Rapport de projet IA41 - Rasende Roboter

January 6, 2024

Contents

1 Mise en contexte

Le but de ce projet est de mettre en application les différents algorithmes vus en cours pendant le semestre d'automne 2023 en **IA41** (Breadth-First Search, A*, Depth-first search).
Ces algorithmes vont être appliqués sur un jeu de société allemand, **Rasende Roboter** ou Ricochet Robots en français, créé en 1999 par Alex Randolph.

Nous étions trois à réaliser ce projet. Pour nous permettre de s'organiser au mieux, nous avons choisi de faire un répertoire **Github**. Notre raisonnement a été assez simple concernant l'approche du projet. Nous avons décidé d'implémenter premièrement le jeu pour ensuite y incorporer les algorithmes de résolution.

Pour la suite du rapport, "Intelligence artificielle" sera remplacée par "IA".

2 Implémentation du jeu

Nous avons fait le choix de prendre la première version du jeu. En effet une autre est apparu en 2003, qui ajoutait un robot noir, une case mission multicolore et des agencements de murs différents.

2.1 Règles du jeu

Les règles sont assez simple. Le joueur dispose d'un plateau de 16x16 cases. Sur celui-ci est disposé 4 robots de couleurs différentes (rouge, vert, bleu et jaune) et des jetons "missions". Une case centrale permet d'afficher la mission à atteindre par le robot de la couleur de celle-ci.
Le but est pour le joueur d'atteindre la mission avec le robot de la couleur correspondante en un minimum de coup et un minimum de temps. Pour cela il peut déplacer tous les robots comme il le souhaite.

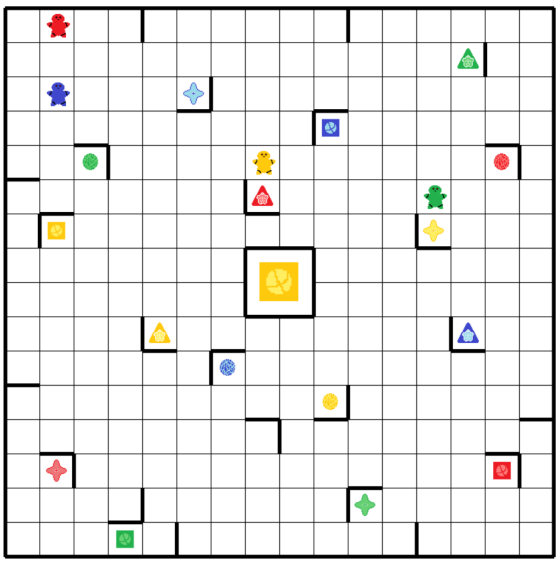

Figure 1: Image du plateau de jeu "Rasende Roboter"

Les déplacements sont assez simple également. Les robots ne peuvent suivre que 4 directions (haut, bas, gauche, droite). De plus une fois qu'une direction est prise, le robot en question ne s'arrête que lorsqu'il croise un mur ou un autre robot.

Chaque partie commence par le tirage d'une mission. Une fois la mission choisit, un compte à rebours est lancé. Pendant ce temps les différents joueurs analysent la position de chaque robot et essayent, dans leur tête, de trouver un moyen de résoudre le problème. Pour faciliter la compréhension, si on regarde la Figure 1, le robot jaune doit atteindre la cible affichée au centre du plateau. Ici elle se situe aux coordonnées (7,2).

Si un joueur à une solution, il fait signe et donne sa réponse. Si elle est bonne, il gagne, sinon les autres tentent de résoudre le problème.

2.2 Choix et explications de notre implémentation

Le but du projet étant d'implémenter différents algorithmes de résolutions, il nous est paru évident de modifier certains points du jeu.

Dans la version physique, le jeu se joue à plusieurs, chacun pour soi. Nous avons décidé que notre jeu pouvait se jouer seul ou à plusieurs contre les algorithmes de résolutions. Ainsi le but de chaque partie est de résoudre le problème en moins de coups possibles certes, mais surtout en moins de coups que ces algorithmes.

Lorsqu'on arrive sur le jeu, on peut choisir la difficulté de l'IA (easy, medium, hardcore).

On peut également choisir de "reset" la manche si on voit que l'on bloque. Un mode revisionnage a été implémenté pour voir les coups effectués par l'IA. Ce mode est uniquement disponible lorsque la manche est terminée et que l'IA a trouvé une solution.

Au niveau de l'ergonomie de jeu. Il se joue uniquement grâce à une souris ou un pavé tactile. Lorsque l'on clique sur un robot, un chemin s'affiche pour aider le joueur à voir les cases atteignables. Ensuite il peut cliquer sur les différents chemins possibles pour déplacer le robot.

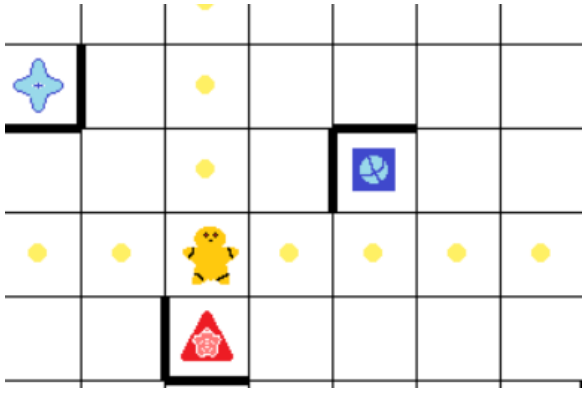

Figure 2: Exemple de déplacement du robot jaune

Pour ce qui est de l'implémentation du jeu uniquement, nous vous détaillerons certaines méthodes dans la partie réservée aux algorithmes de résolution. Mais pour le reste, nous avons décidé de faire uniquement un diagramme de classes pour rendre plus visuel nos explications.

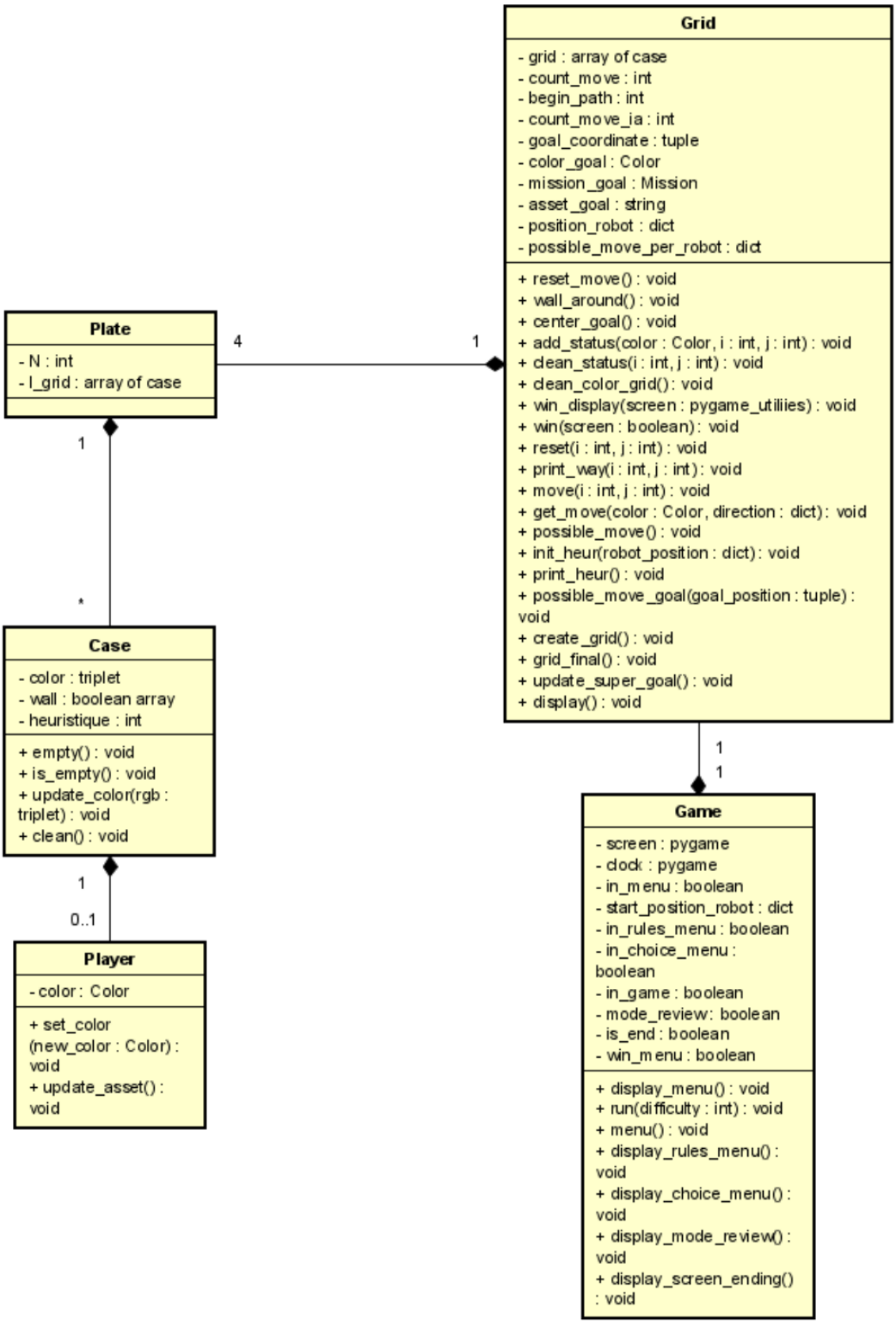

Figure 3: Diagramme de classes du jeu Rasende Roboter

3 Algorithmes de résolution

Le jeu, Rasende Roboter, est à la fois un problème de **planification** et à la fois un problème de **décision séquentielle**. Plus précisément un problème de planification résolu de manière séquentielle.

Avant de commencer à implémenter la résolution, il nous a fallu **comprendre** le problème. Cette étape à été plutôt simple grâce aux règles du jeu. Les **données** du problème sont :

1. la position des murs dans la grille
2. la couleur, l'identité et les coordonnées de la mission à atteindre dans la grille
3. la couleur et les coordonnées des robots
4. les déplacements possibles des robots

Notons que (1) et (4) peuvent être fusionnés en une unique donnée.

Ensuite la **condition** de résolution du problème est la suivante : le robot de la couleur de la mission à atteindre doit se situer sur les coordonnées de cette mission. Ce problème ne présentait pas d'**inconnues** mise à part le choix de la mission à atteindre en début de partie.

Une fois toutes ces étapes de reconnaissance effectuées et toutes les zones d'ombres écartées, il nous a fallu **élaborer un plan de résolution du problème**. Nous avons utilisé une approche ***"Divide and Conquer"*** pour y voir plus clair et résoudre plus efficacement ce problème.

3.1 Travail préliminaire

La première partie de notre travail a été d'implémenter le jeu tout en gardant à l'esprit l'ajout des algorithmes de résolutions. En effet un mauvais démarrage aurait été chronophage par la suite alors que l'inverse nous a permis d'être très efficace.

En implémentant le jeu, il nous a fallu un moyen d'extraire les **données** explicitées (***voir 3***) lors de notre préparation. L'élément central de notre implémentation est la classe "grid" (***voir Fig:3***). En effet les méthodes suivantes permettent d'extraire des données cruciales.

- actualize_robot_position() permet de créer un dictionnaire, position_robot, ayant pour clefs la couleur de chaque robot et pour valeurs leur coordonnée dans la grille.

Ainsi lors de l'appel de cette méthode, le dictionnaire suivant est créé, enregistrant à l'instant t la position de chaque robot. Le point (3) a été résolu.

```
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (2, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
```

Figure 4: Dictionnaire position_robot

Ensuite pour résoudre le point (2) nous avons créé un tableau statique de missions sous cette forme.

```
mission_tab.append((6,1,Color.YELLOW,Mission.SQUARE))
```

Figure 5: Tableau de missions

Ainsi lorsque l'on initialise la grille on peut remplacer ses attributs goal_coordinate et color_goal par les élements du tableau statique de missions, en tirant aléatoirement une position dans le tableau.

Les points (1) et (4) ont été les plus délicats. Ils impliquent une multitude de méthodes de la classe "grid".

Le but était d'avoir une méthode qui initialisait un dictionnaire qui avait pour clefs la couleur de chaque robot et pour valeur une liste de dictionnaires de case atteignable avec pour clefs la direction et pour valeur la coordonnée de la case. Si cela vous paraît trop compliqué voici un exemple.

```
{<Color.BLUE: 3>: [{'RIGHT': (2, 5)}, {'LEFT': (2, 0)}, {'UP': (1, 1)}, {'DOWN': (5, 1)}]
```

Figure 6: Mouvements possibles par robot

Pour arriver à ce résultat il faut actualiser la position des robots dans la grille grâce à la méthode actualize_robot_position() et ensuite il faut voir sur les 4 directions (UP, RIGHT, DOWN, LEFT) quels chemins peuvent emprunter le robot.

Pour cela on se sert de deux informations. La première c'est que le robot avance jusqu'à ce qu'il croise un mur. Pour cette condition il suffit de regarder l'attribut "wall" de la classe "case" qui est un tableau de 4 booléens représentant la présence ou non de mur respectivement dans les 4 directions à une case donnée de la grille.

Ainsi si le robot veut aller en haut, il suffit de le faire avancer jusqu'à ce que la case sur laquelle il est, contient un mur dans la direction "UP" du tableau de booléen "wall". Il suffit de vérifier " wall[0] == True ? Stop : Continue ".

La deuxième information est la suivante. Un robot peut arrêter sa progression si il croise un autre robot sur son chemin.

Cette condtion est assez simple à verifier si on se sert du dictionnaire position_robot. On fait avancer le robot dans une direction en vérifiant à la n-ième "case + 1" qu'un autre robot ne s'y trouve pas. Si c'est le cas, on s'arrête, sinon on continue.

3.2 Représentation d'un état

Pour résoudre ce problème, il a fallu définir la représentation **d'un état** qui permettait d'illustrer l'entièreté de notre problème à un instant donné.

Grâce à notre travail préliminaire, un état à l'instant t sera représenté par le dictionnaire vu à la figure 4. Il ne faut pas oublier que la position des robots ne vaut rien sans la grille correspondante.

Ensuite il nous a fallu définir **un système de production**. Pour cela, on se servira du dictionnaire vu à la figure 6.

Il a fallu ensuite définir un **état initial** et un ou des **état/s final/finaux**. Pour cela il suffit de prendre la position de départ de chaque robot (figure 4) pour l'état initial.

Pour les états finaux il suffit que la robot de la couleur de la mission soit sur les coordonnées de celle-ci et ça peu importe où se trouvent les autres robots.

Une fois tout ce travail effectué, il faut maintenant appliquer **des stratégies de résolutions**.

3.3 Stratégies de recherche dans un arbre

3.3.1 Représenter les mouvements du jeu

Pour résoudre un problème du jeu Rasende Roboter, il faut produire des états à l'instant t + 1 valident à partir d'un état à l'instant t. Ces états produits sont liés à leur état père. Si on applique ce raisonnement sur chaque état produit à l'étape t + 1, t + 2 , etc... On se retrouve à constuire une arborescence de tous les coups possibles pour chaque robot au cours de la partie. La racine de cette arbre sera alors **l'état initial** et le but sera alors de chercher un des **états finaux** dans l'arborescence.
Chaque étage de l'arbre représente le nombre de déplacements depuis l'état initial. La figure 7 montre l'état au niveau n = 0 et les états au niveau n = 1. Faisons maintenant un peu de mathématiques, si chaque robot peut se déplacer dans les 4 directions à chaque tour. Sachant qu'il y a 4 robots sur le plateau. Les états au niveau n = 1 seront au nombre de 16 car 4 x 4 = 16.

Maintenant si on suppose qu'au niveau n = 2 chaque état du niveau n = 1 peut encore générer 16 autres états. On aura 16 x 16 = 256 + 1 états dans l'arbre générer. Et cela seulement pour vérifier toutes les configurations possibles en deux coups.

Bien sûr, chaque état ne génère pas tout le temps 16 autres états car certains robots ne peuvent que faire certains déplacements suivant leur position.

Mais en extrapolant, on voit vite la croissance exponentielle du nombre d'états générés suivant le nombre de coups nécessaires à la résolution du problème. En règle générale, le nombre d'états selon le nombre de coups n se calcul:

$$\forall n \in \mathbb{N}^*, 16^n + 1$$

Comment à partir d'un état, peut-on générer les autres états ?

Ceci est plutôt simple, en se servant de la méthode de la classe "grid" possible_move() , des dictionnaires position_robot et possible_move_per_robot.

En effet quand on arrive sur un état on actualise la position des robots pour être à jour, ensuite on parcourt le dictionnaire possible_move_per_robot et on créer un nouvel état en remplaçant uniquement une coordonnée d'un robot et en gardant les autres coordonnées intactes.

Puis on suit cet algorithme pour générer autant d'états que l'on veut. La fonction next_state() dans le fichier bfs_dfs.py est chargée de ce processus de création d'état.

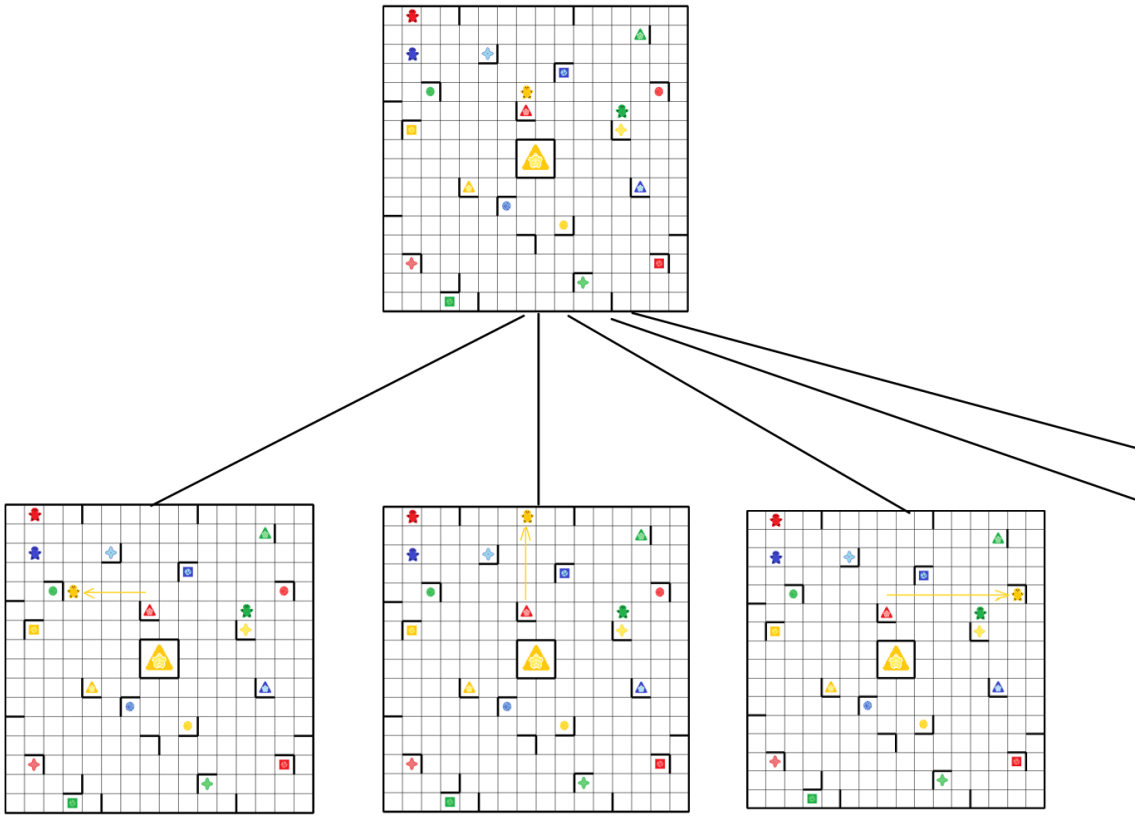

Figure 7: Construction de l'arborescence

Deux fonctions pratiques ont été implémentées pour la fonction next_state(). add_status_empty_grid() et clean_all_status() qui permettent respectivement d'ajouter à partir d'un dictionnaire de la forme du dictionnaire position_robot les robots sur la grille et de retirer les robots de la grille.

Ainsi pour la suite une seule grille suffira pour tous les traitements et tous les algorithmes de résolutions. Ce qui préserve le stockage en mémoire durant l'éxécution.

Maintenant que tous les problèmes liés à la représentation des états ont été réglé, nous pouvons passer aux choses sérieuses.

3.4 Breadth First Search : BFS

Pour résoudre une partie de Rasende Roboter, il suffit maintenant de constuire l'arbre des états dynamiquement et de le parcourir pour trouver un des états finaux.

Le breadth first search ou largeur d'abord en français est un algorithme de recherche dans un arbre qui dans notre cas nous permettra de trouver, si solution il existe, le meilleur état final possible. C'est à dire celui en moins de coups. Ci-dessous, l'algorithme du BFS.

Algorithm 1 Breadth First Search

```
1: Entrée: noeud racine N
2: Créer FIFO Q, Créer Liste L
3: Q.append(N)
4: while Q n'est pas vide et que le but ne soit pas atteint do
5:    Enlever N en tête de liste
6:    Ajouter L.append(N)
7:    for tous les fils de N, noté f do
8:       f.père = N
9:       Q.append(f)
10:   end for
11: end while
12: Si le but a été trouvé retourner Succès sinon retourner Echec
```

Après avoir implémenté l'algorithme. Nous avons pu effectuer les premiers essais. Pour être rigoureux nous avons mesuré le temps que prenait l'algorithme pour trouver une solution, le nombre d'états parcouru et ainsi le nombre d'états parcouru par seconde.

Mais nous avons vite remarqué les limites de cette méthode. Sans améliorations il était presque impossible de trouver des solutions à plus que 7 coups. On le rappel chercher une solution à 7 coups revient au pire des cas à constuire et visiter un arbre de 16 x 16 x 16 x . . . x 16 (7-fois) + 1 états soit 268 435 457 états.

Au meilleur de sa performance notre algorithme visitait 5000 états / seconde. Pour 7 coups il lui aurait fallu 268 435 457 / 5 000 = 53 687 secondes soit 14 heures pour tenter de trouver cette solution. Ceci n'est pas raisonnable et bien souvent il y avait un stack overflow au bout d'un certain temps d'interprétation.

3.4.1 Améliorations : BFS

Nous avons ensuite vu que les solutions qu'il trouvait rapidement étaient celles nécessitant le déplacement d'un seul robot. Nous avons donc réduit la génération des états. La fonction is_already_visited()

permet de générer seulment les états où un robot n'est pas passé. Ainsi les 4 robots vont aller sur toutes les cases possibles une unique fois. De plus, au lieu de vérifier si le noeud qu'on traite est le noeud final. On vérifie à la génération de chaque état si celui n'est pas un état final. Ainsi on s'évite la vérification et la génération inutile d'états.

Avec toutes ces améliorations voici un exemple sur un échantillon, même cible mais avec les paramètres différents sur la vérification.

```
We have found a path
Final node:  {<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (9, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
Mission color:  Color.BLUE
Mission color:  (9, 13)
Time elapsed: 0.036047935485839844 seconds
Number of state that have been visited:174
State/s: 4827
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (2, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (1, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (1, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (9, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
```

Figure 8: BFS sans la vérification à la génération

```
We have found a path
Final node:  {<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (1, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
Mission color:  Color.BLUE
Mission color:  (9, 13)
Time elapsed: 0.008579015731811523 seconds
Number of state that have been visited:43
State/s: 5012
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (2, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (1, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (1, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
{<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (9, 13), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (5, 12)}
```

Figure 9: BFS avec la vérification à la génération

On peut voir que la vérification à la génération épargne la visite de 131 états, ce qui permet d'augmenter la vitesse de résolution avec un rapport de 3,75 ce qui n'est pas négligeable.

Pour conclure notre BFS permet de résoudre 80 % des problèmes sur la grille fournit dans notre jeu. Le niveau de difficulté associé dans le jeu sera "Medium".

3.5 Depth First Search : DFS

En ce qui concerne le depht first search, son implémentation a été rapide puisque son principe est le même que le breadth first search mais au lieu de parcourir l'arbre en largeur , le DFS lui le parcours en profondeur. La solution trouvée sera alors avec un nombre de coups beaucoup plus élevés. Voici son algorithme.

Algorithm 2 Depth First Search

Entrée: noeud racine N
Créer LIFO Q, Créer Liste L
Q.append(N)
while Q n'est pas vide et que le but ne soit pas atteint **do**
  Enlever N en tête de liste
  Ajouter L.append(N)
  for tous les fils de N, noté f **do**
    f.père = N
    Ajouter en tête de Q,les fils notés f
  end for
end while
Si le but a été trouvé retourner Succès sinon retourner Echec

3.5.1 Améliorations : DFS

Les mêmes améliorations que le BFS ont été implémenté. Il faut néanmoins noter que cet algorithme est plus adapté lorsque l'arbre de recherche a un grand facteur de branchement c'est à dire lorsque le nombre de noeuds au niveau suivant est plus important que celui au niveau actuel. Ce qui est notre cas ici.

```
We have found a path
Final node:  {<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (2, 1), <Color.YELLOW: 1>: (4, 7), <Color.GREEN: 2>: (13, 2)}
Mission color:  Color.GREEN
Mission color:  (4, 2)
Time elapsed: 0.20952725410461426 seconds
Number of state that have been visited:487
State/s: 2324
*********BFS_SOLUTION**********
```

Figure 10: Algorithme BFS sur la cible verte en (4,2)

```
We have found a path
Final node:  {<Color.RED: 4>: (0, 1), <Color.BLUE: 3>: (2, 1), <Color.YELLOW: 1>: (0, 0), <Color.GREEN: 2>: (13, 2)}
Mission color:  Color.GREEN
Mission coordinate:  (4, 2)
Time elapsed: 0.009041786193847656 seconds
Number of states that have been visited: 26
States/s: 2876
*********DFS_SOLUTION**********
```

Figure 11: Algorithme DFS sur la cible verte en (4,2)

On le voit mieux sur cet exemple ici, sur une même cible, le DFS parcours seulment 26 états contre 487 pour le BFS pour trouver une solution. Certes, le DFS propose une solution en 26 coups contre 5 pour le BFS. Mais le DFS va environ 23 fois plus vite que le BFS.

Le DFS sera le niveau de difficulté "Easy" dans notre jeu puisque qu'il propose des solutions avec un grand nombre de coups.

3.6 A*

L'objectif de l'algorithme A* est de rechercher le plus court chemin dans un graphe partant d'un état initial pour aller à un état final. En principe on va associer à chaque état une fonction qui donnera une valeur.

Cette valeur va être ensuite utilisée dans l'algorithme. La fonction se décompose en deux parties, un coût qui représente un déplacement du robot et une **heuristique** qui sert à donner une estimation de la distance qu'il reste à parcourir pour le robot.

Un de nos problèmes a été le choix de notre heuristique. La première a été la distance euclidienne avec des résultats peu concluant, car l'heuristique n'était pas admissible pour ce problème. La deuxième a été celle que nous avons retenue. A chaque début de partie on créer une grille d'entier parallèle à celle du jeu.

On initialise la case où se trouve la cible à 0. Puis on remplit de 1 les cases accessibles en ligne et en colonne depuis cette case. On effectue ça ainsi de suite en augmentant de n+1 le numéro des cases. Ce numéro représente alors le nombre de coups qu'il faudra faire pour atteindre la cible si le robot se situe sur cette case. Voici un exemple.

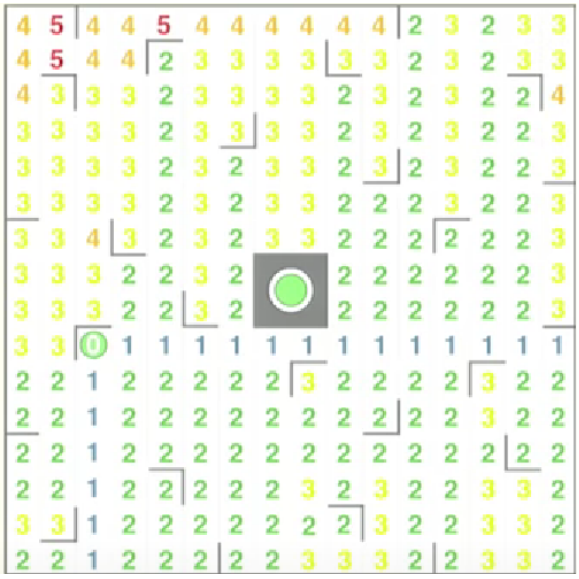

Figure 12: Exemple de grille heuristique par Randy Coulman

En pratique, une fois l'heuristique mise en place grâce à la méthode initHeur() de la classe grid, il suffit d'appliquer l'algorithme à notre problème.

Algorithm 3 A*

```
    Entrée: noeud racine N
    Créer liste Opened, Créer Liste Closed
 3: Opened.append(N)
    while Opened n'est pas vide do
      Enlever le noeud dont f(noeud) est minimal
 6:   Ajouter Closed.append(noeud)
      if noeud appartient aux noeuds finaux then
        return Succès
 9:   else
        for tous les fils du noeud, noté y do
          if y n'appartient pas à Closed union Open ou g(y)> g(noeud)+(noeud,y) then
12:         g(y) = g(noeud) + c(noeud,y)
            f(y) = g(y) + h(y)
            pere(y) = noeud
15:         Opened.append(y)
          end if
        end for
18:   end if
    end while
    Si le but a été trouvé retourner Succès sinon retourner Echec
```

Après des tests nous n'avons pas eu des résultats très positifs puisque le taux de réussite était presque nul. Nous avons donc implémenté une "quête secondaire". Plus précisément, avant de déplacer le robot devant aller sur la cible on place bien les autres robots. Ainsi le robot principal utilise le A* pour atteindre la cible.

Malheureusement, cet algorithme devant donner des meilleurs résultats que le BFS n'est pas abouti. Le code est disponible mais il n'est pas utilisé dans notre implémentation.

3.6.1 Améliorations : A*

Malgré nos difficultés pour la mise en place de cet algorithme, nous avons des pistes pour l'améliorer grandement !

On pourrait associer un poids à un état. C'est-à-dire grâce à la grille décrite dans la figure 12, on créé une méthode qui somme les poids de chaque robot. Ainsi quand on va tirer le noeud avec le poids minimal dans la liste *Opened*, on choisira l'état avec le poids de plus faible.

C'est-à-dire que tous les robots se rapprocheront de la cible. Comme ça on évite l'implémentation des quêtes secondaires. On poserait évidemment une condition sur le déplacement des robots non principaux pour qu'ils ne prennent pas place sur les cases de poids strictement inférieur à deux.

4 Améliorations possibles du jeu

Notre jeu est jouable dans 2 niveaux de difficulté sur 3. On peut, à la fin de chaque manche visualiser les coups qui ont été produit par l'IA.

On pourrait augmenter le nombre de manche contre l'IA durant une partie. On pourrait également générer aléatoirement de nouvelles grilles pour permettre aux joueurs de se challenger davantage

et tester nos algorithmes sur ces nouvelles grilles. Le A* doit être implémenté correctement par la suite.