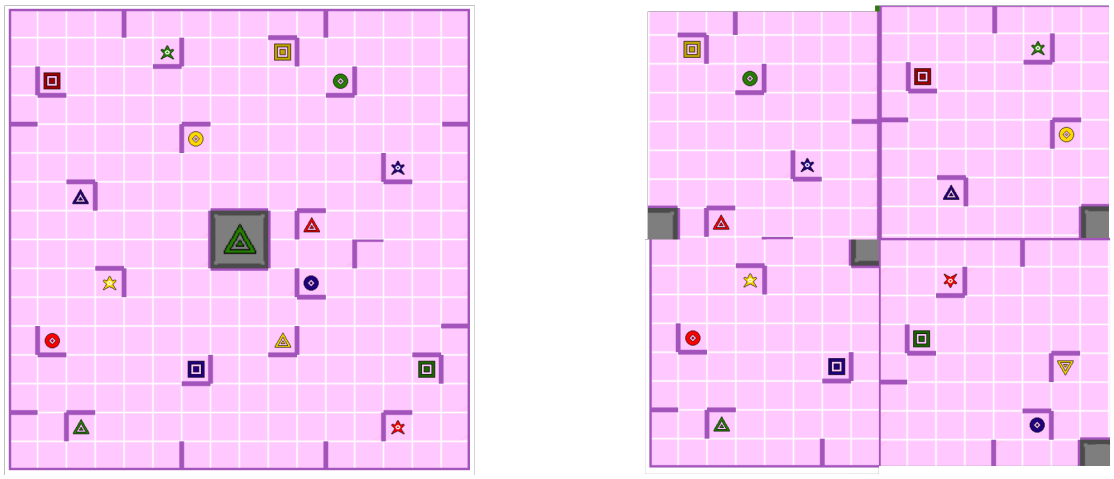


FIGURE 9 – Bon (à gauche) et mauvais plateau (à droite)

Pour cela, il faut 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 apr rapport à sa position :

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

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

[1,0,0,1] - [1,1,0,0] - [0,1,1,0] - [0,0,1,1]

Le code est le suivant :

```
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);
}
```

4.1.3 Création du plateau

Une fois que chaque mini-plateau était bien positionné, c'est à ce moment là que nous devions créer le plateau. Pour cela, nous avons crée un tableau à deux dimensions de 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
plateau[][] ← nouvelle instance de Case
 demiTabX ← divise le tableau en largeur par 2
 demiTabY ← divise le tableau en hauteur par 2
si la position du mini-plateau doit être à 1 alors
    pour y ← 0 à demiTabY faire
        pour x ← 0 à demiTabX faire
            plateau[x][y] ← miniPlateau[x][y]
        fin
    fin
fin
sinon si la position du mini-plateau doit être à 2 alors
    pour y ← 0 à demiTabY faire
        pour x ← demiTabX à tableau.length faire
            plateau[x][y] ← miniPlateau[x − demiTabX][y]
        fin
    fin
sinon si la position du mini-plateau doit être à 3 alors
    pour y ← demiTabY à tableau.length faire
        pour x ← demiTabX à tableau.length faire
            plateau[x][y] ← miniPlateau[x − demiTabX][y − demiTabY]
        fin
    fin
sinon si la position du mini-plateau doit être à 4 alors
    pour y ← demiTabY à tableau.length faire
        pour x ← 0 à demiTabX faire
            plateau[x][y] ← miniPlateau[x][y − demiTabX]
        fin
    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 où se situait les jetons, ainsi que la partie du milieu, comme montré ci-dessous :

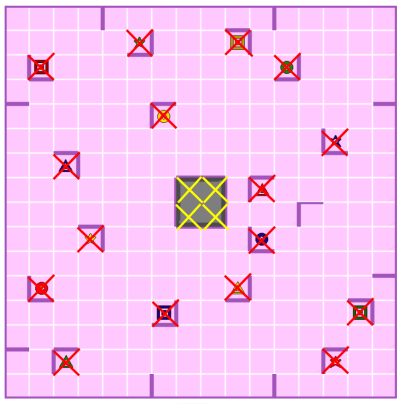


Figure 10 – Positions initiales interdites au lancement du jeu pour les robots

Bien évidement, deux robots ne pouvaient pas être 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 disponible 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 X du robot, et un autre pour la position Y. 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 donné a un robot. Dans le cas où la coordonné tirée n'était pas valide, le processus recommençait tant que celle-ci n'était pas satisfaisante.

L'algorithme correspondant à la position des 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 do
2      aleaX ← tirage d'un nombre alatoire entre 0 et 15
        aleaY ← tirage d'un nombre alatoire entre 0 et 15 si surJeton alors
3          continue
4      fin
5      surCaseInterdite = estSurCaseInterdite() si surCaseInterdite alors
6          continue
7      fin
8      si robotsExistants! ← ∅ alors
9          surRobot ← estSurAutresRobots(aleaX, aleaY)
10         si surRobot alors
11             continue
12         fin
13     fin
14     position[] ← aleaX, aleaY
15 while surJeton or surCaseInterdite or surRobot
16 retourner position[]
```

4.3 Déplacements et collisions des robots

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

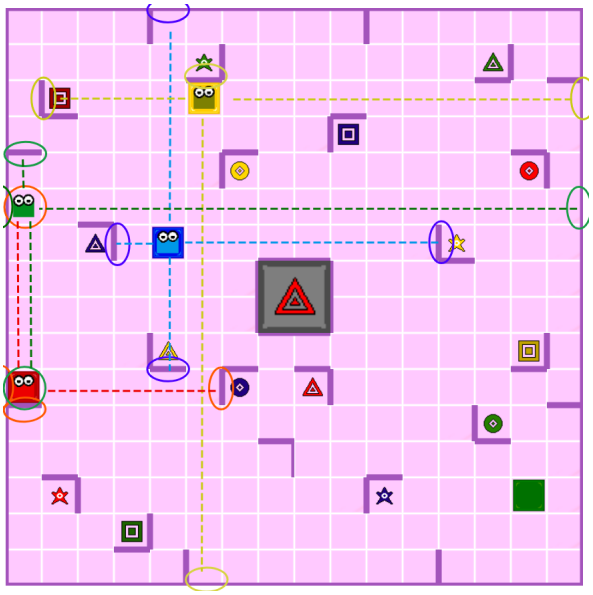


FIGURE 11 – Ensemble de collisions pour chaque déplacement des robots à un état donné

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.

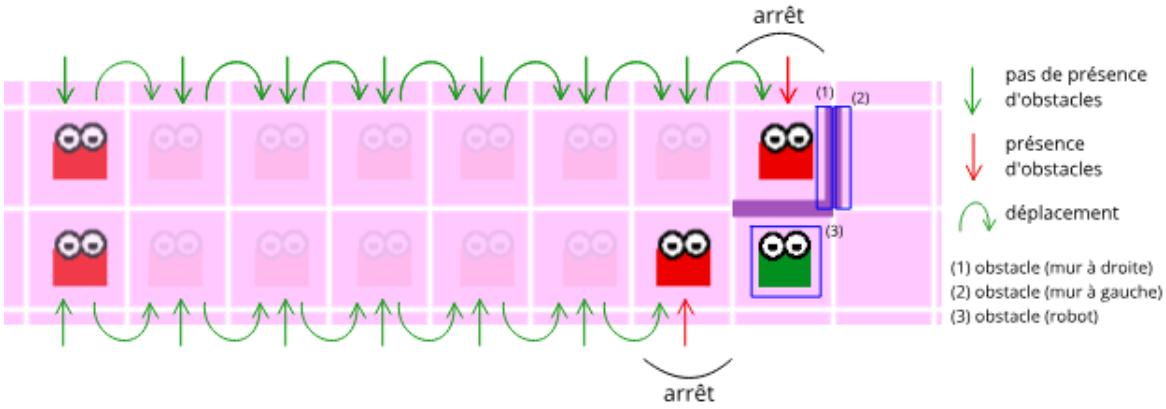


FIGURE 12 – Vérifications du robot lors du déplacement

l'algorithme correspondant à ces vérification est le suivant :

Algorithme 6 : Déplacement d'un robot jusqu'à collision

Entrées : La direction de déplacement

```
si direction == haut alors
    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
        deplacement d'une case vers le haut
    fin
fin
sinon si direction == bas alors
    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
        deplacement d'une case vers le bas
    fin
sinon si direction == gauche alors
    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
        deplacement d'une case vers la gauche
    fin
sinon si direction == droite alors
    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
        deplacement d'une case vers la droite
    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 de récupérer au robot devant jouer (celui qui doit aller sur la case correspondante) de récupérer le jeton avec 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 peut 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 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, la classe Case et la classe Robot représente les observables. Le pattern observer est mis en place de la manière suivante :

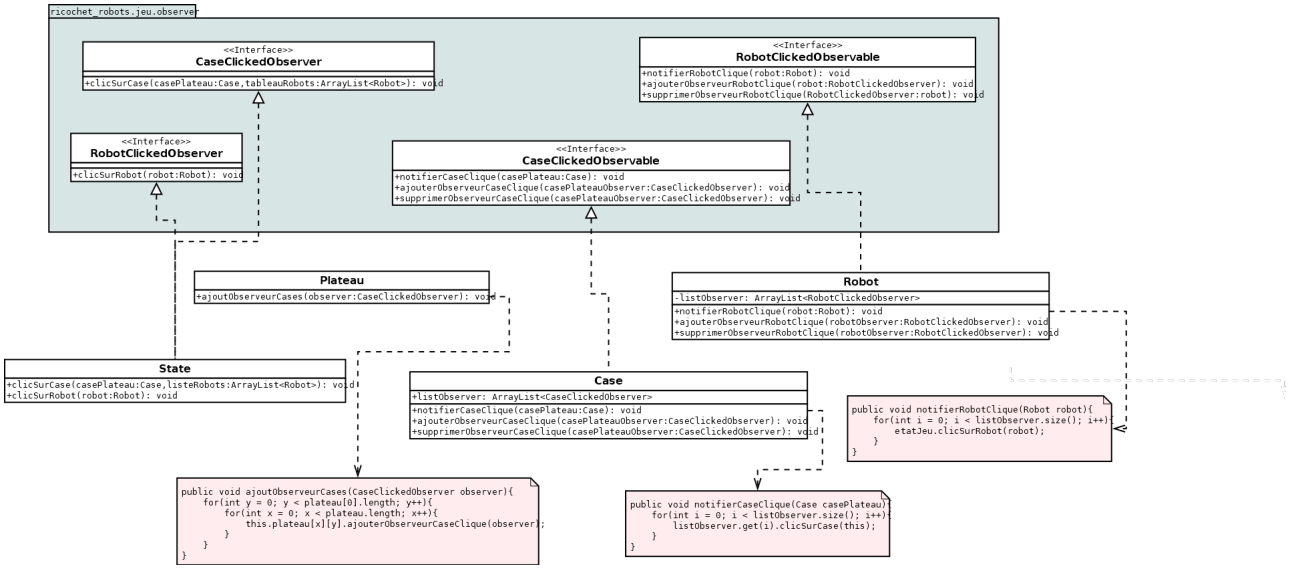


FIGURE 13 – Diagramme de l'implémentation du pattern observer

Lors de la création de chaque robot, nous ajoutons à chaque robot, la classe State comme observeur de chaque robot. Dans la classe Plateau, nous réalisons la même chose sur chacune des cases (en ajoutant la classe State comme observer). Dans la classe State, nous avons également les méthodes redéfinies des interfaces CaseClickedObserveret RobotClickedObserver, c'est-à-dire la méthode clicSurCase(Case case) et la méthode clicSurRobot(Robot robot). Cest 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 appelés lorsque le robot (ou la case) est cliquée. Cela permet de prévenir les observeurs du clic réalisé sur l'objet (en l'occurrence ici nous avons que la classe State qui est observeur). Plus précisément, la méthode notifierRobotClique appelle directement la méthode clicSurRobot() pusqu'elle à accès à la classe State, et la la méthode notifierCaseClique() va récupérer dans son Arraylist d'observeurs, les différents observeurs et va appeler la méthode clicSurCase() de chaque observateur, donc la classe State.

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

```
@Override
public void clicSurRobot(Robot robot){
        System.out.println("Robot" + robot.getCouleur() + "clique");
        robotSelect = robot;
}
```

Cela permet de changer le robot actuel, car au déplacement d'un robot, le déplacement se fait par défaut en fonction du de la variable contenue dans robotActuel.

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

```
@Override
public void clicSurCase(Case casePlateau){
        System.out.println("case " + casePlateau +"clique");
        robotAJouer();
}
```

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, et donc le joueur récupérait le contrôle du robot devant jouer.

Quand l'état de la classe change elle doit envoyer un signal a tout ses observateurs qui doivent effectuer l'action nécessaire en fonction du nouvel état de la classe.

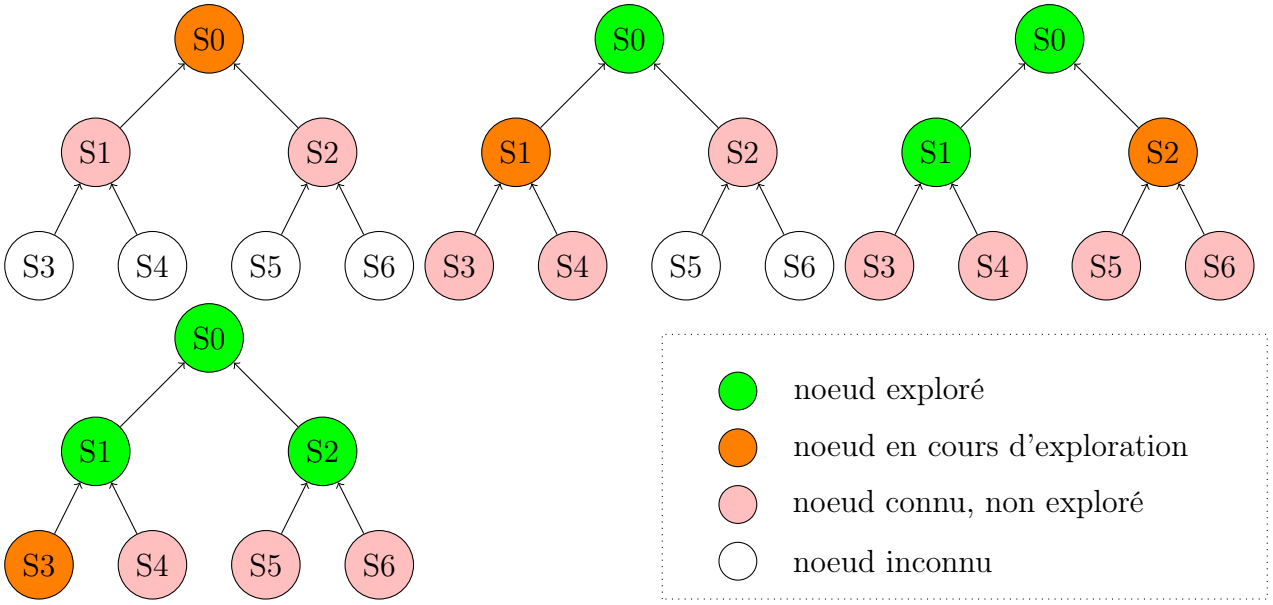
5 Implémentation de l'algorithme

5.1 Algorithme de parcours en largeur (BFS)

5.1.1 Principe

D'après Wikipédia[2], 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 comencer par explorer un nœud parent, puis l'ensemble de ces successeurs, puis les successeurs non explorés de ces successeurs, et ainsi de suite jusqu'a trouver une solution.

Il s'agit d'un algorithme très utile, à la fois pour la recherche de chemin, mais aussi pour certains types d'analyse de carte avec notamment l'analyse des cartes de distance.



5.1.2 Implémentation dans le Ricochet Robots

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, si ce robot ne reste pas sur la même case, un nouvel état de jeu est donc crée. Dès lors que l'on trouve un état gagnant, l'algorithme doit s'arrêter puisque le chemin vers ce noeud sera forcément le chemin le plus court avec son exploration étage par étage.

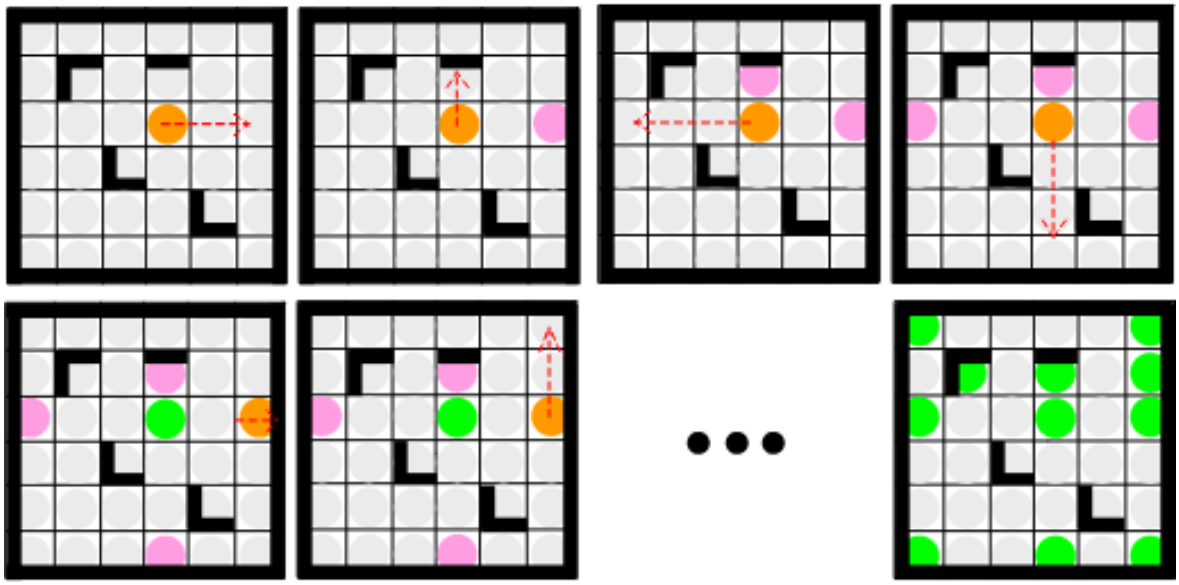


FIGURE 14 – Visualisation de l'algorithme BFS sur le Ricochet Robots pour un robot

Bien que cet algorithme fonctionne, il s'agit d'un algorithme qui met un certain temps pour résoudre le problème puisque le nombre de mouvements est souvent supérieure à 4 (observations faîtes durant les nombreuses simulations réalisées). Il faut donc pour cela trouver des méthodes d'optimisation.

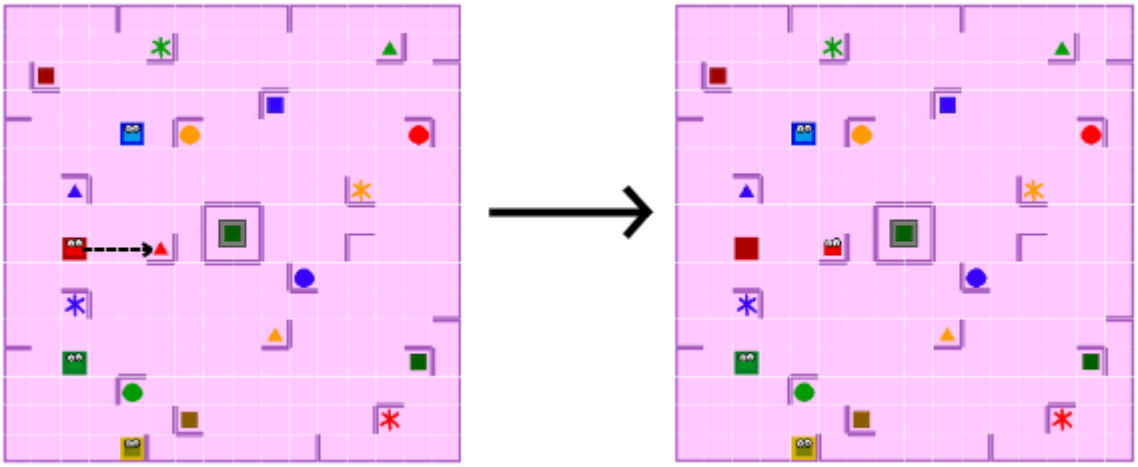


FIGURE 15 – État A vers un nouvel état B après déplacement

5.2 Optimisation

L'algorithme A* est un algorithme regroupant l'algorithme BFS et l'ajout d'une heuristique. Cette heuristique permet d'aller plus rapidement vers la solution, et ainsi permet d'éviter d'explorer des noeuds "inutiles". Parmi les heuristiques existants, le plus connu est celui du "vol d'oiseau". Cet heuristique permet de prioriser certains déplacements. Par exemple dans l'image ci-dessous, dans un déplacement d'une case par une case, les mouvements prorisées seront les déplacements vers le haut et la gauche.

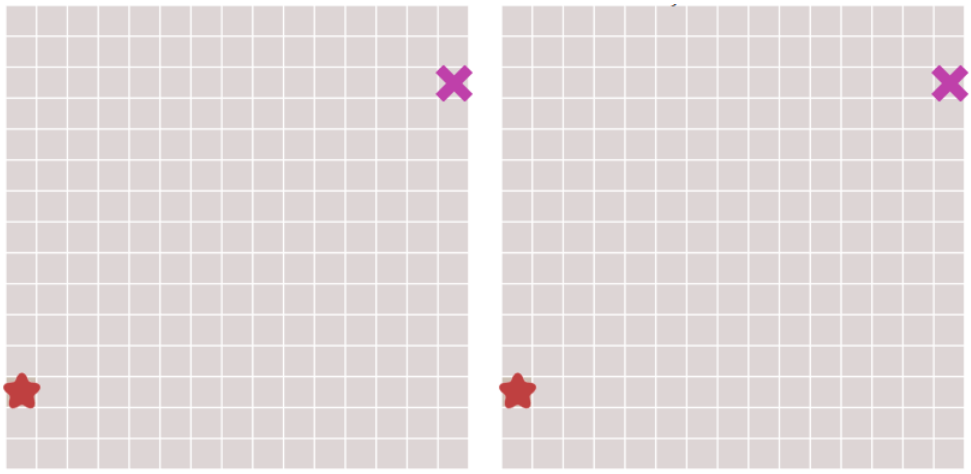


FIGURE 16 – image[3] avec et sans heuristique à l'état initial

Cet heuristique permet une réelle optimisation. Nous pouvons ainsi le voir sur cette image l'importance de l'heuristique.

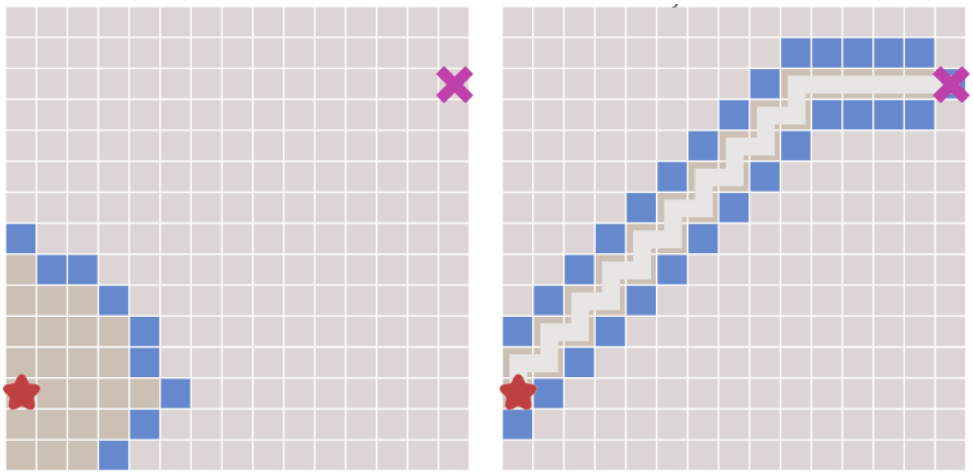


FIGURE 17 – image[3] sans et avec heuristique après 23 noeuds explorés

Ainsi, pour connaître l'ensemble des états possibles d'une partie, le but était de déplacer chaque robot, dans chaque direction, et de sauvegarder chaque nouvel état. Lorsque qu'il y avait deux états similaires, c'est-à-dire lorsque les quatre robots avaient déjà connu ces positions, cet état ne se sauvegardait pas puisqu'il a déjà été exploré.

6 Conclusion

6.1 Objectifs remplis

6.2 Pistes d'améliorations

Références

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

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

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