Projet Ricochet Robots - RP Cours de Sorbonne Université - 2024

SIDKI Noureddine [2008210]

April 22, 2024

L'objectif de ce projet était d'implémenter différentes méthodes de recherche de chemins optimaux basées sur le jeu Ricochet Robots [1], en utilisant le langage de programmation Python.

Modélisation, instances et résolution par recherche arborescente

Initialement, nous avons reçu un code de départ [2] permettant de générer des instances de grilles pour le jeu Ricochet Robots et de les afficher. L'objectif de ce jeu est de déplacer le robot cible vers la cible en un nombre optimal de coups. D'autres robots peuvent se déplacer pour réduire le nombre de coups de la solution, mais nous les considérerons plus tard. Seul un robot peut être déplacé à chaque coup.

Une instance est caractérisée par une grille carrée de dimensions $n \times n$, où n est un entier pair supérieur ou égal à 8, et par k robots, avec $k \in [1, 5]$. Les instances sont représentées par trois grilles : deux grilles binaires représentant respectivement les murs verticaux et horizontaux, et une autre grille de cellules représentant les positions des robots avec des entiers de 1 à 5, ainsi que la position de la cible à atteindre (représentée par -1).

Les robots et la cible sont représentés par des positions $(i, j) \in \{1, 2, \ldots, n\}$, i (respectivement j) représentant les lignes (respectivement les colonnes).

Nous avons choisi de représenter nos états avec les positions des robots $R^1_{i,j}, \ldots, R^k_{i,j}$, où $(i, j) \in \{1, 2, \ldots, n\}$. Si l'on nomme la position de la cible $T_{k,l}$, alors l'état final est caractérisé par $R^1_{m,n}$, où $m = k$ et $n = l$.

Nous allons désormais définir une borne supérieure de la taille de l'espace d'états. Une case ne peut être occupée que par un seul robot. Il y a donc $(n \times n) - (k - 1)$ états possibles pour un robot dans notre grille, et vu que l'on a k robots, alors on élève le tout à la puissance k, ce qui implique que la borne supérieure de la taille de l'espace d'état est de $((n \times n) - (k - 1))^k$. Si l'on considère l'îlot central inatteignable de taille 2×2 imposé par le code fourni, alors nous aurons $(((n \times n) - 4) - (k - 1))^k$ états possibles.

Par ailleurs, un robot engendrera 4 fils, car le robot peut se déplacer dans 4 directions : nord, sud, est et ouest. Ce qui signifie que nous aurons comme borne maximale $4 \times k$ états successeurs d'un état. Parfois ce sera moins car il se peut qu'un robot soit entouré d'obstacles l'empêchant de se mouvoir.

Nous avons décidé d'implémenter une procédure arborescente de recherche en profondeur pour déterminer une borne supérieure de l'évaluation de la solution optimale. La recherche en profondeur se concentre sur le développement d'une unique branche et effectue un balayage vertical jusqu'à trouver une solution (quelconque), ce qui en fait un choix approprié pour notre objectif.

Rappel : Nous considérons que seul le robot cible peut se déplacer.

Notre recherche en profondeur fait appel à deux classes Python :
1.) La classe `Noeud` représente un état dans notre recherche. Ses attributs sont :

- la *position* du robot cible

- le *nombre de coups* effectués pour atteindre cet état
- le *chemin* parcouru jusqu'à ce nœud, représenté sous forme d'un tableau contenant les positions précédentes

2.) La classe `RicochetRobotSolverProfondeur` est responsable de la recherche en profondeur dans l'arbre de recherche. Elle contient les attributs suivants :

- la taille de la grille n
- le nombre de robots k
- la profondeur maximale de recherche m, qui permet de définir une limite de profondeur pour arrêter le programme si la profondeur est trop grande
- les grilles *cellules*, *horizontaux* et *verticaux* caractérisant l'instance
- O, une liste contenant les nœuds à développer
- F, une liste contenant les positions des nœuds déjà explorés
- un booléen *trouve* qui indique si une solution a été trouvée ou non
- *target*, contenant la position de la cible à atteindre

La classe implémente diffèrentes méthodes.

La méthode `Recherche` est la principale. Elle implémente la recherche en profondeur en initialisant le départ de la recherche et en itérant jusqu'à ce qu'il n'y ait plus de nœuds à ouvrir ou qu'une solution soit trouvée. Elle récupère le prochain nœud à ouvrir en choisissant celui avec le nombre minimal de coups, selon le principe "dernier arrivé, premier servi" (pile). Si le nœud est un état final, la méthode s'arrête et retourne le chemin obtenu ainsi que le nombre de coups de la solution. Si le nombre de coups du nœud dépasse la profondeur maximale, alors la méthode passe au nœud suivant. Sinon, elle développe le nœud en utilisant la méthode `Voisins`, qui génère les états fils du nœud actuellement ouvert en itérant sur chaque direction possible (4 états successeurs) en appelant la méthode `Traverser`. Cette dernière itère en avançant la position du robot case par case dans la direction en paramètre jusqu'à ce qu'il rencontre un obstacle. Les collisions peuvent être causées par les murs verticaux et horizontaux ou par les autres robots.

De plus, si le nœud fils engendré fait déjà partie des nœuds à ouvrir prochainement O ou des nœuds déjà ouverts F, alors nous ne le prenons pas en compte et passons au prochain voisin, ce qui permet d'éviter des boucles infinies.

Enfin, nous avons la méthode `PrintSolution` qui prend le chemin solution en paramètre et affiche l'évolution de la grille pour chaque coup effectué dans la solution.

Nous avons choisi de définir la variable m, représentant la profondeur maximale, à n, la dimension de la grille, ce qui nous semble être un bon compromis. Bien sûr, elle peut être ajustée pour ne pas surcharger le GPU.

Après l'exécution de la recherche en profondeur, nous obtenons une solution non optimale au pire des cas, ainsi que le nombre de coups associés, qui sert de borne supérieure pour le nombre de coups requis pour la solution optimale.

Par la suite, pour obtenir un chemin optimal, nous devrons désormais considérer les déplacements de **tous les robots**, pour cela nous avons choisi d'implémenter une recherche en largeur qui est appropriée car contrairement à la recherche en profondeur qui se concentre sur une branche et la développe au maximum, la recherche en largeur va plutôt effectuer un balayage horizontal en se concentrant à développer les noeuds de l'arbre étage par étage.

Pour cela, nous implémentons une classe `RicochetRobotSolverLargeur` qui est un héritage de la classe `RicochetRobotSolverProfondeur`. Désormais, au lieu de contenir uniquement la position du robot R1 dans les nœuds, nous aurons désormais les positions de tous les robots sous la forme d'un tableau [[R1.i,R1.j], ..., [Rk.i,Rk.j]]. Par conséquent, au lieu de générer uniquement les quatre états fils issus des déplacements du robot R1 pour le nœud courant, elle va également générer les états fils de tous les autres robots de la grille. Ce qui nous donne 4*k états successeurs. Désormais, lors du choix d'un nœud O à ouvrir, nous utiliserons le principe du premier arrivé, premier servi (queue). Lorsque nous engendrons un état fils, nous vérifions toujours s'il n'appartient ni à l'ensemble O ni à l'ensemble F en prenant le soin de comparer les positions de tous les robots. Si tel est le cas, nous rajoutons le voisin à l'ensemble O. Nous n'avons pas besoin de comparer les nombres de coups car on évolue dans l'ordre croissant du nombre de coups en balayant chaque étage, donc il est impossible de retomber plus tard sur la même position avec un nombre de coups inférieur. Au pire des cas, il sera égal et donc les deux états seront les mêmes (même positions et même nombre de coups).

Nous allons démontrer l'efficacité de notre recherche en largeur. Pour cela, nous avons généré une grille, puis nous avons essayé l'algorithme de recherche en profondeur, il n'a trouvé aucune solution, ce qui est normal car comme on le constate, aucune solution n'est possible en déplaçant uniquement R1 (voir Figure 1).

Nous avons observé la grille et imaginé une solution optimale en 5 coups (voir Figure 1).

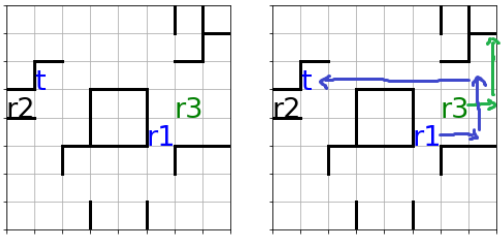

Figure 1: Instance généré et solution imaginée

Nous avons donc lancé la recherche en largeur en nous attendant à ce que notre algorithme trouve cette solution, mais contre toute attente, la recherche en largeur a trouvé une solution meilleure que toutes nos espérances : une solution optimale en 4 coups (voir Figure 2).

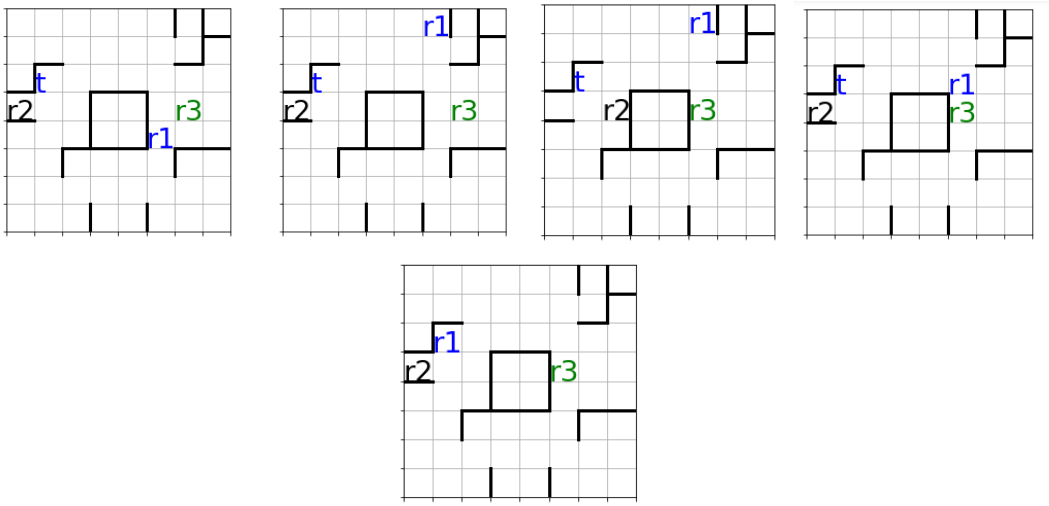

Figure 2: Solution optimale de la recherche en largeur en 4 coups

Maintenant, nous allons étudier les performances de la méthode de recherche en largeur. Comme on le voit dans les figures 3, 4, 5 et 6 nous constatons que plus la taille de la grille n et le nombre de robots k augmentent, plus le temps de recherche et le nombre de coups de la solution optimal augmentent.

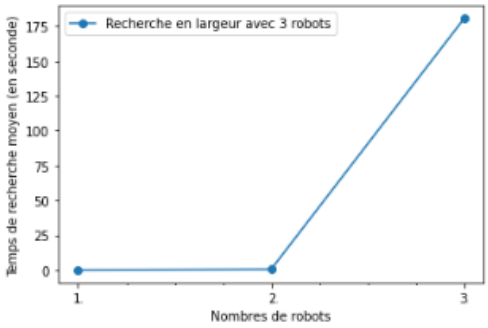

Figure 3: Comparaison des temps moyens de 20 instances - recherche en largeur pour $n = 8$ et différents nombres de robots

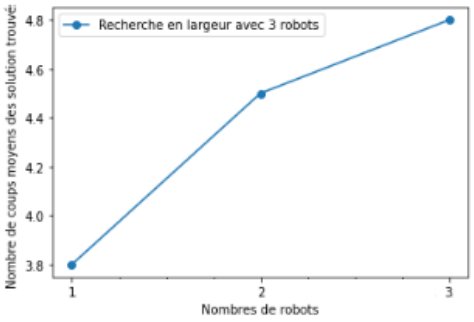

Figure 4: Comparaison du nombre moyen de coups de 20 instances - recherche en largeur pour $n = 8$ et différents nombres de robots

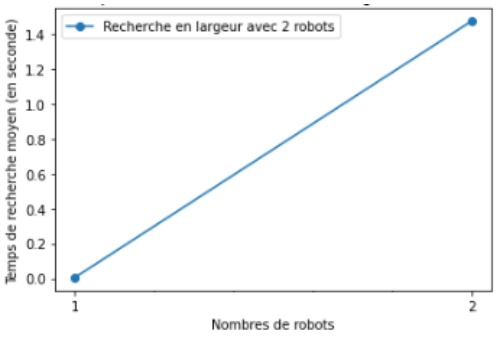

Figure 5: Comparaison des temps moyens de 20 instances - recherche en largeur pour $n = 10$ et différents nombres de robots

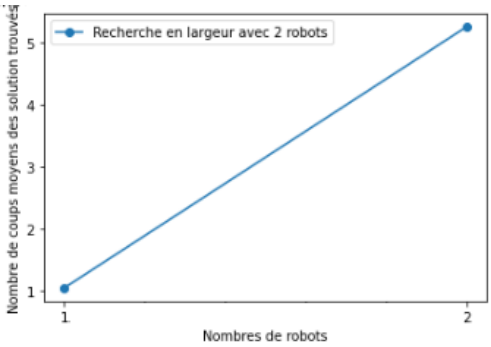

Figure 6: Comparaison du nombre moyen de coups de 20 instances - recherche en largeur pour $n = 10$ et différents nombres de robots

Résolution par A*

Désormais, nous allons étudier la recherche avec l'algorithme A*. La première heuristique, h1, est la suivante : si le robot et la cible se situent sur la même ligne ou la même colonne, l'évaluation est de 1. Si le robot et la cible se situent sur des lignes et colonnes différentes, l'évaluation est de 2. Enfin, si le robot et la cible sont à la même position (même ligne et même colonne), l'évaluation est de 0. Nous allons vérifier si cette heuristique est minorante. Le plus simple serait de vérifier si elle est à la fois monotone et cohérente.

$$\forall \gamma \in \Gamma, h(\gamma) = 0 \tag{1}$$

$$\forall n \in \mathbb{N}, \forall m \in S(n),\ h(n) - h(m) \leq k(n,m) \tag{2}$$

Une heuristique coïncidente est définie par cette formule (1). L'évaluation des états finaux doit être égale à 0, ce qui est bien le cas de cette heuristique, car il est mentionné que si le robot et la cible sont à la même position, alors l'évaluation est de 0.

Une heuristique monotone est définie par cette formule (2). Soit m l'état succédant à l'état n. Le coût $k(n,m)$ est constant à 1 pour tout n et m, par définition du jeu.

Nous allons simplement vérifier toutes les situations :
1.) Si n est évalué à 2 et m est évalué à 1, alors $h(n) - h(m) = 1 \leq k(n,m) = 1$. Validé.
2.) Si n est évalué à 1 et m est évalué à 0, alors $h(n) - h(m) = 1 \leq k(n,m) = 1$. Validé.
3.) Si n est évalué à 2 et m est évalué à 2, alors $h(n) - h(m) = 0 \leq k(n,m) = 1$. Validé.
4.) Si n est évalué à 1 et m est évalué à 1, alors $h(n) - h(m) = 0 \leq k(n,m) = 1$. Validé.
5.) Si n est évalué à 0 et m est évalué à 1, alors $h(n) - h(m) = -1 \leq k(n,m) = 1$. Validé.
6.) Si n est évalué à 0 et m est évalué à 2, alors $h(n) - h(m) = -2 \leq k(n,m) = 1$. Validé.
7.) Si n est évalué à 2 et m est évalué à 0, alors $h(n) - h(m) = 2 \geq k(n,m) = 1$. Non validé, mais cette situation ne peut pas arriver car les seuls déplacements autorisés sont horizontalement et verticalement. Ainsi, si le nœud n n'est pas sur la même ligne ou colonne que le nœud cible (heuristique à 2), il ne pourra pas atteindre la cible en un coup unique (heuristique à 0). Cette situation n'est donc pas comptabilisée.

Ainsi, nous pouvons conclure que l'heuristique est monotone.
Étant donné que l'heuristique est à la fois monotone et coïncidente, par la preuve (7), nous pouvons conclure que l'heuristique H1 est minorante.

Proposition

Toute heuristique monotone et coïncidente est minorante

Preuve

Soit $n, m_1, m_2, \ldots, m_q, \gamma$ un chemin optimal de n vers le but :

$$\begin{array}{rcl} h(n) - h(m_1) & \leq & k(n, m_1) \\ h(m_1) - h(m_2) & \leq & k(m_1, m_2) \\ \vdots & \leq & \vdots \\ h(m_q) - h(\gamma) & \leq & k(m_q, m_\gamma) \\ \hline h(n) - h(\gamma) & \leq & h^*(n) \end{array}$$

Comme h est coïncidente alors $h(\gamma) = 0$, donc $h(n) \leq h^*(n)$

Figure 7: Preuve heuristique monotone et coïncidente est minorante [2]

Pour implémenter l'algorithme A* avec l'heuristique 1, nous avons créé une classe NoeudAlgoA avec les attributs suivants :

- la *position* des robots pour cet état
- g, le coût pour arriver à cet état (équivalent au nombre de coups effectués)

- f, le coût du meilleur chemin solution passant par cet état
- le *chemin* parcouru jusqu'à ce nœud, représenté sous forme d'un tableau contenant les positions précédentes

Nous avons également implémenté une classe RicochetRobotSolverA_H1 qui hérite de la classe RicochetRobotSolverLargeur. Nous avons ajouté une fonction CalculerH1 qui prend la position du robot R1 en paramètre et qui évalue l'heuristique. Elle est notamment utilisée lorsque l'on engendre les états fils pour initialiser g et f. Soit m l'état succédant à l'état n. Alors $g^m = g^n + k(n, m)$ sachant que $k(n, m) = 1$ et $f^m = g^m + h^m$.

Nous avons également redéfini la fonction ChoixNoeud pour qu'elle sélectionne le prochain nœud à ouvrir en minimisant la valeur f tout en maximisant la valeur g. Pour des raisons de performance qui ont drastiquement impacté notre programme, nous n'avons pas pris en compte le fait qu'un nœud fils, ayant les mêmes positions de robots qu'un nœud déjà exploré F ou bientôt ouvert O, mais possédant un g inférieur, soit ajouté à la liste des nœuds à ouvrir O. Nous l'avons tout de même implémenté en commentaire.

Par la suite, nous avons implémenté l'algorithme A* avec l'heuristique 2.
Cette heuristique relâche la contrainte stipulant que le robot devant atteindre la cible ne peut pas s'arrêter en cours de déplacement. Une fois un déplacement amorcé, le robot peut donc s'arrêter sur n'importe quelle case (mais il doit toujours s'arrêter s'il rencontre un mur). Les autres robots ne sont pas considérés dans l'évaluation de cette heuristique. Bien sûr, les contraintes sont relâchées uniquement pour le calcul de l'heuristique.

Nous avons donc implémenté une classe RicochetRobotSolverA_H2 qui hérite de la classe RicochetRobotSolverA_H1.
Cette classe possède un attribut TableH2 qui contiendra les heuristiques que l'on calculera à l'avance pour toutes les positions de la grille uniquement à l'aide de la position de départ. La TableH2 est initialisée au départ grâce à la fonction CalculerTableH2.

Pour ce faire, nous utilisons le principe de la queue : nous commençons par insérer la position de départ associée à la valeur 0, puis nous engendrons les états fils dans toutes les directions et les insérons dans la queue qui seront ensuite dépilés. Ensuite, nous engendrons leurs états fils et ainsi de suite.
Pour générer les états fils, nous utilisons la fonction GetBorneH2 qui prend une direction donnée et la position du robot R1 retiré de la queue. Cette fonction nous retourne la dernière position avant de toucher un obstacle (elle prend uniquement en compte les collisions avec les murs verticaux et horizontaux).
Enfin, nous déplaçons la position du robot R1 jusqu'à la borne en remplissant les cases de la TableH2 situées sur le trajet avec la valeur du nœud additionnée de 1.

Le fait de précalculer le tableau nous évitera des calculs redondants. Pour calculer les variables g et f des nœuds, nous accédons directement à l'évaluation située dans la TableH2 à la position du robot R1 du nœud courant. Pour les mêmes soucis de performance évoqués précédemment, nous n'avons pas pris en compte le fait qu'un nœud avec les mêmes positions qu'un nœud de l'ensemble des nœuds ouverts O ou des nœuds fermés F, mais possédant un g inférieur, soit ajouté aux nœuds à ouvrir prochainement O.

Pour finir, nous comparons les performances des trois méthodes sur des instances identiques. Pour cela, nous calculons et affichons dans des graphes les temps de recherches et les nombres de coups de la solution pour chaque instance. Dans les figures (8), (9), (10), nous voyons que pour des instances de

différentes tailles, les temps de recherche sont toujours en faveur de l'algorithme A* avec l'heuristique 1, car c'est la plus efficace et est minorante. Elle est suivie par la méthode de recherche en largeur, et enfin de l'algorithme A* avec l'heuristique 2 (considérant que c'est valable pour les n et k utilisés, cela peut changer avec d'autres valeurs, peut-être plus grandes).

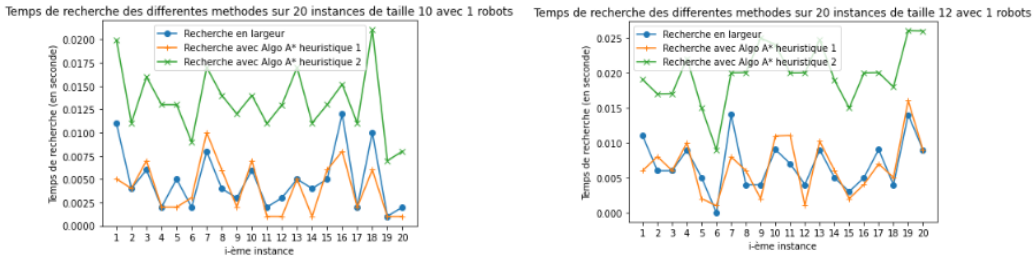

Figure 8: Comparaisons des temps de recherche pour 20 instances de 1 robot et différentes tailles

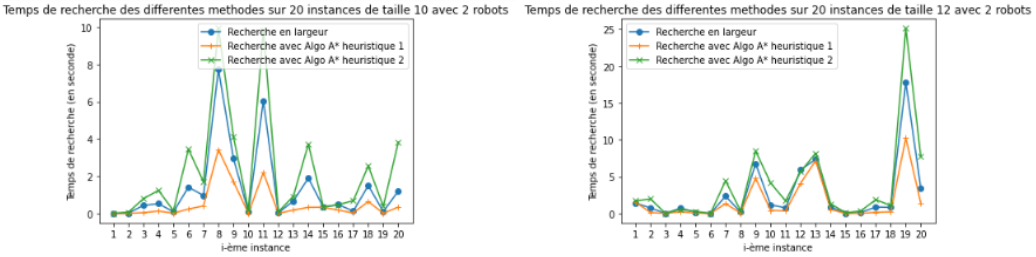

Figure 9: Comparaisons des temps de recherche pour 20 instances de 2 robots et différentes tailles

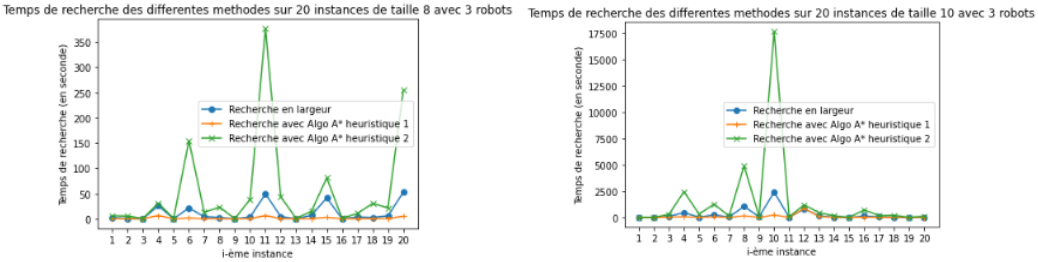

Figure 10: Comparaisons des temps de recherche pour 20 instances de 3 robots et différentes tailles

À l'aide des figures (11), (12), (13), nous voyons que les nombres de coups sont identiques, ce qui indique que les algorithmes trouvent tous les mêmes solutions optimales (en considérant que c'est valable pour les n et k utilisés, cela peut changer avec d'autres valeurs, peut-être plus grandes).

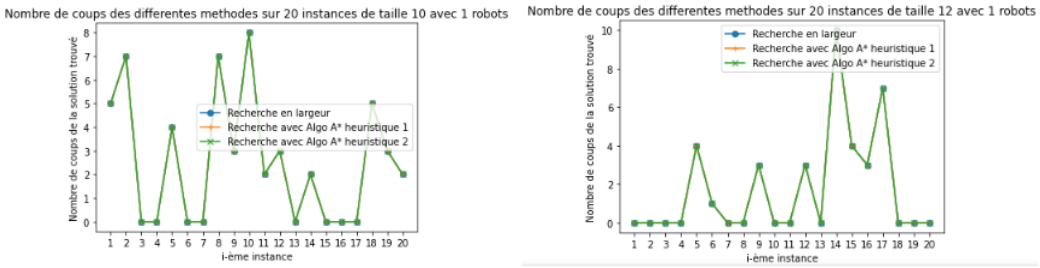

Figure 11: Comparaisons nombre de coups optimales pour 20 instances — 1 robot — différentes taille

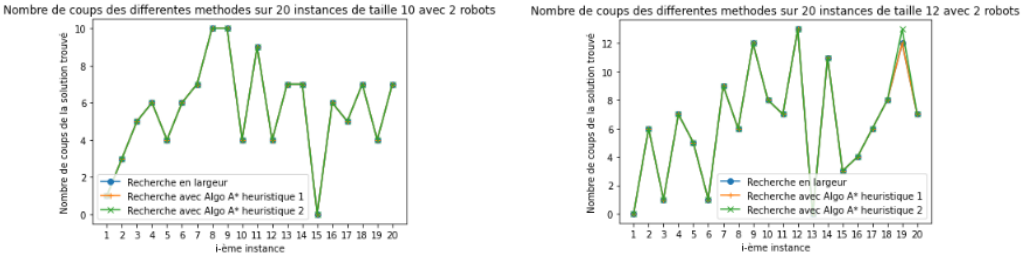

Figure 12: Comparaisons nombre de coups optimales pour 20 instances — 2 robots — différentes taille

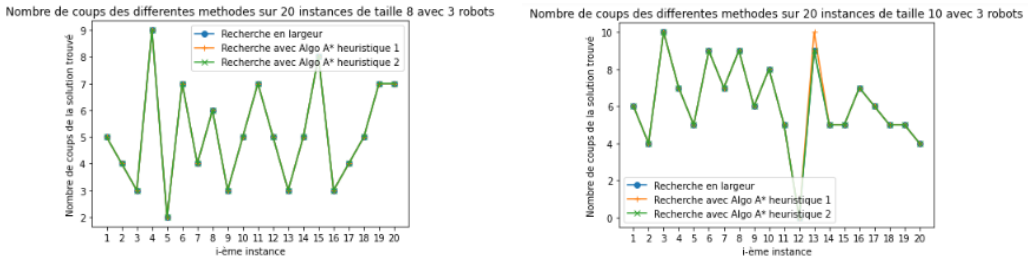

Figure 13: Comparaisons nombre de coups optimales pour 20 instances — 3 robots — différentes taille

Conclusion

Ainsi s'achève notre analyse. Nous avons présenté nos différentes implémentations des méthodes de recherche de solution pour le jeu Ricochet Robots et avons comparé leurs performances à différentes échelles.

References

[1] Ricochet Robots, May 2021. https://fr.wikipedia.org/w/index.php?title=Ricochet_Robots&oldid=183427013.

[2] S. Université. Sujet du projet. https://moodle-sciences-23.sorbonne-universite.fr/pluginfile.php/277516/mod_resource/content/1/ProjetRP_RicochetRobot.pdf.