Simulation de robots mineurs

SAÉ 2.01-SAÉ 2.02

Réalisé par TRAN Minh-Tue

Membre de groupe :
TRAN Minh-Tue
SAINZ Miquel
SHIDOU sidali

Introduction

Dans le cadre de la SAE 2.02, les robots peuvent effectuer des tâches d'exploration et d'exploitation de manière autonome. Ils doivent respecter les règles suivantes : une action par tour.

Pendant un tour, le robot peut effectuer l'une des actions suivantes :

- Exploiter une mine d'or ou de nickel (extraction de 1 à 3 unités)
- Déposer les ressources dans un entrepôt (tout déposer en une fois)
- Se déplacer d'une case dans une direction quelconque

Le monde doit être inconnu.

Github

https://github.com/Minhtue1203/robot-miner

Environnement développement

- Java 21
- Javafx
- Junit
- Maven

Exécution du projet

Nous pouvons lancer le jeu via le fichier .jar ou en démarrant le fichier HelloApplication.java du projet.

Interface du jeu

L'interface du jeu est développé par Javafx

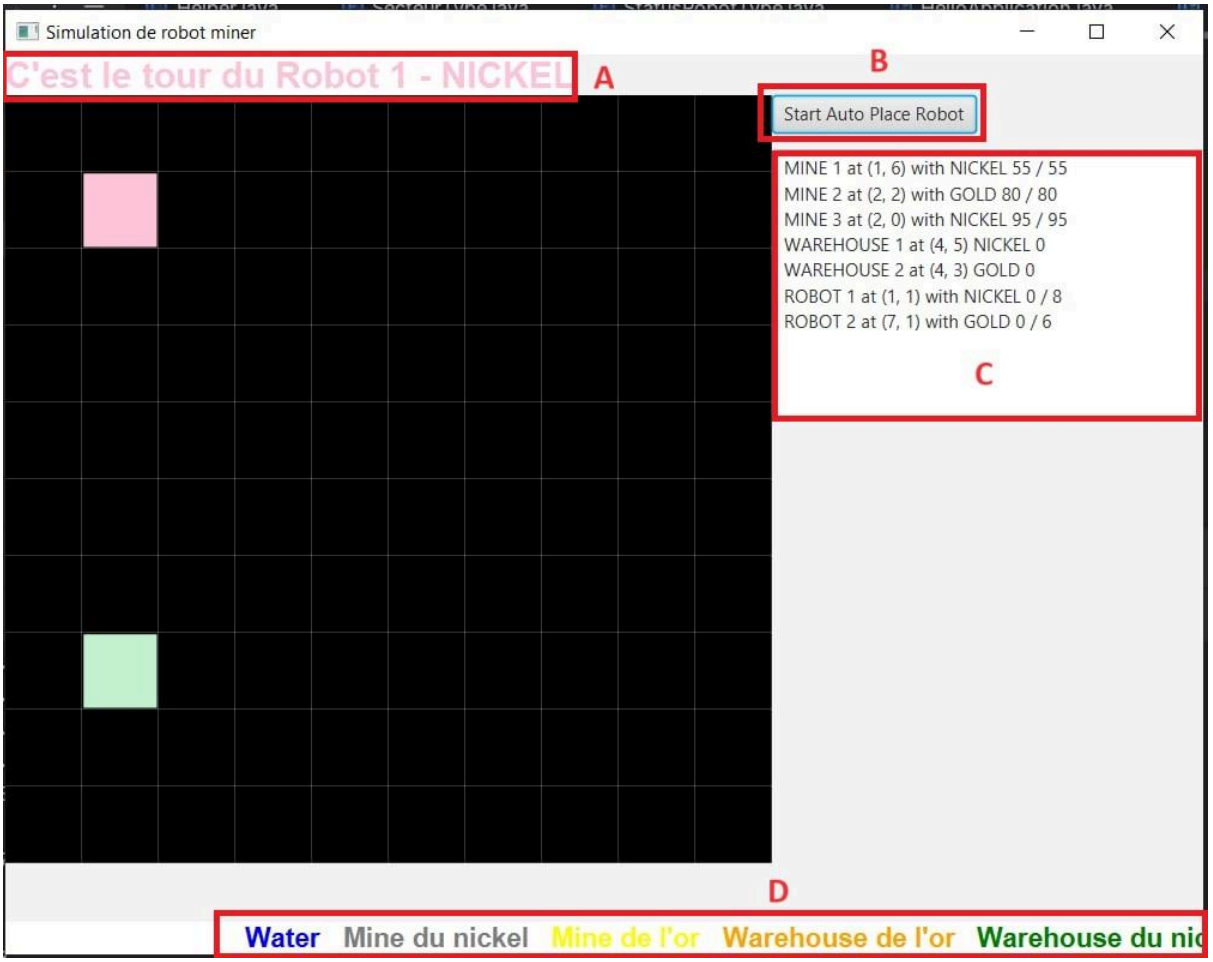
Note:
A - Le libellé affiche l'état du jeu, notamment le tour du robot, et indique si le jeu est terminé. Sa couleur peut changer en fonction de celle du robot. Par exemple, le Robot 1 est en rose.

B - Le bouton permet de démarrer le jeu en mode automatique.

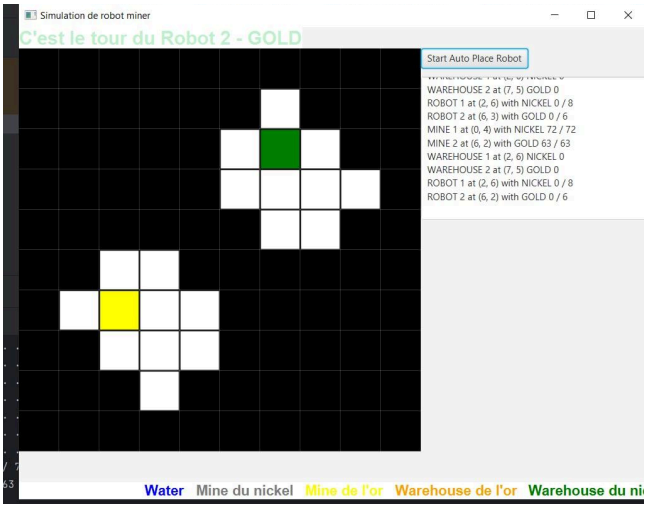
C - Les logs permettent de voir l'état des robots, des minéraux et des entrepôts dans le monde.

D - L'explication des couleurs représente les objets dans la grille.

Les secteurs non encore explorés par le robot sont en noir.



Le secteur exploré par le robot, ce qui est en blanc.

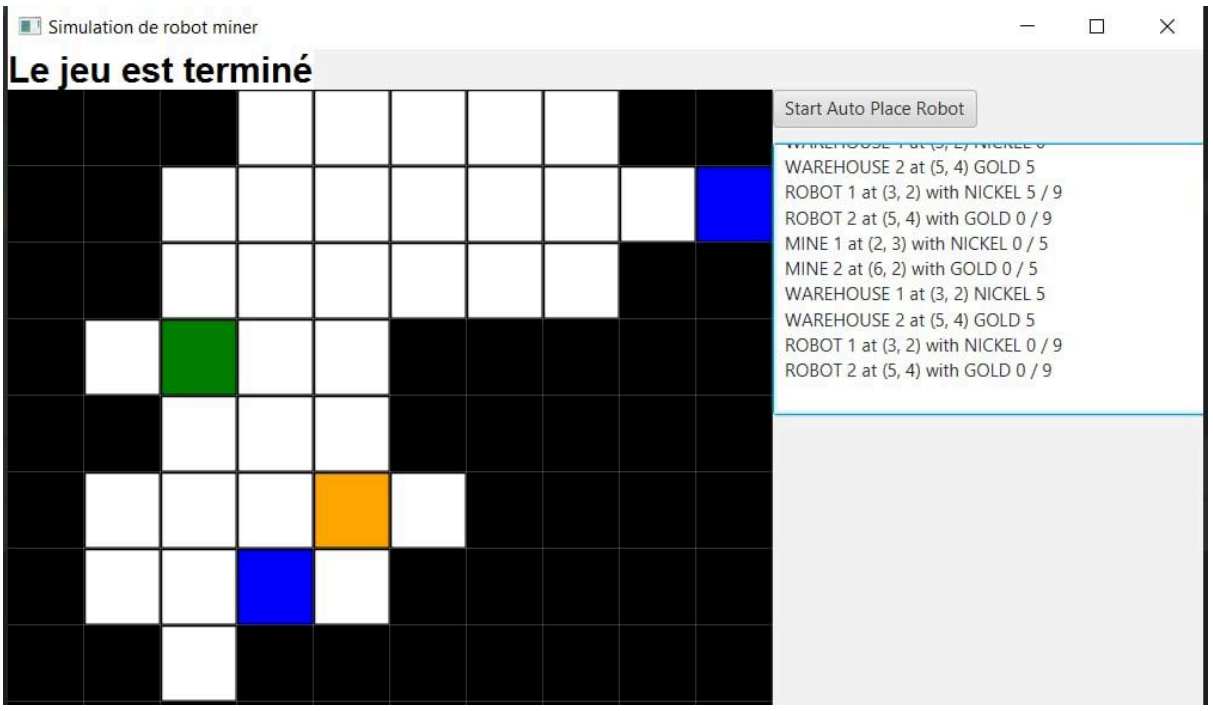


À partir du moment où l'on clique sur le bouton "Start Auto", le jeu se déroule automatiquement en boucle dans l'ordre suivant :

- Selon le type du robot, chercher les mines.
- Les récolter jusqu'à la capacité maximale.
- Chercher l'entrepôt correspondant au type de minéraux
- Les déposer
- Revenir aux mines où il était. Dans le cas où ces mines n'ont plus de minéral, il cherche d'autres mines.

Le jeu se termine lorsqu'il n'y a plus de mines dans le monde et que les robots ont déposé tous les minéraux récoltés.

P.S. : Sur la photo, j'ai testé avec des mines contenant 5 minerais.



Explication de l’algorithme

Dijkstra

J’utilise l’algorithme Dijkstra pour explorer le monde et chercher le chemin le plus court du départ de robot à la destination (la mine/l’entrepôt). L’algorithme est développé par la class *AutoMine.java - models/Automine.java* et la *Node.java - models/Node.java*.

La classe ***Node*** qui représente un nœud dans la grille. Cette classe est utilisée dans la file de priorité pour l'algorithme de Dijkstra. Elle permet d’enregistrer la position x, y et la distance minimale de la source à ce nœud (utilise à l’aide de *Comparable*)

La classe ***AutoMine*** qui gère la logique pour trouver le chemin le plus court vers une mine ou un entrepôt.

1. Fonction ***dijkstra(int startX, int startY)*** :

Implémentez l'algorithme de Dijkstra pour trouver le chemin le plus court à partir d'une position de départ (startX, startY)

Pour ce faire,j'initialise les distances minimales de la source à chaque cellule à l'infini.

Je crée une file de priorité en utilisant PriorityQueue pour explorer les nœuds avec la distance minimale. J'ajoute le nœud de départ avec une distance de 0 à la file

Ensuite, je mets cette exploration dans une boucle qui continue tant que la file de priorité n'est pas vide. La boucle extrait le nœud avec la distance la plus faible. Si ce nœud a déjà été visité, il est ignoré. Sinon, il est marqué comme visité.

Pour l’exploration des voisins, chaque direction possible (haut, bas, gauche, droite), je calcule les nouvelles coordonnées (newX, newY). Je vérifie si la position est valide et si elle n'est pas de l'eau. Je calcule la nouvelle distance pour atteindre cette position. Si cette nouvelle distance est plus courte que la distance actuellement enregistrée pour cette position, je mets à jour la distance et le prédécesseur. Ensuite, j'ajoute le nouveau nœud à la file de priorité.

2. Fonction ***List<int[]> reconstructPath(int startX, int startY, int endX, int endY)***

Cette fonction reconstruit le chemin du point de départ au point cible en utilisant les prédécesseurs enregistrés. Elle est appelée après j’ai trouvé la position (x, y) du cible dans la fonction *findPathToNearestMine*

p/s: les prédécesseurs est un tableau 2D pour stocker le prédécesseur de chaque cellule dans le chemin le plus court.

3. Fonction ***List<int[]> findPathToNearestMine(Robot robot)***

Elle retourne le chemin le plus court vers la mine la plus proche qui correspond au type de minéral du robot, ou vers l'entrepôt pour y déposer le minéral.

Pour ce faire, j'exécute la fonction de *Dijkstra* pour explorer le monde et déterminer la distance minimale du départ aux positions dans la grille.

Ensuite, je lance une boucle pour rechercher la mine ou l'entrepôt le plus proche. Selon le statut du robot, s'il est en mode "Finding", je cherche une mine qui contient encore des minéraux. S'il est en mode "Depositing", je cherche l'entrepôt.

P.S. : Dans le cas où une mine n'a plus de minéraux, elle sera supprimée de la grille.

Enfin, j'appelle la fonction *reconstructPath* pour reconstruire le chemin du départ jusqu'à la position cible.

Résumé de la stratégie de l'algorithme

Voici les étapes principales :

1. Initialiser toutes les distances à l'infini et la distance de la source à zéro.
2. Utiliser une file de priorité pour explorer les nœuds avec la distance minimale.
3. Mettre à jour les distances des cellules adjacentes si un chemin plus court est trouvé.
4. Marquer les cellules comme visitées une fois qu'elles sont extraites de la file de priorité.
5. Continuer jusqu'à ce que toutes les cellules accessibles soient visitées.
6. Utiliser les prédécesseurs pour reconstruire le chemin vers la destination

Ces fonctions collaborent pour permettre au robot de trouver efficacement le chemin le plus court vers une cible pertinente en tenant compte des obstacles et des exigences spécifiques du robot.

Déplacement

Le robot a 3 mode:

- **Finding** : Chercher une mine et se diriger vers la mine.
- **Mining** : Extraire des minéraux.
- **Depositing** : Chercher l'entrepôt, déposer des minéraux ou se diriger vers l'entrepôt.

En cliquant sur le bouton "Start Auto", on déclenche la fonction *startAutoPlaceRobot () - view/GridView.vue*. Le robot commence en mode Finding. Je démarre la fonction *autoPlaceRobotsForMine()* qui retourne une liste des chemins les plus proches pour chaque robot. Ensuite, je mets la liste dans une boucle en utilisant *ScheduledExecutorService*, ce qui me permet de mettre à jour la vue toutes les secondes pour voir le déplacement du robot depuis son départ jusqu'à sa destination (la mine dans ce cas).

Lorsque le robot arrive la mine (correspondant au type du robot), il passe en mode Mining.

Ici, on entre dans une boucle : de l'entrepôt à la mine <-> de la mine à l'entrepôt, en utilisant les fonctions *loopHarvestResources et loopDepositResources (situées dans controller/GridController.java)*

Fonction *loopHarvestResources* : Elle permet au robot de récolter et de mettre à jour la quantité de minéraux. Dans le cas où le robot atteint sa capacité maximale ou que la mine n'a plus de minéraux, le robot passe en mode Depositing et j'appelle *autoPlaceRobotsForDeposit()* pour chercher et mettre à jour le chemin du robot.

Fonction *loopDepositResources* : Elle permet au robot de déposer les minéraux et de mettre à jour la quantité de stockage. Dès que le robot a terminé de déposer, il passe en mode Finding et j'appelle *autoPlaceRobotsForMine()* pour chercher la mine la plus proche et s'y diriger.

Le jeu se termine lorsqu'il n'y a plus de mines dans le monde et que tous les robots ont déposé les minéraux extraits.

Conclusion

Grâce à ce projet, j'ai eu l'occasion de découvrir et de pratiquer les algorithmes PEL, PEP et Dijkstra. Dans mon projet, j'ai choisi Dijkstra parce qu'il répond aux conditions que je recherche en termes de performances pour trouver les chemins les plus courts. J'ai réussi à l'appliquer dans mon projet, et l'application peut se jouer en mode automatique.