UNIVERSITÉ CAEN NORMANDIE

- Rapport Projet -
Conception Logicielle avancée
Solveur Robot Ricochet

Erwan Phillipe MENSAH
TOURE Papa Samba Khary
Daouda TRAORE

L2 Informatique 2021-2022 Université Caen Basse-Normandie.

Table des matières

1 Introduction

Dans le cadre de notre second semestre , il nous a été proposé multiples sujets nous permettant de mettre en pratique nos connaissances et nos compétences. Parmis tous ces sujets celui qui nous a le plus intéressé fut le **solver** du jeu de société "**Ricochet Robots**.
Comme c'est le cas pour beaucoup de jeux de société, il est possible d'en réaliser une application.
Le but du projet etait de dévolopper un programme capable de trouver une solution optimale pour toute situation du jeu.
Trois objectifs étaient à remplir. Dans un premier temps il faillait développer le **moteur du jeu** puis d'implémenter un **algorithme de résolution A*** et de l'optimiser étant donné la compléxité du problème et finalement de réaliser une **interface graphique**.
Ce projet nous a donc amené à travailler sur des connaissances qui sont nouvelles pour nous, que ce soit les algorithmes de résolution ou encore des choses complexes du développement des applications. Ce rapport a pour objectif de présenter comment nous avons travaillé et quels ont été les multiples obstacles que nous avons du faire face.
Le présent rapport est organisé en trois grandes parties. La première est consacrée à la présentation du jeu, puis nous parlerons de la répartition des tâches, du développement du moteur, de la réalisation du solveur, enfin de la création de l'interface graphique et des problèmes rencontrés.

2 Présentation

Figure 1: Ricochet Robots

Ricochet robots est un jeu de société pour deux personnes voir plus, imaginé par Alex Randolph. L'objectif du jeu est de déplacer les robots afin d'atteindre la cible en un minimum de mouvements possibles avec des restrictions strictes sur les movements des robots. Le jeu fut publier pour la première fois en Allemagne sous le nom de **Rasende Roboter** en **1999**.

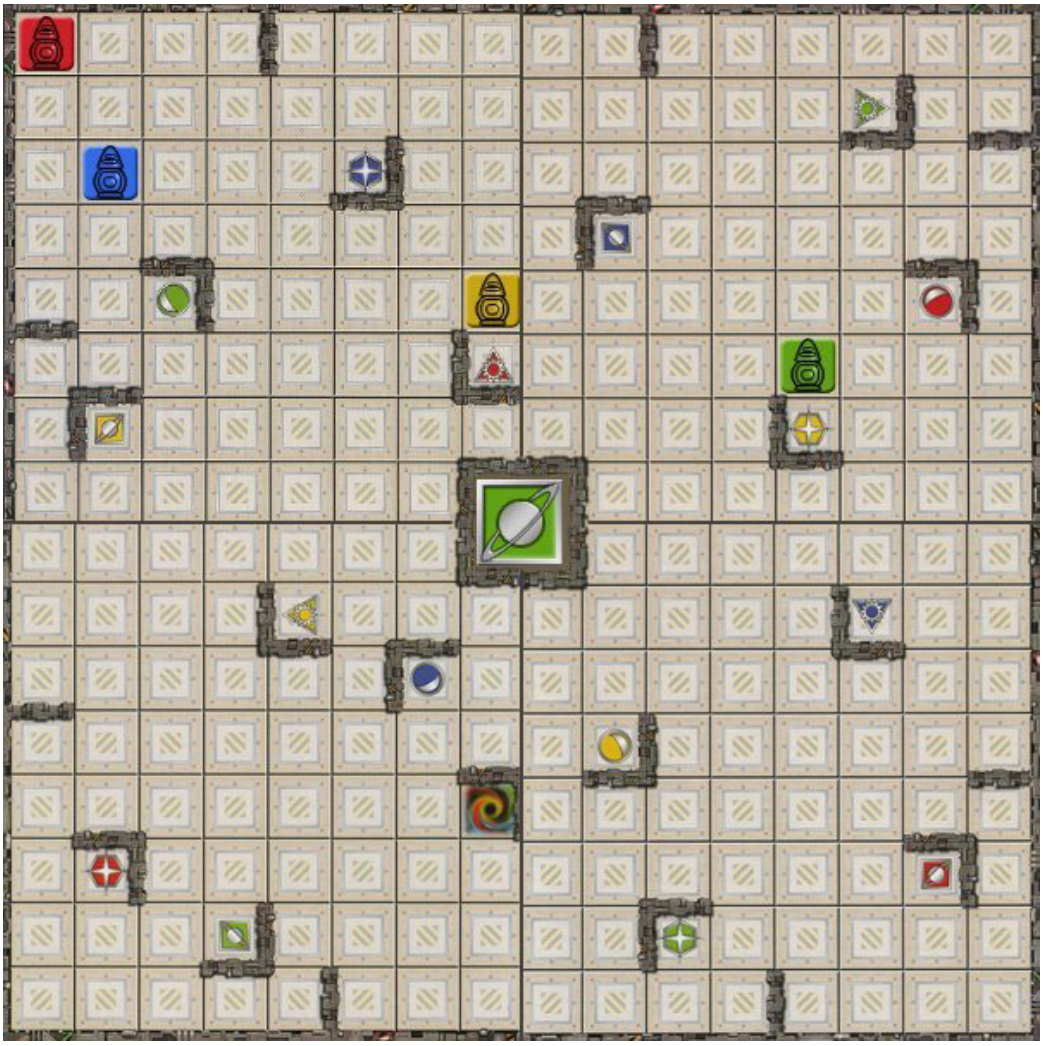


Figure 2: Dans l'exemple ci-dessus, l'objectif, indiqué par le jeton placé au centre, est la case verte contenant la planète saturne. Tous les joueurs devront donc s'efforcer de trouver en un minimum de coups le moyen de déplacer le Robot vert jusqu'à cette objectif

Contenu

- 4 plateaux imprimés sur les deux faces.
- 1 pièce centrale en plastique
- 4 robots en 4 couleurs.
- 4 jetons carrés aux couleurs des robots.
- 17 jetons de cible ronds.
- 1 sablier (environ 1 minute)

Préparatifs

- Placez les quatre plateaux avec le coin troué au centre. Il y a 96 façons différentes de poser les plateaux. Fixez les plateaux à l'aide de la pièce de blocage en plastique.
- Posez au hasard les quatre jetons colorés sur des cases libres. Placez les robots sur les carrés de leurs couleurs respectives.
- Mélangez les 17 cibles faces cachées sur la table.
- Placez le sablier à côté du plateau de jeu. Retournez une cible au hasard et placez-la face visible sur la pièce de blocage centrale. Le jeu peut commencer.

But du jeu

Lors de chaque tour, le but est de gagner le jeton cible au centre. Sur le plateau, le même dessin est représenté une fois, avec le même symbole et la même couleur. C'est la cible du tour. Votre but est de trouver comment le robot de même couleur, dit robot actif, peut atteindre cette cible en le moins de coups possible. Le joueur qui atteint la cible en le moins de mouvemenrts gagne le jetob. Celui qui gagne le plus de jetons gagne la partie !

Mouvements

Au début d'un tour, les robots ne bougent que dans les cerveaux des joueurs. Chaque joueur essaye de découvrir le moyen le plus économe en mouvements pour amener le robot actif sur la cible. Les robots se déplacent horizontalement ou verticalement, sans tourner, selon les directives des joueurs, mais une fois en mouvement, ils ne peuvent plus s'arrêter jusqu'à ce qu'ils rencontrent un obstacle devant lequel ils s'arrêtent. Les obstacles sont les bords du plateau, les murs dessinés, la pièce centrale et les autres robots. Lorsqu'un robot heurte un obstacle, il peut s'arrêter ou ricocher à angle droit, à droite ou à gauche, jusqu'à ce qu'il heurte un nouvel obstacle, et ainsi de suite. Chaque mouvement d'un robot jusqu'à un obstacle compte comme un mouvement.

Fin du jeu

Une partie à 2 se termine dès qu'un joueur a gagné 8 jetons ; une partie à 3 se termine avec un gain de 6 jetons et une partie à 4 avec un gain de 5 jetons. Si plus de 4 joueurs participent, continuer jusqu'à ce que tous les jetons soient gagnés

A noter

La plupart des situations peuvent être résolues en moins de 10 mouvements, mais il arrive dans certains cas qu'il faille vingt mouvements ou plus. De telles situations sont intellectuellement intéressantes.
Il existe également d'autres variantes, 54 au totale. Certaines versions ont un robot en plus tandis que d'autres ont en plus des murs placés diagonalement augmentant ainsi grandement la difficulté.

3 Organisation

Pour la répartition des tâches elle a été faite sans véritable organisation vu que au début nous ne savions pas vraiment ou nous allions mais nous pouvons considérons la répartition suivante:

Erwan MENSAH :

- Réalisation du moteur
- Partie Graphique
- rapport

Daouda TRAORE :

- Réalisation du moteur
- Partie Graphique
- Rapport

TOURE Papa :

- Réalisation du moteur
- Solveur
- Rapport

Pour la prochaine fois une meilleure organisation sera notre premier objectif, mais étant de amateurs avancer à l'aveuglette était la seule chose à notre portée.
Le tableau suivant donne une répartition plus en détail des tâches.

Tâches	TOURE	TRAORE	MENSAH
Board			
Création de la classe Board	X	X	X
Création des quarts de plateau	X	X	X
Assemblage des quarts de plateau	X	X	X
Génération aléatoire du plateau	X		
Rotation des quarts de plateau	X		
Println du plateau		X	X
Robot + Coord			
Création des classes	X	X	X
Target + Move			
Création des classes	X	X	X
GameView			
Création de la classe GameView		X	X
Affichage graphique des robots		X	X
Affichage graphique de la cible		X	X
Affichage graphique des murs		X	X
GUI			
Création de la classe GameGUI		X	X
Fonctionnalité des bouttons	X	X	X
Pattern MVC			
Implémentation du pattern MVC	X	X	X
State			
Création de la classe State	X		
Création de l'état initial du jeu	X		
Positionnement aléatoire des robots	X		
Séléction aléatoire de la cible	X		
Définition de l'état gagnant	X		
Calcul des différents moves	X		
Héritage de Runnable	X		
Algorithme A*			
Création de la classe aStar	X		
Création de la classe aStarMax	X		
Mise en place de l'heuristique	X		
Création de la Classe aStarFinalForm	X		
Documents			
Documentation du code		X	X
Rédaction du rapport	X	X	X
Rédaction du support de soutenance	X	X	X

3.1 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.

3.1.1 Hébergement du code

Afin de faciliter la gestion du projet, nous avons utilisé à la fois la Forge d'Unicaen qui permet de créer et d'administrer des dépôts sous Git très facilement par l'intermédiaire d'une interface web. D'ailleurs pour tous les membres l'utilisation de GIT fut une première, donc un temps d'acclimatisation et d'apprentissage nous a été necessaire. D'autres fonctionnalités sont disponibles sur cette plateforme 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(plus à but personnel).

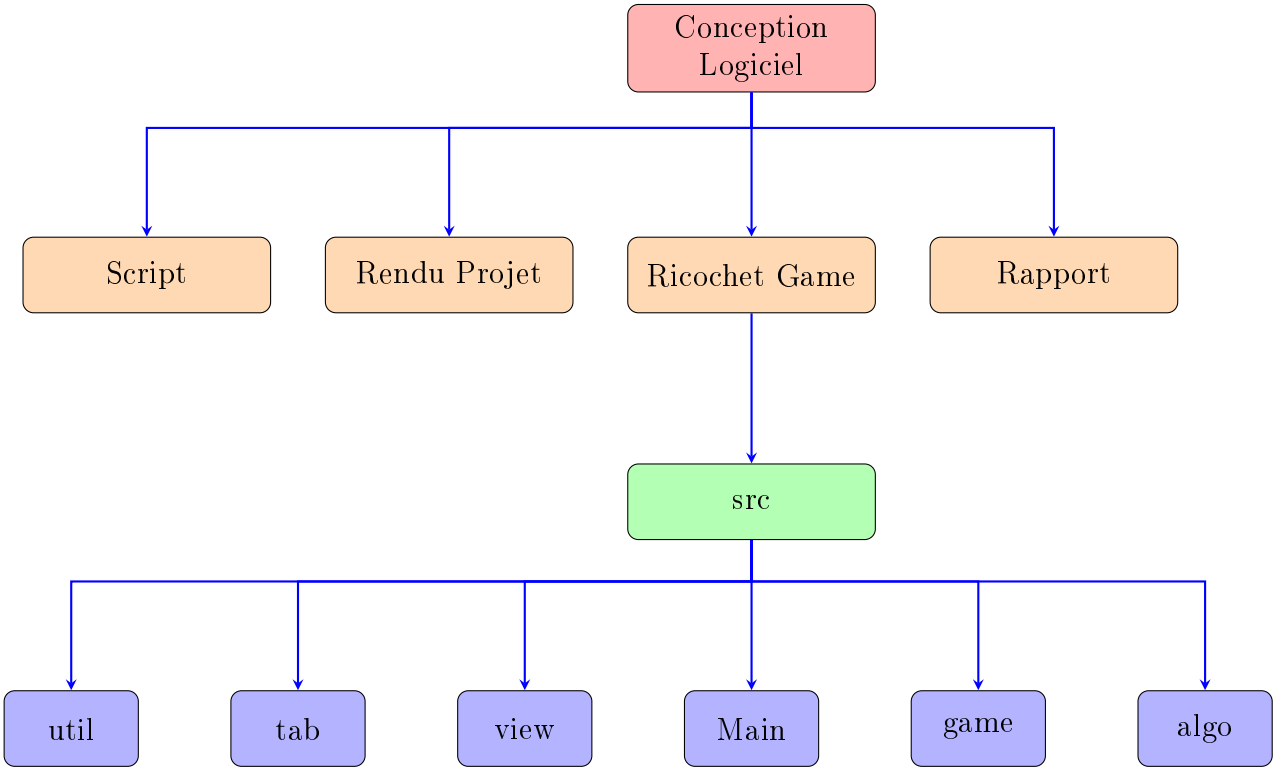
3.1.2 Gestionnaire de version

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 comme gestionnaire de fichier sans raison particulière juste parce que c'était celui sur lequel nous étions tombé en premier, qui en fin de compte s'avérait être un bon choix. 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 est correctement implémentée. 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.

4 Conception

4.1 Arborescence du projet

Au fur et à mesure de l'avancement le projet a pris pour arborescence la suivante **??** :



RicochetGame:
Représente l'espace de travail.
Rapport:
Le répertoire contenant les différents éléments du rapport.
Script:
Contient différents scripts.
RenduProjet:
Répertoire contenant le code à rendre.

4.2 Architecture du programme

4.2.1 Diaggramme des packages

Le diagramme suivant est sa forme finale prise une fois que tous les éléments ont été correctement implémentés.

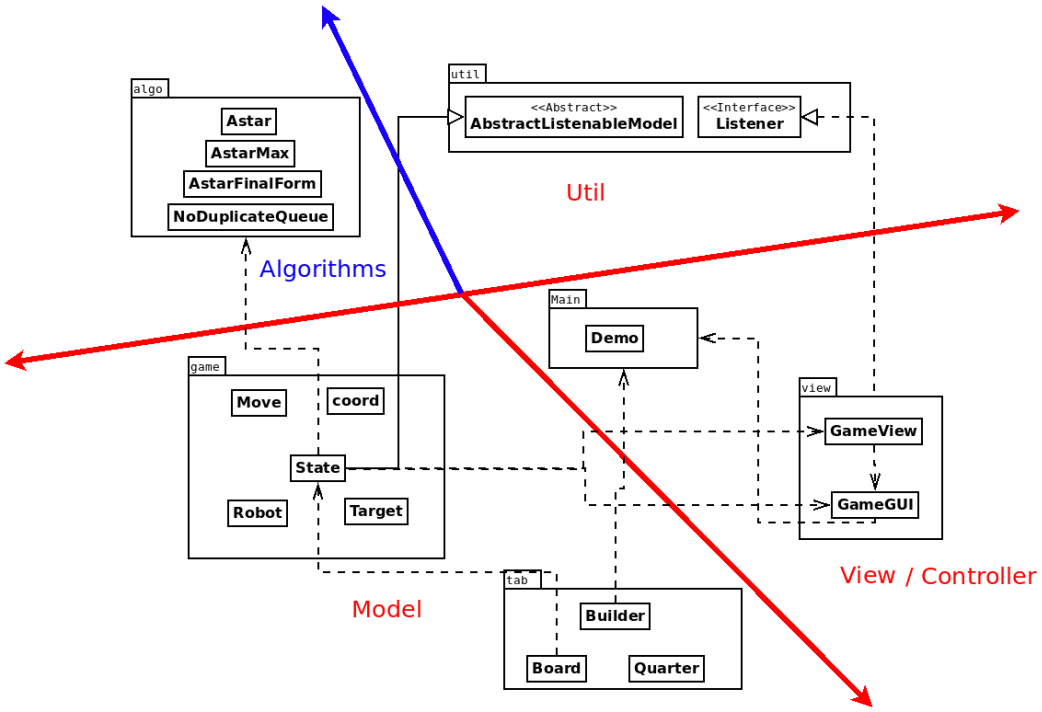


Figure 3: Diagramme UML

Comme nous pouvons le voir le code source est décomposé en 6 packages avec le pattern **MVC** en plus pour une plus facile implémentation de l'interface graphique.

- Les packages **tab** + **game** constitue le moteur du jeu et représente le **modèle**.

- Le package **view** constitue la partie **vue** + **controlleur**
- Le package **algo** contient les différents solveurs.

5 Réalisation du Moteur

La réalisation du moteur était la première étape que nous avions entamé et la dernière à être achevée car jusqu'au bout de multiples buggs ont du être corrigés et de nouvelle fonctionalités ajoutées afin que les autres parties puissent fonctionner comme prévu.

5.1 Réalisation du Board

5.1.1 UML des classes

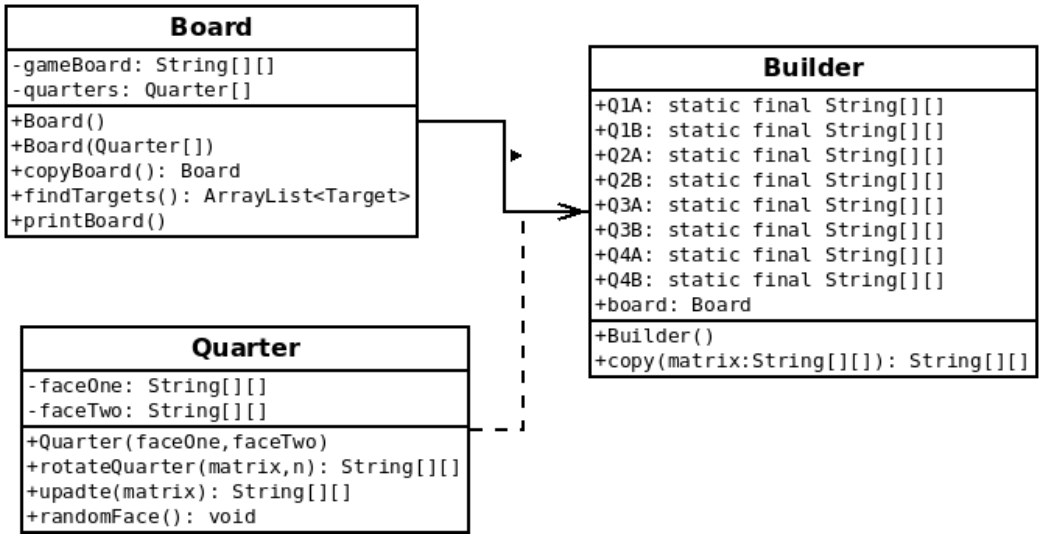


Figure 4: Diagramme UML du package tab

5.1.2 Création des quartiers

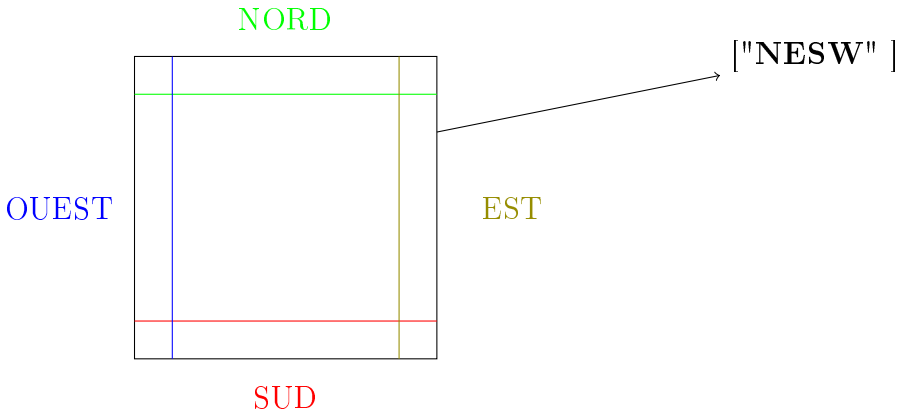
Pour concevoir le Ricochet Robot, la première question que nous nous sommes poser était: "comment devons-nous créer le plateau?". Le plateau du Ricochet Robots est un ensemble de "mini-plateaux" qui, une fois assemblés, forment ce plateau. Dans la version du jeu de société, il existe quatre morceaux de plateaux, chaque morceau ayant deux faces. Ici aussi nous avons utilisé les mêmes propriétés, donnant ainsi 96 configurations de plateforme possibles.
Pour créer le plateau, nous devions commencer par la création de ces mini-plateaux. Ces quatre mini-plateaux sont chacun représenté sous la forme d'un tableau à deux dimensions, contenant des cases et des murs.
Pour la réprésentation du plateau nous avons repris l'idée de Micheal Fogleman [?], qui pour indiquer la nature de chaque case utilise des caractères.

```
1  Q1A = {{"NW","N","N","N","NE","NW","N","N"},
2         {"W","S","X","X","X","X","SEYH","W"},
3         {"WE","NWGT","X","X","X","X","N","X"},
4         {"W","X","X","X","X","X","X","X"},
5         {"W","X","X","X","X","X","S","X"},
6         {"SW","X","X","X","X","X","NEBQ","W"},
7         {"NW","X","E","SWRC","X","X","X","S"},
8         {"W","X","X","N","X","X","E","NWX"},};
```

Comme nous pouvons le voir chaque case est désignée par une **chaîne de caractères(String)** représentant l'état de la case. Ici :

- **'N'** indique la présence d'un mur au nord de la case.
- **'S'** indique la présence d'un mur au sud de la case.
- **'E'** indique la présence d'un mur à l'est de la case.
- **'W'** indique la présence d'un mur à l'ouest de la case.
- **'X'** indique que la case vide.

- les lettres **'R'**, **'B'**, **'G'**, **'Y'** désignent réspectivement les couleurs des targets **red, blue, yellow, green**
- les lettres **'C'**, **'T'**, **'H'**, **'Q'** désignent respectivement la forme des targets **circle, triangle, hexagone, square**

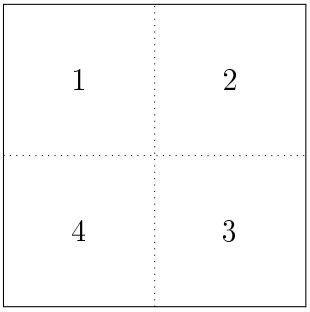


Case
ayant un mur au nord et à l'ouest

De cette manière nous avions obtenu une modélisation assez simple d'une face de quartier sous la forme d'une matrice de **string**. Ainsi les huits faces différentes ont été formées par la suite et chaque quartier a été affecté à deux de ces faces.

5.1.3 Placement des quartiers

Le plateau est composé de quatre quartiers, comme nous pouvons le voir sur la figure suivante :



Pour former à chaque fois un plateau aléatoire nous prenions au hasard l'une des faces de chaque quartier puis plaçons au hasard les quartiers dans chacun des emplacements. Sauf que se limiter à ça impliquer la génération d'un tableau incorrecte, car les murs de chaque quartiers devaient en plus être mises à jour.
La solution adoptée fut de placer le premier quartier dans l'emplacement **Nord-Ouest** sans le changer, puis d'opérer une rotation de 90° pour le suivant et le placer à l'emplacement **Nord-Est**, puis une double rotation de 90° pour le troisième quatier et le placer à l'emplacement **Sud-Est** et enfin une triple rotation pour le dernier quatier.
L'algorithme suivant **??** montre comment marche la rotation d'un quartier et le seconde **??** montre comment créer une grille aléatoire.

Algorithm 1: ALGORITHME DE ROTATION D'UNE MATRICE

```
   Input  : matrice
   Input  : entier n réprésentant le nombre de rotation
   Output: matrice tournée n fois
   /* boucle sur le nombre de fois que l'on décide de
      tourner la matrice                                    */
 1 for k = 0; k ≤ n; k++ do
   |  /* Determiner la transposé                            */
 2 |  for i = 0; i < 8; i++ do
 3 |  |  for j = 0; j < 8; j++ do
 4 |  |  |  tmp ← matrice[i][j];
 5 |  |  |  matrice[i][j] ← matrice[j][i];
 6 |  |  |  matrice[j][i] ← tmp;
 7 |  |  end for
 8 |  end for
   |  /* Renverser les éléments de chaque lignes          */
 9 |  for l = 0; l < 8; l++ do
10 |  |  low ← 0;
11 |  |  high ← 7;
12 |  |  while low < high do
13 |  |  |  tmp ← matrice[l][low];
14 |  |  |  matrice[l][low] ← matrice[l][high];
15 |  |  |  matrice[l][high] ← tmp;
16 |  |  |  low++;
17 |  |  |  high--;
18 |  |  end while
19 |  end for
20 |  update(matrice); /* fonction mettant à jour les murs  */
21 end for
22 return matrice
```

Algorithm 2: ALGORITHME DE CREATION DE LA GRILLE 16X16

Input : Tableau de taille 4 de quartiers : quartiers
Output: Grille

```
1  grille ← String[16][16];
2  list ← ∅;
3  Q1 ← quartiers[0].randomFace();
4  Q2 ← quartiers[1].randomFace();
5  Q3 ← quartiers[2].randomFace();
6  Q4 ← quartiers[3].randomFace();
7  list.add(Q1); list.add(Q2); list.add(Q3); list.add(Q4);
8  list.shuffle();
9  for i = 0; i ≤ 3; i++ do
10 |  rotationMatrice(list.get(i),i);
11 end for
12 for i = 0; i < 8; i++ do
13 |  for j = 0; j < 8; j++ do
   |  |  /* placement Nord-Ouest                          */
14 |  |  grille[i][j] ← list.get(0)[i][j];
   |  |  /* placement Nord-est                           */
15 |  |  grille[i][j + 8] ← list.get(1)[i][j];
   |  |  /* placement Sud-est                           */
16 |  |  grille[i + 8][j + 8] ← list.get(2)[i][j];
   |  |  /* placement Sud-Ouest                         */
17 |  |  grille[i][j + 8] ← list.get(3)[i][j];
18 |  end for
19 end for
20 return grille
```

5.2 Fonctionnement du moteur

Une fois la grille générée, nous nous sommes pencher sur le coeur de l'application c'est à dire le moteur du jeu dont la majorité des fonctionalités appartiennent à la classe **State** qui également joue le rôle de modèle dans l'architecture **MVC**.

5.2.1 Diaggramme UML

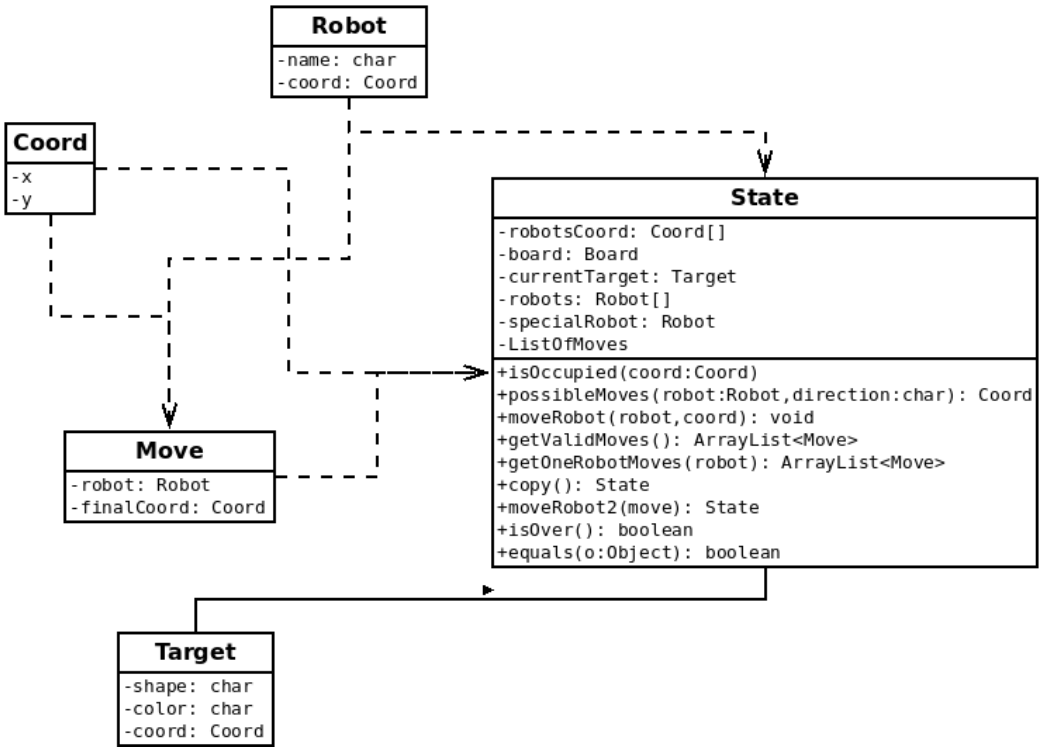


Figure 5: Diagramme UML du package game

5.2.2 Description

La classe **State** donne l'état du jeu à tout instant donné. Elle initialise le jeu en créant et en plaçant les robots à des positions aléatoires, de même elle choisit aléatoirement une target parmis les 16 présentes sur le plateau. Elle assure les différents mouvements des robots grâces aux méthodes :

- **possibleMove** prenant en argument un **robot et une direction** et retournant la **coordonnée finale** d'arrivée.
- **moveRobot** prenant un move et modifiant la position du robot.

- **moveRobot2** faisant la même opération que moveRobot mais retournant un nouvel état, fonction essentielle pour l'agorithme **A***.
- **getValidMoves** retournant l'ensemble des moves possibles pour l'ensemble des robots, également essentielle pour **A***. Elle fonctionne de la manière suivante :

Algorithm 3: ALGORITHME RETOURNANT L'ENSEMBLE DES MOVES POSSIBLES

Input : Tableau des robots
Output: List des moves possibles pour les 4 robots

```
1  listOfMoves ← ∅
2  directions ← {'N','S','E','W'};
3  foreach robot in listOfRobots do
4  |   foreach direction in directions do
5  |   |   coordFinal ← possibleMoves(robot, dir);
6  |   |   if robot.coord not equal to coordFinal then
7  |   |   |   move = newMove(robot, coordFinal);
   |   |   |    listOfMoves.add(move);
8  |   |   end if
9  |   end foreach
10 end foreach
11 return listOfMoves
```

6 ALGORITHME A*

6.1 Présentation

L'algorithme A*[?] 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.Il est souvent utilisé grâce a sa complétitude, son optimisation et son efficience. 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.
Il est également qualifié d'algorithme **Best-First-Search** car trouvant toujours la meilleure solution en premier.

Il a quand même un défaut qui est sa **compléxité en espace** car il garde en mémoire tous les noeuds/états générés.

Il existe d'autres algorithmes de recherche rapides et populaires tel que celui de recherche en profondeur **Breadth-First-Search(BFS)** [?] ou encore **Dijkstra** [?] qui peut être considéré comme le grand frère de A*.

A* fonctionne en maintenant un arbre des trajets originant de la racine(le point de départ) et explorant ces trajets un à un jusqu'à satisfaire sa condition terminale.
A chaque itération de sa boucle principale A* doit déterminer quel trajet il doit explorer. Il le fait en se basant sur le coût de ce chemin qui est une estimation du coût requis pour explorer ce chemin jusqu'au point d'arrivé.
Plus précisément A* choisit le chemin qui minimisera :

$$f(n) = g(n) + h(n) \tag{1}$$

Avec **n** étant le noeud suivant à explorer, **g(n)** le coût du trajet à partir du départ jusqu'à n(**gScore**) et **h(n)** l'heuristique qui estime le coût du chemin de n jusqu'à l'arrivée.
A* s'arrête lorque le trajet qu'il choisit d'explorer est celui du départ à l'arrivée ou lorsqu'il n'y a plus d'autres possibilités de chemins.
La fonction heuristique varie en fontion du type de problème. Si la fonction est admissible c'est à dire qu'elle ne surestime pas le coût pour atteindre

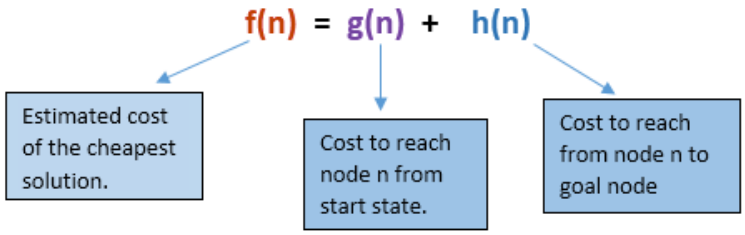
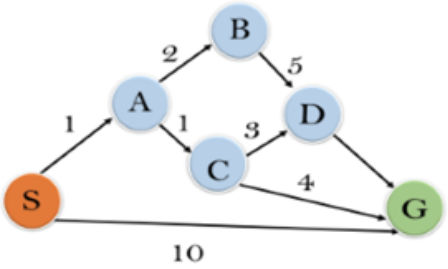


Figure 6: Formule de calcul du coût(**fScore**)

l'arrivée alors A* retournera toujours le chemin le moins coûteux.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Figure 7: Problème de parcours

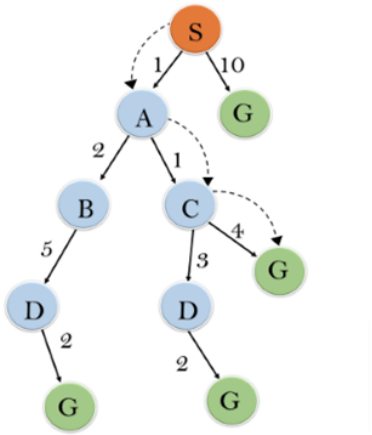


Figure 8: Solution

Pour des explications plus en détailles et plus démonstratives nous recommendons la video de **Computerphile** parlant de A* sur youtube. Elle fut d'une grande aide afin de pouvoir cerner le fonctionnement l'algorithme.

6.1.1 UML

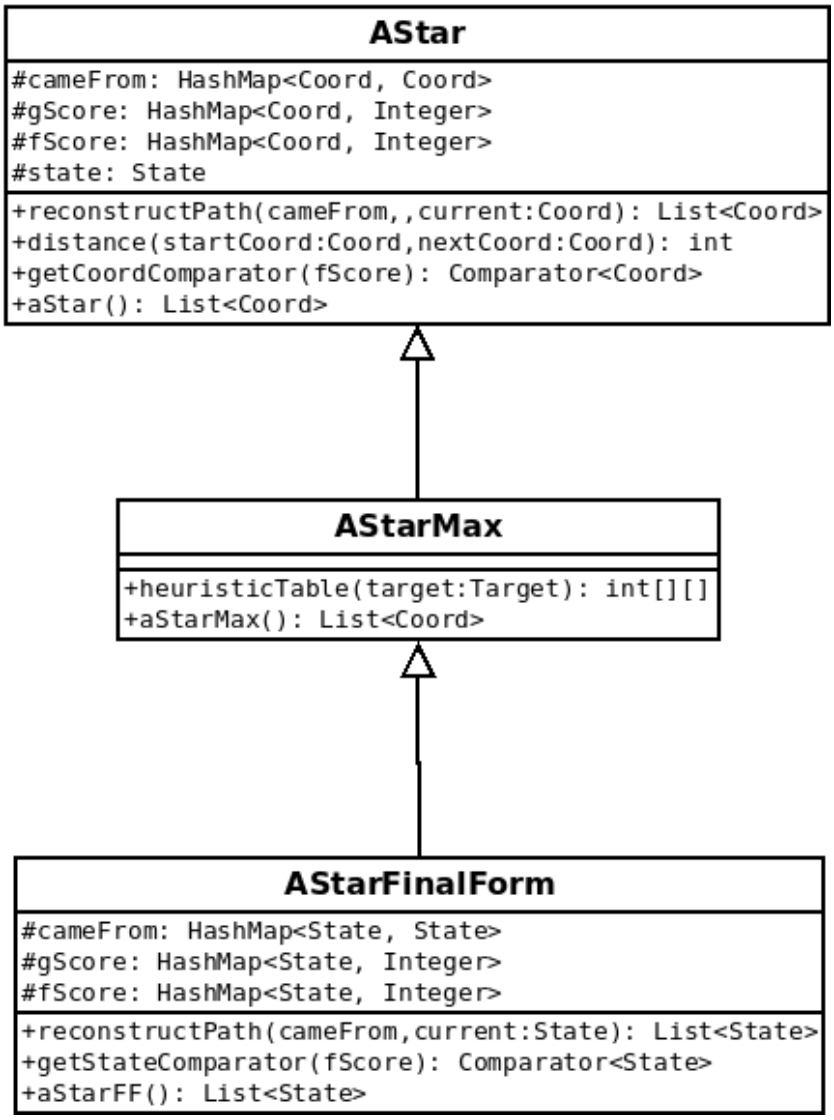


Figure 9: UML du package Algo

6.2 Implémentation par rapport à Ricochet Robot

L'implémentation de A* s'est faite en trois étapes.

- La première étape avec l'algorithme ne prenant en compte que les mouvements du robot qui avait la même couleur que la cible.(**robot actif**)
- La deuxième étape avec l'utilisation d'une heuristique plus appropriée.
- Et finalement la version finale en prenant en compte l'ensemble des robots.

6.2.1 Première Version Astar

AStar comporte la majeur partie des décisions d'optimisation et de conception.
Parlons d'abord de pourquoi nous ne faisions bouger qu'un seul robot. En effet au début à partir de toutes les documentations lues et exemples d'applications de A* étudiés, l'algorithme prenait toujours un seul objet situé en un point donné et cherchait à trouver le plus court chemin afin d'atteindre l'objectif. Sauf que dans notre problème la solution ne limite pas toujours qu'aux mouvements d'un seul robot mais plutôt des 4. En effet bouger qu'un seul robot ne garantit que une fois sur trois(grosse approximation) d'arriver à la cible. Les autres robots doivent être utilisés comme obstacles afin d'aider à arriver à la cible.
Le problème venait de là. Comment donc dire à l'algorithme de prendre en compte **les autres robots**? Solution qui ne sera apportée que par la dernière version.
Durant l'implémentation de A* plusieurs choix d'optimisation ont été pris dont certains involontairement.
Dés le début nous comptions utiliser une PriorityQueue comme choix de strucuture de données car recommender en CM et dans la plupart des articles parlant de A*. En effet la priorityQueue dispose d'un comparateur permettant d'ordonner les éléments de la queue. Ici notre comparateur était fait en sorte de mettre en tête le noeud dont le score etait le plus petit, score désignant ici $f(n) = g(n) + h(n)$ **??**.

Ainsi nous avions directement accès au meilleur noeud en temps constant. L'utilisation d'un PriorityQueue donne un gain énorme en perfomance comparée à une liste ou il aurait d'abord fallut chercher le meilleur élement dans la liste(temps linéaire), le récupérer puis le supprimer(temps linéaire). Tandis que pour la queue nous pouvons à la fois avoir accès et supprimer l'élément en temps logarithmique(voir figure ci dessous **??**).
Les valeurs suivantes ont été directement prise à partir de **Github** [**?**].

List	Add	Remove	Contains	Get
ArrayList	$\mathcal{O}$(1)	$\mathcal{O}$(n)	$\mathcal{O}$(n)	$\mathcal{O}$(1)
LinkedList	$\mathcal{O}$(1)	$\mathcal{O}$(1)	$\mathcal{O}$(n)	$\mathcal{O}$(n)
Queue	Add	Remove	Poll	Peek
PriorityQueue	$\mathcal{O}$(logn)	$\mathcal{O}$(n)	$\mathcal{O}$(logn)	$\mathcal{O}$(1)

Figure 10: List vs Queue

Le deuxième choix de strucuture de donnée fut l'utilisation d'un **HashSet** afin de garder en mémoire les noeuds déjà visité pour ne pas avoir à les revisiter, choix qui était un total hasard sans aucune réflexion au préalable, mais qui s'est avéré être à la fois excellent et un vrai casse tête(nous y reviendrons plus tard). En effet vérifier qu'un noeud appartient au Set se fait en temps constant($\mathcal{O}$(1)).

6.2.2 Deuxième version AStarMax

AstarMax hérite de **Astar** et fait la même chose. La seule différence entre les deux est l'**heuristique**. En effet après un long de temps de recherche nous nous sommes dit que peut être une meilleure heuristique réglerai tous nos soucis. La première version utilisait comme heuristique une **distance de Manhattan** plus facile et rapide à calculer que la distance euclidiènne mais qui n'est pas optimale comme estimation. En effet l'éloignement du robot par rapport à la cible n'a aucun rapport avec le nombre de coups nécessaire pour l'atteindre.
La nouvelle heuristique consistite en un tableau de même dimension que le plateau du jeu et chaque cellule de ce tableau contient le nombre de coups

necessaire pour atteindre la cible à partir de cette position en assumant les autres robots étant idéalement placés.
Cette idée nous est venu de Randycouleman [?] qui lui même s'est inspiré de Micheal Fogleman [?].

Figure 11: Table des heuristiques

La création de la table fonctionne de la manière suivante:

- Mettre **1** sur toutes les cases pouvant atteindre la cible en un mouvement.
- Mettre **2** sur toutes las cases vides et pouvant atteindre une case contenant **1** en un mouvement.
- Mettre **3** sur toutes las cases vides et pouvant atteindre une case contenant **2** en un mouvement.
- Mettre **4** sur toutes las cases vides et pouvant atteindre une case contenant **3** en un mouvement.
  Ainsi de suite.

Cette méthode a pour avantage de limiter grandement le nombre de calculs. En effet le tableau est généré une fois au début, puis à chaque fois que l'on

a besoin de connaitre l'heuristique il nous suffit de récupérer la valeur de la case correspondant aux coordonnées du robot **actif**(celui qui doit atteindre la cible).

6.2.3 Dernière version AstarFinalForm

Cette dernière version appelée **AstarFinalForm** reprend tous les éléments des deux précédentes et rajoute la prise en compte des autres robots.
Premièrement nous ne représentons un état comme étant uniquement la position de ses 4 robots. Ainsi un état est représenté par un tableau de taille 4 de coordonnées.
Ensuite l'état de départ est affecté avec un score équivalent au nombre de mouvements nécessaires pour le robot actif d'atteindre la cible. Puis nous rajoutons cette état au Set de noeuds visités et à la priorityQueue.
A partir de l'état de départ nous générons tous les états possibles correspondant à chaque mouvement de robot. Si un état n'a pas encore était visité ou est plus avantageux que les précédents, il est rajouté à la priorityQueue, et à une HashMap en même temps que l'état qui le précéde.
La boucle principale de l'algorithme continue tant qu'il ne tombe pas sur un état verifiant la condition que le robot est arrivé à la position de la cible.
Les algorithmes suivants sont directement inspiré de ceux présents sur le site de Wikipedia [**?**]

Algorithm 4: ALGORITHME RECONSTRUCTPATH : RETOURNANT LE TRAJET

Input : HashMap cameFrom<State,State>
Input : currentState
Output: ListOfStates

```
1 listOfStates ← new List < State > ()
2 listOfStates.add(currentState)
3 while currentState ∈ cameFrom.Keys do
4 |   currentState ← cameFrom.get(currentState)
5 |   listOfStates.add(currentState)
6 end while
7 return listOfStates
```

Algorithm 5: ALGORITHME ASTAR_FINAL_FORM

```
   Input  : initialState, h
   Output: listOfStates
 1 openSet ← ∅
 2 priorityQueue ← ∅
 3 gScore ← new map < State, Int > with Default Value of +∞
 4 fScore ← new map < State, Int >
 5 startingState ← initialState
 6 openSet.add(startingState)
 7 priorityQueue.add(startingState)
 8 gScore.put(startingState, 0)
 9 fScore.put(startingState, h(startingState))
10 while priorityQueue is not empty do
11 |   currentState ← priorityQueue.poll()
12 |   if currentState.isOver() then
13 |   |   return reconstructPath(cameFrom, currentState)
14 |   end if
   |   /* On explore tous les états résultant d'un mouvement
   |      de robot                                          */
15 |   foreach neighbor of currentState do
16 |   |   tentativeGscore ← gScore[currentState] + 1
   |   |   /* S'il est meilleur que tout ce que nous avions
   |   |      jusqu'a présent nous le gardons               */
17 |   |   if tentiveGscore < gScore[neighbor] then
18 |   |   |   cameFrom[neighbor] ← currentState
   |   |   |    gScore[neighbor] ← tentativeGscore
   |   |   |    fScore[neighbor] ← tentativeGscore + h(neighbor)
19 |   |   |   if neighbor not in openSet then
20 |   |   |   |   openSet.add(neighbor)
21 |   |   |   |   priorityQueue.add(neighbor)
22 |   |   |   end if
23 |   |   end if
24 |   end foreach
25 end while
   /* openSet est vide et la target jamais atteinte       */
26 print("NO SOLUTION")
27 return ∅
```

7 Problèmes rencontrés

7.1 A*

Durant l'implémentation de A* un problème semblait être insurmontable, pendant quasiment un mois nous sommes restés bloquer dessus et à faillit finir par nous traumatiser.
En effet l'algorithme semblait tourner à l'infini quel que soit la situation du plateau. Dans l'algorihme avant d'ajouter un élément à la queue nous verifions à chaque fois s'il a déjà été exploré ou pas. Pour cela nous utilisions la méthodes **contains()**. Pendant longtemps en aucun moment nous n'avions soupconné l'origine du problème, ce n'est que apres avoir fait un **println()** de la queue que l'on s'est rendu compte que la queue grandissait à l'infini et que des noeuds déjà visités étaient quand même revisités.
La première solution que l'on avait adopté fut de créer une classe **NoDuplicateQueue** qui est priorityQueue qui ne prenait pas double en allant redéfinir sa méthode **add()**.
Mais même aprés cela le problème était toujours présent. Ce ne que bien plus tard que l'on s'est mis a soupconné la méthode **contains()**.
On est allé lire la documentation et on s'est rendu compte que contains comparait les **références au lieu des valeurs**.
A partir delà nous avons décidé de redéfinir **equals** afin de régler le problème. Mais à la grande suprise le problème persistait.
Pour le contourner le problème on s'est mis à faire une boucle sur tous les éléments du Set et les comparer un à un avec l'élément que l'on voulait ajouter ce qui est lent (temps linéaire) et va l'encontre de l'utilisation d'un HashSet.
C'est à ce moment que notre chargé de TP Etienne LEHEMBRE nous a expliqué l'origine du problème.
En effet un HashSet vérifie si deux éléments sont égaux fait à la fois appel à la méthode **equals** et la méthode **hashcode()**.
Si les deux éléments ont des codes différents, ils sont considérés comme différents. Ce qui faisait que même si on avait redéfinit equals() cela ne suffisait pas.
Il nous a fallut donc également aussi override la méthode hascode(). Après

cela le problème n'était plus.
De tout le projet c'est surement la chose la plus impotante que nous avons eut à apprendre.

7.2 GUI

Durant la mise en place de l'interface graphique, nous avons voulut ajouter un boutton qui joue automatiquement tous les moves renvoyés par A* en faisant une boucle sur la liste retournée, mais nous avons vite constaté que l'interface ne se mettait à jour qu'à la fin de la boucle.
Une fois de plus le problème semblait imcomprehensible car nous avions correctement implémenté le pattern **MVC**. Un autre boutton qui exige que l'on click à chaque fois pour joueur un move mettait à jour l'interface à chaque fois mais l'autre non.
Une fois de plus est venu à la rescousse notre chargé de tp qui aprés investigation nous a montré que l'interface était gelée et ne reprenait la main que une fois la boucle terminée.
Pour résoudre le problème il a fallut faire tourner l'interface et la méthode faisant une boucle sur les moves retournés sur différents **threads**. Qui fut une première car jusque là nous n'avions jamais considéré cette éventualité.
La liste de problèmes rencontrés durant ce projet étant extremment longue nous nous limiterons à ces deux qui furent les plus problématiques.

8 Conclusion

8.1 Avis Général

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.
Ce projet nous aura permis de beaucoup apprendre et de nous améliorer.

8.1.1 Eléments à améliorer

Bien que tout ce qui a été demandé dans le sujet a été remplit, pas mal de choses restent quand même à améliorer mais faute de temps et du nombre de projets à finir en ce fin semestre tout ce que l'on souhaité n'a pas pu être accomplit. Par exemple:

- Amélioreration de l'interface graphique, car l'interface finale est trés basique.
- Rajouter d'autres fonctionnalités comme pourvoir jouer par soi même.
- Faire des tests comparant différents type d'algorithmes.