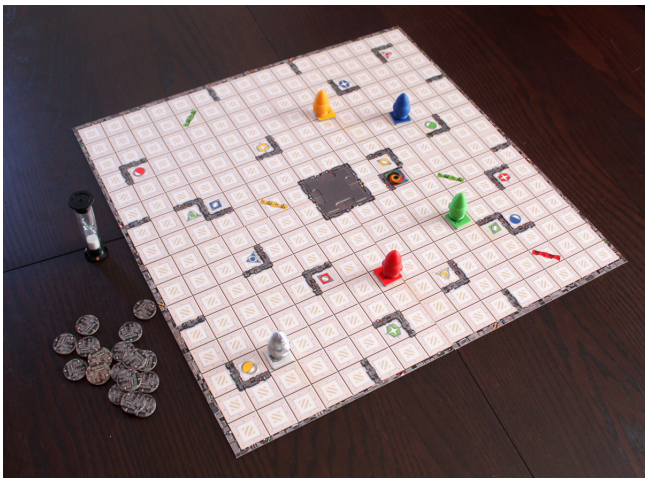
UNIVERSITÉ DE CAEN NORMANDIE

TRAVAIL PERSONNEL APPROFONDI

Ricochet Robot Solver



Membres du groupe :

1. AIT KHEDDACHE Wissam
2. BOUAOUD Malik
3. SINI Lynda
4. SI-MOHAMMED Sonia Taous
5. CUQUEMELLE Mathieu

Enseignants :
M.BONNET GREGORY
M.CHATEL ROMAIN
M.SASSI TAOUFIK

Groupe :
4A
L2 INFORMATIQUE

2019 / 2020

Table des matières

1 Introduction :

La programmation orientée objet est un paradigme informatique que chaque informaticien doit maîtriser. Ainsi, pour appliquer nos connaissances théoriques acquises tout au long de notre parcours académique, nous avons choisi de réaliser le projet de "Ricochet Robot Solver". Le but de ce projet est de développer un programme qui permet de trouver une solution optimale pour toute situation du jeu, c'est-à-dire trouver le plus court chemin entre deux objets. Les parties principales du projet sont les suivantes :

- Développement du moteur de jeu.
- Implémentation d'un algorithme de résolution A*.
- Implémentation des tables de transposition.
- Réalisation d'une interface graphique avec JavaFx.

2 Présentation du jeu :

Ricochet Robot est un jeu de société sorti à l'origine en Allemagne sous le nom de Rasende Roboter. Il s'agit d'un jeu de parcours se jouant de un à plusieurs joueurs. Son objectif est d'atteindre une case objective en un minimum de déplacements possibles.

2.1 Règles du jeu :

- Assembler quatre sous-plateaux afin de construire le plateau du jeu.
- Placer les quatre robots au hasard sur les cases du plateau.
- Mélanger les jetons objectifs face cachée.
- Retourner un jeton objectif : il indique la case objective et le robot qu'on doit bouger.
- Trouver une solution pour atteindre la case cible en minimum de coups possibles, en respectant les règles de déplacement citées ci-dessous.

2.2 Règles de déplacements :

- Un robot se déplace horizontalement ou verticalement sans pouvoir s'arrêter jusqu'à ce qu'il rencontre un obstacle : bords du plateau, murs, plaque centrale, un autre robot.
- Lorsqu'une barrière oblique colorée de la même couleur que le robot est rencontrée, le robot rebondit de 90°suivant la bonne direction.

3 Organisation du projet

3.1 Modifications apportées sur les règles du jeu

Les règles de ce jeu sont un peu compliquées à implémenter, et vu que nous sommes limités par le temps, nous nous sommes mis d'accord à faire certaines modifications sur le jeu.

1. Un obstacle est représenté par une case du plateau.
2. Un robot rencontré lors d'un déplacement est considéré comme un obstacle.
3. Un robot diffèrent du robot courant, ne peut être déplacé pour créer de nouveaux chemins.
4. L'assemblage aléatoire, et les rotations des plateaux, ne se font que du côté face.

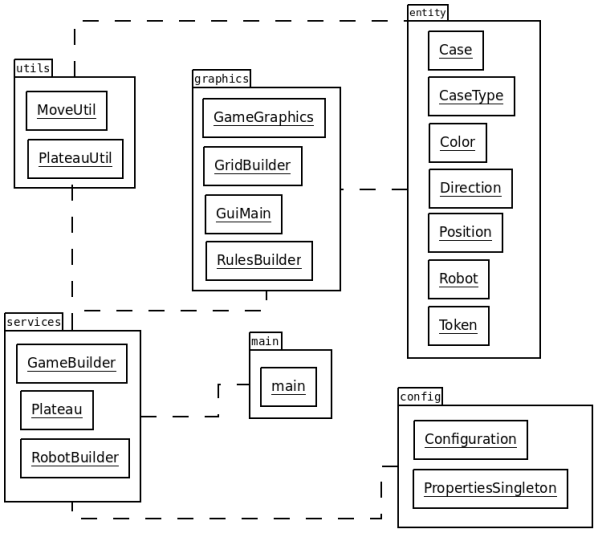
3.2 Répartition des tâches :

Ce travail a été réalisé par tous les membres du groupe ,on a essayé de travailler sur toutes les parties qui composent le jeu .

Tâches	Construction du projet		Implémentation de A*	interface graphique			
Participants	Construction du plateau	Mouvement des robots	A*	menu	plateau et son	mouvement des robots	rules
Malik	x		x		x		
Wissam	x		x			x	
Lynda		x	x				x
Sonia		x	x	x			
Mathieu	x	x					

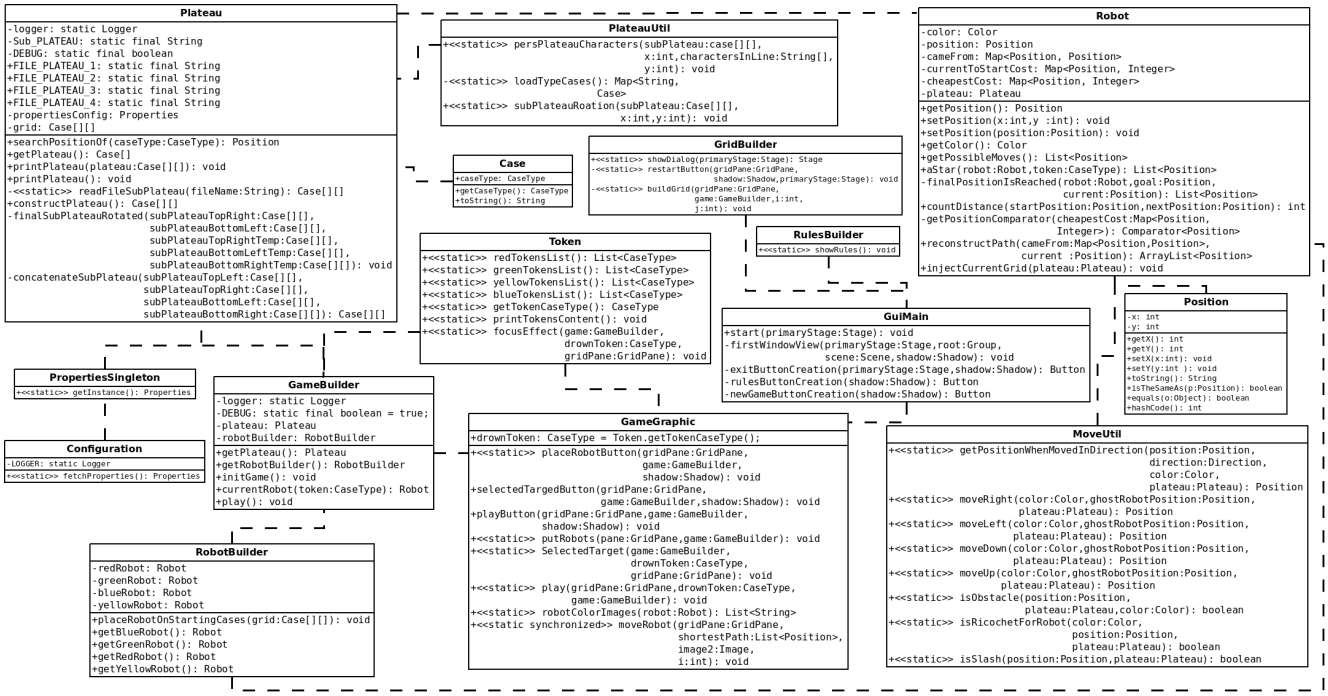
3.3 Architecture du code :

3.3.1 diagramme des packages :



Concernant l'architecture du projet, nous avons opté pour une décomposition en six packages :
`config` : Implémente le modèle de conception Singleton qui vise à contrôler l'initialisation des objets d'une classe particulière en garantissant qu'une seule instance de l'objet existe dans la machine virtuelle Java..
`entity` : Les entités nécessaires pour la construction de l'application .
`graphics` : La liaison entre le moteur du jeu et l'interface graphique
`main` : Classe test du jeu pour le lancer sur la console.
`services` : La couche métier qui correspond à la partie fonctionnelle de l'application.
`utils` : Il porte des classes utiles, comme la classes qui gère les mouvements des robots.

3.3.2 Diagramme des classes :



4 Éléments techniques :

Les éléments techniques se reposent sur les 5 points cités ci-dessous, c'est-à-dire la lecture des fichiers pour générer les plateaux, l'algorithme de recherche A* amélioré grâce aux tables de transposition, l' interface graphique développée avec JavaFx et le design pattern Singleton.

4.1 Lecture des fichiers :

Les fichiers évoqués représentent les 4 sous plateaux formant le grand plateau du jeu. Nous avons tenu à garder impérativement cette caractéristique importante du jeu, c'est-à-dire le fait de pouvoir mélanger les 4 sous plateaux à chaque début de partie. Ce sont les classes `Plateau` et `PlateauUtil`, qui nous permettent de faire cet assemblage avec les méthodes suivantes :

- *readFileSubPlateau(Plateau)* : Consiste à la lecture d'un fichier (.txt) et retourne une matrice de case **case[][]**
- *finalSubPlateauRotated(Plateau)* : Permet de réaliser la rotation des sous plateaux.
- *subPlateauRotation (PlateauUtil)* : Réalise la rotation des éléments internes des sous plateaux .
- *concatenateSubPlateau(Plateau)* : Concatène les 4 sous-plateaux
- *constructPlateau (Plateau)* : Consiste à lire les 4 sous plateaux et les assembler aléatoirement à l'aide des méthodes citées ci-dessus .

L'assemblage du plateau de jeu se fait donc dans la méthode *constructPlateau*, et se déroule comme ceci :

- On passe en paramètre un nom de fichier tiré aléatoirement avec la méthode *readFileSubPlateau*.

— Appel à *finalSubPlateauRotated*, le côté haut-gauche reste inchangé, les autres font une rotation avec inversement des éléments internes de 90°(les ricochets) cet inversement concerne seulement le côté haut droit et bas-gauche. Le côté bas-droit fait une rotation de 180°, donc ses ricochets ne changent pas.
— Après la réalisation de toutes ces opérations, il ne reste plus qu'à concaténer les 4 sous plateaux avec *concatenateSubPlateau*.

4.2 Algorithme A :*

4.2.1 Présentation :

A* est un algorithme qui consiste à trouver le plus court chemin entre un nœud initial et un nœud final dans un graphe, il est basé sur l'algorithme **Dijkstra**. A* a été créé pour que la première solution trouvée soit la meilleure. De plus cette extension est surtout une amélioration privilégiant la vitesse et la rapidité de calcul grâce à l'utilisation d'une **heuristique** d'évaluation, d'où sa célébrité dans le domaine de l'intelligence artificielle, plus précisément les jeux vidéo. Notre implémentation de l'algorithme a été perfectionnée pour rendre des résultats plus optimaux par rapport à l'original, et cela grâce à l'utilisation de l'heuristique adéquate, des tables de transposition, et le choix des différentes structures de données .

4.3 Heuristique :

Une heuristique dans le sens mathématique est un calcul fournissant rapidement une solution, pas nécessairement optimale ou exacte, à des fins d'optimisation. Son utilisation dans A* consiste à évaluer la distance qui sépare chaque nœud visité du but à atteindre, et cela pour estimer le meilleur chemin, et explorer ensuite les nœuds restants par rapport à cette évaluation. Mais il existe plusieurs façons pour calculer une heuristique, et donc le choix de cette dernière repose sur le fait de ne pas surestimer la distance du point de départ vers le point d'arrivée, et c'est son utilisation qui distingue A* des autres algorithmes existants. Comme notre jeu impose 4 directions possibles de mouvements aux robots, nous utiliserons donc : une taxi-distance alias *distance de Manhattan*.

4.4 Implémentation :

Algorithme 1 : ALGORITHME A*

Entrées : Robot robot, CaseType token
Sortie : Une Liste de Positions
Variables :
tentativeGscore : entier
startPosition, targetPosition, nextMove, current : Position
cheapestCost, currentToStartCost : Map<Position, Integer>
finalSmallestPath, possibleMoves : List<Position>
cameFrom : Map<Position, Position>
comparator : Comparator<Position>
openQueuePositions : Priorityqueue<Position>
openSetPositions : Set<Position>
début
$startPosition \leftarrow newPosition(robot.getPosition().getX(), robot.getPosition().getY())$
$targetPosition \leftarrow plateau.searchPositionOf(token)$
$comparator \leftarrow getPositionComparator(cheapestCost)$
$openQueuePositions \leftarrow newPriorityQueue <> (1, comparator)$
$openSetPositions \leftarrow newHashSet <> ()$
$openQueuePositions.add(startPosition)$
$openSetPositions.add(startPositions)$
$gScore.put(startPosition, 0)$
$fScore.put(startPosition, countDistance(startPosition, targetPosition))$
tant que !$openQueuePositions.isEmpty()$ **faire**
$current \leftarrow openQueuePositions.poll()$
$openSetPositions.remove(current)$
$gScore.put(current, countDistance(startPosition, current))$
$finalSmallestPath \leftarrow finalPositionIsReached(robot, targetPosition, current))$
si $finalSmallestPath.isEmpty()$ **alors**
retourner finalSmallestPath ;
fin
$robot.setPosition(current)$
$possibleMoves \leftarrow getPossibleMoves()$
pour $nextMove \in possibleMoves$ **faire**
$tentativeGscore \leftarrow gScore.getOrDefault(current$
$, Integer.MAX\_VALUE) + 15$
si $tentativeGscore < currentToStartCost.getOrDefault(nextMove,$
$Integer.Max\_VALUE))$ **alors**
$cameFrom.put(nextMove, current)$
$fScore.put(nextMove, tentativeGscore$
$+countDistance(nextMove, targetPosition))$
si !$openSetPositions.contains(nextMove)$ **alors**
$openQueuePositions.add(nextMove)$
$openSetPositions.add(nextMove)$
fin
fin
fin
fin
retourner *Collections.emptyList()*
fin

4.4.1 Déroulement :

Après l'initialisation des structures de données, nous ajoutons au gScore, et fScore la position initiale du robot, avec comme valeur 0 pour le gScore, et la valeur de la Map fScore.

$$(valeur)fScore = gScore + heuristique \tag{1}$$

Initialement openQueuePosition est non vide, donc nous récupérons une position avec la méthode *poll* (ayant le fScore le plus intéressant). On ajoute à la Map gScore la position et la distance entre le point de départ et le point courant.
Ensuite, on appelle la méthode *finalPositionIsReached* ,qui retourne une liste de positions, si une liste non vide est retournée cela signifie que l'objectif a été atteint, et qu'on doit arrêter de faire tourner notre algorithme.
Sinon(cas de la liste vide) le robot se place dans la case courante, et on appelle la méthode *getpossiblemoves*.
Après cela tentativeGscore reçoit soit, le score du noeud, s'il a été visité, sinon :

$$tentativeGscore = +\infty + N(N >= 15) \tag{2}$$

Dans le pseudoCode de A*, N vaut exactement 1 ,mais comme nos robots n'explorent pas forcément leurs cases voisines, nous avons remplacé ce 1 qui génère des boucles infinis par $N >= 15$ pour faire "grossir" artificiellement tentativeGscore, car à l'origine A* cherchera à minimiser le nombre de coups vers une destination.
Si tentativeGscore < gScore, donc ce nœud est plus intéressant que tous les nœuds précédemment visités, car son coût est inférieur, cameFrom représente une map ayant comme clé la position suivante à celle de la valeur, en d'autres termes, chaque position a comme clé sa position suivante.
On ajoute le nœud satisfaisant la condition précédente dans la map, et on enregistre son fScore dans sa Map, si le nœud nextMove n'est pas dans le set donc il n'est pas aussi dans la priorityQueue donc il n'a pas été visité, on doit alors l'enregistrer. Cette boucle continue à tourner tant que *finalpositionIsReached* ne nous retourne pas de liste vide, dans le cas échéant la méthode ne retourne jamais de liste non vide ce qui signifie donc qu'il n'y a pas de solution pour atteindre le jeton cible.
Pseudo-code de *finalpositionIsReached* :

Algorithme 2 : FINALPOSITIONISREACHED

Entrées : Robot robot, Position goal, Position current
Sortie : Une Liste de Positions
Variables :
aStarArray : ArrayList<Position>
cameFrom : Map<Position, Position>
début
  si $((current.getX() == goal.getX())$ ***et*** $(current.getY() == goal.getY()))$ **alors**
    $aStarArray \leftarrow reconstructPath(cameFrom, current)$
    $robot.setPosition(aStarArray.get(aStarArray.siz() - 1)$
    retourner aStarArray ;
  fin
  retourner *Collections.emptyList()*
fin

finalPositionIsReached à l'aide de la Map cameFrom qui contient comme **clé** la position suivante de celle de la **valeur**,nous retourne une ArrayList de positions contenant le chemin à prendre depuis le départ vers l'arrivée.

4.4.2 Tables de transposition

Une table de transposition est un cache de positions précédemment vues et d'évaluations associées, dans une arborescence de jeux générée par un programme de jeu informatique, voici une illustration meilleur ci-dessous (figure 1) :

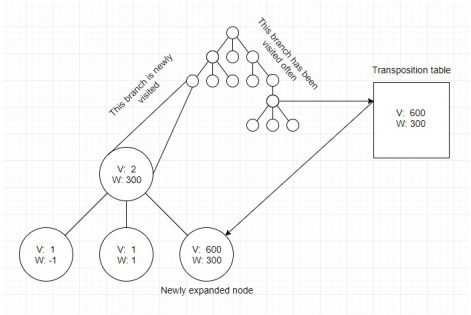


FIGURE 1 – Graphe d'une table de transposition

Notre implementation de l'algorithme A*, repose sur l'utilisation des tables de transposition, c'est le sens des HashMap **gScore** et **fScore**. Elles sont utilisées pour ne pas ré-étendre les nœuds ayant été déjà visités, ou un nœud avec un fScore inférieur à ce qui a déjà été trouvé, et donc A* gagne en rapidité considérablement.

4.4.3 La complexité

La complexité de A* dépend de l'heuristique utilisée, dans notre cas c'est *la distance de Manhattan*, elle est dans le pire des cas égale à celle de Dijkstra une complexité exponentielle $\mathbf{O}(e^n)$ où n est le nombre de nœuds du graphe, on rappelle qu'on a utilisé un Set au lieu de faire *priorityQueue.contains* et cela diminue la complexité de cette vérification qui passe de $\mathbf{O}(\log(n))$ à **O(1)**.

4.5 Design pattern Singleton

Un modèle de conception décrit une solution établie aux problèmes les plus fréquemment rencontrés dans la conception de logiciels. Le pattern Singleton vise à garder un contrôle sur l'initialisation d'une classe, en veillant à ce qu'une seule instance de l'objet existe dans toute la machine virtuelle Java. C'est ce qu'on a fait dans le package `config`, qui assure qu'une seule instance de classe Properties soit crée. Tout d'abord on crée une variable statique qui va permettre de stocker l'unique instance de la classe, c'est l'équivalent de l'attribut static "instance" crée dans la classe `PropertiesSingleton`, Si celui-ci est nul alors on crée une instance de la classe et on stocke sa valeur dans cet attribut. Sinon c'est que l'attribut possède déjà une instance de la classe. Dans tous les cas la méthode retourne la valeur de l'attribut possédant l'unique instance de la classe. Dans notre implémentation, nous avons créé une classe interne statique `SingletonHolder`, qui détient l'exemple de la classe Singleton. Il ne crée l'instance que si un appelle à la méthode *getInstance()* advient ,et non pas lorsque la classe extérieure est chargée. En outre, il faut noter que le constructeur a le modificateur d'accès privé. Il s'agit d'une exigence pour créer un Singleton, car un constructeur public signifierait que n'importe qui pourrait y accéder et commencer à créer de nouvelles instances.

4.6 Interface graphique avec Java Fx

JavaFx offre des interfaces graphique plus élégante que celle du swing , c'est pour cela qu'on l'a choisi pour l'implémentation graphique. le paquet *graphics* contient toutes nos méthodes implémentant une interface graphique. Nous avons 3 classes :

- `GridBuilder` : Contient des méthodes *showDialog, buildGrid, restartButton* créant un GridPane représentant notre plateau du jeu.
- `RuleBuilder` : Contient une méthode *showRules* affichant les règles du jeu.
- `GameGraphic` : Abrite plusieurs méthodes faisant marcher le jeu après l'affichage de la grille :
  - *placeRobotButton* charge les images des robots dans la grille grâce à un bouton.
  - *selectedTargetButton* crée un bouton qui ajoute un effet lumineux sur le jeton courant grâce au filtre RGB utilisé par la méthode *focusOnSelectedTarget*.
  - *playButton* crée un bouton "play" faisant appel à la méthode *play()* qui retrace le parcours du robot.
- `GuiMain` : Affiche la présentation du jeu (le Menu du jeu) avec 3 bottons : New Game, Rules, Exit.
  - New Game : Pour commencer la partie.
  - Rules : Pour afficher les règles de jeu.
  - Exit : Pour quitter le jeu.

Remarque : Musique enclenchée au lancement du jeu ,et tous les boutons ont un effet shadow d'animation lorsque la souris pointe sur un d'entre eux .

4.7 Utilisation de l'application :

Le jeu est un projet gradle. Gradle est un système de build qui permet de construire un projet java (structure et héirarchie).

4.7.1 Lignes de commandes :

Voici quelques commandes à lancer afin d'éxécuter le projet :
Placez vous dans le dossier robot-ricochet-mc-mb-wa-ls-ss (cf README).

- Sous Linux :
  - *./gradlew build* : génération du dossier build qui contient des sous dossiers contenant les binaires ...(les outputs).
  - *./gradelew run* : lancer le jeu .
  - *./gradelew javadoc* : génération d'un dossier docs/javadoc dans build qui contient la javadoc du projet.
  - *./gradelew jar* : génération du jar exécutable dans build/libs.
  - *./gradelew clean* : supprimer le dossier build.
- Sous Windows : Pour lancer le jeu il suffit de lancer la commande *gradle.bat run*.
- Sous MacOs : Les mêmes commandes que linux mais au lieu de mettre *./gradlew run* (exemple) il suffit de mettre *gradle run*.

4.7.2 Utilisation de l'interface graphique :

Après le lancement du programme, vous obtenez la figure 2 ci-dessous, qui représente le Menu du jeu :

FIGURE 2 – Menu du Jeu

Si vous cliquez sur le bouton **NewGame** vous obtiendrez la figure 3 :

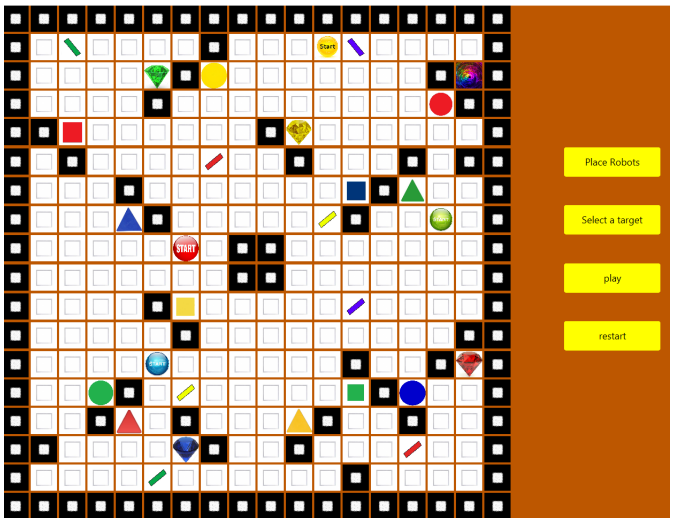


FIGURE 3 – Lancement du Jeu

Vous pouvez jouez en cliquant successivement sur les boutons suivants : **Place Robots –> Select a target –> Play**.
Le bouton **Restart** c'est pour rejouer.
Et si vous cliquer sur **Rules**, vous provoquez l'affichage de cette fenêtre (figure 4), qui explique les règles du jeu implémentés.

FIGURE 4 – Règles du jeu

5 Conclusion

5.1 Objectifs remplis

A terme de ce rapport, qui présente les détails d'un projet qu'on a réalisé en groupe de cinq étudiants, nous sommes arrivés à implémenter et optimiser l'algorithme d'intelligence artificielle **A*** qui cherche le plus court chemin qu'un robot peut emprunter pour atteindre l'objectif.

Ce travail nous a permis d'enrichir nos connaissances en informatique et précisément en Java, on a appris à utiliser **Git, Gradle, JavaFx**, on a réussi à implémenter un algorithme difficile comme **A***, ainsi que l'application des bases acquises en cours.

Le travail d'équipe, la répartition équitables des tâches, la bonne gestion de temps et le partage d'idées nous a apporté une bonne expérience dans le déroulement de ce projet .

5.2 Améliorations possibles

Pour conclure on pourrait apporter des modifications sur ce projet, par exemple, dans la construction du plateau on pourrait utiliser des murs autour des cases pour représenter les obstacles au lieu de représenter une case entière comme obstacle, ceci nous permettra d'avoir moins de blocage lors de la recherche du chemin .

Concernant les règles du jeu, nous pourrions également ne pas considérer les autres robots différents du robot courant comme des obstacles fixes, mais dynamiques, c'est à dire déplaçable de sorte que A* trouve d'autres chemins si la situation est bloquée, aussi l'utilisation d'une autre formule que la distance de Manhattan pour calculer l'heuristique

Finalement, concernant l'interface graphique elle pourrait être plus animée, par exemple création d'un effet graphique indiquant que le robot a gagné le tour,ou bien un effet sonore des applaudissements ...

Références

[1] wikepedia [en ligne] Disponible sur
https://en.wikipedia.org/wiki/A*-search-algorithm.

[2] openclassrooms [en ligne] Disponible sur
https://openclassrooms.com/fr/courses/26832-apprenez-a-programmer-en-java

[3] Stackoverflow [en ligne] Disponible sur
https://stackoverflow.com/questions/2049380/reading-a-text-file-in-java.

[4] Developpez.com [en ligne] Disponible sur
https://mikarber.developpez.com/tutoriels/java/introduction-javafx.

[5] Codeflow [en ligne] Disponible sur
https://www.codeflow.site/fr/article/gradle.

[6] Gimps [logiciel du dessin] téléchargeable sur
https://www.gimp.org/downloads/.

[7] Google images [en ligne] Disponible sur
https://www.google.com/.

[8] Heuristics From Amit's Thoughts on Pathfinding [en ligne] Disponible sur
https://theory.stanford.edu/ amitp/GameProgramming/Heuristics.html.

[9] Baeldung [en ligne] Disponible sur
https://www.baeldung.com/creational-design-patterns.