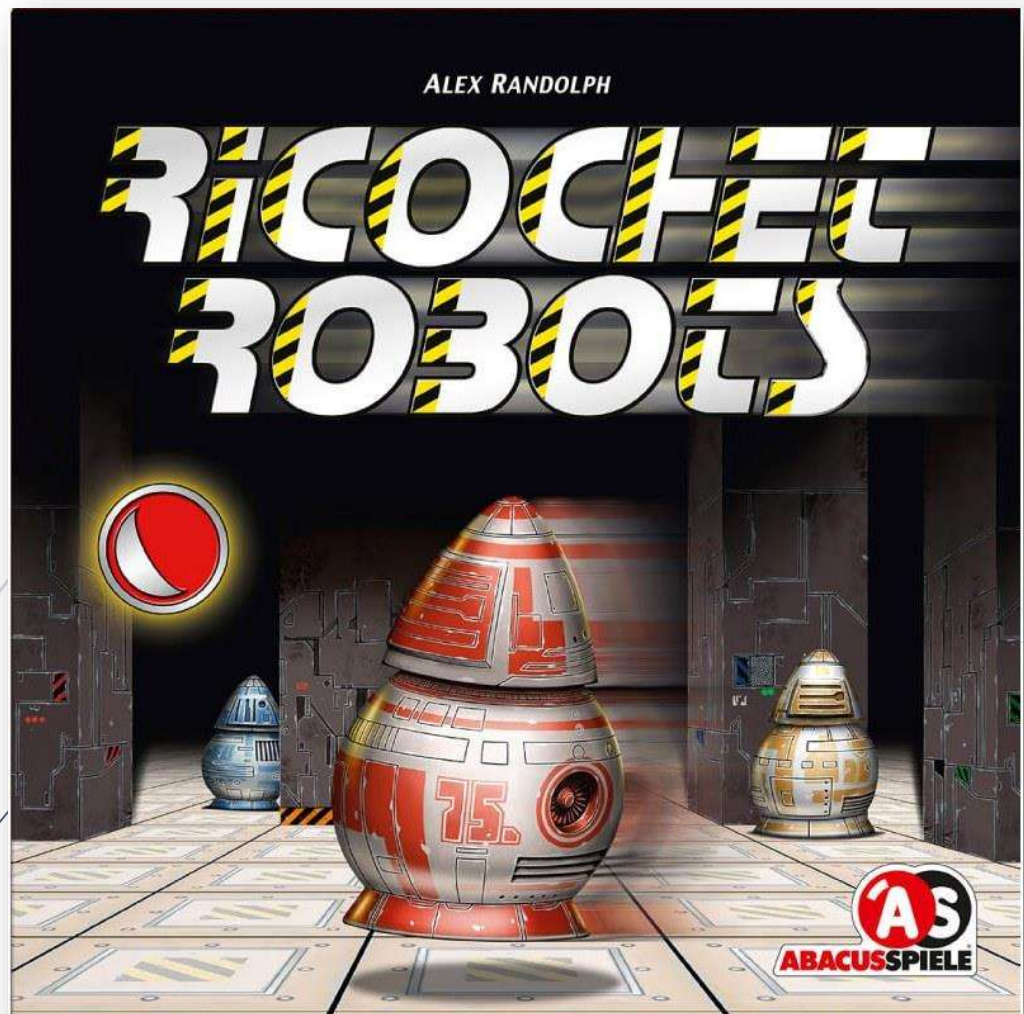


# Ricochet Robots

Résolution par l'intelligence artificielle



## Sommaire

|  |           |
|--|-----------|
| <i>I – Présentation du sujet .....</i>               | <b>2</b>  |
| <i>II – Spécification du problème .....</i>          | <b>3</b>  |
| Objectif.....  | 3         |
| Représentation.....                                  | 3         |
| <i>III – Analyse du problème .....</i>               | <b>5</b>  |
| Algorithmes basiques.....                            | 5         |
| Algorithme A* .....                                  | 5         |
| Recherche de l'heuristique .....                     | 5         |
| <i>IV - Exemples, analyses &amp; résultats .....</i> | <b>7</b>  |
| <i>V - Améliorations envisageables.....</i>          | <b>10</b> |
| <i>VI - Perspectives .....</i>                       | <b>11</b> |
| <i>VII - Annexe.....</i>                             | <b>12</b> |
| <i>Bibliographie .....</i>                           | <b>12</b> |

## I – Présentation du sujet

Le jeu Ricochet Robots, créé par Alex Randolph en 1999 est un jeu de société mêlant calcul et réflexion. Les règles du jeu peuvent être consultées ici:

[https://fr.wikipedia.org/wiki/Ricochet\\_Robots](https://fr.wikipedia.org/wiki/Ricochet_Robots).

L'objectif de notre projet est de créer une IA capable de trouver la solution comportant le minimum de coups possibles, tout en effectuant les calculs en un temps raisonnable. Pour ce faire, nous avons dans un premier temps créé une copie du jeu en langage Python, puis nous avons développé l'intelligence artificielle permettant de trouver les solutions. Le jeu est jouable soit par l'utilisateur, soit par l'intelligence artificielle.

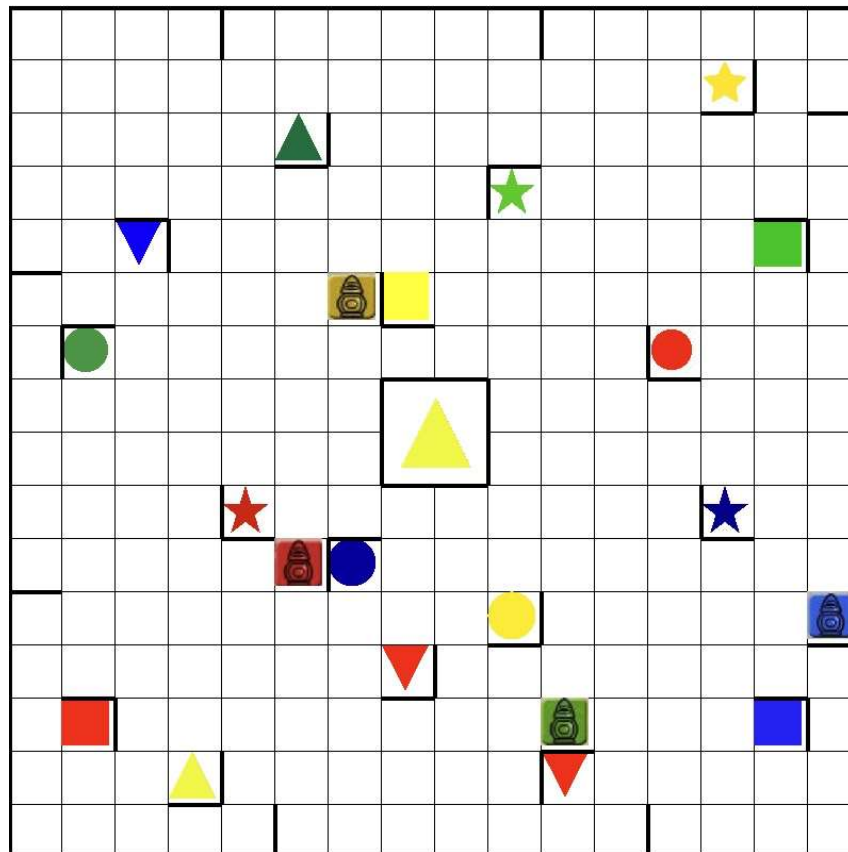


Figure 1 : Plateau de jeu

## II – Spécification du problème

### Objectif

L'objectif de notre programme est d'arriver à un état final (le robot qui se situe sur la case cible) en partant d'un état de départ (le robot qui se situe sur sa case d'apparition). On choisira parmi les chemins possibles celui qui utilise le moins de coups. L'état initial comporte donc les positions de tous les robots ainsi que celle de la cible, alors que l'état final répond à la seule condition du robot correspondant se situant sur la cible.

### Représentation

Nous avons décidé de décomposer les éléments du jeu en plusieurs objets (classes). Chaque état se définit donc par les valeurs attribuées à chaque objet.

#### Plateau

Le plateau de jeu est une matrice de 16x16 cases. La disposition des murs reste fixe, cependant la disposition des cibles et des robots est aléatoire.

#### Robots

La classe robot comporte seulement les positions du robot et sa couleur. Elle est surtout utile pour le déplacement de ceux-ci.

#### Mouvements

Pour déplacer un robot par l'utilisateur, il suffit de cliquer dessus pour le sélectionner puis d'utiliser les flèches directionnelles pour le faire bouger. Le mouvement se base donc sur la lecture des cases dans la direction choisie, afin d'ordonner au robot de s'arrêter lorsqu'un obstacle est rencontré (un mur ou un autre robot). Cette logique est aussi utilisée pour obtenir les nœuds voisins dans le processus de résolution du problème.

#### Cases

Chaque case possède donc une position x (ligne) et une position y (colonne), elle possède aussi un type entier indiquant ce qui se trouve sur cette case. 0 pour une case vide, 1 pour une case comportant un ou plusieurs murs, 2 pour un robot et 3 pour une cible. Enfin pour indiquer les différents murs on possède 4 booléens correspondant aux 4 présence possibles d'un mur.

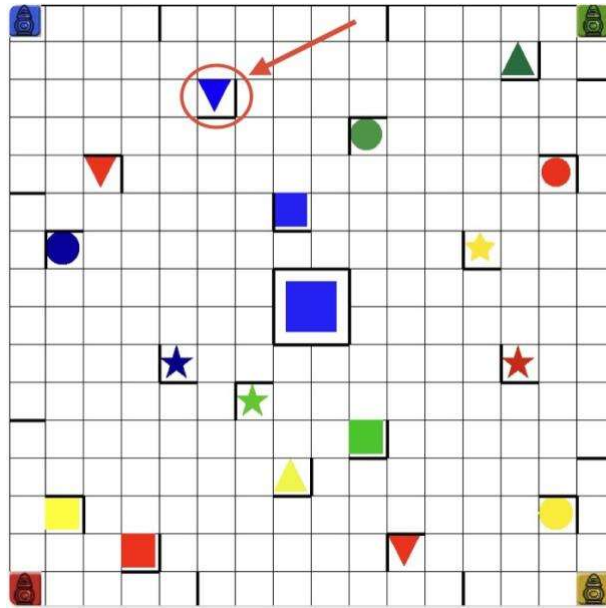


Figure 2 : Exemple de case

Prenons par exemple la case entourée. En partant de 0, ses coordonnées sont x=2 et y=5. Il s'agit d'une cible, son type est donc 3. Elle possède des murs à droite et en bas, de fait on a : top=false, bottom=true, left=false et right=true. La case située juste en dessous aura top=true.

## Cibles

Chaque cible possède aussi des positions x, y, une couleur et un type (triangle, carré, etc.). Au début de chaque partie, les cibles sont disposées de manière aléatoire sur le plateau, dans chaque case « coin ». Une cible principale est alors choisie aléatoirement aussi et affichée au centre du plateau.

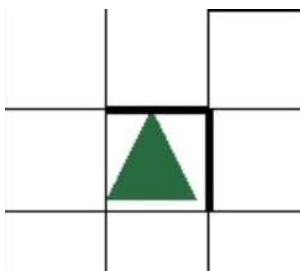


Figure 3 : Case d'apparition d'une cible

## III – Analyse du problème

### Algorithmes basiques

Dans un premier temps, nous avons utilisé des algorithmes de résolution tel que le Breadth First Search de se familiariser avec le problème. Rapidement, ces algorithmes ont montré leurs limites et ne sont donc pas utilisables dans la grande majorité des cas. Le code du BFS est consultable dans le fichier ai.py. et il trouve parfois des solutions efficaces.

### Algorithme A\*

Afin de trouver le plus court chemin, nous utilisons un algorithme A\* prenant en compte une heuristique prédéfinie ainsi que le nombre de coups permettant d'arriver à chaque position.

### Recherche de l'heuristique

Le plus important dans un algorithme A\* afin d'obtenir des solutions optimales, est de trouver une heuristique adaptée au problème prenant en compte les caractéristiques spécifiques de Ricochet Robots. La recherche de cette heuristique est donc une étape essentielle pour garantir la qualité de notre intelligence Artificielle.

Dans un premier temps, nous avons utilisé en tant qu'heuristique la distance euclidienne séparant la cible du robot à déplacer. Nous n'étions pas complètement satisfaits des solutions que délivrait cette méthode, mais nous l'avons gardée en attendant de pouvoir l'améliorer.

Il nous est alors venu l'idée d'associer des poids à chaque case, c'est-à-dire à chaque déplacement possible, en s'inspirant de la méthode résolution en sens inverse. Ce poids est déterminé en calculant le nombre de coups nécessaires pour aller de la cible jusqu'à ladite case. Cependant dans Ricochet Robots, les chemins ne sont pas bidirectionnels généralement. Pouvoir aller au point B en partant du point A ne signifie pas que l'on peut aller a du point A en partant du point B.

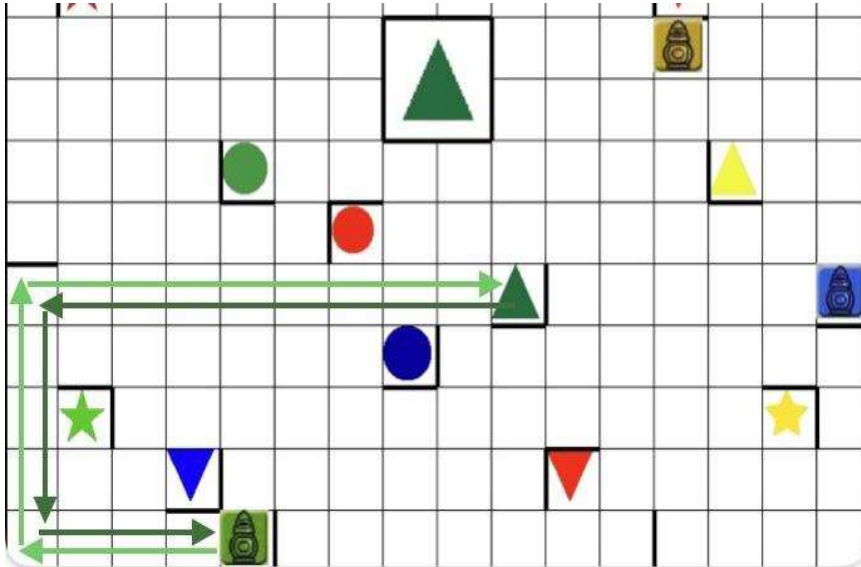
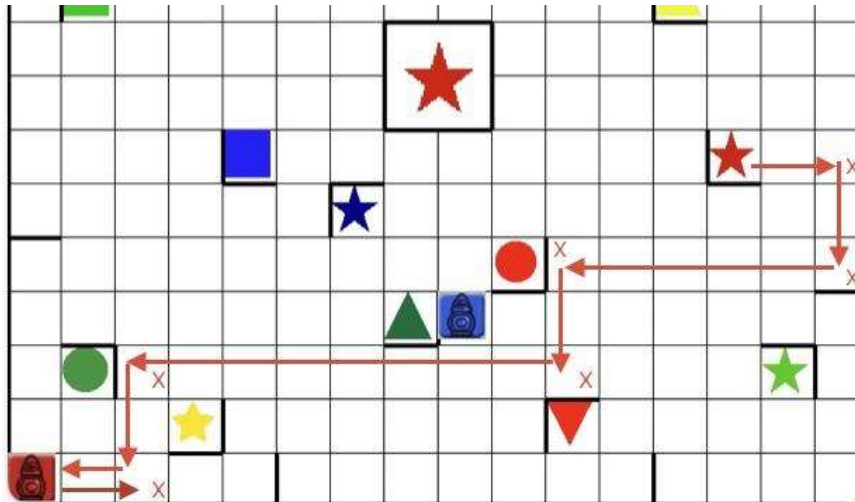


Figure 4 : Chemin bidirectionnel



X : position innatignable par le robot rouge dans cette situation

Figure 5 : Chemin unidirectionnel

Le caractère unidirectionnel de certains chemins pose des limites à l'heuristique des poids, cependant elle reste efficace dans de nombreux cas.

## IV- Exemples, analyses & résultats

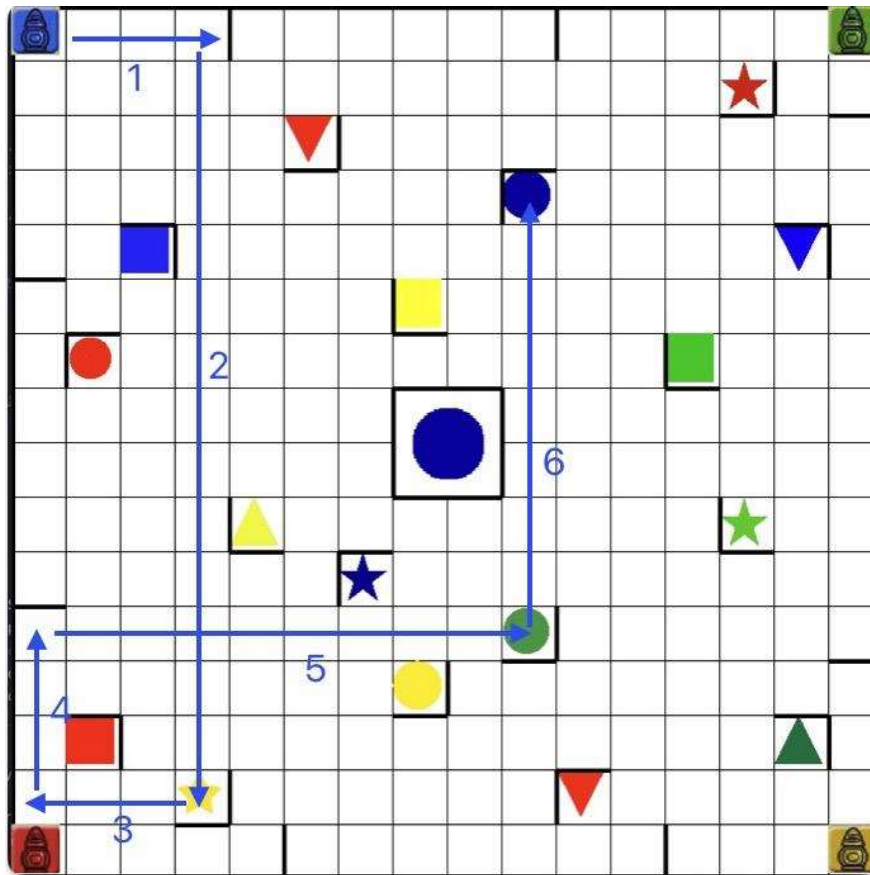


Figure 6 : Résolution en utilisant la distance euclidienne

Dans cette situation, on comprend rapidement les avantages et les inconvénients d'utiliser la distance euclidienne entre le robot et la cible comme heuristique. On voit que les coups 2 et 3 augmentent cette distance, et font donc parti de chemins visités plus tard par A\*. Cependant après le coup 1, il s'agit des uniques mouvements possibles et donc cela compense l'augmentation de la distance euclidienne. Tous les autres coups rapprochent le robot de la cible et montrent donc une utilisation possible de la distance euclidienne comme outil pour évaluer un nœud.







|   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|---|
|    | 5   | inf | 4   | 6   | inf | 5   | inf | 12  | 6   | 5   | 12  | inf | 11  | 6   |    |
| 9   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | 10  | inf | inf   |
| inf   | inf | inf | inf | inf | inf | 4   | inf | inf | 7   | inf | inf | inf | inf | inf | 3   |
| 11  | inf | inf | inf | inf | 10  | inf | inf | 11  | 1   | inf | inf | inf | inf | 7   | 2   |
| 8   | inf | 7   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| 7   | 6   | inf | inf | inf | inf | 7   | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| inf   | 13  | inf | inf | inf | inf | inf | inf | 12  | inf | inf | 13  | inf | inf | inf | inf   |
| inf   | inf | inf | inf | inf | inf | inf | inf | inf | 8   | inf | inf | 7   | inf | inf | 8   |
| inf   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| inf   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| inf   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| 8   | inf | inf | inf | 9   | 9   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| 1   | inf | inf | inf | inf | inf | inf | inf | inf | 0   | 4   | inf | inf | inf | inf | 3   |
| 15  | 14  | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
| 10  | 9   | 6   | inf | inf | inf | inf | inf | inf | inf | 5   | inf | inf | inf | 6   | inf   |
| 2   | inf | inf | 3   | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf | inf   |
|  | 8   | 7   | inf | 8   | 10  | inf | inf | inf | inf | inf | 11  | 7   | inf | 7   |  |

Figure 7 : Assignment des poids pour  $A^*$

On part donc de la cible principale, à laquelle on assigne un poids nul, et on assigne un poids infini à toutes les autres cases. A partir de là, on regarde les coups possibles depuis la cible et on assigne un poids 1 aux cases d'arrivées de ces coups. Ainsi de suite on obtient des poids pour chaque case atteignable par le robot. Ces poids représentent alors le résultat  $h(x)$  de l'heuristique. Dans le traitement des nœuds,  $A^*$  privilégiera donc ceux au poids moindre.

Un problème subsiste toutefois : le mouvement des autres robots est nécessaire dans certains cas pour obtenir la meilleure solution, ou même pour obtenir une solution tout court.

Pour les résolutions impliquant plusieurs robots, une idée aurait été de détecter les cibles nécessitant le déplacement d'un autre robot et de déplacer ce robot sur une case voisine à celle de poids 2 afin de rendre la résolution possible. Ici, en cas de poids 3 au-dessus de celui de poids 2, on peut alors accéder à la case de poids 2 par un chemin différent de celui passant par la case de poids 1.

Une autre idée plus générale serait de faire effectuer 1 coup ou 2 par les autres robots et de recalculer les poids du plateau pour explorer un maximum de possibilités de nœuds optimaux.

Cependant nous n'avons pas réussi à implémenter efficacement ces idées dans notre programme. Ce dernier ne prend donc pas en compte les mouvements possibles des autres robots.

Néanmoins, si l'utilisation d'un autre robot n'est pas nécessaire à la résolution du problème, on peut alors affirmer avec certitude que l'on tracera un chemin bidirectionnel partant de la cible jusqu'au robot, et la résolution peut alors s'effectuer.

Finalement, une combinaison des 2 heuristiques de poids et de distance euclidienne suggérerait un algorithme A\* efficace. Cela pourrait passer par soit par la somme des 2 dans le calcul des couts, soit en déterminant laquelle est la plus adaptée selon la situation.

## V- Améliorations envisageables

- Comme évoqué ci-dessus, prendre en compte les déplacements des autres robots est crucial pour une version optimale de l'IA.
- Des optimisations sont toujours possibles ici et là pour améliorer la rapidité de l'algorithme.
- L'implémentation en elle-même pourrait être réalisée de manière plus conventionnelle, plus « propre » et en respectant les normes de la programmation orientée objet.
- L'utilisation d'un A\* ad hoc nécessite d'approfondir encore les connaissances spécifiques sur Ricochet Robot et sur les modèles choisis pour représenter le jeu (Algorithmes de Dijkstra, Bellman-Ford, théorie des graphes en général, etc). On pourrait alors trouver une heuristique plus adaptée encore.

## VI- Perspectives

Ricochet Robots représente notre toute première expérience de développement mettant en jeu de l'intelligence artificielle, mais nous aura aussi permis de découvrir le processus de reproduction d'un jeu, qui pourrait s'apparenter de manière plus générale à un processus de reproduction d'un phénomène quelconque afin d'y impliquer une intelligence artificielle. Cette étape de modélisation d'un système et de ses états est cruciale dans le développement d'une IA, peu importe le sujet. Avoir accompli ces étapes s'avèrera probablement très utile et transposable pour les projets à venir.

Ce projet nous aura permis de mettre en pratique et les connaissances acquises tout au long de l'U.E d'IA41. Mais aussi d'aller plus loin en devant rechercher des connaissances un peu plus spécifiques par nous-même, touchant à la théorie des graphes ou plus simplement au langage Python par exemple.

## VII- Annexe

Voici 2 des fonctions essentielles de notre programme :

### **Get\_neighbors (dans ai.py)**

Cette fonction prenant en paramètre une position et le plateau de jeu permet de renvoyer une liste de voisins accessibles depuis ladite position. Cela est bien sûr utile dans notre algorithme A\* pour obtenir les nœuds voisins d'un nœud.

### **poidplateau (dans poid.py)**

Il s'agit de la fonction calculant les poids pour toutes les cases d'un plateau en fonction d'une cible donnée. Elle utilise la logique inhérente au jeu Ricochet Robots pour simuler les mouvements possibles depuis une position, et retourne un plateau comportant les poids associés à chaque case.

## Bibliographie

Image de couverture : <https://www.cageauxtrolls.com/jeux-familiaux/ricochet-robots.html>