RAPPORT CONCEPTION LOGICIEL

Robot-Ricochet

22008897 : KERMEZIAN AXEL
22009099 : IHANNOUBA NIHAL
22011592 : KRIMI IBRAHIM

20 avril 2021

Table des matières

1 Introduction

Le module de conception logicielle permet de mettre en pratique et de perfectionner les connaissances acquises en programmation objets lors du premier semestre. Chaque groupe composé au maximum de quatre personnes , doit réaliser une application complète. Différents sujets ont été proposés notre choix est le suivant : *L*e jeu du Ricochet-Robot.

Règles du jeu :

« Le Ricochet-robot est un jeu de société se jouant sur un plateau,possédant quatre robots de couleur différentes, ainsi qu'une "tuile" qui est tiré aléatoirement parmi les couleurs des quatre robots. Le but est alors d'amener le robot de la couleur de la tuile sur la case objectif. Les joueurs jouent simultanément, chacun réfléchissant sur le moyen d'amener le robot en utilisant les règles de déplacement. Les déplacements sont uniquement horizontal ou vertical.Lorsque le robot rencontre un obstacle celui-ci met fin à son avancement.

Lorsque l'un d'entre eux pense avoir trouvé une solution, il annonce en combien de mouvements il compte réaliser l'objectif puis il retourne le sablier. Les autres joueurs ont jusqu'à la fin du sablier pour proposer de meilleures solutions, utilisant moins de mouvements.

Après l'écoulement du sablier, le joueur qui a la solution comptant le moins de mouvement montre sa solution et remporte la tuile. S'il échoue dans sa démonstration, le joueur qui proposait le nombre de mouvements immédiatement supérieur montre sa solution, etc. jusqu'à ce qu'une solution soit valide.»

Le choix du sujet n'est pas du au hasard : le développement d'une intelligence artificielle est pour nous la principal cause de notre choix. L'intelligence artificielle en question est A* ,qui jusqu'à présent nous est inconnus. Le principe de cette intelligence est de trouver le chemin le plus cours d'un point de départ jusqu'à son point d'arrivé. Cela poursuit les différentes implémentation d'intelligence artificielle étudié et nous permet d'approfondir nos connaissance une nouvelle fois sur le sujet.

Le projet comporte trois phases de développement importantes :

* Premièrement : Les fonctions propre implémentant les règles du jeu.
* Deuxièmement : Le tirage d'un plateau totalement aléatoire pour chaque nouvelle partie exécutée.
* Troisièmement : Le développement de l'intelligence artificiel A*

2 Organisation

2.1 Répartition des tâches

Dans un premier temps le groupe c'est concerté pour définir l'implémentation du projet ainsi que la traduction de ses règles en pseudo code.
L'implémentation de la recherche de chemin A* a été négligé lors de cette étape et reporter lorsque le jeu seras implémenté correctement.

La concertation étant terminé , le groupe a tout d'abord opté pour la génération d'une grille de jeu totalement aléatoire. *(voir page 7)*.
Ainsi que l'intégration des règles du jeu dans une classe spécifique.
Étant composé de trois membres cela a impliqué une répartition des tâches spécifique.
Krimi Ibrahim et Ihannouba Nihal ont implémenter les différentes classes visant au bon déroulement du tirage aléatoire d'un plateau et de sa composition.
Kermezian Axel était chargé d'implémenter les règles du jeu celon le cahier des charges.
Une fois ces tâches terminer impliquant un jeu fonctionnel, le groupe c'est de nouveau réunis afin d'implémenter A*.

L'implémentation de A* a été effectue en cohésion par la totalité du groupe.

3 Architecture

3.1 Répartition des classes

La répartition des classes a été étudiée minutieusement.
Cinq Packages au total ont permis de séparer les différentes classes selon leur utilisation.

1. **Le package games** : contient les classes propre au développement du ricochet-Robot.
   - Jeu : implémente les fonctions permettant le respect des règles du jeu ;
   - Plateau : Définis la grille de jeu ;
   - Robots : Définis les quatre robots ;
   - Objectif : Définis l'objectif à atteindre.

2. **Le package joueur** : contient les classes implémentant un joueur.
   - Joueur : Implémente la saisie d'un joueur clavier ;
   - Astar : Implémente une recherche de chemin par l'IA ;
   - Noeud : Définis un Noeud(positionnement Robots Plateau).

3. **Le package randomfonction** : contient la classe Aleatoire
   - Aleatoire : Implémente les fonctions permettant le tirage aléatoire d'un plateau.

4. **Le package plays** : contient la classe Orchestrator
   - Orchestrator : Définis le déroulement d'une partie ainsi que l'affichage de son menu.

5. **Le package main** : Classe Main

3.2 Diagramme

FIGURE 1 – Diagramme Package games

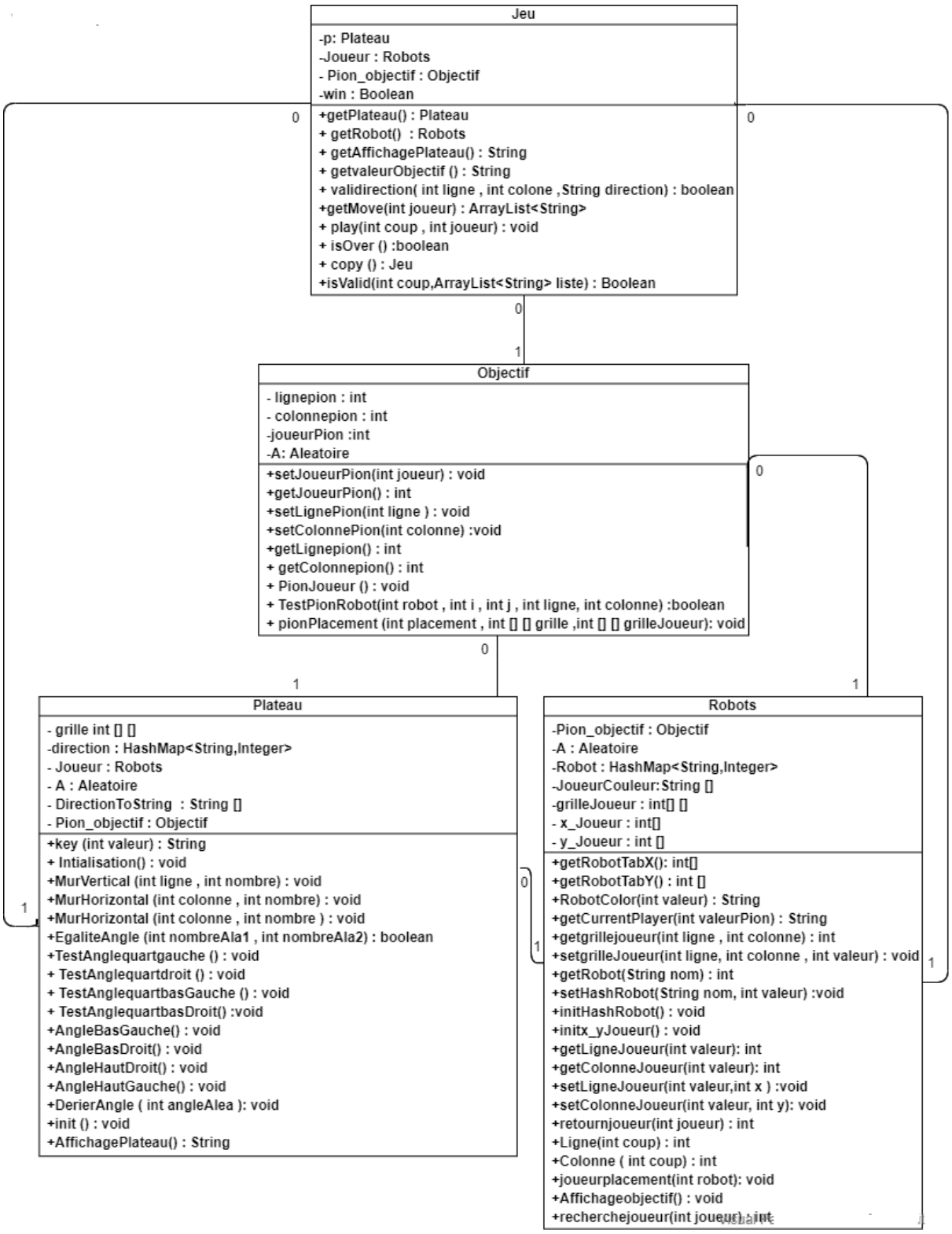
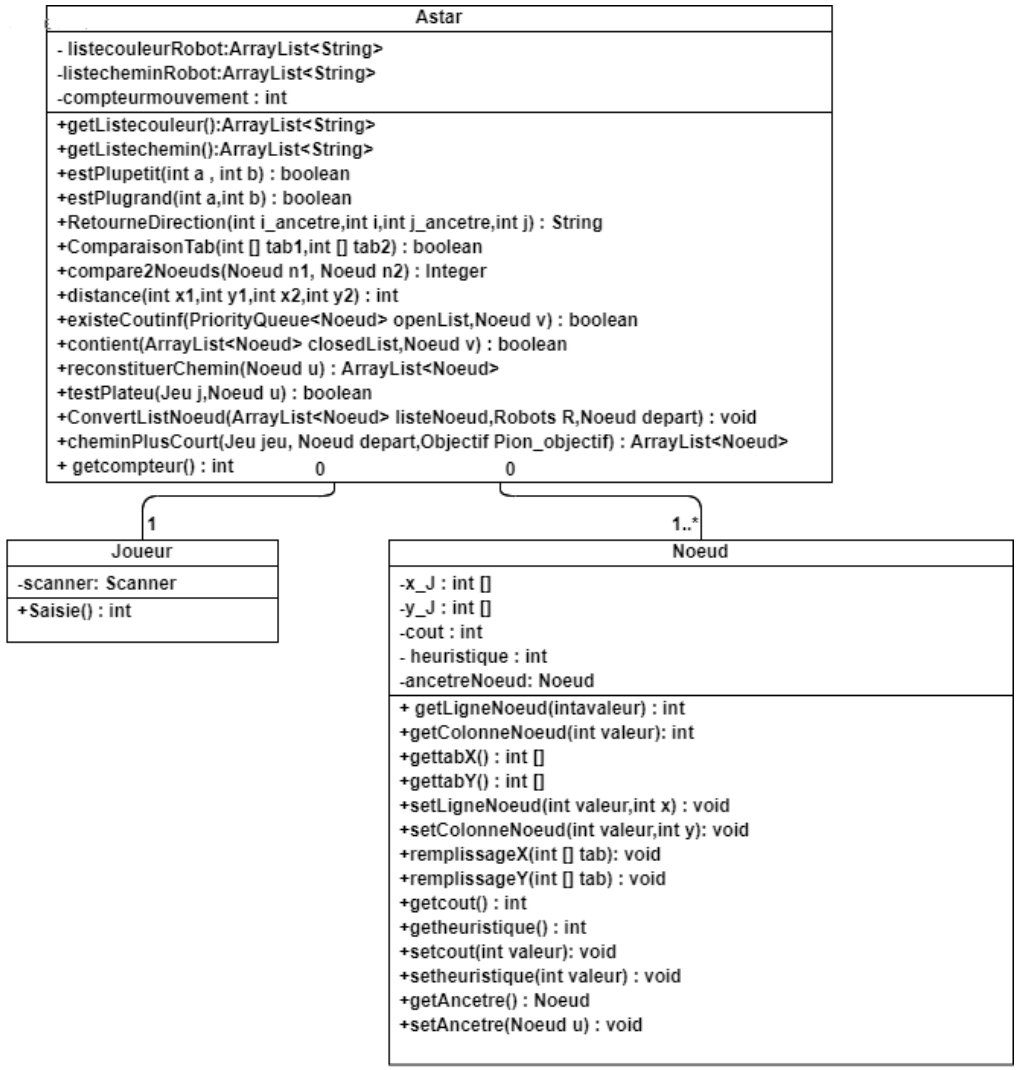


FIGURE 2 – Diagramme classe joueur



1

FIGURE 3 – Diagramme classe Orchestrator

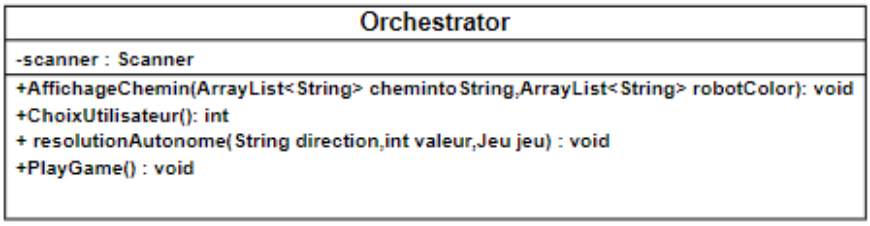


FIGURE 4 – Diagramme classe Aleatoire

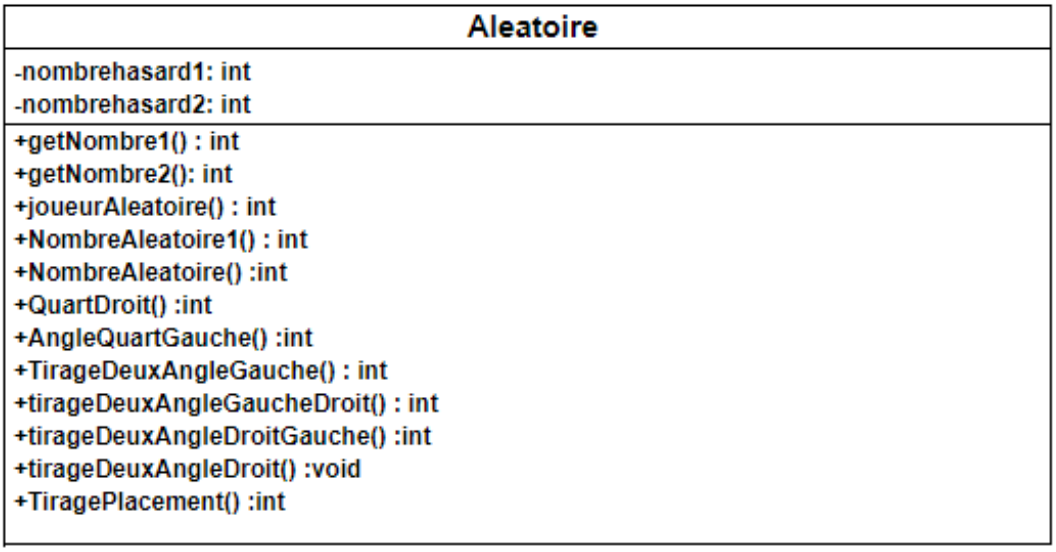
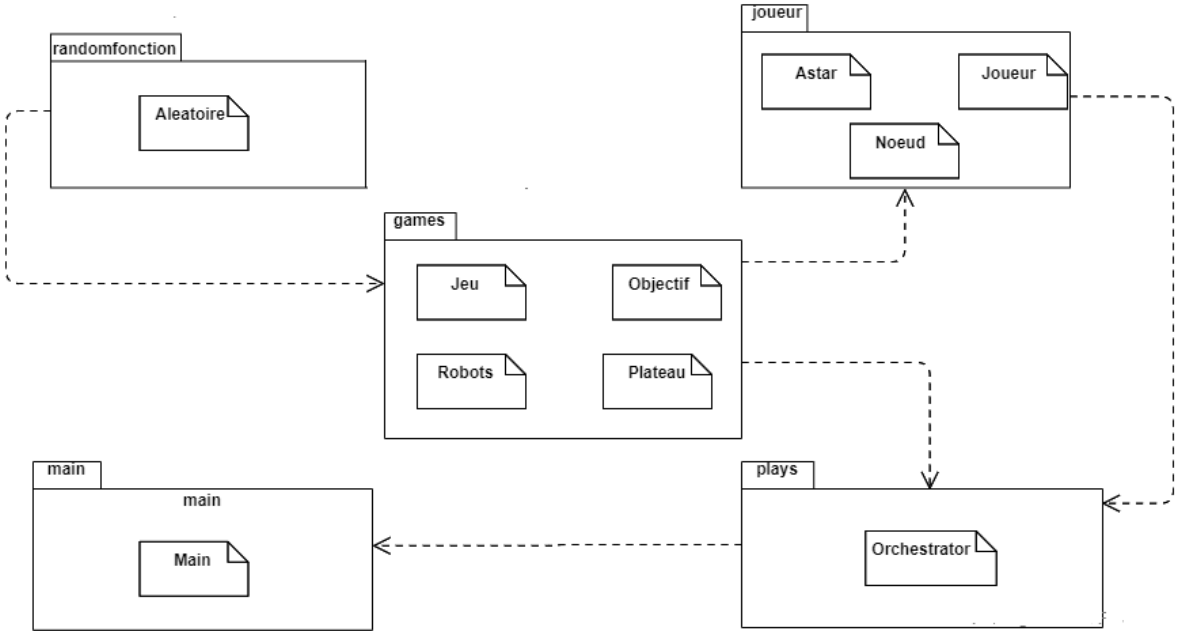


FIGURE 5 – Diagramme des Packages



4 Fonctionnalité

4.1 Tirage Aléatoire du plateau

Le choix d'un tirage totalement aléatoire des éléments du plateau a été décidé par l'ensemble des membres du groupe.
Chaque plateau subis un tirage totalement aléatoire de toutes les positions d'élément inclus dans celui-ci.La possibilité de tirer un plateau identique au précédent est quasiment nul ou relève de l'impossible.Le détails de ce tirage est détaillé plus bas.

Détail de la composition du plateau

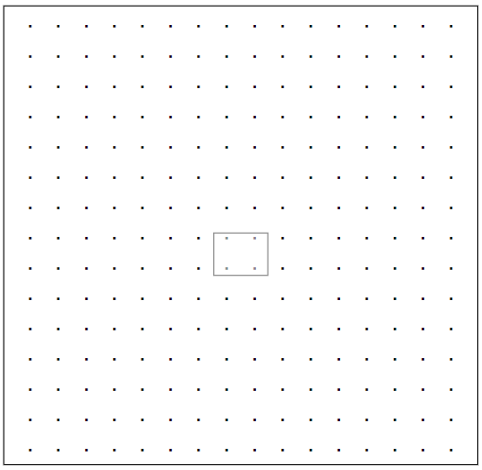
Le plateau de jeu est composé de deux tableaux d'entier : grille et joueurPlacement de tailles 16x16.

Le tableau 'grille', contient les éléments ayant pris la valeur des contraintes de directions : angles , murs , centre et case vide. Le second tableau joueurPlacement , contient l'objectif à atteindre ainsi que les robots.
La superposition de ces deux tableaux permet de modifier uniquement les valeurs du tableau contenant les robots.Cela réduit le nombre d'opération car pour le déplacement d'un robot effectué la case de celui-ci est mis à zéros tandis que la case actuellement jouer prends la valeur du robot.Cela implique uniquement deux opérations.
Une fois initialisé la grille des directions ne subis aucune modification.

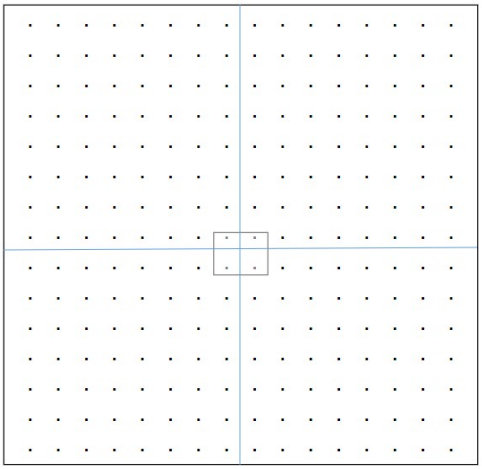
Description des éléments du plateau

Élément	Quantité	Quantité-Par-Quart	Valeur possible
Mur	8	2	Haut, Bas, Droit, Gauche
Angles	16	4	HautGauche, BasGauche, HautDroit, BasDroit
Dernier-Angle	1	Aléatoire	HautGauche, BasGauche, HautDroit, BasDroit
Robots	4	Aléatoire	Bleu, Rouge, Jaune, Vert
Objectif	1	Aléatoire	Bleu, Rouge, Jaune, Vert

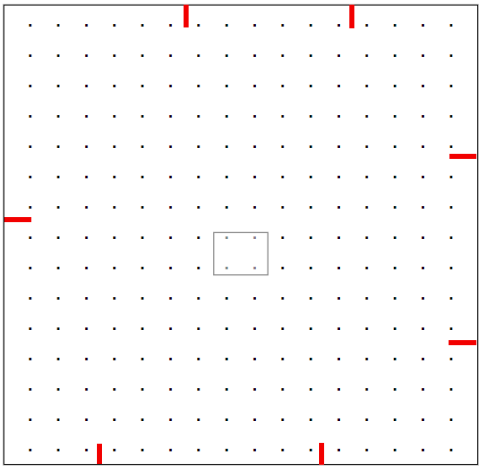
4.1.1 Étape tirage aléatoire d'un plateau



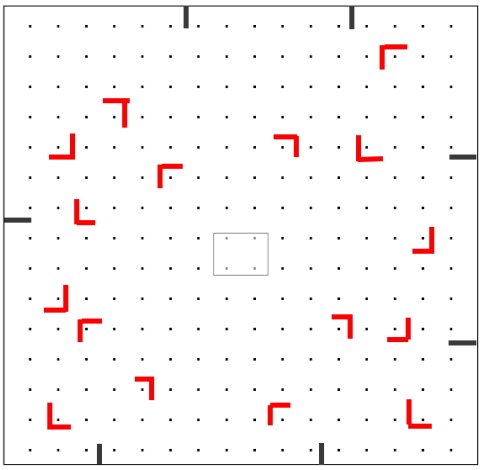
1. Au commencement, le plateau est vide. Seulement le centre est initialisé ainsi que les quatre coins du plateau avec une valeur propre à leur angle.



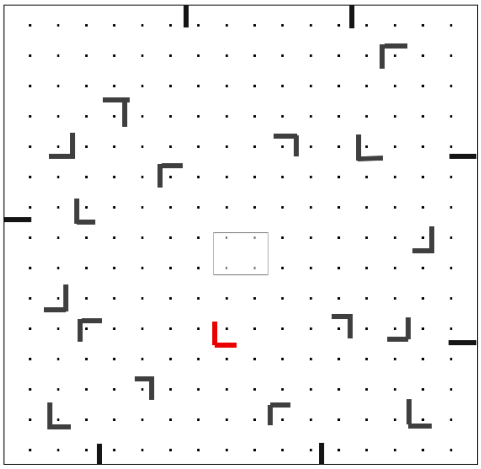
2. Le plateau est "divisé" en quatre quart. Chaque quart est composé d'un mur vertical,un mur horizontal et de quatre angles.



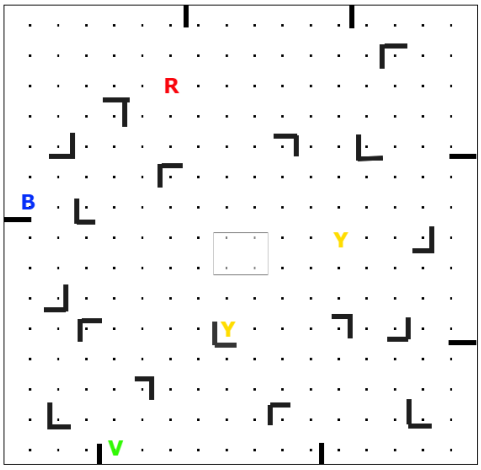
3. Les murs horizontaux et verticaux de chaque quart sont tiré selon un chiffre aléatoire correspondant au indice inclus dans leur quart.
Les indices des quatre coin initialisé plus haut sont exclus.Pour les murs horizontaux l'indice minimum et maximum de la ligne du quart forme l'intervalle du tirage.
Les murs verticaux fonctionne de façon similaire en choisissant la colonne à la place de la ligne pour la formation de l'intervalle.
Le nombre aléatoire est également tester tel que celui-ci exclus les quatre coins du plateau et les cases n'étant pas vide.



4. Le placement des angles de chaque quart est tiré aléatoirement en excluant le tirage du 0 et du 15 pour les lignes et colonnes. L'angle est ensuite placé sur une case du plateau tel que les cases adjacentes Haut,Bas,gauche,droites sont vides. Cela permet d'éviter le chevauchement d'angles.



5. Le placement du dernier angle est placé aléatoirement dans un des quatre quart.La valeur ,le quart et la position de ce dernier dans le quart sont tiré aléatoirement



6. Le placement des robots et de l'objectif.Une couleur correspondant à l'un des robots est tirée aléatoirement. Cette couleur seras affecté à l'objectif. Le placement de cette objectif est obligatoirement sur une case du plateau contenant un angle (angles coin plateau exclus). Aléatoirement , parmi les 17 angles une position est tiré aléatoirement. Les joueurs sont alors positionnées de façon aléatoire sur une case du plateau hormis les quatre cases centrales.

4.1.2 Nombre de compositions différentes possible d'un plateau

Le calcul du nombre de composition différente possible prends en compte le nombre d'éléments aléatoires composant le plateau (voir tableau page 5) :

Mur : Soit huit murs au total , chaque quart de plateau possède deux murs pouvant être positionnées chacun à sept positions différentes. Ce qui nous donne un total de $(7 * 7)^4$.

Angles : Seize angles doivent être placés , soit quatre angles différents par quart qui peuvent être positionnées dans 48 positions. En ajoutant les restrictions ainsi que le positionnement au fur et à mesure de chaque angle par quart cela nous retourne le résultats suivant. $((48 * 4) * (43 * 3) * (38 * 2) * (33))^4$

Dernier-Angle : L'angle final est tiré aléatoirement parmi les quatre valeur d'un angle ainsi que le quart attribué. Les angles précédents étant actuellement positionné cela réduit les places possibles à 48-16 soit 32. Soit (32*4)*4.

Robots : Le nombre de robots est de quatre et doivent être placés totalement aléatoirement sur les 256 case - 4 case du centre. Soit 252*251*250*249.

Objectifs : L'objectif est positionné sur l'un des 17 angles.

Élément	**Probabilité total**
Mur	$(49)^4$
Angles	$(62118144)^4$
Dernier-Angle	512
Robots	3 937 437 000
Objectif	17

Total multiplication	$2,9 * 10^{51}$

Le génération d'un plateau aléatoire est assez complexe pour ne jamais récupérer un plateau identique aux précédents.

4.2 Algorithme A*

4.2.1 Introduction

Le but premier du développement d'un jeu tel que le ricochet-robot est bien évidement l'implémentation d'une IA permettant la recherche du meilleur chemin.

L'algorithme A* est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Il utilise une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin y passant, et visite ensuite les nœuds par ordre de cette évaluation heuristique. C'est un algorithme simple, ne nécessitant pas de pré-traitement, et ne consommant que peu de mémoire.

L'IA choisi A* est donc la plus optimale à résoudre notre problème.Cependant le problème reste complexe, ainsi A* peut avoir un temps de calcul très long selon le plateau utilisé.Nous avons donc ajouté quelques modifications à celui-ci et effectuer des expérimentations au regard du temps émis(voir page 16).

Le pseudo code étant disponible sur la page de Wikipédia , cela nous a permis de l'étudier et de s'en inspirer.Lors de son étude nous avons cernés les différents aspects important au bon fonctionnement de la recherche chemin. Notamment la définition de la classe Noeud et de ces attributs mais également et ceux plus que tout : la fonction rechercheChemin. La composition d'un Noeud a été notre premier objectif.

Noeud wikipédia

```
Structure nœud = {
      x, y: Nombre
      cout, heuristique: Nombre
   }
```

Noeud Redéfinition

```
public class Noeud {

    protected int [] x_J =new int [4];
    protected int [] y_J = new int [4];
    private int cout;
    private int heuristique;
    private Noeud ancetreNoeud;
}
```

Dans un premier temps nous avons redéfinis la classe Noeud. Le pseudo code munis des attributs x,y étant les coordonnée d'un unique robot : le robot devant atteindre l'objectif.
Selon la disposition des éléments du plateau , il est souvent impossible d'atteindre l'objectif en déplacent uniquement ce robot.

Nous avons donc remplacés cela par des tableaux de coordonnées : x-J pour les lignes et y-J pour les colonnes. Le principe est identique à la classe Robot : les cordonnées définissent l'emplacement des robots selon le plateau actuelle.
L'ajout également d'un attribut extrêmement important : ancetreNoeud.Celui-ci est essentiel dans la recherche du chemin car il permet d'implémenter la fonction reconstituerChemin(voir page 14).

Les attributs cout et heuristique sont quand à eux inchangés. Le cout d'un Nœud est le résultat de l'addition entre le cout de son ancêtre et 1.
L'heuristique, prends en compte le cout du Nœud additionné à la distance de Manhattan entre les coordonnées du robot principale dans ce Nœud (ligne,colonne) et ceux de l'objectif.

4.2.2 Recherche du meilleur chemin

La fonction de comparaison entre deux Noeud n'a subis aucune modification. Nous utiliserons cette fonction pour permettre d'ordonnée une liste de Noeud avec l'élément ayant la plus petite heuristique en tant que premier élément de liste.

Algorithm 1: COMPARE2NOEUDS Effectue une comparaison entre deux Noeud

```
Input: Noeud n1,Noeud n2
Output: Entier
if n1.heuristique < n2.heuristique then
    retourner 1
if si n1.heuristique == n2.heuristique then
    retourner 0
else
    retourner -1
```

Algorithm 2: RECHERCHE CHEMIN WIKIPÉDIA

```
Input: g :Graphe, But : Nœud, depart : Nœud
Output: Void
closedList = File()
openList = FilePrioritaire(comparateur=compare2Noeuds)
openList.ajouter(depart)
while openList n'est pas vide do
    u = openList.defiler()
    if u.x == But.x et u.y == But.y then
        reconstituerChemin(u)
        terminer le programme
    for chaque voisin v de u dans g do
        if non(v existe dans closedList ou v existe dans openList avec cout inf) then
            v.cout = u.cout +1
            v.heuristique = v.cout + distance([v.x, v.y],[But.x, But.y])
            openList.ajouter(v)
    closedList.ajouter(u)
terminer le programme (avec erreur)
```

Algorithm 3: RECHERCHE CHEMIN Final

```
Input: jeu :Jeu, Pion-objetif : Objetif, depart : Nœud
Output: ArrayList de Noeud
LigneObjectif = Pion-objectif.getLignepion()
ColonneObjectif = Pion-objectif.getColonnepion()
valeurjoueurCourrant = Pion-objectif.getJoueurPion()
Depart.setAncestor(null)
closedList : ArrayList<Noeud>()
openList : FilePrioritaire(comparateur=compare2Noeuds)
listeJeu : ArrayList<Jeu>
openList.ajouter(depart)
listeJeu.ajouter(jeu)
while openList n'est pas vide do
    u = openList.defiler()
    if u.getLigne(valeurjoueurCourant) == LigneObjectif et u.getColonne(valeurjoueurCourrant) ==
     ColonneObjectif then
        retourner reconstituerChemin(u)
    for chaque jeu j dans listeJeu do
        if coordonnée Robot jeu j corresponds au coordonée Noeud u then
            jeuCopy = j.copy()
            listejeu.remove(j)
            break
    for chaque valeurdujoueur de 11 jusqu'a 14 do
        liste : Liste de string
        liste <- jeuCopy.getmove(valeurjoueur-10)
        for chaque move i dans liste do
            jeuCopy2 : Jeu
            JeuCopy2 = jeuCopy.copy()
            jeuCopy2.play(i,valeurjoueur-10)
            compteurmouvement = compteurmouvement+1
            v=Noeud(jeuCopy2.getRobotTabX(),jeuCopy2.getRobotTabY(),u.getcout()+1,0))
            v.setheuristique(v.getcout()+distance(v.getLigneNoeud(valeurjoueurCourrant),
             v.getColonneNoeud(valeurjoueurCourrant),LigneObjectif,ColonneObject
            v.setAncetre(u)
            if non(v existe dans closedList ou v existe dans openList avec cout inf) then
                listeJeu.add(jeuCopy2)
                openList.ajouter(v)
    closedList.ajouter(u)
retourner null
```

La méthode rechercheChemin a été adaptée à notre implémentation du ricochet-Robot.Pour cela nous avons ajouté trois éléments important : L'ancêtre d'un Noeud (mentionné ci-dessus),une possibilité de copier l'état d'un jeu et pour finir une liste de jeu contenant les copies effectuées.

Copie du jeu : Permet de copier l'état d'un jeu pour effectuer des déplacements de Robots sans que cela impact la disposition réelle du plateau initiale.

Liste de jeu : Permet d'ajouter des éléments de type jeu. Les copies de jeu respectant certaine conditions seront donc ajoutés à la liste de jeu(explication ci-dessous).

Déroulement recherchechemin :

Lors du parcours de la listeJeu , le jeu correspondant au Nœud "u" est copié dans un nouveau jeu.Ce jeu est ensuite supprimé de la listeJeu.

Chaque Robot va alors exécuter ses déplacements valides dans une nouvelle copie(pour ne pas modifier la copie précédente car les déplacements sont valide uniquement dans la précédente copie).Ce qui engendras la création de nouveaux Nœuds "v".Si celui-ci n'est ni dans la closedList, ni dans l'OpenList avec un coup inférieur alors il est ajouté à l'openList.Le nouvel état de jeu étant créé via se déplacement est également ajouté à la listeJeu.

Nous ajoutons donc uniquement les jeux dont les Nœuds "v" sont ajoutés à l'openList. Si le Nœud n'est pas ajouté dans l'openList cela n'apporte aucun intérêt d'effectuer un ajout de la copie du jeu. Du fait que celui-ci ne correspondras au coordonnée d'aucun Nœud il ne seras jamais extrait.

4.2.3 ReconstituerChemin

La fonction reconstituerChemin est implémentée très simplement en recherchant l'ancêtre d'un Nœud entré et ce jusqu'au Noeud de départ.Le Noeud de départ ayant un ancêtre "null" permet de remonter jusqu'à celui a l'aide d'une boucle while.

FIGURE 6 – Algorigramme reconstituerchemin

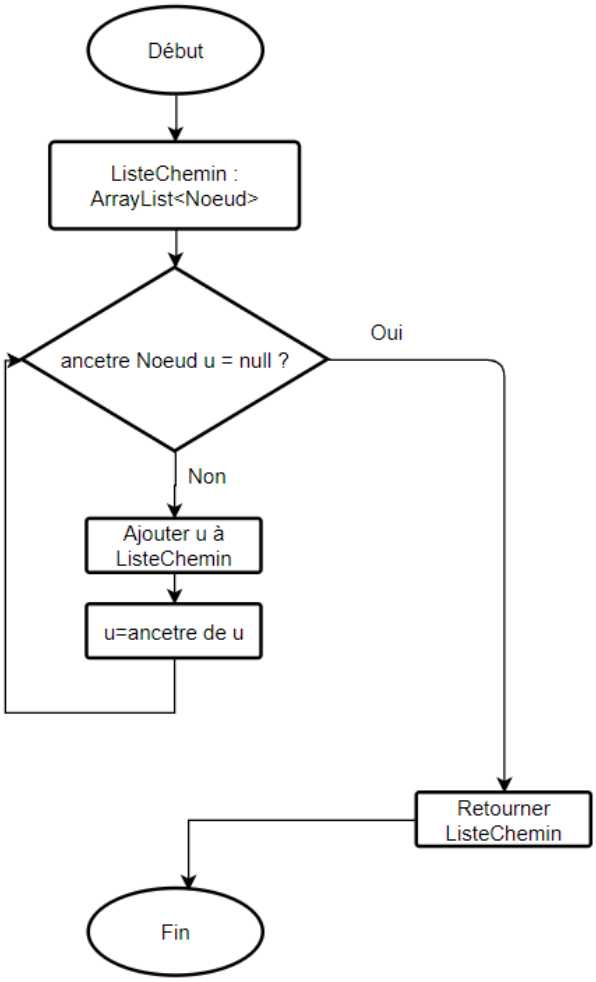
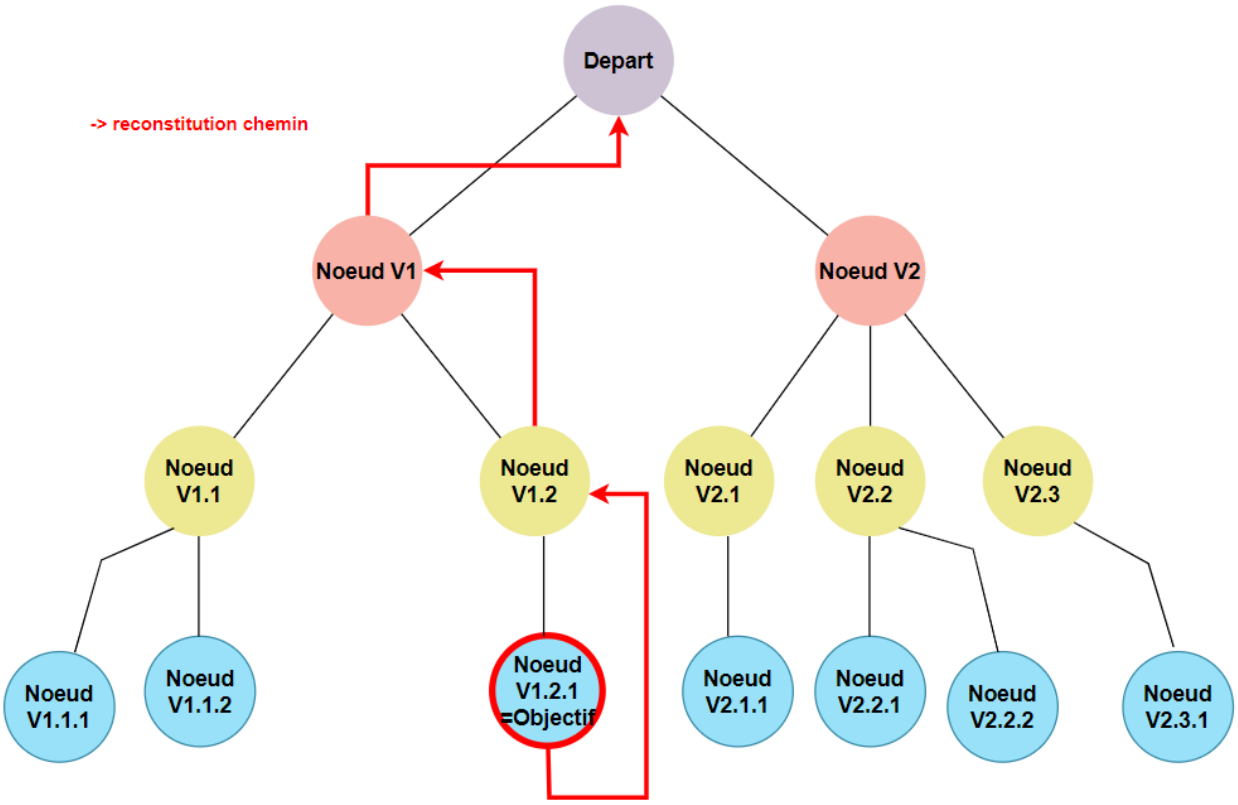


Schéma de la méthode reconstituerchemin

Le schéma ci-dessous détail de façon visuel le processus engendré pour une reconstitution d'un chemin.
Ceci est un exemple représentant une situation en miniature.Le nombre de coup total exécute serait le nombre de noeud (départ exclus) tandis que le nombre de coup à jouer pour trouver l'objectif serait de trois. Lorsque la méthode rechercheChemin est appelée celle-ci débute avec le noeud de départ (Noeud ayant les coordonnées de départ du jeu actuelle).
Les coups des différents robots sont ensuite joués dans une copie du jeu. Ces coups engendrons la créations de nouveau Noeuds qui auront pour ancêtre le Noeud de départ. Ce processus se poursuis jusqu'à ce que le Robots principale ait des coordonnées similaire à l'objectif.

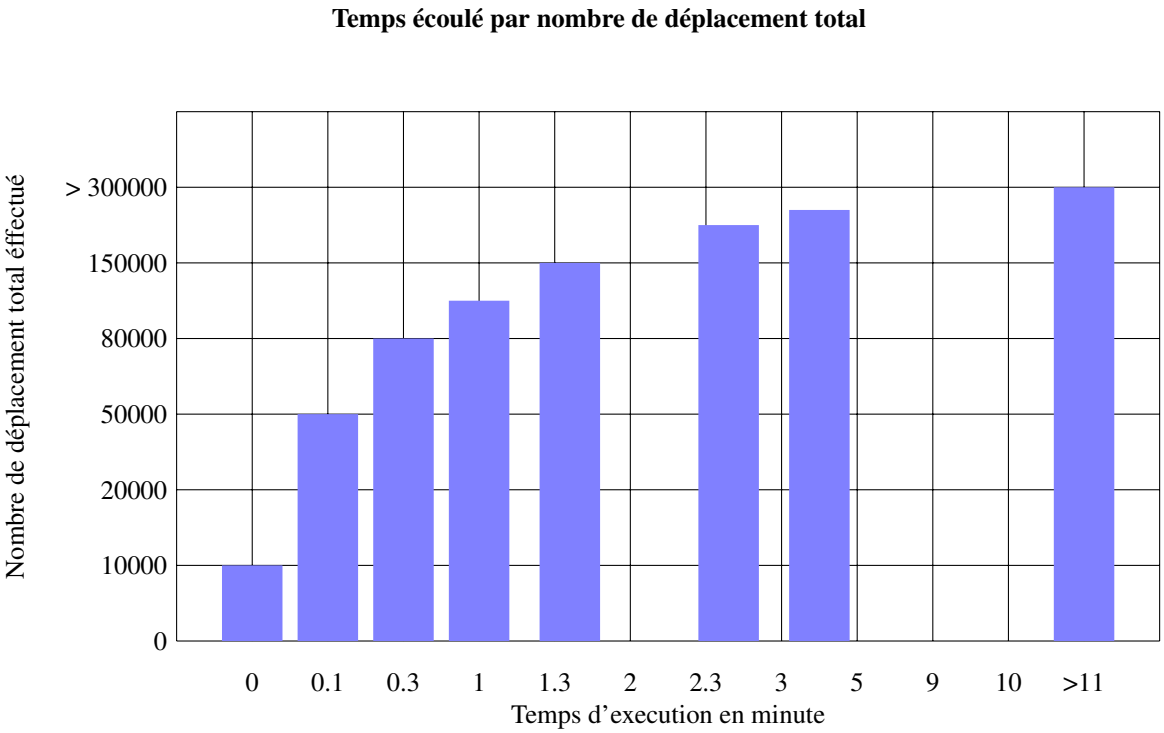


V "numéro" : représente le numéro du nœud.
Flèches rouges : représente la reconstitution du chemin
Nœud V1.3... Bleu : Représente le nœud ayant les coordonnées égale à l'objectif

5 Expérimentation

Lorsque la conception du ricochet-robot fut terminé nous avons décidés d'expérimenter le temps écoulé par rapport au nombre de déplacement total effectué pour la recherche du meilleur chemin. Pour cela nous avons ajouté un compteur : "compteurmouvement" au attributs de la classe Astar. Celui ci est incrémenté lors de la recherche du meilleur chemin pour chaque coup joué dans la copie (cité la ligne du code Astar).

La fonction cheminPlusCourt retourne les déplacements à effectuer pour atteindre l'objectif. Le nombre de ces déplacements à effectuer se situe dans un intervalle de 2 à 16. Cela est dû à la disposition des éléments formant le plateau.



6 Manuel d'utilisation

Architecture du dossier principale :

Build : permet de contenir les fichier .class.
src : contient les packages ainsi que les différentes classes.
rapport : contient le rapport sous format pdf.
compilation.sh : Script permettant l'execution du programme.
sources.txt : contient les sources.

Compilation du projet :

Ouvrir un terminal dans le dossier principale contenant src ,build et compilation.sh ;
Lancer le script de compilation avec la commande ./compilation.sh

L'exécution va générer un plateau aléatoire , qui seras affiché sur la console en mentionnant l'objectif a atteindre plus bas.

Affichage plateau :
Robots = R,B,V,Y.
Mur = - ou | ou -
Angles = -| , |- etc..
Case vide = .

Le choix d'un menu est alors affiché :
Choissir un choix de jeu :
Choix 1 : Résolution automatique = '1'
Choix 2 : Résolution saisie clavier = '2'
Choix 3 : Affichage du chemin le plus court = '3'
Choix 4 : Quitter le jeu = '4'

Il faut alors saisir un entier entre 1 et 4. L'entier 1 permet une résolution autonome du plateau affiché en affichant également les directions empruntées ainsi que le plateau final.
L'entier 2 permet à l'utilisateur de jouer au clavier.
L'entier 3 permet uniquement d'afficher le chemin pour le plateau actuelle.
L'entier 4 permet de quitter le programme.

Le terminal vous demanderas alors de saisir vos Différents choix :

7 Conclusion

Ce projet nous a permis d'utiliser la connaissance apportée par l'université sur le langage Java pour implémenter un jeu et ses règles en autonomie. L'étude et l'écriture d'une intelligence artificielle nous a permis d'aborder une nouvelle fois ce vaste domaine très intéressant et d'expérimenter différentes situations de jeu.

Dans l'ensemble le cahier des charges a été respecté :
Le moteur du jeu , le tirage d'un plateau totalement aléatoire ainsi que l'algorithme A* comprenant le recherche de chemin sont implémentés correctement. Cela dis malgré la rapidité général du calcul de la recherche chemin, l'optimisation via des tables de transposition n'a pas été effectué.

7.1 Optimisation possible

De nombreuses optimisation sont possibles pour le jeu du ricochet-Robot.Cela dis, le manque de temps a complètement résilier la possibilité de les implémenter.

1. **Les tables de transpositions :** permettant d'optimiser la recherche du chemin est une optimisation possible qui réduirait fortement le coup de calcul de l'exécution.

2. **L'implémentation d'une interface graphique :** Malgré un affichage sur la console plus que correct, l'interface aurait ajouter un aspect visuels qualitatif au jeu développé. Également une jouabilité du jeu via des cliques souris par un mouseListener aurait permis d'effectuer des parties en toute simplicité, par exclusion de l'utilisation du clavier.