Rapport de projet IA41 - Rasende Roboter

SENGEL Noé, POURCINE Mattéo, FLEURET Gabriel

December 18, 2023

Contents

1 Mise en contexte

Le but de ce projet est de mettre en application les différents algorithmes vu 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 choisit 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.

Plusieurs sources ont été utiles à la réalisation de ces travaux. Vous les trouverez en fin de rapport. 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 mur différents.

2.1 Règles du jeu

Les règles sont assez simple. Le joueur dispose d'un plateau de 16x16 cases. Sur celui-ci est diposé 4 robots de couleurs différentes (rouge, vert, bleu et jaune) et des jetons "missions". Une case centrale permet d'afficher la mission a 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.

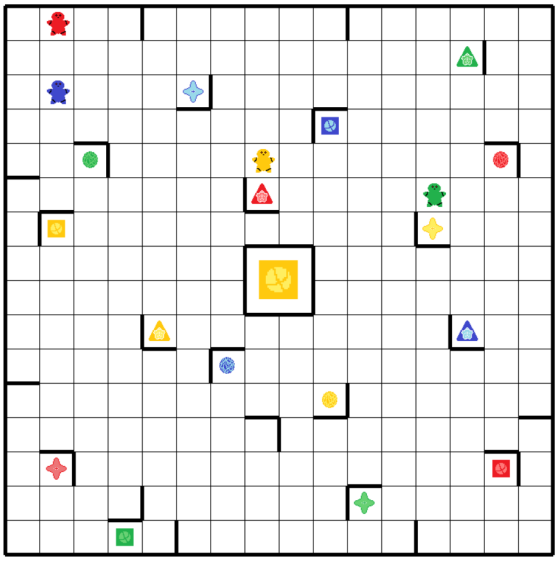

Figure 1: Image du plateau de jeu "Rasende Roboter"

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 rebourt est lancé. Pendant ce temps les différents joueurs analyse la position de chaque robot et essaye, 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 afficher 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 ait paru évident de modifier certains points du jeu.

Dans la version physique, le jeu se joue à plusieurs, chacun pour soi. Nous avons décider 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 coup possible certes, mais surtout en moins de coup que ces algorithmes.

Lorsque qu'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 à 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 peu cliquer sur les différents chemin possible pour déplacer le robot.

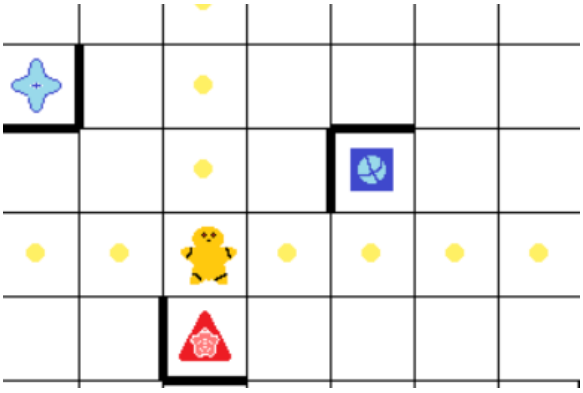

Figure 2: Exemple de déplacement du robot jaune

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

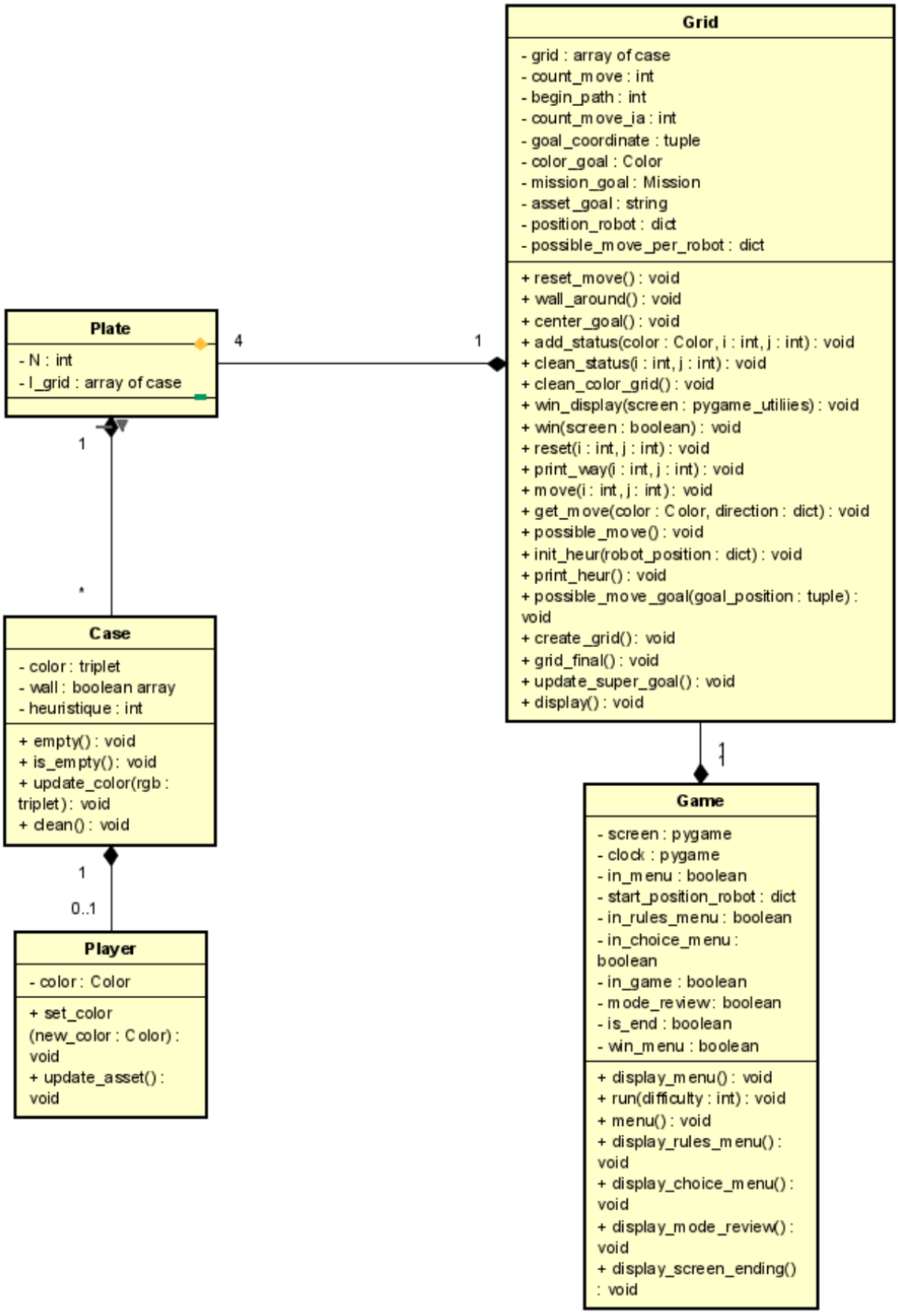

Figure 3: Diagramme de classe du jeu Rasende Roboter

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 fallait **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 ses étapes de reconnaissance effectuée et toutes les zones d'ombres écartées, il nous a fallu **élaborer un plan de résolution du problème**. Nous avons utilisée 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.

Une fois le jeu implémenté, il nous a fallu un moyen d'extraire les **données** explicitées (***voir 3***) lors de notre préparation. L'élément centrale de notre implémentation est la classe "grid" (***voir Fig:3***). En effet les méthodes suivantes permettre 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 valeur leur coordonnée dans la grille.

Ainsi lors de l'appel de cette méthode, le dictionnaire suivant est créé, enregistrent 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 (***voir Améliorations***) 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 attibuts goal_coordinate et color_goal par les élements du tableau statique de mission. En tirant aléatoirement une position dans le tableau.

Les points (1) et (4) ont été les plus délicats. Ils impliquent une multitudes 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 dictionnaire de case atteignable avec pour clefs la direction et pour valeur la coordonnées de la case. Si cela vous parait trop compliqué voici un exemple.

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

Figure 6: Tableau de missions

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) quelles chemins peut 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.