RAPPORT DE PROJET – EI4 AGI

Application Ricochet Robots

Projet réalisé par :

CAILLAUD Alain
MICHEL Pierre

Projet encadré par

M. AUTRIQUE Laurent

ISTIA - Université d'Angers

Remerciements

Avant d'aller plus loin, nous tenons à remercier M. Laurent AUTRIQUE, notre tuteur de projet pour nous avoir proposé ce projet et pour nous avoir accompagnés durant son déroulement.

Nous souhaitons également remercier tous ceux qui, nous ont permis d'avancer dans cet objectif de projet.

Nous remercions les différentes personnes qui ont travaillé sur les algorithmes solveur de Ricochet Robots et qui ont mis en ligne leurs techniques, leurs optimisations et leurs résultats pour avoir partagés leurs connaissances.

Nous remercions également l'ensemble du personnel de l'ISTIA pour nous avoir permis de travailler sur ce projet.

I. Table des matières

I. Introduction

Durant notre projet d'année EI4, nous nous intéressons à un jeu de société qui existe sous le nom de Ricochet Robots. C'est un jeu de société où les joueurs doivent déplacer des robots en faisant le moins de coups possible pour atteindre un objectif. Chaque partie fait ainsi appel à beaucoup de réflexions.

Comme c'est le cas pour beaucoup de jeux de société, il est possible de réaliser une version virtuelle du jeu, que ce soit sur ordinateur ou sur tablette/smartphone. Notre premier objectif durant le projet est donc de réaliser une version de Ricochet Robots utilisable sur tablette Android. L'intérêt de versions virtuelles des jeux de société est que nous pouvons utiliser les ressources de calculs informatiques dans l'intérêt du jeu. De même qu'il existe des solveurs pour les jeux d'échecs, nous pouvons réaliser un solveur pour le jeu Ricochet Robots. Notre objectif de projet est donc également d'étudier comment on peut utiliser les ressources informatique pour réaliser un solveur pour notre application.

Ce projet nous a donc amené à travailler sur des connaissances qui sont nouvelles pour nous, que ce soit le développement Android jusqu'aux algorithmes de résolution. Ce rapport a pour objectif de présenter comment nous avons travaillé et quels ont été les résultats de ce projet.

A travers ce document, nous présenterons dans un premier temps le jeu et le développement sous Android. Ensuite, nous présenterons l'application que nous avons réalisée suivi de l'algorithme de résolution.

II. Présentation

II.1 Le jeu Ricochet Robots

Le jeu Ricochet Robots est un jeu de société qui a été réalisé en 1999 par l'Allemand Alex Randolph. Ce jeu peut se jouer seul tout comme avec beaucoup de participants. Le plateau du jeu représente une grille sur laquelle on a différents murs. Ce plateau est divisé en 4 parties double-face avec des murs positionnés de manière différente de chaque côté. On assemble les 4 parties en choisissant pour chacune d'elle un des deux côtés aléatoirement. On obtient ainsi un plateau avec une des 16 combinaisons de grille de jeu possible. On place ensuite les 4 robots aléatoirement sur la grille.

Figure 1 : Exemple de situation de jeu

Comme nous pouvons l'observer sur la photographie, des cibles de couleurs sont positionnées dans les angles que forment les murs. Durant une partie, un des joueurs pioche une des cibles, l'objectif est d'amener le robot de la couleur correspondante sur la cible piochée en faisant un minimum de coups possible. Les robots se déplacent en ligne ou en colonne, cependant, une fois une direction choisie (haut, bas, gauche ou droite) ils ne peuvent s'arrêter tant qu'ils ne rencontrent pas un obstacle qui peut être un mur ou un autre robot. Les robots qui ne sont pas de la couleur de la cible ne permettent pas de gagner mais l'utilisateur peut les déplacer (en conservant les règles de déplacement citées précédemment) de manière à créer des obstacles pour le robot principal selon ses besoins.

Chaque déplacement d'un des 4 robots compte pour un coup. Dès qu'un des joueurs a trouvé une solution, on lance un compte à rebours et les autres joueurs ont une minute pour trouver une solution avec moins de coups que le premier joueur. A l'issue de ce délai, le joueur ayant la solution comptant le moins de coups présente sa solution et remporte la manche.

En 2003, une seconde version du jeu est sortie, celle-ci diffère par la présence d'un 5ème robot de couleur noir ainsi que d'obstacles constitués de 3 murs au lieu de deux comme sur la photo :

Version du jeu sortie en 2003

Pour notre projet, nous avons choisi de travailler avec la première version du jeu, celle avec 4 robots.

II.2 Android

Android est un système d'exploitation pour mobile qui a été réalisé par une startup du même nom. Il se base sur un noyau Linux et est open source. Depuis le rachat de la startup en 2005 par Google, Android n'a cessé d'évoluer et de conquérir le marché, selon l'entreprise américaine de conseil Gartner, 80 % des smartphones vendus en 2014 intégraient l'OS Android.

II.3 Développement Android

Les applications fonctionnant sous Android sont généralement écrites en Java. C'est donc en Java que nous avons réalisé notre application. Il existe différents logiciels (IDE) permettant de programmer une application pour Android notamment Eclipse. Cependant, le 8 décembre 2014, alors que nous commencions le projet, Google sort officiellement son

environnement de développement Android Studio qu'il conseil désormais. Nous avons ainsi naturellement choisit d'utiliser de logiciel pour la réalisation de notre projet.

II.4 Les applications déjà existantes

Avant de commencer notre application, nous avons naturellement regardé les applications déjà existantes sur le thème. Sur Google Play, nous avons trouvé 4 applications reprenant le jeu Ricochet Robots.

- Ricochet Robots : Cette application permet de jouer à Ricochet Robots sur 3 grilles différentes conformes au jeu original. On y joue avec 5 robots. Depuis le début de notre projet, cette application a disparu de Google Play.

- Ricochet Robot : Cette application permet également de jouer dans des conditions semblables au jeu original. Les grilles sont formées de la même manière que le jeu originale, c'est à dire parmi les 16 combinaisons possibles. Les scores des joueurs sont stockés sur un serveur et permettent de créer un classement parmi les joueurs.

- Ricochet Racer : Dans cette application, les robots ont été remplacés par des voitures. Les grilles sont de taille plus petite 8 x 8, elles peuvent cependant être générées de manière aléatoire.

- Escaping Droids : Cette application qui est encore en version beta à ce jour permet de jouer parmi 24 challenges différents. Les grilles diffèrent par rapport au jeu original, il n'y a pas de murs réservés au milieu et il peut y avoir jusqu'à 4 murs placés en forme de croix.

III. Organisation

Si l'on souhaite découper le projet en différentes parties on pourrait discerner 3 taches majeures :

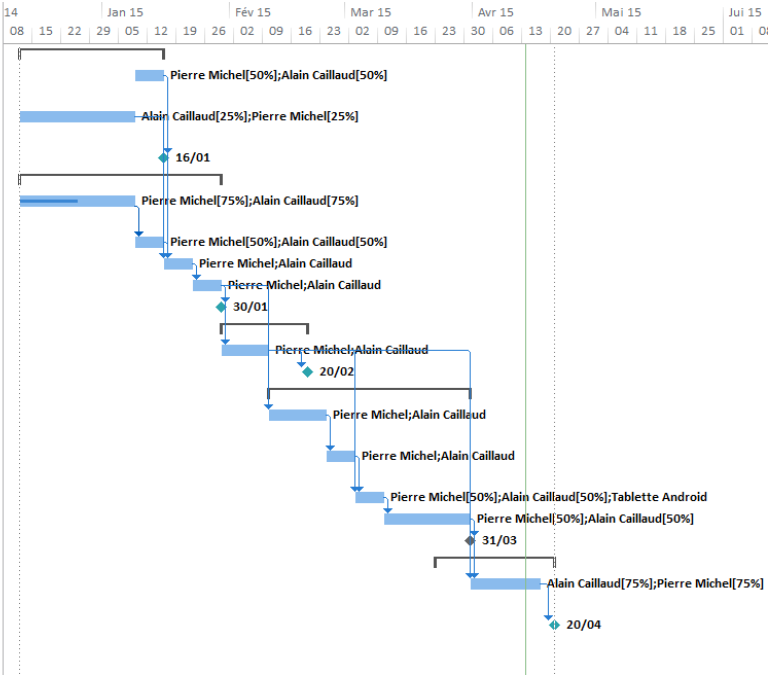
- La partie étude : Une bonne partie de notre travail au début du projet fut consacré à rechercher beaucoup d'informations. Nous devions en effet nous informer sur le jeu lui-même et sur les différentes applications "concurrentes" de la nôtre. Nous avons

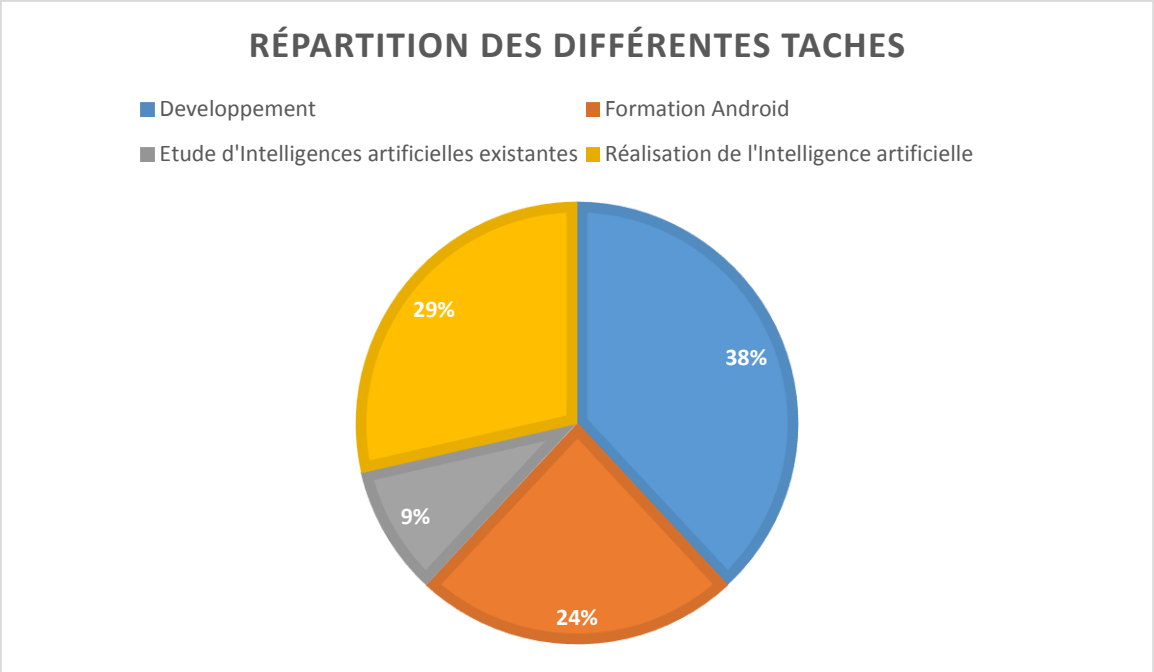
ensuite réalisé un cahier des charges. Enfin, nous avons passé du temps à apprendre comment programmer sur Android. Bien que nous ayons en parallèle des cours de Java, aucun de nous deux n'avait eu l'occasion de programmer sur Android, ce fut pour nous l'occasion d'apprendre à programmer sur ce système d'exploitation.

- La partie réalisation du jeu : Une fois que nous avions notre cahier des charges et découvert les bases de la programmation Android, nous avons commencé à programmer l'application.
- L'intelligence artificielle : Une fois que l'application était un peu avancée et qu'il était possible de jouer, nous avons en parallèle commencé à réaliser l'Intelligence artificielle censé résoudre une situation de jeu. Au fur et à mesure qu'elle avançait, nous l'intégrions à l'application du jeu.

Le diagramme de Gant suivant représente l'historique de nos travaux :

Nom de la tâche	Durée	Début	Fin
Elaboration des besoins	**25 jours**	**Ven 12/12/14**	**Ven 16/01/15**
Recherche Propriété intellectuelle	1 sm	Ven 09/01/15	Jeu 15/01/15
Elaboration cahier des charges	4 sm	Ven 12/12/14	Jeu 08/01/15
Jalon 1	0 jour	Ven 16/01/15	Ven 16/01/15
Etude de faisabilité	**35 jours**	**Ven 12/12/14**	**Ven 30/01/15**
Formation Programmation Android	4 sm	Ven 12/12/14	Jeu 08/01/15
Analyse des risques	1 sm	Ven 09/01/15	Jeu 15/01/15
Elaboration du gameplay	1 sm	Ven 16/01/15	Jeu 22/01/15
Modélisation du projet	1 sm	Ven 23/01/15	Jeu 29/01/15
Jalon 2	0 jour	Ven 30/01/15	Ven 30/01/15
Realisation du jeu uniquement	**15 jours**	**Ven 30/01/15**	**Ven 20/02/15**
Développement du jeu	1,5 sm	Ven 30/01/15	Mar 10/02/15
Jalon 3	0 jour	Ven 20/02/15	Ven 20/02/15
Ajout intelligence artificielle	**35 jours**	**Mar 10/02/15**	**Mar 31/03/15**
étude de l'intelligence Artificielle	2 sm	Mar 10/02/15	Mar 24/02/15
élaboration de l'Intelligence Artificielle	1 sm	Mar 24/02/15	Mar 03/03/15
Test de l'application	1 sm	Mar 03/03/15	Mar 10/03/15
Optimisation	3 sm	Mar 10/03/15	Mar 31/03/15
Jalon 4	0 jour	Mar 31/03/15	Mar 31/03/15
Documentation et livraison	**21 jours**	**Lun 23/03/15**	**Lun 20/04/15**
Réalisation du rapport de projet	2,67 sm	Mar 31/03/15	Ven 17/04/15
Jalon 5	0 jour	Lun 20/04/15	Lun 20/04/15





IV. Développement du jeu

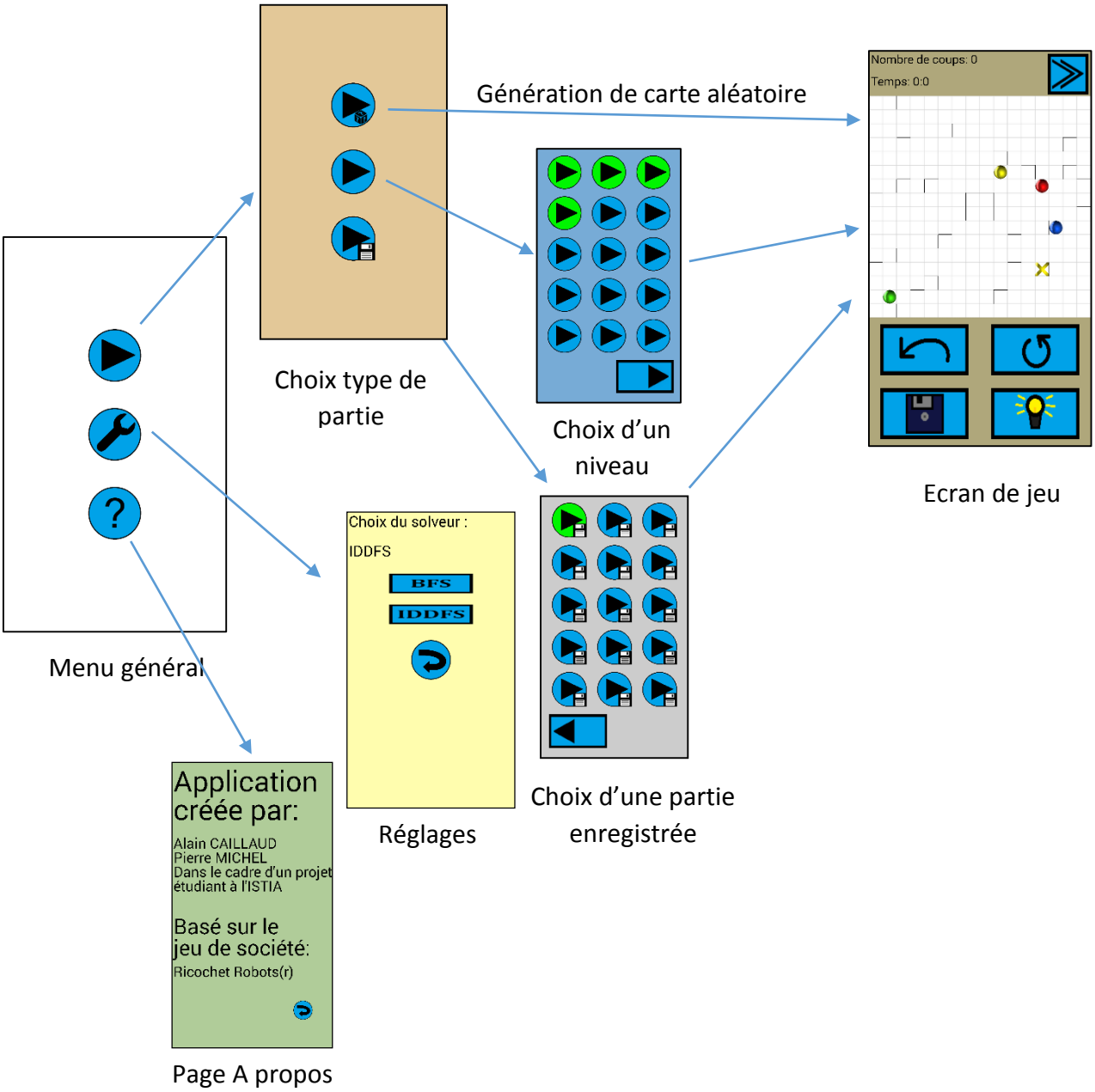
IV.1 L'application réalisée

L'application que nous avons réalisée permet à un possesseur de tablette ou smartphone Android de jouer au jeu Ricochet Robots.

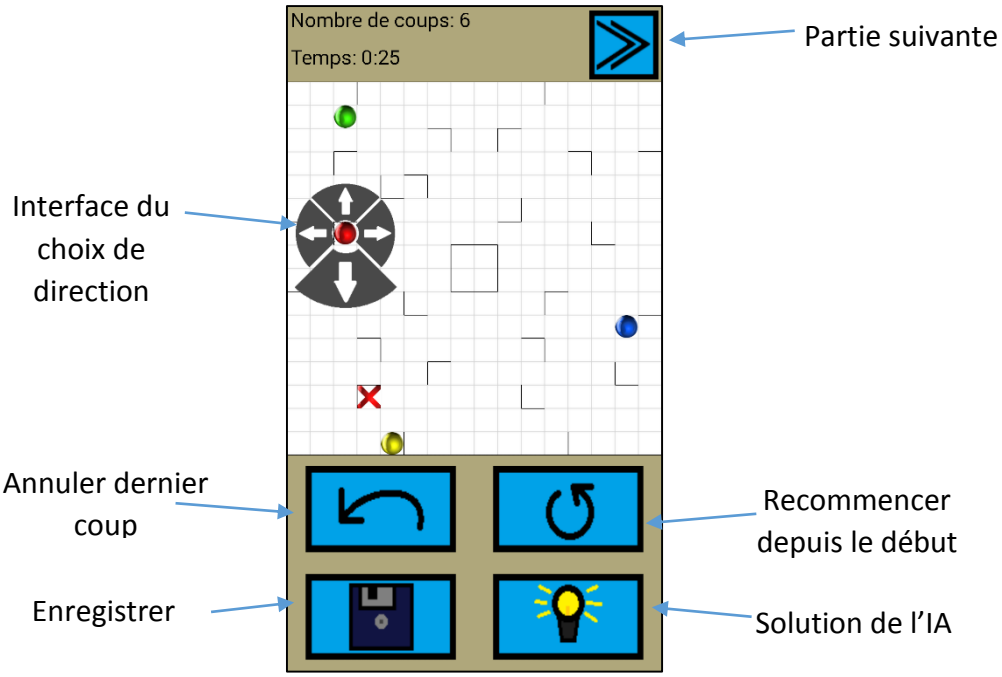
Lors du lancement d'une partie, l'utilisateur dispose de plusieurs choix, il peut :

- Générer une carte aléatoire : Dans ce cas, l'application génère une carte aléatoire pour la partie ;
- Choisir un niveau : Différentes cartes sont déjà fournies avec l'application, l'utilisateur peut choisir de jouer sur une de ces cartes.
- Choisir une carte enregistrée : L'utilisateur peut enregistrer une carte, il peut ainsi la rejouer dessus plus tard.

Le schéma suivant présente les choix possible et les différents écrans de l'application :



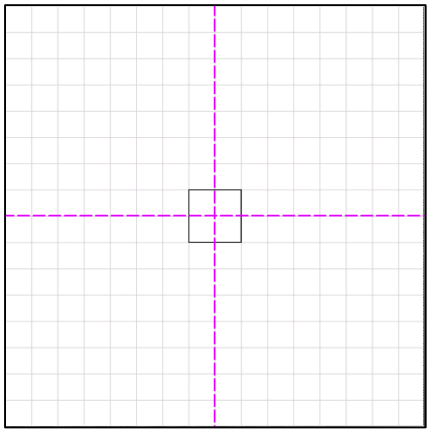
Peu importe le choix de l'utilisateur au lancement de la partie, il arrive à un écran de jeu. En restant appuyé sur un robot, le joueur fait apparaitre une interface permettant le choix de la direction. Une fois la direction choisie, le robot se déplace jusqu'à rencontrer un obstacle. Un compteur de temps et de coups est affiché ainsi que différents boutons qui permettent d'enregistrer la carte, de la change, d'annuler un ou tous les coups ou encore de montrer la solution de l'Intelligence artificielle. La capture d'écran suivante représente ces différents aspects :



Ecran de jeu

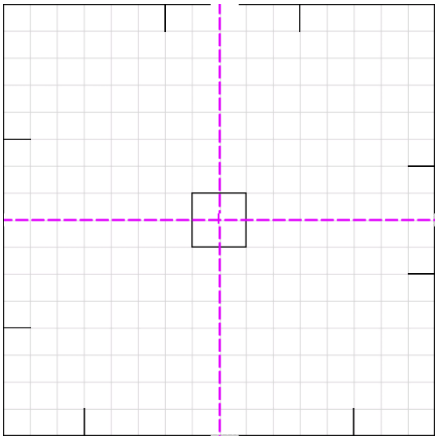
IV.2 Génération de grilles

Concernant les grilles du jeu, on a choisi de réaliser un générateur de grilles aléatoires. Ainsi, on avait beaucoup plus de possibilités de grilles que le jeu original et cela permet ainsi une durée de vie bien plus longue. On souhaitait cependant que nos grilles soient conformes aux grilles du jeu concernant le nombre de murs et leurs répartitions. Ci-dessous, voici comment nous procédons pour générer des grilles aléatoires :

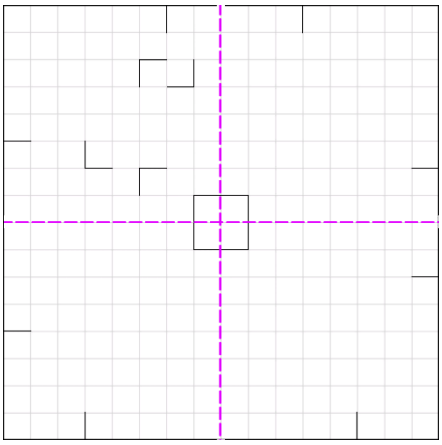


1- Tout d'abord, on place des murs tout autour de la grille, ainsi que les murs qui forment le carré du milieu.

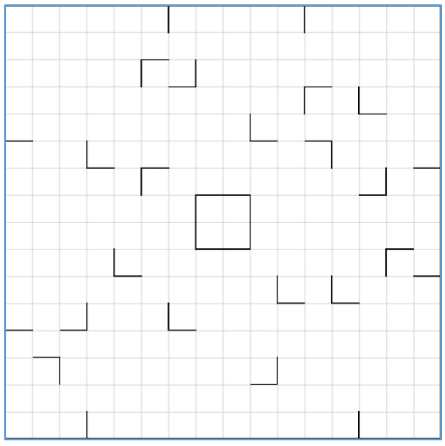
   Pour la suite de la génération de la carte, nous allons travailler sur chaque quart séparément (traits violet).



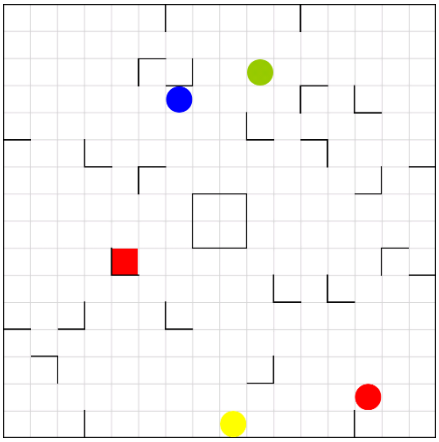
2- Sur chaque quart, on place deux murs extérieurs, un coté vertical et un coté horizontal. On utilise une fonction permettant de générer des nombres pseudo-aléatoires afin d'obtenir un placement aléatoire pour ces murs.



3- On ajoute ensuite les murs intérieurs. Sur chaque quart, on place 4 « angles » constitués de de deux murs chacun. Leur placement est aléatoire, cependant il est fait en sorte qu'un angle ne puisse ni en toucher un autre, ni toucher un mur extérieur.



4- Une fois que nous avons placé 2 murs extérieurs et 4 « angles » par quart, on ajoute un nouvel « angle » de deux murs placé dans un des quarts choisi aléatoirement. De cette manière, on obtient une carte contenant 17 « angles » et 8 murs extérieurs comme sur les grilles du jeu original.



5- Enfin, il nous reste plus qu'à placer les robots, là encore de manière aléatoire. Le choix de la cible (rouge, vert, bleu, jaune ou multi couleur) est fait aléatoirement, son placement est également aléatoire mais forcément dans un angle de deux murs.

Dans chaque quart de grille, on a 7 positions possibles pour chacun des deux murs extérieurs et 35 pour chacun des 4 « angles », on a 252 positions différentes pour les robots et 37 pour les cibles, on peut donc estimer l'ordre de grandeur du nombre de grilles ainsi :

$$\left(7 * 7 * (35 * 4) * (34 * 4) * (33 * 4) * (32 * 4)\right)^4 * 252 * 251 * 250 * 249 * 37 = 9 * 10^{51}$$

Cette estimation nous mène à un résultat de 9 * 10^{51}, à titre de comparaison, le nombre d'atomes sur terre est de l'ordre de 10^{50}. On a donc un choix de grilles extrêmement vaste grâce au générateur.

IV.3 Enregistrements

Notre application permet d'enregistrer des cartes pour pouvoir les lire plus tard. L'utilisateur peut ainsi enregistrer une carte qu'il aurait générée aléatoirement afin d'y jouer de nouveau plus tard. Pour cela, nous avons choisi de les enregistrer dans un simple fichier texte. Cela nous permettait de facilement lire ou modifier une carte avec un simple éditeur de texte sur un ordinateur. Sur Android, ces fichiers sont stockés dans un espace qu'Android met à disposition pour l'application. Cet espace étant privé à l'application, l'utilisateur ne risque pas de modifier ou supprimer involontairement un des fichiers.

Pour chaque carte on enregistre un fichier avec des positions. Chaque position est constituée d'une abréviation désignant l'objet (mur, robot, cible) suivi de ses coordonnées. Exemple :

mh1,13;	mur horizontal de coordonnées x = 1, y = 13
mv5,2;	mur vertical de coordonnées x = 5, y = 2
rb6,3;	robot bleu de coordonnées x = 6, y = 3
cr4,15;	cible rouge de coordonnées x = 4, y = 15

Les cartes de niveaux sont également stockées dans des fichiers de ce genre. La principale différence avec les enregistrements de l'utilisateur est qu'elles sont fournies avec l'application et ne peuvent pas être modifiées par l'utilisateur.

V. L'intelligence Artificielle

Une fois le développement du jeu Android bien amorcé, nous avons débuté à travailler en parallèle sur l'IA (Intelligence Artificielle) du jeu. Le but de cette IA est de trouver en un minimum de temps une solution optimale à une situation dans Ricochet Robots. C'est-à-dire qu'à partir de la disposition du plateau de jeu et de la position de chaque robot, l'IA doit fournir en un temps raisonnable une liste de coups à effectuer afin de mener le robot principal à son objectif. La solution devant être optimale, la liste fournie doit donc être la plus petite possible. Afin d'être compatible avec le jeu, cette IA devait donc être codée en Java.

V.1 Etudes de recherches antérieures

Afin de ne pas foncer la tête baissée dans le problème, nous nous sommes réservés un certain temps afin de nous renseigner sur des travaux réalisés dans le passé. Parmi tous les résultats étudiés, trois études (disponibles dans la bibliographie) sont ressorties comme étant les plus pertinentes.

Le premier document est une étude menée en Septembre 2005 par Nicolas Butko, Katharina A. Lehmann, et Veronica Ramenzoni. Cette étude se concentre plus particulièrement sur les méthodes employées par les humains afin de trouver une solution au jeu. Bien que ne présentant pas de solution technique à la résolution de problèmes, l'étude fut néanmoins utile dans l'optimisation de notre IA.

Le second document est une étude plus récente menée en Novembre 2012 par Michael Fogleman. Cette étude apporte tout d'abord la présentation du résonnement employé dans le développement d'un solveur de Ricochet Robots. De plus, cette étude nous fournit le code source un solveur codé en langage C. Cette étude nous a essentiellement servie de base pour l'algorithme de notre solveur codé en Java.

Le dernier des trois documents, créé en 2014 par Michael Henke est un solveur codé en Java basé sur celui de Michael Fogleman. Ce solveur nous a été utile afin d'avoir une idée de l'efficacité que pourrait avoir notre solveur.

V.2 BFS en général

Afin de résoudre un problème, nous avons employé un algorithme de parcours en largeur (*en anglais BFS = Breadth First Search*). Cet algorithme est généralement employé dans le parcours de graphes. Le but d'un BFS est de parcourir un graphe étage par étage. Par

opposition, il existe également un algorithme nommé DFS (*Depth First Search*) qui consiste en explorer le plus loin possible dans une voie avant de revenir en arrière. Sachant que le but de l'IA est de trouver une solution au jeu en un minimum de coups, l'algorithme BFS ressort immédiatement comme étant plus judicieux. En effet, un BFS va examiner tous les coups possibles pour un nombre *n* de coups avant d'examiner la partie pour *n+1* coups.

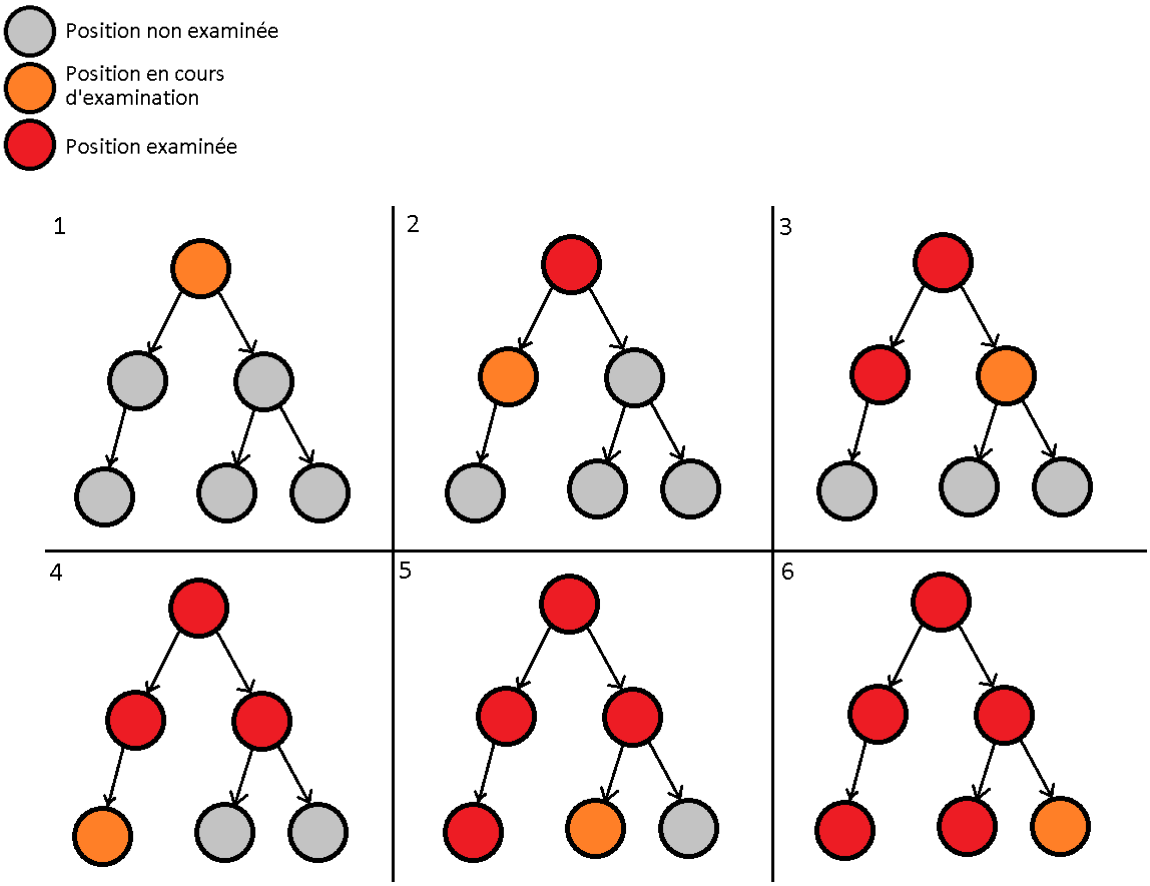


Figure 1: *Fonctionnement du BFS*

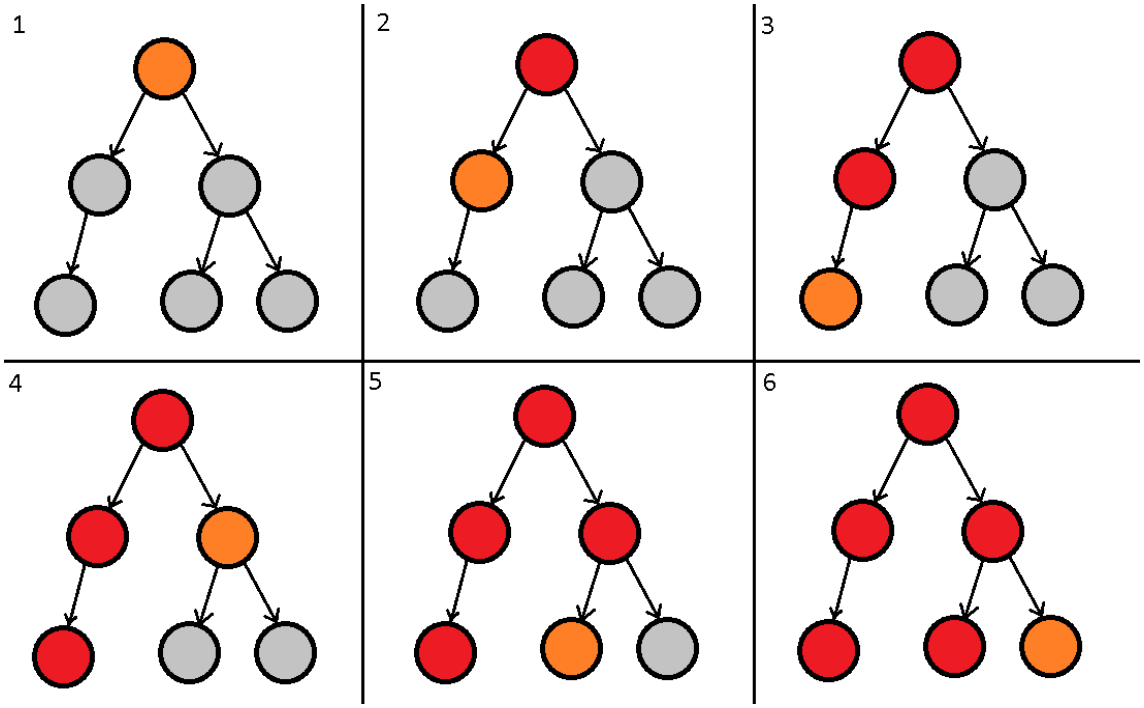


Figure 2: *Fonctionnement du DFS*

Une recherche par BFS est généralement employée afin de trouver la distance la plus petite entre deux points. Par exemple, cet algorithme peut être employé par un GPS afin de trouver le chemin entre deux villes se faisant en un minimum d'étapes.

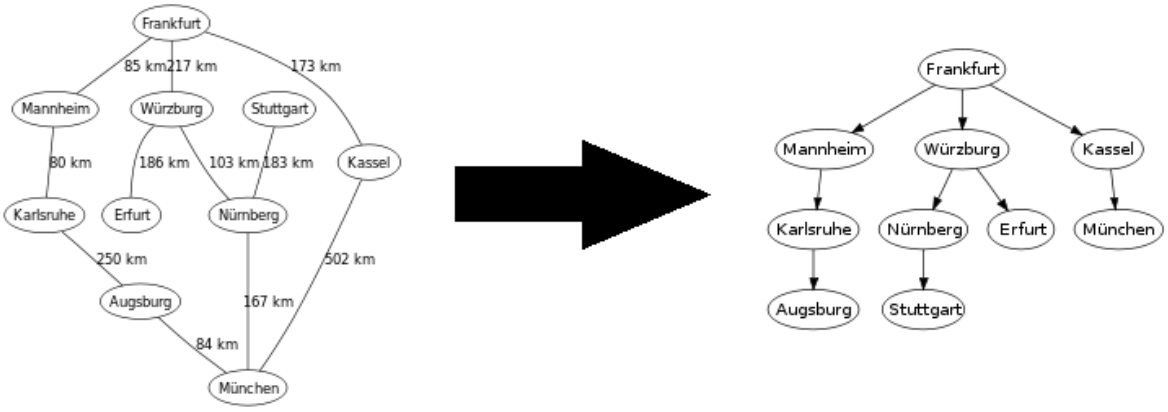


Figure 3: *Application du BFS à la recherche de chemin le plus court*

Une recherche par DFS est utilisée lorsque l'on souhaite trouver une solution le plus rapidement possible. Cette solution, en revanche, n'est pas optimale. C'est-à-dire qu'il existe certainement une meilleure solution. Cet exemple peut donc être utilisé pour trouver la sortie d'un labyrinthe. En effet, dans un labyrinthe, l'objectif est généralement de sortir le plus vite

possible. De façon analogue, il existe des algorithmes permettant de générer des labyrinthes basés sur la recherche DFS.

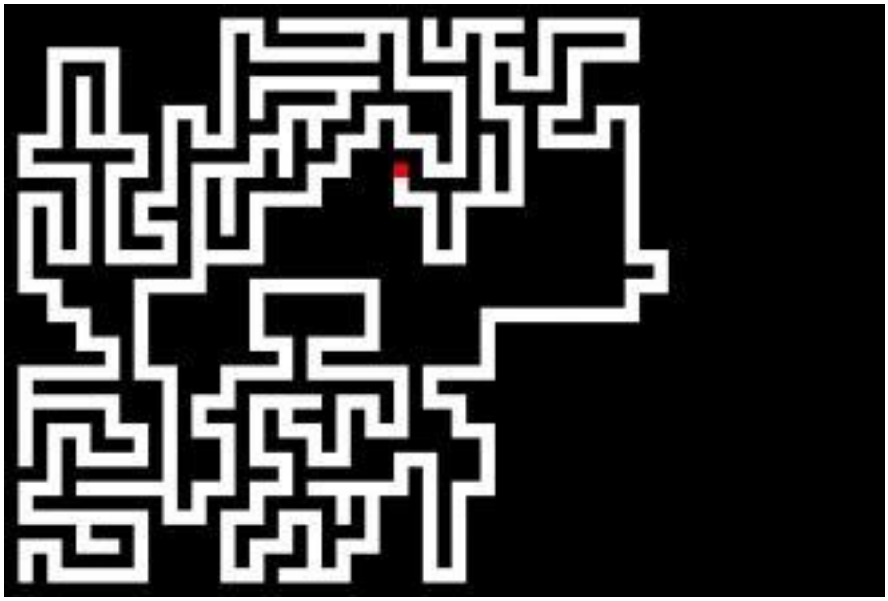


Figure 4: *Application du DFS à la génération de labyrinthes*

Il existe un algorithme nommé IDDFS (*Iterative Deepening DFS*) qui permet de faire un compromis entre vitesse et obtention d'une solution optimale. Le but de cette recherche est d'effectuer plusieurs DFSs successifs en se fixant une profondeur maximale au-delà de laquelle on arrête le DFS. À chaque DFS, on augmente cette profondeur maximale. Par conséquent, l'algorithme est capable de trouver une solution optimale tout en restant rapide. En revanche, cette méthode est excessivement gourmande en mémoire. Cette méthode est employée par certains algorithmes de recherche de fichiers.

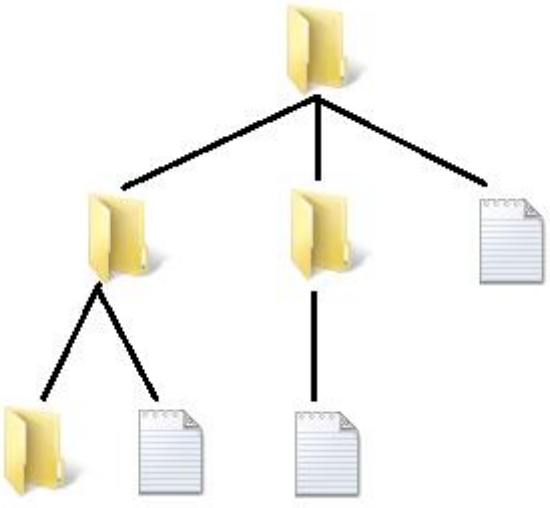


Figure 5: *Application de l'IDDFS pour la recherche de fichiers*

V.3 Application du BFS au jeu Ricochet Robots

Dans le cas de Ricochet Robots, les *nœuds* du graphe que nous allons parcourir représentent les différents états que peut prendre le jeu au cours d'une partie et les arêtes représentent les coups qui peuvent être effectués. La *racine* (nœud à l'origine de tous les autres) est donc l'état initial de la partie. C'est-à-dire la position de tous les pions au début de la partie. En appliquant un BFS, l'IA va ainsi tester toutes les solutions en 1 coup, puis toutes celles en 2 coups, etc. En théorie, l'IA est donc sûre de de trouver la solution optimale à condition d'avoir le temps de chercher et d'avoir suffisamment de mémoire.

Le problème de cette méthode, si elle n'est pas optimisée, est que l'IA prendra beaucoup trop de temps à résoudre le problème. En appliquant l'IA sans optimisations, résoudre un problème de 10 coups peut déjà prendre quelques minutes. D'après l'étude de Michael Fogleman, la majorité des problèmes (près de 90%) peuvent être résolus en 10 coups. Les situations les plus complexes dans Ricochet Robots peuvent être résolues un 22 coups minimum. Pour un tel problème l'IA mettrais des milliards d'années avant de trouver la solution.

V.4 Les optimisations

L'objectif de ce projet étant de programmer une application pour Android en Java capable de résoudre une partie de Ricochet Robots, nous sommes ainsi obligés d'optimiser l'IA afin qu'elle puisse produire un résultat en un temps satisfaisant. L'application de Michael Fogleman en C est capable de résoudre un problème de 22 coups en près de 3 secondes. Son adaptation en Java peut résoudre ce même problème en environ 25 secondes. Sachant qu'un smartphone est bien moins puissant qu'un ordinateur de bureau, nous pouvons donc nous attendre à obtenir une IA capable de résoudre ce problème en quelques minutes.

Dans le jeu Ricochet Robots, il existe 16 coups différents. En effet, chacun des quatre pions peut se déplacer en haut, en bas, à gauche et à droite. Par conséquent, appliquer une recherche revient à exécuter chacun de ces 16 coups dans des ordres différents jusqu'à trouver la solution. Pour un problème qui peut être résolu en x coups, il faut examiner au maximum 16^{x-1} positions. Ainsi, pour résoudre un problème de 10 coups, il faut chercher dans 68,719,476,736 positions et résoudre un problème en 22 coups revient à visiter 19,342,813,113,834,066,795,298,816 positions. Voyant la complexité de nature exponentielle, il devient évident qu'il est nécessaire d'éliminer des cas inutiles.

- *Eliminations des doublons*

Au cours d'une partie, il est possible que les pions se trouvent tous aux mêmes positions. Le cas typique illustrant cette situation est lorsque l'on souhaite déplacer un pion dans une direction alors qu'il y a un mur juste à côté du pion. Suite à ce coup, le pion n'aura pas bougé. Par conséquent, l'IA aura traité la situation initiale et tenté ce coup. Ensuite, l'IA va traiter tous les états qui dérivent de la situation initiale même si le pion n'a pas bougé. La même situation est donc traitée deux fois.

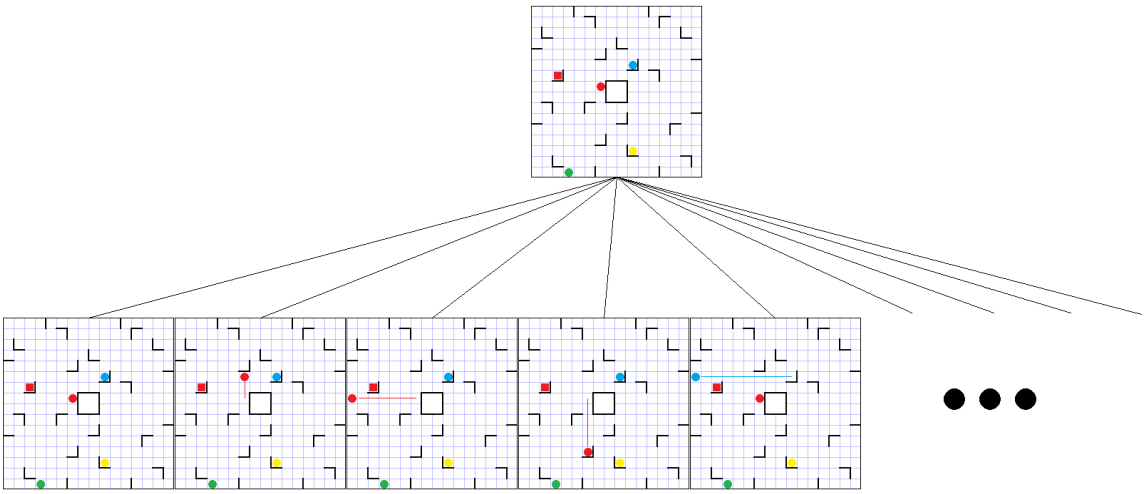


Figure 6: *États dérivés d'une situation initiale*

La première optimisation que nous avons donc mise en place est d'éliminer les cas déjà traités. C'est-à-dire que lorsqu'un cas est traité, l'IA va mémoriser la position de tous les pions. Ainsi, si jamais cette configuration revient plus basse dans la recherche, l'IA saura qu'il est inutile de la traiter de nouveau. En pratique, il est possible de condenser les positions de tous les pions dans un entier de 32 bits. En effet, les coordonnées x et y des pions, variant de 0 à 15, peuvent êtres encodés sur un entier de 4 bits. Ainsi, un pion peut tout simplement être représenté par un entier d'un octet. Il y a 4 pions, la situation dans laquelle se trouve la partie peut alors être réduite à 4*8 = 32 bits. Ainsi, dès qu'une situation est examinée, l'entier 32 bits est mémorisé par l'IA. Si cet entier revient plus tard, l'IA saura qu'il est inutile d'aller plus loin dans cette direction.

De plus, il est possible de faire une distinction entre les pions "principaux" et les pions "secondaires". Les pions principaux sont les pions qui doivent aller vers l'objectif et les pions secondaires sont tous les autres. Par exemple, si l'objectif est rouge, le pion principal sera le pion rouge et les pions secondaires seront les autres. Si l'objectif est multicolore, tous les pions sont principaux (il n'y a en a donc aucun de secondaire). Cette distinction se fait car échanger deux pions principaux ou deux pions secondaires n'a aucune conséquence car n'importe quel des pions principaux peut aller à la sortie et n'importe quel pion secondaire peut aider. En

pratique, les pions principaux ainsi que les pions secondaires sont triés selon leurs coordonnées x et y et ensuite placés sur l'entier 32 bit dans l'ordre de tri.

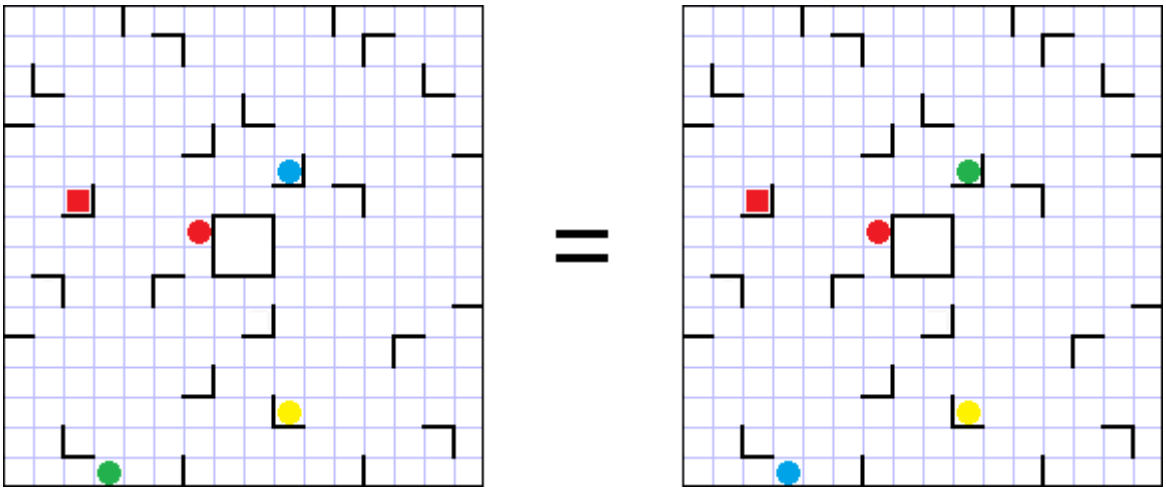


Figure 7: *Illustration des états équivalents*

En appliquant cette optimisation nous pouvons estimer que pour chaque situation, environ 9 situations peuvent en dériver. En moyenne, 7 coups mènent vers une situation qui a déjà été explorée. Ainsi, résoudre un problème de x coups revient à examiner au minimum 9^{x-1} situations. Par conséquent, un problème de 10 coups peut maintenant se faire en explorant 387,420,489 positions; soit 177 fois moins que sans optimisations. Et résoudre un problème de 22 coups revient à scruter 109,418,989,131,512,359,209 positions; soit 176,777 fois moins. Grâce à cette optimisation, nous pouvons estimer que résoudre un problème de 10 coups prendrait quelques minutes. Résoudre un problème de 22 coups durerait certainement quelques années; ce qui est bien trop long dans le contexte d'une application Android.

- *Elimination des coups contre-productifs*

Après avoir éliminé les doublons, on cherche à éliminer les coups contre-productifs. Ce que nous appelons des coups contre-productifs, sont des coups dont l'exécution ne conduirait pas à une meilleure situation de jeu. Exemple simple, supposons que la cible soit rouge et que le robot rouge soit sur la même ligne sans aucun obstacle entre eux, il serait maladroit d'essayer de le déplacer de manière verticale, cela conduirait le robot à s'éloigner de la cible. On cherche donc à éviter tous les coups qui éloigneraient le robot de sa cible. Pour cela, on établit une carte que nous appellerons carte des distances minimales. Sur cette carte, on associe pour chaque case un chiffre. Ce chiffre indique que, si tous les pions secondaires sont bien placés, le nombre de minimum de coups pour aller vers l'objectif sera égal à ce chiffre. La case sur laquelle se situe la cible à une distance de 0. Ensuite, on associe à toutes les cases qui sont sur la même ligne et même colonne que cette case 0 sans aucun obstacle entre elles la valeur 1. Ensuite, on associe à toutes les cases qui ont une case de valeur 1 sur leur ligne ou

colonne et pas d'obstacles entre les deux la valeur 2. On continue ainsi jusqu'à remplir la carte. Le graphisme suivant donne une représentation d'une carte de distance minimale :

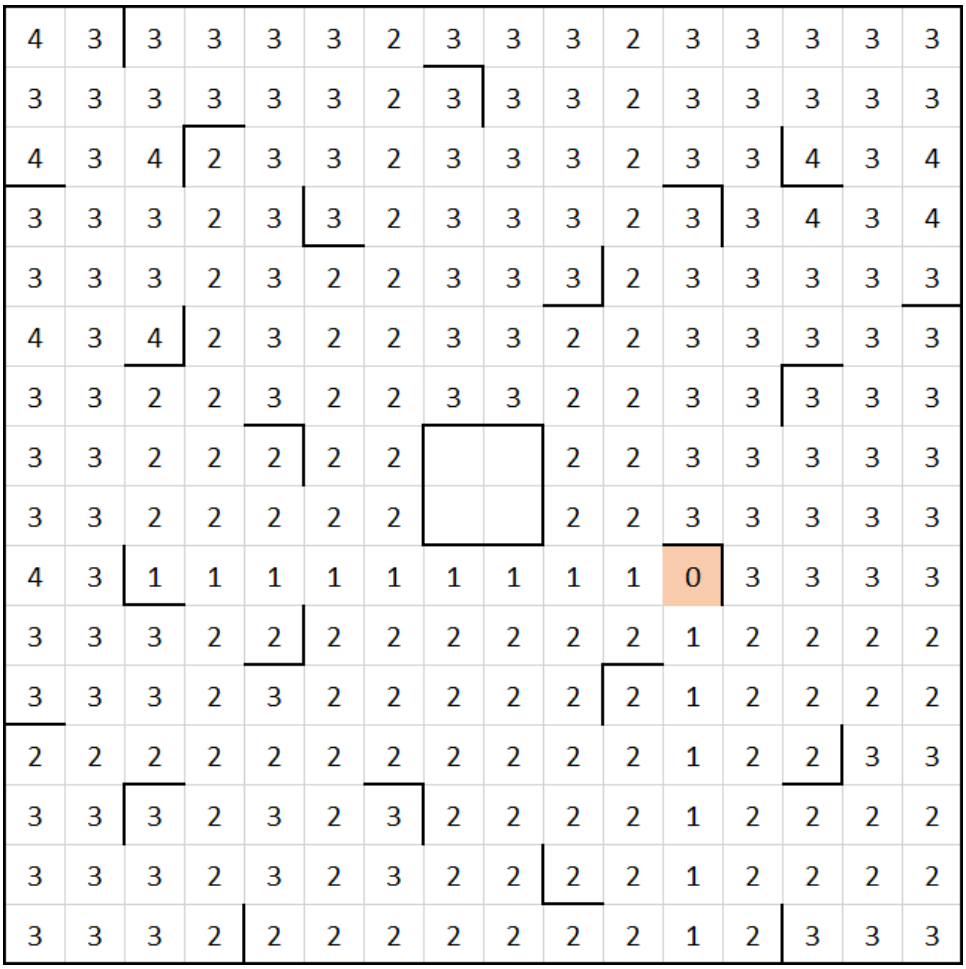


Figure 8: *Grille de jeu avec les distances minimales*

Ensuite, lors de la réalisation de la recherche, on veille à ce que chaque déplacement du robot principal (celui qui doit aller sur la cible) ne l'éloigne pas de la cible. Exemple, si la cible est rouge et que le robot rouge est sur une case marquée de distance 2, faire un mouvement qui l'emmènerait sur une case marquée de distance 3 serait contre-productif car on éloignerait le robot de sa cible, inutile dans ce cas de continuer cette branche du BFS. En appliquant cette seconde optimisation, cela nous permet d'éliminer une partie des branches du BFS et ainsi de gagner en temps d'exécution et de mémoire utilisée.

Grâce à cette optimisation, nous pouvons estimer le nombre de positions à examiner à environ 8^{x-1} pour une solution de x coups. Ainsi, un problème de 10 coups nécessite 134,217,728 examinations et un problème de 22 coups requiert 9,223,372,036,854,775,808 examinations.

Résoudre un problème un problème en 10 coups peut maintenant prendre environ 1 minute.

- *Optimisation du code*

Comme mentionné précédemment, le langage communément employé pour programmer des applications pour Android est le Java. Or, le Java est un langage orienté-objet passant par une machine virtuelle afin de permettre une plus grande flexibilité. En revanche, tout code produit en Java sera inévitablement moins performant. De plus, en raison de sa grande flexibilité, il est facilement possible de créer du code peu efficace. C'est pour cela qu'il est capital de respecter minutieusement certaines règles liées au langage.

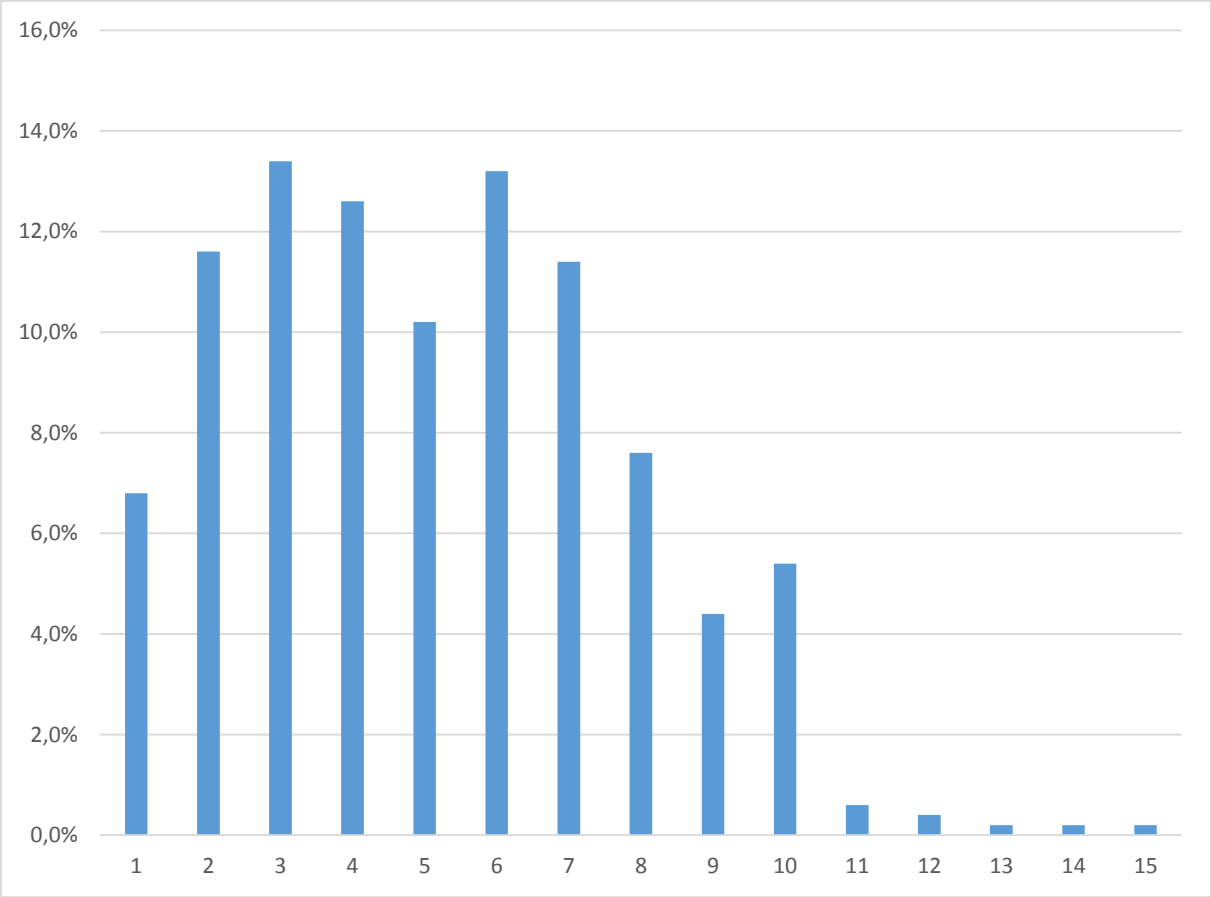
La plus grosse contrainte liée au Java est qu'il faut limiter l'allocation de nouveaux objets. C'est-à-dire que le Java connait des pertes de performances lorsqu'il y a trop d'opérations sur la mémoire. En effet, allouer régulièrement de gros espaces de mémoire peut s'avérer particulièrement problématique pour un support tel qu'un smartphone. Afin de remédier à ce problème, nous avons pris deux mesures. Tout d'abord, afin de limiter la cadence des allocations en mémoire, nous avons fixé dès le début une quantité de mémoire maximale que pourra emprunter l'IA. Ainsi, avoir ce bloc de données directement sous la main permet de ne pas avoir à constamment demander plus de mémoire. Ensuite, nous avons limité les quantités de données allouées à chaque fois. En effet, lors d'une partie, la disposition de la grille et des objectifs restent fixe. Les données à mémoriser ne sont que les positions de tous les pions.

En appliquant ces optimisations nous avons constaté que pour un problème solvable en 10 coups, la durée de la recherche a été divisée par 4. En revanche, en raison de la nature fixe de la quantité de mémoire utilisée, l'IA peut être sujette à des problèmes si la solution nécessite un nombre de coups élevés.

V.5 Nos résultats / statistiques

Ces études sur l'intelligence artificielle nous ont permis d'établir des statistiques. On peut ainsi avoir des statistiques sur le nombre de coups minimum par exemple. Les résultats ont été obtenus à partir d'un échantillon de 500 grilles, toutes obtenues à l'aide du générateur que nous avons intégré à l'application. Nous avons passés chacune de ces cartes à l'aide de l'intelligence artificielle que nous avons réalisée. Afin que le temps d'exécution pour les 500 études soit correct, nous avons préféré faire les tests sur un ordinateur, plus performant qu'un mobile ou une tablette.

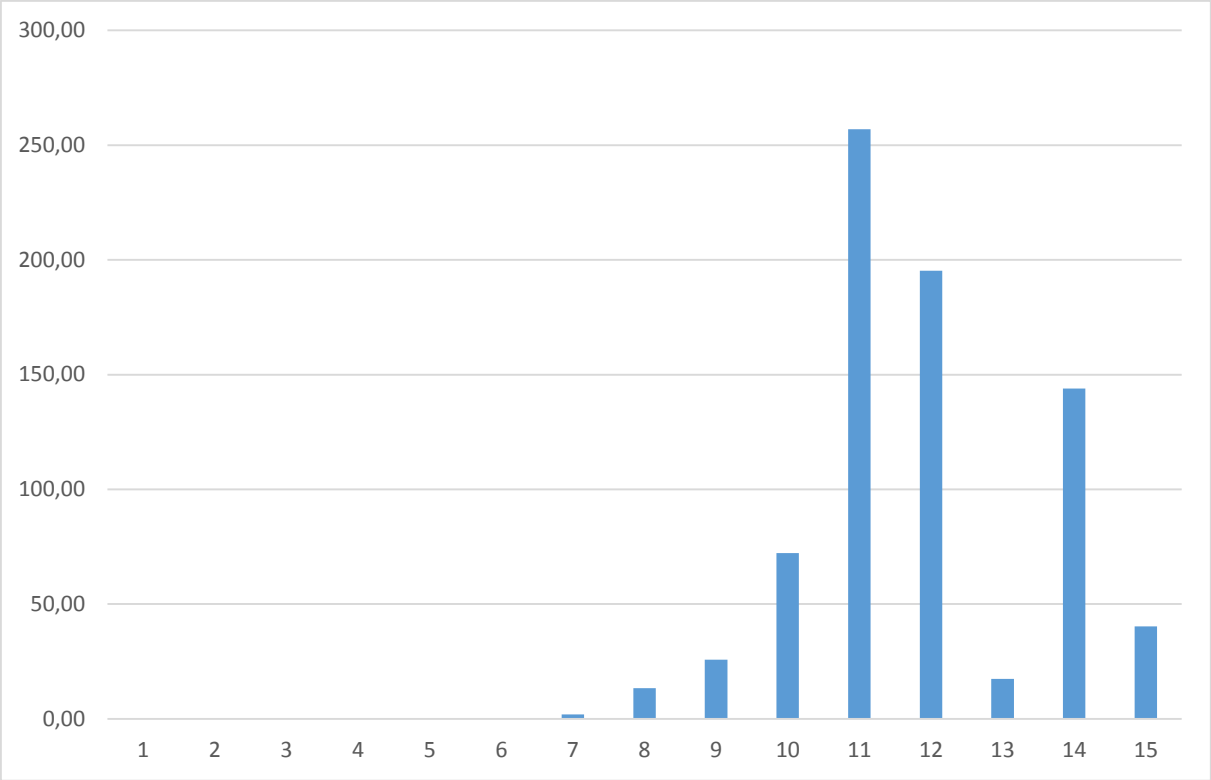
Sur les 500 grilles différentes, notre solveur a réussi à en résoudre 491, les 9 qui n'ont pas pu être résolue demandaient trop de mémoire vive pour leur résolution, cependant, elles représentent moins de 2 % des cartes que nous avons étudiées.



Répartition du nombre de coups minimum en %

On déduit ainsi du graphe que la médiane se situe à 5 coups, c'est-à-dire qu'il y a autant de cartes qui sont solvables en 5 coups ou moins que de cartes qui demandent au moins 6 coups.

On peut également s'intéresser au temps moyen de résolution des problèmes selon le nombre de coups minimum qu'ils demandent. A titre d'information, nous avons effectué le test suivant à l'aide d'un ordinateur équipé d'un processeur Intel Core2 Duo et de 4 Go de mémoire vive.



Durée moyenne de calcul d'une solution selon le nombre de coups minimum (en secondes)

On observe clairement que la majorité des grilles qui peuvent être résolues en moins de 8 coups le sont en un temps négligeable (quasiment invisible sur le graphe), on observe cependant aussi que l'augmentation du temps de calcul est exponentielle en fonction du nombre de coups minimum.

V.6 Les évolutions possibles

Il serait possible de faire évoluer notre application de différentes manières. Voici une liste non-exhaustive des évolutions que nous aurions pu apporter au projet :

- Réaliser un éditeur de carte : Un utilisateur de l'application n'a aucun moyen de créer sa propre carte. Pourtant il pourrait être intéressant pour un utilisateur de pouvoir jouer ou faire réfléchir l'IA sur une carte qu'il a lui-même réalisé. On pourrait imaginer le profil de quelqu'un qui possèderait le jeu de société originale et qui voudrait connaitre la solution la plus optimisée à sa situation de jeu. Il serait donc intéressant d'intégrer à l'application un éditeur de cartes.

- Réaliser un système de score : Il serait possible de faire évoluer l'application de manière à intégrer un système de score. Chaque joueur verrait son score augmenter en fonction de sa progression. On pourrait imaginer que le joueur gagerait plus ou

moins de points en fonction qu'il finit une partie facile (réalisable en quelques coups) ou plus compliqué (nécessitant par exemple plus de 8 coups…). A la manière de l'application déjà existante Ricochet Robot™, nous aurions pu intégrer un système de score avec un classement sur un serveur permettant aux joueurs de se comparer voir même de se défier.

- Réaliser une campagne de jeu complète : Actuellement, notre application permet déjà au joueur de jouer parmi des cartes déjà existantes. Nous avons un ensemble de cartes de différents niveaux et le joueur peut jouer sur ces cartes. Mais nous aurions pu pousser le concept plus loin et créer une campagne de jeu avec beaucoup plus de cartes mais aussi des niveaux. On pourrait ainsi faire des niveaux avec des cartes plus ou moins simples selon le niveau, exemple réalisable en moins de 3 coups pour le niveau 1, réalisable en 4 à 6 coups pour le niveau 2. Ensuite on pourrait ajouter des conditions de déblocages des niveaux, exemple avoir fini 75 % des cartes du niveau 1 pour débloquer le niveau 2…

- Amélioration du design : Le choix du graphisme est subjectif, chacun ayant des gouts différents, mais il serait possible d'adopter un graphisme plus complexe, avec pourquoi pas une grille en 3 dimensions, des animations quand l'utilisateur gagne…

- Permettre de jouer sur différents taille de cartes : Nous avons fait le choix d'utiliser uniquement des grilles de taille 16 * 16, mais une évolution possible serait de laisser à l'utilisateur le choix de la grille, il pourrait ainsi jouer sur une grille plus petite ou plus grande.

- Permettre de jouer à la version 2003 du jeu : Comme nous l'avons expliqué précédemment, il existe deux version du jeu, celle que nous avons choisi (1999) utilise que 4 robots, celle de 2003 en utilise 5 et a des murs plus complexes, une évolution possible serait de permettre au joueur de choisir quelle version du jeu utiliser.

VI. Conclusion

Pour conclure, nous pouvons tout d'abord dire que ce projet s'est révélé autant intéressant au niveau des connaissances qu'au niveau du défi. En effet, au début du projet, nous ne connaissions pas du tout la programmation Android et nous commencions tout juste à nous initier à la programmation en Java. Ainsi, ce projet nous a permis de nous former en Android et consolider nos capacités en Java. De plus, le projet étant centré autour du développement d'une intelligence artificielle pour un problème compliqué, nous avons été en mesure de mettre à l'épreuve nos capacités. Travailler en équipe a également été une expérience enrichissante car cela nous a permis de découvrir les contraintes que nous pouvons rencontrer lorsque nous ne travaillons pas seuls.

L'objectif du projet était de développer une application pour Android pour le jeu Ricochet Robots™ tout en implémentant une intelligence artificielle capable de résoudre des parties du jeu. Bien que nous ayons été capables de créer le jeu et faire marcher l'intelligence artificielle, ces deux aspects peuvent néanmoins être améliorés. En effet, le jeu connait certains ralentissements en raison du manque d'optimisation de certains morceaux du code. De plus, le côté esthétique du jeu est également à revoir. Nous avons passé beaucoup plus de temps à travailler sur le contenu de l'application que sur sa forme. Finalement, bien que notre intelligence artificielle parvienne à résoudre des problèmes de moins de 10 coups sans trop de difficulté et que nous ayons également su intégrer un solveur nettement plus efficace, notre intelligence artificielle peut encore être améliorée et optimisée.

Si ce projet devait être à refaire, nous adopterions dès le début du projet des normes de codage à respecter impérativement. En effet, il nous est arrivé bien trop souvent que l'application ne fonctionne pas de la façon désirée car, travaillant tous les deux sur le même code, certains éléments n'étaient pas implémentés de la même façon partout. Par exemple, à un moment, nous avions des conventions différentes pour exprimer la couleur des pions. De plus, nous adopterions également une organisation plus modulaire au projet. Cela nous permettrait de modifier des bouts de code sans pour autant casser tout le projet. Finalement, le développement de l'application serait bien plus rapide car, à présent, nous connaissons les contraintes du développement Android. Ainsi, nous aurions nettement plus de temps pour optimiser l'intelligence artificielle.

VII. Bibliographie

Règles du jeu Ricochet Robots [En ligne] (1999)
Disponible sur : http://maludo.chez.com/regles/RASEN.pdf

Michael Fogleman : Ricochet Robots Solver Algorithms [En Ligne] (Novembre 2012)
Disponible sur : https://speakerdeck.com/fogleman/ricochet-robots-solver-algorithms

Nicolas Butko, Katharina A. Lehmann, Veronica Ramenzoni : Ricochet Robots - A Case Study for Human Complex Problem Solving [En ligne] (15 Septembre 2005)
Disponible sur : http://www-pr.informatik.uni-tuebingen.de/mitarbeiter/katharinazweig/downloads/ButkoLehmannRamenzoni.pdf

Smack42 : DriftingDroids (Octobre 2014)
Disponible sur : https://github.com/smack42/DriftingDroids/wiki

Alain CAILLAUD, Pierre MICHEL : *Bouncing Spheres* (Mai 2015):
https://github.com/imefGames/ProjetEi4

Application Ricochet Robots

Projet réalisé par : CAILLAUD Alain, MICHEL Pierre

Projet encadré par : M.AUTRIQUE Laurent

Résumé : *Bouncing spheres* est une application Android permettant de jouer au jeu de société Ricochet Robots™ directement sur tablette ou smartphone. Dans ce jeu, les utilisateurs peuvent non seulement essayer de résoudre les problèmes mais également se défier contre une intelligence artificielle. Cette application fut créée dans le cadre d'un projet étudiant à l'ISTIA, école d'ingénieurs à Angers, sous l'encadrement de l'enseignant M.AUTRIQUE Laurent.

Ce projet étudiant a pour objectif de réaliser une application intégrant un solveur fonctionnant sur un système Android. Dans ce but, des compétences en programmation Java, Android mais également en algorithmiques de résolution de problème sont nécessaires.
Pour réussir à atteindre cet objectif, une formation sur la programmation Android fut nécessaire ainsi que beaucoup d'études concernant les différents moyens de résoudre une partie de Ricochet Robots™.

L'application est à présent terminée et nous espérons pouvoir la publier sur le Play Store de Google d'ici peu.

Mots-clés : Ricochet Robots™, Android, Java, Solveur, Parcours de graphe en largeur, Parcours de graphe en profondeur, Intelligence Artificielle

Summary : *Bouncing Spheres* is an Android application allowing you to play the game Ricochet Robots™ directly on your phone or tablet. In this game, users may attempt to solve puzzles but also see the puzzles be solved for them by an Artificial Intelligence right before their eyes. This application was created during a student project in ISTIA, an engineering school in Angers, under the supervision of Mr Laurent AUTRIQUE.

The objective of this student project was to create from scratch this application for Android. In order to attain this goal, skills in Java and Android programming were needed, but skills in algorithmics were also necessary in order to create the Artificial Intelligence.
Thus, studying those domains, in addition to researching various existing studies were compulsory in order to succeed in completing this project.

The application is now finished and we hope publishing it soon on Google's Play Store.

Keywords: Ricochet Robots™, Android, Java, Solver, Breadth First Search, Depth First Search, Artificial intelligence