RP

Rapport

Ricochet Robots

Réalisé par :
Erisa KOHANSAL
Shirel AMOZIEG

Encadrés par :
Thibaut Lust

Avril 2024

Table des matières

1 Introduction

Ce projet consiste à développer et tester des méthodes de recherche arborescente pour la résolution du jeu "Ricochet Robots". Le principe général du jeu est de déplacer des robots vers des emplacements sélectionnés tout en respectant les limites strictes imposées aux mouvements des robots.

2 Questions

2.1 Question 1

Le plateau de jeu peut être modélisé par une matrice binaire représentant les murs verticaux, une matrice binaire représentant les murs horizontaux et une matrice d'entiers représentant les robots et les cibles.
L'état initial est constitué du plateau de jeu et de la disposition initiale des robots et de la cible sur le plateau, leur coordonnées (x, y). On a donc un vecteur de $n + 1$ paires de coordonnées, avec n étant le nombre de robots présents.
L'état final est atteint lorsque le robot de la couleur correspondante à la cible atteint la case de la cible ; c'est à dire lorsque les coordonnées du robot sont identiques à celle de la cible.

2.2 Question 2

Pour un plateau de taille $n \times n$ et k robots, une borne supérieure de la taille de l'espace est :
$n^2 \times n^2 \times ... \times n^2$ (k fois) $= n^{2k}$
Pour $k = 4$ robots et un plateau de 16×16, la taille est $16^{2\times4}$

2.3 Question 3

On considère que chaque robot peut se déplacer dans 4 directions : haut, bas, gauche, droite. Il existe k robots sur le plateau. Le nombre total de combinaison correspond au produit du nombre de choix qu'a chaque robot (4 directions), soit $4 \times 4 \times 4... \times 4$ (k fois). Le nombre maximal de successeurs possibles d'un état est donc 4^k.

2.4 Question 4

Pour cette question, nous n'allons déplacer que le robot bleu vers sa cible. Nous avons décidé d'implémenter une procédure arborescente de recherche en largeur afin de trouver une solution optimale. Chaque nœud représente un état possible du jeu et chaque branche représente un mouvement possible du robot.

L'indice de la cible dans la grille est marquée par -1. $cible[0][0]$ récupère le premier élément du tableau des indices des lignes, et $cible[1][0]$ récupère le premier élément du tableau des indices des colonnes. La cible $(cible[0][0], cible[1][0])$ retourne donc un tuple représentant la position de la cible en coordonnées (ligne, colonne) dans la matrice.
L'indice de la position du robot bleu dans la grille est marquée par 1. On la récupère de

la même manière que la position de la cible.
Ensuite nous avons deux éléments ; *queue* et *distance*. *queue* est une file d'attente initialisée avec la position du robot bleu et une distance de valeur 0. On regarde dans toutes les directions possibles (en prenant en compte les murs, les robots et les limites de la grille). On avance tout droit jusqu'à un obstacle. On sauvegarde dans *queue* la position courante pour le prochain mouvement. Si la position trouvée est celle de la cible, on ajoute $+1$ à *distance* et on retourne m. Si la file est vide et aucun chemin vers la cible n'a été trouvé, on retourne -1 et la valeur de m.

Détaillons un peu plus la fonction *no obstacle*. Elle prend en entrée les coordonnées actuelles du robot tx et ty, les déplacements en x et y dx et dy, les matrices des murs horizontaux et verticaux, et la taille de la grille n. Ensuite elle détermine la position suivante *(nx, ny)* en ajoutant les déplacements *(dx, dy)* aux coordonnées actuelles *(tx, ty)*.
Puis elle vérifie les murs horizontaux.

* Si $nx == n - 1$: Si le robot est sur la dernière ligne et se déplace vers le bas, il n'y a pas de mur horizontal à vérifier.
* Sinon : Si $nx < n - 1$, vérifie qu'il n'y a pas de mur horizontal bloquant le mouvement :
  — $dx == 1$: Vérifie si le déplacement est vers le bas (incrémentation de x) et s'il y a un mur horizontal à la position actuelle (tx, ty).
  — $dx == -1$: Vérifie si le déplacement est vers le haut (décrémentation de x) et s'il y a un mur horizontal juste au-dessus du robot à la position $(tx + dx, ty)$.

Le même raisonnement est fait pour la vérification des murs verticaux.

* Si $ny == n - 1$: Si le robot est sur la dernière colonne et se déplace vers la droite, il n'y a pas de mur vertical à vérifier.
* Sinon : Si $ny < n-1$, vérifie qu'il n'y a pas de mur vertical bloquant le mouvement :
  — $dy == 1$: Vérifie si le déplacement est vers la droite (incrémentation de y) et s'il y a un mur vertical à la position actuelle (tx, ty).
  — $dy == -1$: Vérifie si le déplacement est vers la gauche (décrémentation de y) et s'il y a un mur vertical juste à gauche du robot à la position $(tx, ty + dy)$.

(Dans *main* les murs horizontaux et verticaux sont enregistrés).

La fonction vérifie ensuite de l'absence de robots : si la cellule à la nouvelle position (nx, ny) est vide (0) ou contient la cible (-1), indiquant qu'il n'y a pas d'autres robots à cette position.
Elle renvoie *True* si aucun obstacle n'empêche le mouvement vers la nouvelle position, sinon *False*.

2.5 Question 5

On prend à présent en compte tous les robots. La fonction bfs_tous_robots implémente une procédure de recherche arborescente de recherche en largeur.

Comme dans la question précédente, la cible est marquée par -1 et est localisée de la même manière.
On crée une file d'attente *queue* pour stocker la configuration des robots. Après quoi, pour

chaque robot on trouve sa position et on ajoute cette position à une distance égale à 0 dans une liste de configuration. Chaque configuration est ensuite ajoutée à *queue*.
On exécute le *bfs* du seul robot bleu pour récupérer la valeur de m (borne supérieure de recherche) qui va nous servir de comparaison avec celle calculée pour les itérations *mprime* que l'on initialise à 0.
Tant que la file n'est pas vide et que $mprime < m$, on itère. On récupère la première configuration de robot de la file. On calcule quel est le meilleur itinéraire (droite, gauche, haut, bas). Si le robot bleu atteint sa cible, on retourne la distance $+1$ et la valeur du compteur *mprime*. Sinon, on crée une nouvelle configuration avec la position mise à jour du robot actuel et les positions inchangées des autres robots, puis l'ajoute à *queue*. On incrémente le compteur *mprime*. Si à la fin, la file est vide et la cible n'a pas été atteinte, on retourne -1 et la valeur de *mprime*.

2.6 Question 6

À partir de l'instance représentée dans la figure 3 de l'énoncé, il est possible de discerner rapidement la solution optimale, qui est exposée dans la figure ci-dessous.

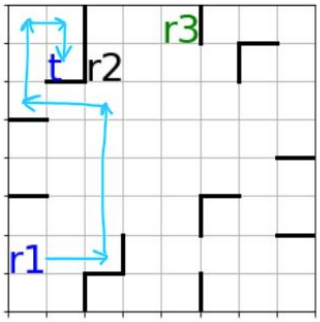


FIGURE 1 – Séquence optimale des mouvements des robots pour atteindre la cible.

Plusieurs stratégies peuvent être envisagées pour cette configuration particulière. Une option consiste à déplacer le robot $r2$ vers le bas et par la suite, de procéder avec des mouvements exclusifs de $r1$. Cette séquence de mouvements résulte en une solution avec un coût de 7. Néanmoins, l'analyse révèle que la solution optimale, illustrée dans la figure 1, atteint l'objectif avec un coût inférieur, soit 6 mouvements. Cette approche optimale élimine le besoin de déplacer $r2$ et concentre l'ensemble des actions sur r1, démontrant une gestion plus efficace des déplacements pour atteindre la cible.

2.7 Question 7

Dans la suite, les instances testées pour les figures 2 et 3 sont les mêmes, tout comme celles pour les figures 4 et 5. Ci-dessous on a deux graphes pour visualiser les temps d'exécution de chaque méthode sur 20 instances avec une grille de taille $n = 8$ et $k = 3$.

Figure 2 illustre les temps d'exécution de la méthode de recherche implémentant uniquement le déplacement du robot bleu. Les résultats montrent une grande variabilité dans le temps de résolution, avec des pics représentant des instances particulièrement difficiles. Néanmoins, la moyenne de ces temps d'exécution est relativement basse, s'établissant à environ 0.91 secondes. On voit bien que lorsque la méthode est limitée au déplacement

d'un seul robot, le problème peut être résolu assez rapidement en moyenne, bien que certaines configuration de la grille peuvent induire des résolutions plus complexes et donc plus longues.

Figure 3 présente les temps d'exécution pour la recherche en considérant tous les robots. On observe immédiatement une augmentation significative des temps d'exécution, avec une moyenne de près de 4.64 secondes. Cette hausse est attribuée à la complexité accrue du problème quand tous les robots sont mobiles. Plusieurs chemins possibles doivent être évalués pour chaque robot, ce qui augmente exponentiellement le nombre de configuration à explorer.

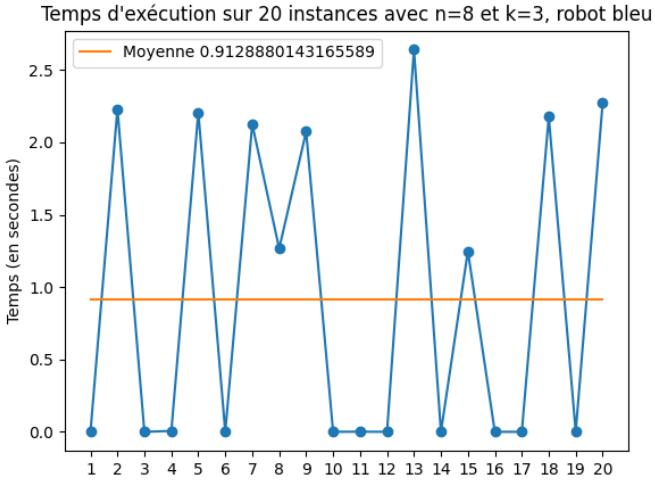


FIGURE 2 – Temps d'exécution de bfs_robot_bleu sur 20 instances avec $n = 8$ et $k = 3$.

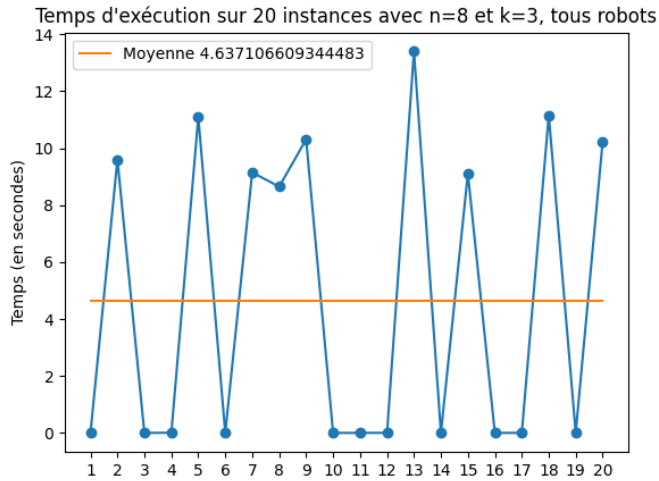


FIGURE 3 – Temps d'exécution de bfs_tous_robots sur 20 instances avec $n = 8$ et $k = 3$.

Les résultats de cette analyse soulignent l'importance de la sélection de la stratégie de résolution pour des jeux de nature NP-difficile comme Ricochet Robots.

Les graphiques suivants présentent les temps d'exécution moyens sur une grille $n = 8$ en faisant varier le nombre de robots k.

Figure 4 (mouvement du robot bleu) illustre que les temps d'exécution varient considérablement, allant de presque immédiat à environ 2.5 secondes. La moyenne de ces temps se situe à environ 1.37 secondes, reflétant un temps de calcul modéré pour résoudre des instances du problème avec seulement le robot bleu actif.

Figure 5 (mouvement de tous les robots) montre un accroissement notable des temps de résolution, avec des pics atteignant près de 17.5 secondes et une moyenne significativement plus élevée d'environ 7.41 secondes.

La comparaison directe des moyennes des deux graphiques démontre que le temps d'exécution augmente de manière exponentielle avec le nombre de robots impliqués dans la résolution. Cela suggère que chaque robot supplémentaire ajoute une couche de complexité à l'ensemble du problème, ce qui se traduit par une augmentation du temps nécessaire pour trouver une solution optimale.

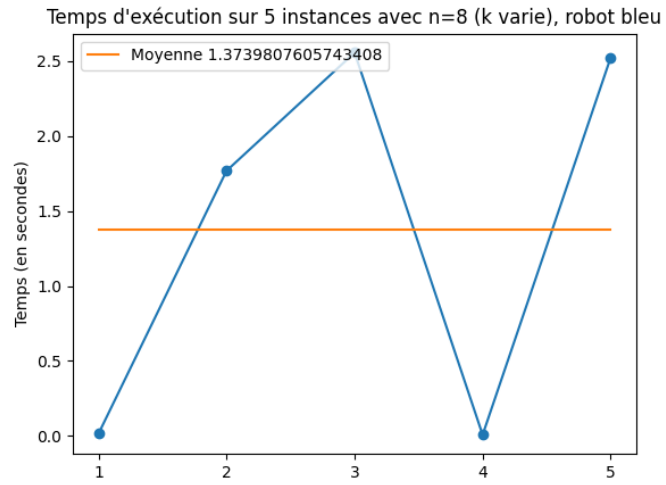


FIGURE 4 – Temps d'exécution de bfs_robot_bleu sur 5 instances de taille $n = 8$, avec k allant de 1 à 5.

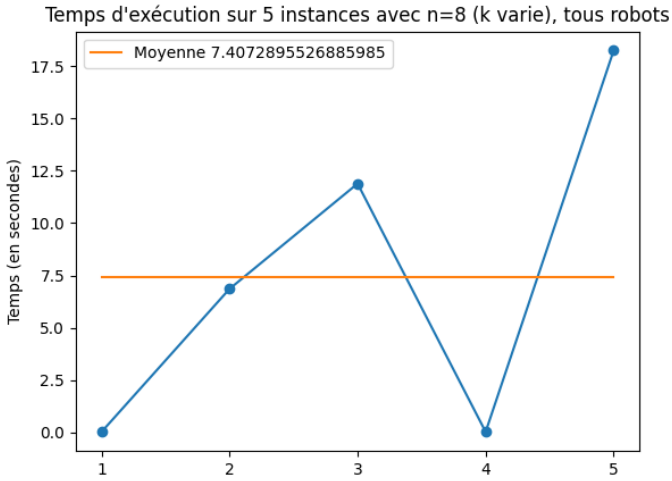


FIGURE 5 – Temps d'exécution de bfs_tous_robots sur 5 instances de taille $n = 8$, avec k allant de 1 à 5.

2.8 Question 8

h1 :
- Sur la même ligne ou la même colonne que les coordonnées de la cible : 1
- Sur des lignes et colonnes différentes des coordonnées de la cible : 2
- Sur la même ligne ET la même colonne (à la même position que la cible) : 0

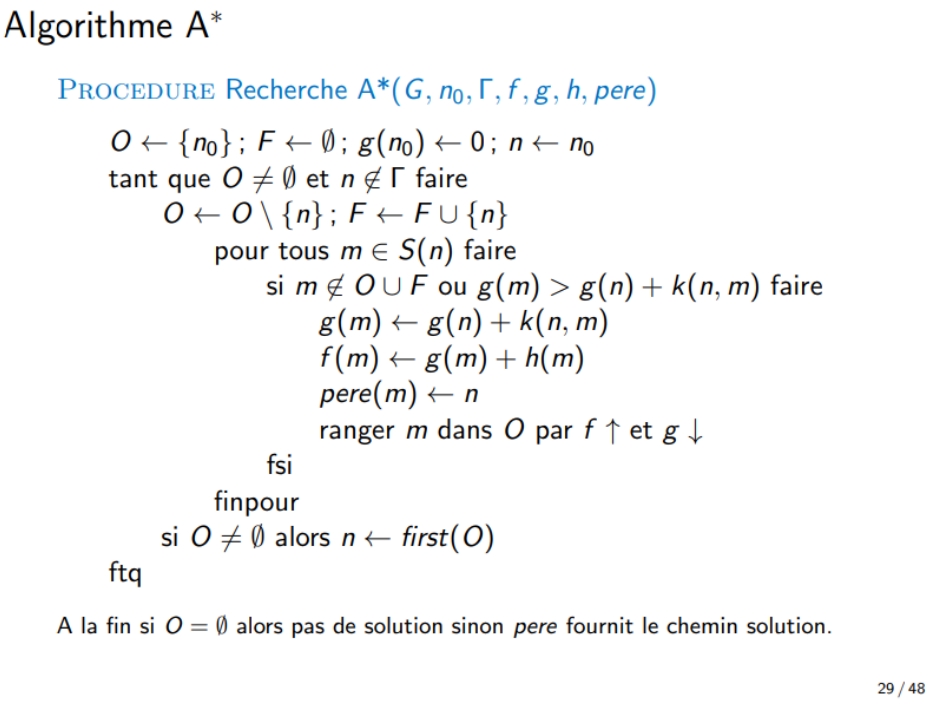
$h1(configurationfinale) = 0$, lorsque le robot et la cible sont sur la même case. Elle est donc coïncidente.

Avec un schéma sur papier, nous avons testé les différentes possibilités de passage d'un état à un autre et la différence les heuristiques des deux bouts de l'arc qui relis les deux états est inférieur au coup de cet arc. C'est à dire $h(n) - h(m) <= k(n, m)$. Ainsi h1 est monotone.

Toute heuristique monotone et coïncidente est minorante. Ainsi, h1 est minorante.

2.9 Question 9

D'après l'algorithme de A* donné dans le cours, on a implémenté la fonction *astar*.


Algorithme A*

PROCEDURE Recherche A*($G, n_0, \Gamma, f, g, h, pere$)

$O \leftarrow \{n_0\}$; $F \leftarrow \emptyset$; $g(n_0) \leftarrow 0$; $n \leftarrow n_0$
tant que $O \neq \emptyset$ et $n \notin \Gamma$ faire
  $O \leftarrow O \setminus \{n\}$; $F \leftarrow F \cup \{n\}$
    pour tous $m \in S(n)$ faire
      si $m \notin O \cup F$ ou $g(m) > g(n) + k(n, m)$ faire
        $g(m) \leftarrow g(n) + k(n, m)$
        $f(m) \leftarrow g(m) + h(m)$
        $pere(m) \leftarrow n$
        ranger m dans O par $f \uparrow$ et $g \downarrow$
      fsi
    finpour
  si $O \neq \emptyset$ alors $n \leftarrow first(O)$
ftq

A la fin si $O = \emptyset$ alors pas de solution sinon *pere* fournit le chemin solution.

29 / 48

FIGURE 6 – L'algorithme de A*

Nous commençons par récupérer les coordonnées du robot bleu et de la cible. Ensuite, nous ajoutons la position du robot bleu à la liste des nœuds ouverts pour commencer le traitement. À chaque itération, nous recherchons le nœud avec la plus petite valeur de f. La valeur f de chaque nœud est calculée comme la somme de g et h, où g est le coût du chemin du nœud de départ au nœud courant et h est une estimation heuristique du

coût pour atteindre le nœud cible depuis le nœud courant. La fonction heuristique h est cruciale pour la performance de l'algorithme A* car elle influence l'ordre d'exploration des nœuds. Si toutes les valeurs de f des nœuds ouverts sont égales, nous regardons alors celui qui a la plus grande valeur de g ; si les valeurs de g sont également égales, nous choisissons le premier élément de la liste des nœuds ouverts.

Il faut ensuite vérifier si la position du nœud courant correspond à celle de la cible. Si c'est le cas, nous retournons le chemin trouvé. Ce chemin peut être construit grâce à la classe *Node*, qui conserve en mémoire les informations relatives à chaque nœud, telles que son parent, sa position, et les valeurs de h, g et f.

Si le nœud courant n'est pas la cible, l'algorithme continue en explorant les mouvements possibles à partir de ce nœud. Le robot peut se déplacer dans quatre directions (haut, bas, gauche, droite). Pour chaque direction, nous utilisons la fonction *no_obstacle* pour vérifier s'il n'y a pas de mur ou d'autre robot bloquant le chemin. Si le déplacement est possible, un nouveau nœud est créé pour cette position, et ses valeurs g, h, et f sont mises à jour en conséquence avant d'être ajouté à la liste des nœuds ouverts pour une exploration future.

L'algorithme se termine soit quand le chemin vers la cible est trouvé, soit lorsque tous les nœuds possibles ont été explorés sans réussir à atteindre la cible, auquel cas il retourne un échec. La performance de l'algorithme A* dépend fortement de la précision de l'heuristique h, et une bonne heuristique peut réduire de manière significative le nombre de nœuds explorés, améliorant ainsi l'efficacité de l'algorithme.

Afin de comparer les résultats obtenus avec ceux de la partie 1, il est essentiel de comprendre les différences fondamentales entre ces méthodes :

* La recherche en largeur explore de manière exhaustive tous les chemins possibles de la manière la plus équitable, en explorant tous les nœuds d'un niveau avant de passer au niveau suivant. Cette méthode garantit la découverte du chemin le plus court, mais peut être très consommatrice en mémoire et en temps.
* L'algorithme A* utilise une fonction de coût $f = g + h$ pour estimer le chemin le plus prometteur vers la cible. Si l'heuristique h est bien choisie, A* peut être nettement plus rapide que les recherches en largeur ou en profondeur, car il explore les chemins les plus prometteurs.

3 Conclusion

En analysant diverses instances du jeu, nous avons comparé l'efficacité des méthodes en termes de temps d'exécution et de qualité des solutions. Nos résultats confirment que la complexité du problème augmente avec le nombre de robots en mouvement et que l'algorithme A*, avec une heuristique bien conçue, peut améliorer significativement les performances par rapport à des méthodes de recherche plus naïves. Cette étude souligne la nécessité d'une sélection stratégique des méthodes de résolution, surtout face à des problèmes de nature NP-difficile, et elle met en lumière la valeur des heuristiques dans l'optimisation des algorithmes de recherche