Université de Caen Normandie
Licence informatique 2ème année

4 avril 2020
Rapport Conception logicielle avancée

Solveur de Ricochet Robot

Lefevre Alex & Malbec Elie & Kablan Yoann & Vouvou Brandon

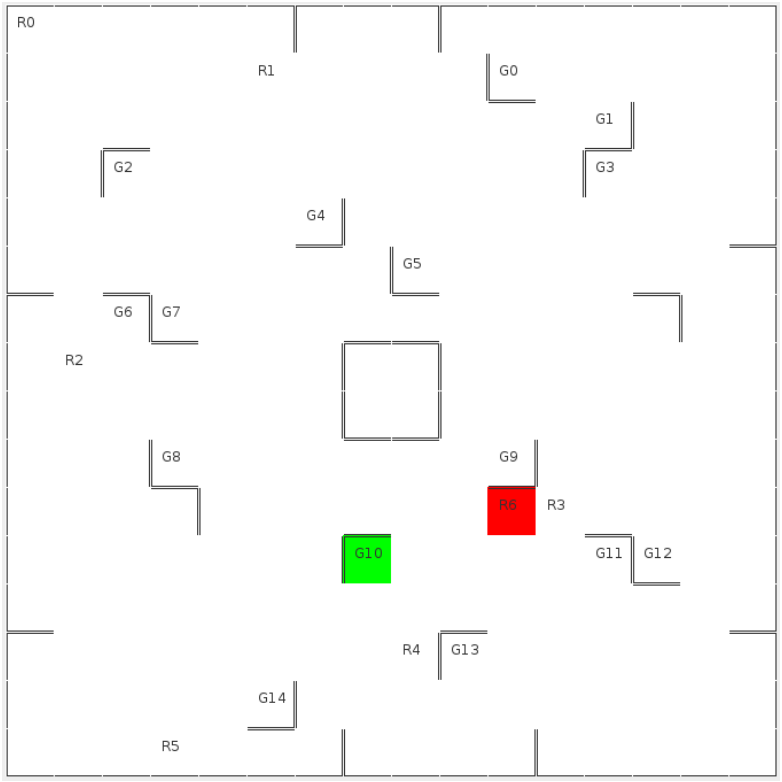

Table des matières

1 Introduction

1.1 Introduction

Dans le cadre de l'unité d'enseignement Conception Logicielle Avancée enseignée à l'Université de Caen en licence 2ème année, nous avons réalisé un solveur du jeu Ricochet Robot. Nous avons formé un groupe de quatre étudiants : Alex Lefevre, Elie Malbec, Yoann Kablan et Brandon Vouvou pour concevoir une IA (intelligence artificielle) résolvant le jeu. Dans ce rapport, nous vous exposons les démarches qui nous ont permis de concevoir un algorithme de résolution ainsi que les difficultés rencontrées durant son élaboration.

Notre réflexion portée sur le concept du jeu et sa méthode de résolution nous ont motivés pour le sélectionner parmi les autres.

Notons également une contrainte imposée pour le développement, le langage de programmation imposé est le Java.

1.2 Le jeu Ricochet Robot

Ricochet Robot est dans sa version originale un jeu de plateau composé d'une grille de taille $16*16$. Chacune des 256 cases est représentée par une paire de coordonnées (x, y). Une case peut comporter des murs à gauche, à droite, en haut et en bas faisant office d'obstacle de déplacement aux quatre entités appelées *robots*. La grille est délimitée par une bordure de murs extérieurs. Les murs sont appelés suivant leur direction : murs 'nord', 'sud', 'est' et 'ouest'. Quatre cases au centre du jeu forment une zone noire et ne sont pas accessibles.

Les robots sont des entités placées aléatoirement sur le plateau du jeu. Deux robots ne peuvent pas être présents sur la même case au même moment ni dans les cases centrales. Chaque robot peut se déplacer suivant les directions cardinales jusqu'à ce qu'il rencontre un obstacle, ce dernier peut être de deux types : un mur ou un autre robot. Pour un robot placé sur une case, il lui est impossible de se déplacer sur la droite si la case sur laquelle il est possède un mur à l'Est.

De multiples cibles sont également placées dans les angles formés par deux murs. Un des robots dit robot principal doit atteindre la cible principale pour terminer le jeu. Les mouvements des robots annexes peuvent servir au robot principal pour créer de nouveaux obstacles de déplacement. Le but est de se rendre sur la cible principale avec le moins de mouvements possible.

Chaque position sur la jeu peut être représentée par des coordonnées sur un plan deux dimensions. Voici les positions utilisées, elles correspondent à l'indexation des objets avec la structure de tableau à double dimension.

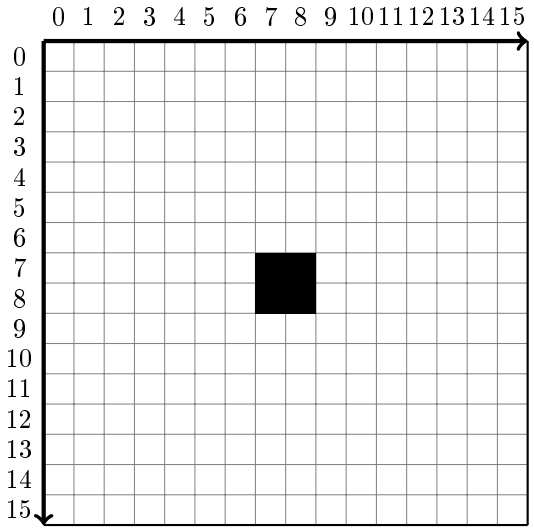

FIGURE 1 – Coordonnées sur le plateau de jeu

Il y a 17 cibles sur le plateau, donc 17 tours de jeu dans le jeu original, cependant, dans notre cas, nous n'avons pas mis en place ces règles de jeu mais uniquement la méthode de résolution et la possibilité de jouer en interface graphique.

1.3 Objectif du solveur

L'objectif de la résolution du jeu est de minimiser le nombre de déplacements nécessaires pour permettre au robot principal de se rendre sur l'objectif. Plusieurs solutions permettent d'arriver au résultat mais certaines sont plus rapides.

Chaque robot déplacé compte pour un mouvement. Le fait de changer la position de robots annexes apporte de nouvelles possibilités. Nous avons développés deux manières de résoudre ce problème, chaque méthode comporte des avantages et des inconvénients que nous détaillerons par la suite. La complexité du problème réside dans le nombre de déplacements possible à chaque état de jeu. Un nombre exponentielle de d'états s'ajoutent à chaque mouvement.

2 Développement du moteur de jeu

La première étape pour la réalisation de ce projet était de concevoir le modèle du jeu. Pour cela, un premier package `modele` comporte l'ensemble des classes nécessaires à sa modélisation.

2.1 Plateau du jeu

Pour modéliser le plateau, nous avons développé une classe `Board`. Cette dernière est constituée de 256 cases et représente l'état actuel du jeu. Les entités sont représentées par des classes `Robot` et `Goal`. Les robots et cibles ne sont pas directement placés sur le plateau mais possèdent leur propre position. Cela permet de ne pas vérifier dans chaque case la présence de telle ou telle entité.

Les robots et les cibles sont stockés dans des listes avec la structure de donnée `ArrayList`. Des références au robot principal et à la cible principale permettent de les distinguer des autres.

La construction du plateau de jeu est basée sur un fichier texte, ce fichier contient des coordonnées et une description des murs et des entités présentes (4.2.1). Cette construction est par défaut basée sur celle du plateau de jeu officiel. Mais il est possible de générer un plateau aléatoirement ou de choisir un autre fichier de configuration.

Voici le diagramme de classe donnant les principales méthodes utilisées sur le plateau mais également la structure des robots et cibles.

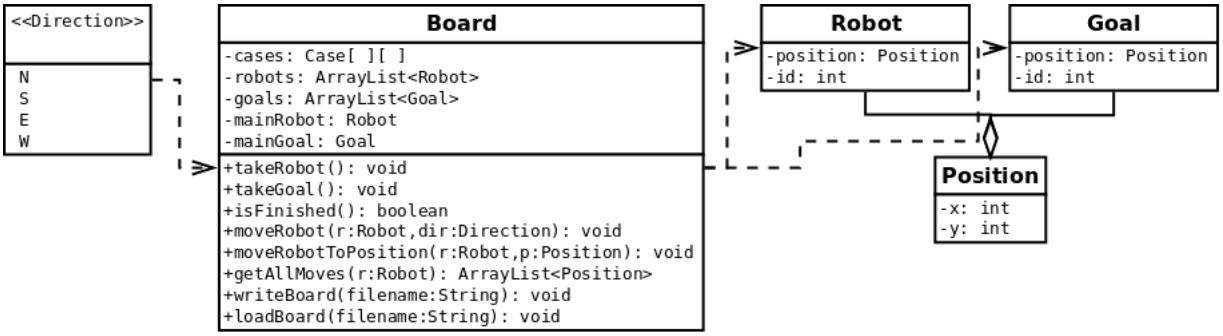

FIGURE 2 – Diagramme de classe du Board et des entités

Le plateau dispose de méthodes qui lui permettent de bouger les robots, de tester si le jeu est terminé mais aussi de tirer une nouvelle cible ou un nouveau robot aléatoirement. Les directions sont listées depuis une énumération et les positions représentées avec deux coordonnées pour un plan deux dimensions.

Nous pouvons également remarquer que les 256 cases sont des objets `Case` stockées dans un double tableau. Cela permet un accès direct aux données mais aussi un parcours simplifié avec une largeur et une hauteur connue. La composition d'une case est la suivante :

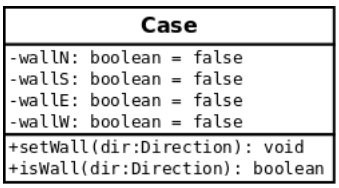

FIGURE 3 – Classe Case

Les quatre murs possibles sont par défaut désactivés puis activés avec la méthode `setWall` dans la direction donnée. Une fois les murs mis en place, ils ne changent plus durant la partie.

2.2 Déplacements

Concernant les déplacements, les méthodes `moveRobot` et `getAllMoves` nécessitent des tests supplémentaires. Il faut vérifier que les déplacement sont valides et donner les position d'arrivée quand un obstacle est rencontré. Pour cela, des méthodes ont été implémentées dans la classe `Board` dont voici le diagramme :

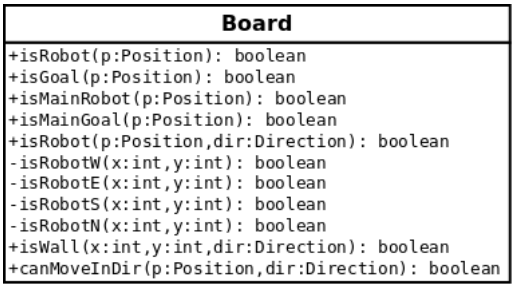

FIGURE 4 – Méthodes pour déplacements dans Board

Les déplacements sont ainsi possibles en donnant spécifiant un robot et une direction à la méthode `moveRobot`. La position d'arrivée est renvoyée et le déplacement est effectué en mettant à jour la position du robot concerné. Notons également qu'une cible ne constitue pas un obstacle à un mouvement.

2.3 Visualisation du jeu

Pour permettre de jouer au clavier et rendre le jeu plus visuel, une interface graphique à été développée en suivant le schéma MVC (Modèle Vue Contrôleur). Les murs sont représentés par des traits et les entités sont placées par rapport à leurs coordonnées respectives. Le robot en cours de déplacement est coloré en rouge et la cible principale en vert. Cette visualisation permet de mieux voir les données comparativement à une lecture sous forme textuelle. La figure 5 donne un aperçu de la vue sur le `Board`.

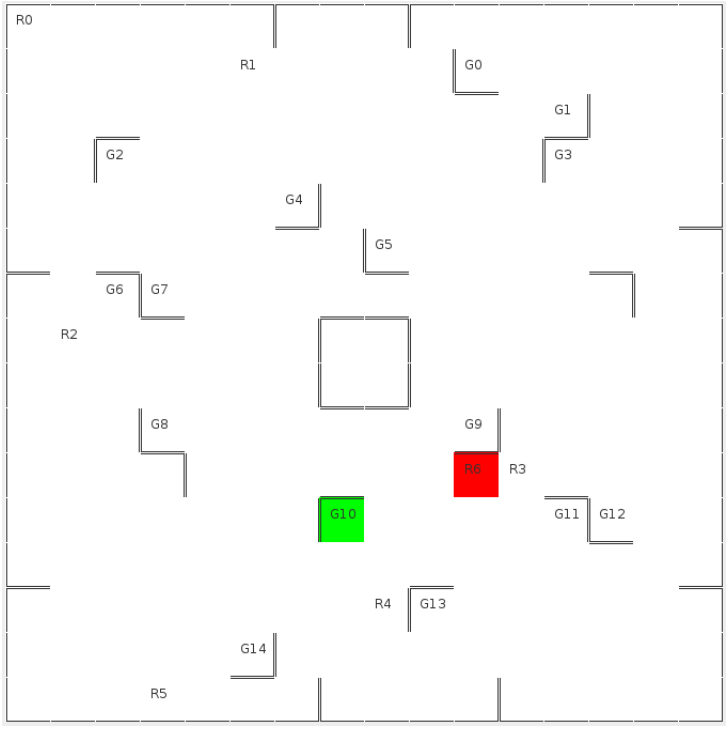

FIGURE 5 – Visualisation graphique du jeu

2.4 Détails de l'implémentation

Lors du développement de ce modèle de jeu, nous avons eu des difficultés quand aux structures de données à utiliser. L'utilisation du gestionnaire de version Git nous à permit de créer plusieurs versions plus ou moins

stables du modèle. Dans un premier temps, nous avions décidé de placer les entités directement dans les cases du plateau de jeu. Cette méthode était très difficile à maintenir car les positions des entités étaient stockées dans des `HashMap`.

Nous avons par la suite abandonné ce système pour réaliser une seconde implémentation du modèle. Dans cette dernière, nous avons gardés la structure des cases, c'est à dire quatre attributs correspondants au murs possibles. Un ensemble de méthode ont été créées pour générer des plateaux aléatoires. C'est sur cette version que sera basée la prochaine utilisant les algorithmes de résolution.

3 Méthodes de résolution

Dans cette partie, nous allons voir les deux algorithmes utilisés pour la résolution du jeu. Il s'agit de l'algorithme $A*$ et de *BFS* (Breadth First Search). Chaque méthode possède des avantages et des inconvénients que nous allons détailler ci-dessous.

3.1 Historique du projet

Durant le développement du projet, nous nous sommes tout d'abord penchés sur l'algorithme de résolution $A*$. Ce dernier possède un avantage théorique car il limite le nombre d'état possible grâce à une heuristique de plateau. L'implémentation de $A*$ à nécessité l'utilisation de méthodes définies dans la classe `Board`. Cependant nous avons eu beaucoup de mal à trouver une heuristique cohérente permettant de trouver la solution. Après certaines recherches, nous avons décidés de se pencher sur l'algorithme *BFS*, ce dernier néglige la mémoire au profit de trouver une solution rapide. C'est cette méthode qui nous à donné le plus de résultats cohérents.

Pour mettre en place ces résolutions, une dernière version du modèle à été mise en place, avec des entités robots et cibles qui possèdent leur propre position. Cela permet de les déplacer virtuellement sur le jeu tout en ne stockant que ces positions pour les différents états.

3.2 Algorithme A*

3.2.1 Principe

L'algorithme $A*$ est un algorithme de recherche de chemin dans un graphe entre un nœud initial et un nœud final. Dans notre cas, le nœud initial est l'état de base du plateau de jeu, les nœuds intermédiaires les possibles chemins empruntés par les robots et le nœud final l'état du jeu terminé. Cet algorithme utilise une évaluation heuristique pour déterminer le meilleur chemin pour se rendre sur la cible.

Nous avons du réaliser une structure de nœud avec une relation d'ordre suivant leur évaluation. En effet, l'évaluation d'un nœud n est notée $f(n)$. La fonction d'évaluation $f(n)$ est un nombre réel positif ou nul, estimant le cout du meilleur chemin du nœud initial passant par n et arrivant au but. $f(n)$ se décompose en deux parties : $f(n) = g(n) + h(n)$

— $g(n)$ le coût du meilleur chemin ayant mené au nœud n depuis le nœud initial.

— $h(n)$ est une fonction d'estimation du coût restant entre un nœud n d'un graphe et le but.

Ici $g(n)$ est définie par la somme des distances de Manhattan de la position du nœud n à la cible *goal*. Cette somme défini le coût nécessaire pour se rendre à ce nœud depuis l'origine. Pour $h(n)$, nous avons utilisé la distance de Manhattan du robot principal à la cible. Cette évaluation vaut 0 si le robot atteint la cible et est maximisée si le robot est loin de la cible.

$A*$ est optimal si l'heuristique $h(n)$ ne surestime jamais le coût pour se rendre au nœud suivant n' et si les nœuds enfants possèdent une heuristique plus basse que celle du nœud courant.

Cet algorithme à besoin d'une liste dite *open* qui contient les nœuds pas encore traités : i.e. la frontière de la partie du graphe explorée jusqu'à maintenant. Les nœuds sont triées selon l'estimé $f(n)$ la fonction d'évaluation. On explore les nœuds les plus prometteurs en premier. Et d'une liste dite *closed* qui contient les nœuds déjà traités, i.e. à l'intérieur de la frontière délimitée par *open*.

3.2.2 Implémentation

Pour implémenter cet algorithme, nous avons créer une classe `Solver` suivant un état du jeu donné. Les nœuds implémentent l'interface `Comparable` permettant d'établir une relation d'ordre entre eux. Les listes *open* et *closed* sont des `ArrayList<Node>`. Chaque nœud contient une position du robot principal. Le diagramme de classe associé est la figure 6 ;

En suivant l'exécution de cet algorithme, nous pouvons trouver une solution n'impliquant que le robot principal. Cette résolution fonctionne si le chemin pour se rendre à la cible est direct. Cependant cela ne garantie pas qu'il s'agit du chemin optimum (avec peu de coups).

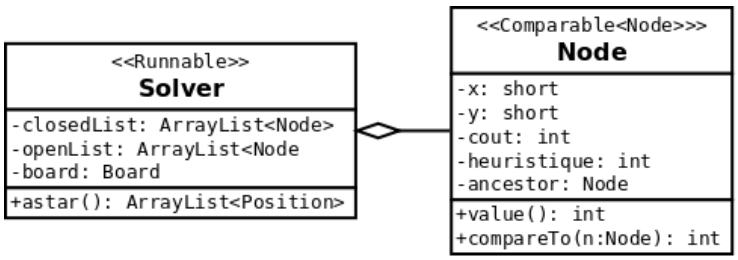

FIGURE 6 – Diagramme de la classe Solver

Nous avons donc tentés d'ajouter les mouvements des autres robots dans la liste *open* après chaque coup joué avec différentes heuristiques :

— Distance de Manhattan entre le robot principal et la cible
— Somme des distances de tous les robots à la cible
— Distance de Manhattan minimum après tous les coups possibles pour le robot principal

Mais aucune ne permet de résoudre le jeu car des coups sont joués jugés bons par leur valeur mais en réalité bloquants, inutiles ou inverses. Il est très difficile de donner un poids différent à chaque état de jeu. Ce constat s'est confirmé par plusieurs rapports qui estiment que chaque mouvement (et donc chaque lien dans l'arbre) à un poids équivalent si c'est un robot annexe qui est déplacé. Dans ce cas précis, la distance entre le robot principal et la cible principale ne change pas, tous les nœuds auront la même valeur, l'algorithme en choisira un au hasard et le jouera. Mais ce ne sera pas forcement un coup utile à la résolution.

Ce type d'algorithme n'est pas performant pour ce jeu, les déplacements sont effectués dans des directions et peuvent éloigner le robot pour qu'au coup suivant il atteigne sa cible. Toute la difficulté réside dans la création d'une heuristique complexe faisant intervenir des état futurs.

Nous avons en conséquence réalisé un $A*$ déplaçant uniquement le robot principal.

3.2.3 Remarques

L'algorithme $A*$ permet de résoudre rapidement un chemin direct pour se rendre sur la cible. Mais dans la plupart des cas, ce chemin n'existe pas et le déplacement de robots secondaires est nécessaires. Il n'y a donc aucune garantie quand au résultat. De plus, le chemin peut contenir plus de mouvements que nécessaire, ne donnant pas pleine satisfaction par rapport au but de notre projet.

3.3 Algorithme BFS (Breadth First Search)

3.3.1 Principe

Dans une recherche en largeur, il faut commencer à la racine *root* qui est l'état initial. Puis ajouter tous les enfants *child* de *root* dans une liste depuis les mouvements possible. Pour chaque état *child*, pris l'un après l'autre, nous testons si c'est un état final, i.e. un jeu terminé dans notre cas avec le robot sur sa cible. Si aucun état final n'est rencontré dans tous les état *child*, génération de tous les états de niveau 2 à partir de chaque *child*. Une nouvelle liste d'état de niveau 2 est testée et ainsi de suite.

Si un état final est rencontré, remonter le chemin jusqu'à la racine *root*.

Cette méthode de résolution garantie un résultat rapidement, mais au détriment de la mémoire. De plus, la première solution est garantie d'être la solution optimum.

Le nombre d'état est exponentiel, ce que pose des problèmes de mémoire et empêche la résolution de long chemins. À l'état initial, il y a 4 robots qui peuvent se déplacer dans 4 directions, cela apporte 16 possibilités. Puis en profondeur 2, pour chacun des 16 cas, il y a 2 à 3 mouvements possibles pour chaque robot. L'optimisation de cet algorithme peut résider d'éviter les mouvements inverses mais aussi de déplacer en premier le robot principal.

3.3.2 Implémentation

Ricochet Robot est un problème complexe à résoudre, le nombre de cas augmente très rapidement dès la profondeur 6. Nous avons choisi de choisir ce algorithme car il permet de résoudre plus de situations de jeu. Mais les performances de notre modèle nous empêchent de trouver des chemins de plus de 5 mouvements en raison de manque de mémoire et d'optimisation. Chaque état est stocké suivant la classe interne `State` contenue dans la classe `BFSSearch`, figure 7.

Ainsi, la mémoire utilisée pour stocker un état de jeu est assez fiable, un lien avec tous les ancêtres depuis l'état courant permet de tracer le chemin parcouru par tous les robots depuis le plateau de base contenu dans la

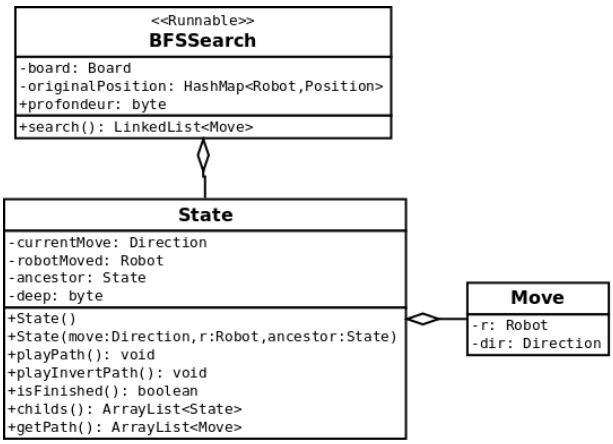

FIGURE 7 – Diagramme de classe BFSSearch

classe `BFSSearch`. Par exemple, pour savoir si l'état est considéré comme terminé, il faut reconstituer le chemin depuis l'origine, puis jouer chaque coup depuis le plateau de base. Ensuite vérifier si le robot est sur la cible, enfin reconstituer le plateau de base en réinitialisant les positions des robots depuis `originalPosition`.

Pour gagner du temps lors de l'exécution il serait possible de déterminer les positions d'arrivée à chaque état et de placer directement le robot à cet endroit à la place de réaliser les multiples tests de déplacement.

Le temps nécessaire pour trouver un chemin est assez court pour des résolutions en 5 mouvements, mais au delà il faut une puissance de calcul supplémentaire et de la mémoire. En effet, chaque état est stocké en mémoire, une optimisation viserait à supprimer ces états et les retrouver en chemin inverse.

3.3.3 Détails

Le pseudo-code de l'algorithme BFS est l'algorithme 1.

Algorithme 1 : SEARCH Algorithme BFS de résolution

Entrées : Plateau initial *Board b*
Output : Liste chaînée de *Move* pour la résolution

```
1  Bool finished ← false
2  State root ← b
3  Add root to nodes list
4  tant que finished = false faire
5      new list childs
6      pour State s in nodes faire
7          add all childs(s) to childs list
8      pour State s in childs faire
9          si isFinished(s) alors
10             retourner path(s)
11     nodes ← childs
12 retourner null
```

3.4 Conclusion

Nous avons eu l'occasion d'implémenter deux algorithmes de recherche. L'un d'eux, $A*$ n'est pas une solution viable sans une heuristique complexe utilisant les états futurs. Mais cette méthode à l'avantage d'être très rapide en explorant uniquement les cas jugés utiles. Nous n'avons pas réussi à déterminer cette heuristique. De plus, le chemin trouvé n'est pas forcement le chemin optimum.

Le second algorithme nous a cependant permis de résoudre une partie du problème, des améliorations pourraient lui être apportées pour explorer des chemins plus complexes et coûteux en mémoire. Cette méthode à l'avantage de trouver la bonne solution la plus rapide pour arriver au but. Sa complexité est exponentielle et son temps d'exécution assez rapide car la majorité des solutions se trouvent en moins de 6 déplacements.

Pour ouvrir sur d'autres possibilités, l'algorithme DFS (Deep First Searching) est une exploration en profondeur qui nécessite peu de mémoire au détriment du temps de résolution. Les branches sont explorées jusqu'au dernier nœud possible (avant de faire des mouvements arrières).

4 Fonctionnalités implémentées et architecture

4.1 Interface graphique

Une interface graphique[8] permet de visualiser l'état d'un plateau de jeu. Le choix de la cible s'effectue via une liste déroulante, de même pour le robot colorié en rouge. Il est possible de déplacer le robot en cours dans la direction souhaitée par l'appui sur les flèches directionnelles. Chaque action entraîne la mise à jour de l'interface graphique conformément à la structure MVC. Un message s'affiche lors de la résolution du jeu. Des boutons permettent de lancer les algorithmes de recherche pour trouver une solution au problème actuel. C'est à dire pour permettre au robot de se rendre sur la cible. La remise à zéro du plateau est un rechargement du fichier de configuration du jeu officiel.

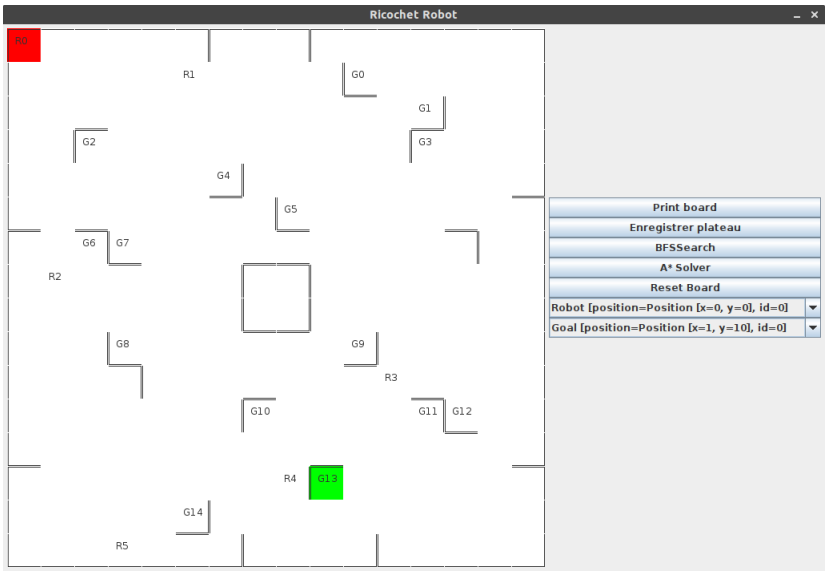

FIGURE 8 – Interface de jeu

4.2 Génération aléatoire de plateau

La génération aléatoire de plateau créé un fichier qui pourra être chargé comme plateau de jeu. Sa génération repose sur certaines règles :

— Création d'un bordure extérieure du jeu.
— Création d'un mur sur chaque coté de chaque quart de terrain.
— Quatre cases centrales inaccessibles.
— Création de 17 angles formés par des murs, ne devant ni toucher les cases centrales, ni les bordures extérieures, ni d'autres angles. Il y a 4 angles par quart de terrain et 1 autre placé aléatoirement.
— Placement des cibles dans les angles
— Placement aléatoire de 4 robots, jamais dans la même case.

Il est également possible d'enregistrer le plateau courant depuis l'interface graphique pour reprendre une configuration.

4.2.1 Format du fichier de configuration

Le fichier contenant les données de configuration d'un plateau de jeu est composé de la manière suivante :

1. Coordonnée x de la case
2. Coordonnée y de la case
3. Description de la case avec des espaces entre chaque caractéristique :
   — N S E W pour les murs
   — R G pour les robots et cibles

Par exemple, *0 0 W N R* donne pour la case en position $(0, 0)$ un mur à l'Ouest, un mur au Nord et un Robot.

Il est possible d'ajouter d'autres robots et de nouvelles cibles en ajoutant de nouvelles lignes de configuration. Elles peuvent être mélangées.

4.3 Architecture du projet

Voici l'architecture de notre projet :

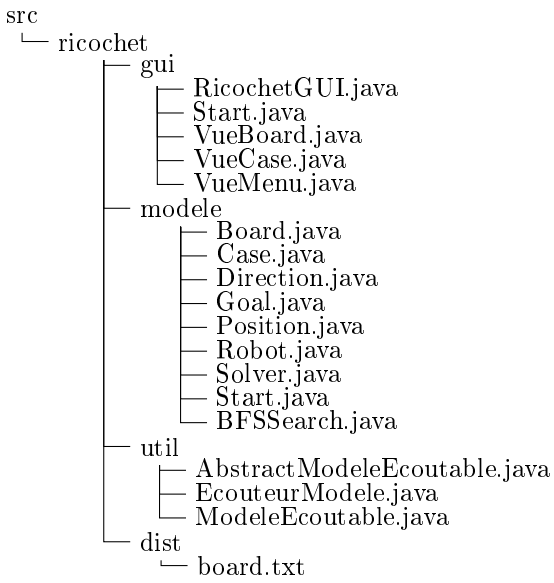

FIGURE 9 – Arborescence des dossiers

5 Expérimentations

5.1 Comparaison du temps de calcul

5.2 Nombre de nœuds explorés

6 Conclusion

Ce projet à été très formateur pour l'ensemble du groupe, nous avons pu découvrir certaines pratiques algorithmiques intéressantes. La gestion du projet avec une succession de versions permet à chacun d'essayer certaines méthodes sans impacter le tronc commun. La question de l'heuristique dans des graphes est complexe et nécessite de l'expérience. De plus, ce projet est perfectible, par une série d'améliorations possible :

- Mise en place d'un timer de jeu pour limiter le temps de résolution à la fois pour un joueur humain et pour les algorithmes.
- Ajout de scores pour un jeu multijoueur.
- Réalisation d'un éditeur de plateau graphique.
- Amélioration des algorithmes de recherche, notamment le BFS.
- Sélection des cibles et robots au clic de la souris et amélioration de l'interface graphique.

Références

[1] Hugo Larochelle, chercheur en apprentissage automatique et chercheur chze Google Brain. `http://info.usherbrooke.ca/hlarochelle/cours/ift615_E2013/contenu.html`

[2] Course :CPSC :Artificial Intelligence/States and Searching, The University of British Columbia `https://wiki.ubc.ca/Course:CPSC:Artificial_Intelligence/States_and_Searching`

[3] A Case Studu for Human Complex Problem Solving, 15 Septembre 2005

[4] MoutainWest RubyConf 2015 - Solving Ricochet Robots `https://www.youtube.com/watch?v=fvuK0Us4xC4`